Verified Complete Test Strategies for Finite State Machines

Robert Sachtleben

September 1, 2025

Abstract

This entry provides executable formalisations of the following testing strategies based on finite state machines (FSM):

1. Strategies for language-equivalence testing on possibly nondeterministic and partial FSMs:
   - W-Method [1]
   - Wp-Method (based on a generalisation of [4] presented in [5])
   - HSI-Method [3]
   - H-Method [2]
   - SPY-Method [10]
   - SPYH-Method [11]
2. Strategies for reduction testing on possibly nondeterministic FSMs:
   - Adaptive state counting (as described in [6])

These strategies are implemented using generic frameworks which allow combining parts of strategies such as reaching and distinguishing of states or distributing traces over classes of convergent traces. Further details are given in the corresponding PhD thesis [8] and tools employing the code generated from this entry are available at https://bitbucket.org/RobertSachtleben/an-approach-for-the-verification-and-synthesis-of-complete.

In addition to formalising different algorithms, this entry differs from my previous entry [7] (see [9] for the corresponding paper) in using a revised representation of finite state machines and by a focus on executable definitions.

Contents

1 Utility Definitions and Properties **9**
1.1 Converting Sets to Maps 10
1.2 Utility Lemmata for existing functions on lists 13
1.2.1 Utility Lemmata for *find* 13
1.2.2 Utility Lemmata for *filter* 15

1.2.3 Utility Lemmata for *concat* . . . 15
1.3 Enumerating Lists . . . 16
1.3.1 Enumerating List Subsets . . . 18
1.3.2 Enumerating Choices from Lists of Lists . . . 21
1.4 Finding the Index of the First Element of a List Satisfying a Property . . . 27
1.5 List Distinctness from Sorting . . . 28
1.6 Calculating Prefixes and Suffixes . . . 31
1.6.1 Pairs of Distinct Prefixes . . . 35
1.7 Calculating Distinct Non-Reflexive Pairs over List Elements . 39
1.8 Finite Linear Order From List Positions . . . 41
1.9 Find And Remove in a Single Pass . . . 45
1.10 Set-Operations on Lists . . . 56
1.10.1 Removing Subsets in a List of Sets . . . 57
1.11 Linear Order on Sum . . . 61
1.12 Removing Proper Prefixes . . . 62
1.13 Underspecified List Representations of Sets . . . 63
1.14 Assigning indices to elements of a finite set . . . 63
1.15 Other Lemmata . . . 65

2 Refinements for Utilities 91
2.1 New Code Equations for *set-as-map* . . . 91

3 Underlying FSM Representation 98
3.1 Types for Transitions and Paths . . . 98
3.2 Basic Algorithms on FSM . . . 98
3.2.1 Reading FSMs from Lists . . . 98
3.2.2 Changing the initial State . . . 99
3.2.3 Product Construction . . . 99
3.2.4 Filtering Transitions . . . 100
3.2.5 Filtering States . . . 100
3.2.6 Initial Singleton FSMI (For Trivial Preamble) . . . 101
3.2.7 Canonical Separator . . . 101
3.2.8 Generalised Canonical Separator . . . 102
3.2.9 Adding Transitions . . . 103
3.2.10 Creating an FSMI without transitions . . . 103
3.3 Transition Function h . . . 104
3.4 Extending FSMs by single elements . . . 104
3.5 Renaming elements . . . 105

4 Finite State Machines 105
4.1 Well-formed Finite State Machines . . . 105
4.1.1 Example FSMs . . . 107
4.2 Transition Function h and related functions . . . 108

4.3 Size . . . 111
4.4 Paths . . . 111
4.4.1 Paths of fixed length . . . 115
4.4.2 Paths up to fixed length . . . 115
4.4.3 Calculating Acyclic Paths . . . 118
4.5 Acyclic Paths . . . 127
4.6 Reachable States . . . 131
4.7 Language . . . 134
4.8 Basic FSM Properties . . . 144
4.8.1 Completely Specified . . . 144
4.8.2 Deterministic . . . 146
4.8.3 Observable . . . 147
4.8.4 Single Input . . . 147
4.8.5 Output Complete . . . 148
4.8.6 Acyclic . . . 148
4.8.7 Deadlock States . . . 159
4.8.8 Other . . . 162
4.9 IO Targets and Observability . . . 162
4.10 Conformity Relations . . . 187
4.11 A Pass Relation for Reduction and Test Represented as Sets of Input-Output Sequences . . . 188
4.12 Relaxation of IO based test suites to sets of input sequences . 189
4.13 Submachines . . . 190
4.14 Changing Initial States . . . 192
4.15 Language and Defined Inputs . . . 200
4.16 Further Reachability Formalisations . . . 204
4.16.1 Induction Schemes . . . 205
4.17 Further Path Enumeration Algorithms . . . 207
4.18 More Acyclicity Properties . . . 213
4.19 Elements as Lists . . . 217
4.20 Responses to Input Sequences . . . 218
4.21 Filtering Transitions . . . 220
4.22 Filtering States . . . 221
4.23 Adding Transitions . . . 224
4.24 Distinguishability . . . 228
4.25 Extending FSMs by single elements . . . 238
4.26 Renaming Elements . . . 240
4.27 Canonical Separators . . . 244

5 Product Machines **248**
5.1 Product Machines and Changing Initial States . . . 260
5.2 Calculating Acyclic Intersection Languages . . . 269

6 Minimisation by OFSM Tables **271**
6.1 OFSM Tables 272
6.1.1 Properties of Initial Partitions 282
6.1.2 Properties of OFSM tables for initial partitions based on equivalence relations 283
6.2 A minimisation function based on OFSM-tables 307

7 Computation of distinguishing traces based on OFSM tables **316**
7.1 Finding Diverging OFSM Tables 316
7.2 Assembling Distinguishing Traces 327
7.3 Minimal Distinguishing Traces 333

8 Properties of Sets of IO Sequences **334**
8.1 Completions 337

9 Observability **340**

10 Prefix Tree **371**
10.1 Alternative characterization for code generation 409

11 Refined Code Generation for Prefix Trees **413**

12 State Cover **415**
12.1 Basic Definitions 415
12.2 State Cover Computation 421
12.3 Computing Reachable States via State Cover Computation . 429

13 Alternative OFSM Table Computation **433**
13.1 Computing a List of all OFSM Tables 433
13.2 Finding Diverging Tables 438
13.3 Refining the Computation of Distinguishing Traces via OFSM Tables 443
13.4 Refining Minimisation 450

14 Transformation to Language-Equivalent Prime FSMs **452**
14.1 Helper Functions 452
14.2 The Transformation Algorithm 453
14.3 Renaming states to Words 456

15 Convergence of Traces **458**
15.1 Basic Definitions 458
15.2 Sufficient Conditions for Convergence 470
15.3 Proving Language Equivalence by Establishing a Convergence Preserving Initialised Transition Cover 521

16 Convergence Graphs **526**
16.1 Required Invariants on Convergence Graphs 527

17 An Always-Empty Convergence Graph **528**

18 H-Framework **529**
18.1 Abstract H-Condition 529
18.2 Definition of the Framework 545
18.3 Required Conditions on Procedural Parameters 546
18.4 Completeness and Finiteness of the Scheme 549

19 SPY-Framework **574**
19.1 Definition of the Framework 574
19.2 Required Conditions on Procedural Parameters 575
19.3 Completeness and Finiteness of the Framework 579

20 Pair-Framework **610**
20.1 Classical H-Condition 611
20.2 Helper Functions 617
20.3 Definition of the Pair-Framework 636

21 Intermediate Implementations **644**
21.1 Functions for the Pair Framework 644
21.2 Functions of the SPYH-Method 652
21.2.1 Heuristic Functions for Selecting Traces to Extend . . 652
21.2.2 Distributing Convergent Traces 670
21.2.3 Distinguishing a Trace from Other Traces 675
21.3 HandleIOPair 689
21.4 HandleStateCover 691
21.4.1 Dynamic 691
21.4.2 Static 698
21.5 Establishing Convergence of Traces 703
21.5.1 Dynamic 703
21.5.2 Static 754
21.6 Distinguishing Traces 795
21.6.1 Symmetry 795
21.6.2 Harmonised State Identifiers 805
21.6.3 Distinguishing Sets 807
21.7 Transition Sorting 818

22 Test Suites for Language Equivalence **818**
22.1 Transforming an IO-prefix-tree to a test suite 819
22.2 Code Refinement 834
22.3 Pass relations on list of lists representations of test suites . . 836
22.4 Alternative Representations 843

22.4.1 Pass and Fail Traces 843
22.4.2 Input Sequences 844

23 Simple Convergence Graphs **847**
23.1 Basic Definitions 847
23.2 Merging by Closure 848
23.3 Invariants 879

24 Intermediate Frameworks **894**
24.1 Partial Applications of the SPY-Framework 894
24.2 Partial Applications of the H-Framework 897
24.3 Partial Applications of the Pair-Framework 902
24.4 Code Generation 905

25 Implementations of the H-Method **906**
25.1 Using the H-Framework 906
25.2 Using the Pair-Framework 907
25.2.1 Selection of Distinguishing Traces 907
25.2.2 Implementation 927
25.2.3 Code Equations 929

26 Implementations of the HSI-Method **930**
26.1 Using the H-Framework 930
26.2 Using the SPY-Framework 931
26.3 Using the Pair-Framework 932
26.4 Code Generation 933

27 Implementations of the Partial-S-Method **937**
27.1 Using the H-Framework 937

28 Implementations of the SPY-Method **938**
28.1 Using the H-Framework 938
28.2 Using the SPY-Framework 940
28.3 Code Generation 941

29 Implementations of the SPYH-Method **944**
29.1 Using the H-Framework 944
29.2 Using the SPY-Framework 945
29.3 Code Generation 947

30 Refined Code Generation for Test Suites **947**

31 Implementations of the W-Method **952**
31.1 Using the H-Framework 952
31.2 Using the SPY-Framework 954
31.3 Using the Pair-Framework 955
31.4 Code Generation 956

32 Implementations of the Wp-Method **958**
32.1 Distinguishing Sets 958
32.2 Using the H-Framework 961
32.3 Using the SPY-Framework 962
32.4 Code Generation 963

33 Backwards Reachability Analysis **967**

34 State Separators **987**
34.1 Canonical Separators 987
34.1.1 Construction 987
34.1.2 State Separators as Submachines of Canonical Separators 994
34.1.3 Canonical Separator Properties 994
34.2 Calculating State Separators 1043
34.2.1 Sufficient Condition to Induce a State Separator . . . 1043
34.2.2 Calculating a State Separator by Backwards Reachability Analysis 1063
34.3 Generalizing State Separators 1068

35 Adaptive Test Cases **1085**
35.1 Applying Adaptive Test Cases 1085
35.2 State Separators as Adaptive Test Cases 1100
35.3 ATCs Represented as Sets of IO Sequences 1111

36 State Preambles **1116**
36.1 Basic Properties 1117
36.2 Calculating State Preambles via Backwards Reachability Analysis 1126
36.3 Minimal Sequences to Failures extending Preambles 1137

37 Helper Algorithms **1149**
37.1 Calculating r-distinguishable State Pairs with Separators . . . 1149
37.2 Calculating Pairwise r-distinguishable Sets of States 1155
37.3 Calculating d-reachable States with Preambles 1157
37.4 Calculating Repetition Sets 1158
37.4.1 Calculating Sub-Optimal Repetition Sets 1158

38 Maximal Path Tries **1163**
38.1 Utils for Updating Associative Lists 1163
38.2 Maximum Path Trie Implementation 1165
38.2.1 New Code Generation for *remove-proper-prefixes* . . . 1182

39 R-Distinguishability **1182**
39.1 R(k)-Distinguishability Properties 1183
39.1.1 Equivalence of R-Distinguishability Definitions 1190
39.2 Bounds 1196
39.3 Deciding R-Distinguishability 1206
39.4 State Separators and R-Distinguishability 1212

40 Traversal Set **1217**

41 Test Suites **1237**
41.1 Preliminary Definitions 1237
41.2 A Sufficiency Criterion for Reduction Testing 1238
41.3 A Pass Relation for Test Suites and Reduction Testing 1240
41.4 Soundness of Sufficient Test Suites 1240
41.5 Exhaustiveness of Sufficient Test Suites 1252
41.5.1 R Functions 1252
41.5.2 Proof of Exhaustiveness 1278
41.6 Completeness of Sufficient Test Suites 1309
41.7 Additional Test Suite Properties 1310

42 Representing Test Suites as Sets of Input-Output Sequences **1310**
42.1 Calculating the Sets of Sequences 1340
42.2 Using Maximal Sequences Only 1346

43 Calculating Sufficient Test Suites **1357**
43.1 Calculating Path Prefixes that are to be Extended With Adaptive Cest Cases 1357
43.1.1 Calculating Tests along m-Traversal-Paths 1357
43.1.2 Calculating Tests between Preambles 1359
43.1.3 Calculating Tests between m-Traversal-Paths Prefixes and Preambles 1361
43.2 Calculating a Test Suite 1361
43.3 Sufficiency of the Calculated Test Suite 1363
43.4 Two Complete Example Implementations 1388
43.4.1 Naive Repetition Set Strategy 1388
43.4.2 Greedy Repetition Set Strategy 1391

44 Refined Test Suite Calculation **1393**
44.1 New Instances . . . 1393
44.1.1 Order on FSMs . . . 1393
44.1.2 Derived Instances . . . 1397
44.1.3 Finiteness and Cardinality Instantiations for FSMs . . 1397
44.2 Updated Code Equations . . . 1400
44.2.1 New Code Equations for *remove-proper-prefixes* 1400
44.2.2 Special Handling for *set-as-map* on *image* 1401
44.2.3 New Code Equations for *h* . . . 1406
44.2.4 New Code Equations for *canonical-separator′* 1406
44.2.5 New Code Equations for *calculate-test-paths* 1407
44.2.6 New Code Equations for *prefix-pair-tests* 1407
44.2.7 New Code Equations for *preamble-prefix-tests* 1410

45 Data Refinement on FSM Representations **1413**
45.1 Mappings and Function *h* . . . 1413
45.2 Impl Datatype . . . 1433
45.3 Refined Datatype . . . 1434
45.4 Lifting . . . 1441

46 Code Export **1464**
46.1 Reduction Testing . . . 1465
46.1.1 Fault Detection Capabilities of the Test Harness . . . 1465
46.2 Equivalence Testing . . . 1468
46.2.1 Test Strategy Application and Transformation 1468
46.2.2 W-Method . . . 1472
46.2.3 Wp-Method . . . 1476
46.2.4 HSI-Method . . . 1478
46.2.5 H-Method . . . 1481
46.2.6 SPY-Method . . . 1485
46.2.7 SPYH-Method . . . 1487
46.2.8 Partial S-Method . . . 1489
46.3 New Instances . . . 1490
46.4 Exports . . . 1496

1 Utility Definitions and Properties

This file contains various definitions and lemmata not closely related to finite state machines or testing.

theory *Util*
imports *Main HOL−Library.FSet HOL−Library.Sublist HOL−Library.Mapping*
begin

1.1 Converting Sets to Maps

This subsection introduces a function *set-as-map* that transforms a set of $('a \times 'b)$ tuples to a map mapping each first value x of the contained tuples to all second values y such that (x,y) is contained in the set.

definition *set-as-map* :: $('a \times 'c)$ *set* $\Rightarrow$ $('a \Rightarrow 'c$ *set option*) **where**
set-as-map $s = (\lambda\ x\ .\ \mathit{if}\ (\exists\ z\ .\ (x,z) \in s)\ \mathit{then}\ \mathit{Some}\ \{z\ .\ (x,z) \in s\}\ \mathit{else}\ \mathit{None})$

lemma *set-as-map-code*[*code*] :
set-as-map (*set xs*) = (*foldl* (λ *m* (*x*,*z*) . *case m x of*
None $\Rightarrow$ *m* ($x \mapsto \{z\}$) |
Some zs $\Rightarrow$ *m* ($x \mapsto$ (*insert z zs*)))
Map.empty
xs)
proof −
let *?f* = λ *xs* . (*foldl* (λ *m* (*x*,*z*) . *case m x of*
None $\Rightarrow$ *m* ($x \mapsto \{z\}$) |
Some zs $\Rightarrow$ *m* ($x \mapsto$ (*insert z zs*)))
Map.empty
xs)
have (*?f xs*) = (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* $\{z\ .\ (x,z) \in$ *set xs*$\}$ *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc xz xs*)
then obtain *x z* **where** *xz* = (*x*,*z*)
by *force*

have $*$: (*?f* (*xs*@[(*x*,*z*)])) = (*case* (*?f xs*) *x of*
None $\Rightarrow$ (*?f xs*) ($x \mapsto \{z\}$) |
Some zs $\Rightarrow$ (*?f xs*) ($x \mapsto$ (*insert z zs*)))
by *auto*

then show *?case* **proof** (*cases* (*?f xs*) *x*)
case *None*
then have $**$: (*?f* (*xs*@[(*x*,*z*)])) = (*?f xs*) ($x \mapsto \{z\}$) **using** $*$ **by** *auto*

have *scheme*: $\bigwedge$ *m k v* . ($m(k \mapsto v)$) = ($\lambda k'$. *if* $k' = k$ *then Some v else m* k')
by *auto*

have *m1*: (*?f* (*xs*@[(*x*,*z*)])) = (λ x' . *if* $x' = x$ *then Some* $\{z\}$ *else* (*?f xs*) x')
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* $\{z\ .\ (x,z) \in$ *set xs*$\}$ *else None*) x = *None*

```
      using None snoc by auto
    then have ¬(∃ z . (x,z) ∈ set xs)
      by (metis (mono-tags, lifting) option.distinct(1))
    then have (∃ z . (x,z) ∈ set (xs@[(x,z)])) and {z' . (x,z') ∈ set (xs@[(x,z)])}
= {z}
      by auto
    then have m2: (λ x' . if (∃ z' . (x',z') ∈ set (xs@[(x,z)]))
                          then Some {z' . (x',z') ∈ set (xs@[(x,z)])}
                          else None)
              = (λ x' . if x' = x
                          then Some {z} else (λ x . if (∃ z . (x,z) ∈ set xs)
                                                then Some {z . (x,z) ∈ set xs}
                                                else None) x')
      by force

    show ?thesis using m1 m2 snoc
      using ‹xz = (x, z)› by presburger
  next
    case (Some zs)
    then have **: (?f (xs@[(x,z)])) = (?f xs) (x ↦ (insert z zs)) using * by auto
    have scheme: ⋀ m k v . (m(k ↦ v)) = (λk' . if k' = k then Some v else m k')
      by auto

    have m1: (?f (xs@[(x,z)])) = (λ x' . if x' = x then Some (insert z zs) else (?f
xs) x')
      unfolding **
      unfolding scheme by force

    have (λ x . if (∃ z . (x,z) ∈ set xs) then Some {z . (x,z) ∈ set xs} else None)
x = Some zs
      using Some snoc by auto
    then have (∃ z . (x,z) ∈ set xs)
      unfolding case-prod-conv using  option.distinct(2) by metis
    then have (∃ z . (x,z) ∈ set (xs@[(x,z)])) by simp

    have {z' . (x,z') ∈ set (xs@[(x,z)])} = insert z zs
    proof −
      have Some {z . (x,z) ∈ set xs} = Some zs
        using ‹(λ x . if (∃ z . (x,z) ∈ set xs) then Some {z . (x,z) ∈ set xs} else
None) x
              = Some zs›
        unfolding case-prod-conv using  option.distinct(2) by metis
      then have {z . (x,z) ∈ set xs} = zs by auto
      then show ?thesis by auto
    qed

    have ⋀ a  . (λ x' . if (∃ z' . (x',z') ∈ set (xs@[(x,z)]))
                          then Some {z' . (x',z') ∈ set (xs@[(x,z)])} else None) a
              = (λ x' . if x' = x
```

```
                        then Some (insert z zs)
                        else (λ x . if (∃ z . (x,z) ∈ set xs)
                                then Some {z . (x,z) ∈ set xs} else None) x′) a
    proof −
      fix a show (λ x′ . if (∃ z′ . (x′,z′) ∈ set (xs@[(x,z)]))
                        then Some {z′ . (x′,z′) ∈ set (xs@[(x,z)])} else None) a
              = (λ x′ . if x′ = x
                        then Some (insert z zs)
                        else (λ x . if (∃ z . (x,z) ∈ set xs)
                                then Some {z . (x,z) ∈ set xs} else None) x′) a
        using ‹{z′ . (x,z′) ∈ set (xs@[(x,z)])} = insert z zs› ‹(∃ z . (x,z) ∈ set
(xs@[(x,z)]))›
      by (cases a = x; auto)
    qed

    then have m2: (λ x′ . if (∃ z′ . (x′,z′) ∈ set (xs@[(x,z)]))
                        then Some {z′ . (x′,z′) ∈ set (xs@[(x,z)])} else None)
              = (λ x′ . if x′ = x
                        then Some (insert z zs)
                        else (λ x . if (∃ z . (x,z) ∈ set xs)
                                then Some {z . (x,z) ∈ set xs} else None) x′)
      by auto


    show ?thesis using m1 m2 snoc
      using ‹xz = (x, z)› by presburger
   qed
  qed

  then show ?thesis
    unfolding set-as-map-def by simp
qed


abbreviation member-option x ms ≡ (case ms of None ⇒ False | Some xs ⇒ x
∈ xs)
notation member-option (‹(-∈o-)› [1000] 1000)

abbreviation(input) lookup-with-default f d ≡ (λ x . case f x of None ⇒ d | Some
xs ⇒ xs)
abbreviation(input) m2f f ≡ lookup-with-default f {}

abbreviation(input) lookup-with-default-by f g d ≡ (λ x . case f x of None ⇒ g d
| Some xs ⇒ g xs)
abbreviation(input) m2f-by g f ≡ lookup-with-default-by f g {}

lemma m2f-by-from-m2f :
  (m2f-by g f xs) = g (m2f f xs)
  by (simp add: option.case-eq-if)
```

lemma *set-as-map-containment* :
assumes $(x,y) \in zs$
shows $y \in (m2f\ (set\text{-}as\text{-}map\ zs))\ x$
using *assms* **unfolding** *set-as-map-def*
by *auto*

lemma *set-as-map-elem* :
assumes $y \in m2f\ (set\text{-}as\text{-}map\ xs)\ x$
shows $(x,y) \in xs$
using *assms* **unfolding** *set-as-map-def*
proof −
assume *a1*: $y \in (case\ if\ \exists z.\ (x,\ z) \in xs\ then\ Some\ \{z.\ (x,\ z) \in xs\}\ else\ None\ of\ None \Rightarrow \{\} \mid Some\ xs \Rightarrow xs)$
then have $\exists a.\ (x,\ a) \in xs$
using *all-not-in-conv* **by** *fastforce*
then show *?thesis*
using *a1* **by** *simp*
qed

1.2 Utility Lemmata for existing functions on lists

1.2.1 Utility Lemmata for *find*

lemma *find-result-props* :
assumes $find\ P\ xs = Some\ x$
shows $x \in set\ xs$ **and** $P\ x$
proof −
show $x \in set\ xs$ **using** *assms* **by** (*metis find-Some-iff nth-mem*)
show $P\ x$ **using** *assms* **by** (*metis find-Some-iff*)
qed

lemma *find-set* :
assumes $find\ P\ xs = Some\ x$
shows $x \in set\ xs$
using *assms* **proof**(*induction xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
by (*metis find.simps*(*2*) *list.set-intros*(*1*) *list.set-intros*(*2*) *option.inject*)
qed

lemma *find-condition* :
assumes $find\ P\ xs = Some\ x$
shows $P\ x$
using *assms* **proof**(*induction xs*)
case *Nil*

```
  then show ?case by auto
next
  case (Cons a xs)
  then show ?case
    by (metis find.simps(2) option.inject)
qed

lemma find-from :
  assumes ∃ x ∈ set xs . P x
  shows find P xs ≠ None
  by (metis assms find-None-iff)


lemma find-sort-containment :
  assumes find P (sort xs) = Some x
shows x ∈ set xs
  using assms find-set by force


lemma find-sort-index :
  assumes find P xs = Some x
  shows ∃ i < length xs . xs ! i = x ∧ (∀ j < i . ¬ P (xs ! j))
using assms proof (induction xs arbitrary: x)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  show ?case proof (cases P a)
    case True
    then show ?thesis
      using Cons.prems unfolding find.simps by auto
  next
    case False
    then have find P (a#xs) = find P xs
      unfolding find.simps by auto
    then have find P xs = Some x
      using Cons.prems by auto
    then show ?thesis
      using Cons.IH False
      by (metis Cons.prems find-Some-iff)
  qed
qed


lemma find-sort-least :
  assumes find P (sort xs) = Some x
  shows ∀ x' ∈ set xs . x ≤ x' ∨ ¬ P x'
  and   x = (LEAST x' ∈ set xs . P x')
proof −
```

obtain i **where** $i < length\ (sort\ xs)$
and $(sort\ xs)\ !\ i = x$
and $(\forall\ j < i\ .\ \neg\ P\ ((sort\ xs)\ !\ j))$
using *find-sort-index*[*OF assms*] **by** *blast*

have $\bigwedge j\ .\ j > i \Longrightarrow j < length\ xs \Longrightarrow (sort\ xs)\ !\ i \leq (sort\ xs)\ !\ j$
by (*simp add*: *sorted-nth-mono*)
then have $\bigwedge j\ .\ j < length\ xs \Longrightarrow (sort\ xs)\ !\ i \leq (sort\ xs)\ !\ j \vee \neg\ P\ ((sort\ xs)\ !\ j)$
using ‹$(\forall\ j < i\ .\ \neg\ P\ ((sort\ xs)\ !\ j))$›
by (*metis not-less-iff-gr-or-eq order-refl*)
then show $\forall\ x' \in set\ xs\ .\ x \leq x' \vee \neg\ P\ x'$
by (*metis* ‹$sort\ xs\ !\ i = x$› *in-set-conv-nth length-sort set-sort*)
then show $x = (LEAST\ x' \in set\ xs\ .\ P\ x')$
using *find-set*[*OF assms*] *find-condition*[*OF assms*]
by (*metis* (*mono-tags*, *lifting*) *Least-equality set-sort*)
qed

1.2.2 Utility Lemmata for *filter*

lemma *filter-take-length* :
$length\ (filter\ P\ (take\ i\ xs)) \leq length\ (filter\ P\ xs)$
by (*metis append-take-drop-id filter-append le0 le-add-same-cancel1 length-append*)

lemma *filter-double* :
assumes $x \in set\ (filter\ P1\ xs)$
and $P2\ x$
shows $x \in set\ (filter\ P2\ (filter\ P1\ xs))$
using *assms* **by** *simp*

lemma *filter-list-set* :
assumes $x \in set\ xs$
and $P\ x$
shows $x \in set\ (filter\ P\ xs)$
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *filter-list-set-not-contained* :
assumes $x \in set\ xs$
and $\neg\ P\ x$
shows $x \notin set\ (filter\ P\ xs)$
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *filter-map-elem* : $t \in set\ (map\ g\ (filter\ f\ xs)) \Longrightarrow \exists\ x \in set\ xs\ .\ f\ x \wedge t = g\ x$
by *auto*

1.2.3 Utility Lemmata for *concat*

lemma *concat-map-elem* :

```
  assumes y ∈ set (concat (map f xs))
  obtains x where x ∈ set xs
              and y ∈ set (f x)
using assms proof (induction xs)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  then show ?case
  proof (cases y ∈ set (f a))
    case True
    then show ?thesis
      using Cons.prems(1) by auto
  next
    case False
    then have y ∈ set (concat (map f xs))
      using Cons by auto
    have ∃ x . x ∈ set xs ∧ y ∈ set (f x)
    proof (rule ccontr)
      assume ¬(∃x. x ∈ set xs ∧ y ∈ set (f x))
      then have ¬(y ∈ set (concat (map f xs)))
        by auto
      then show False
        using ‹y ∈ set (concat (map f xs))› by auto
    qed
    then show ?thesis
      using Cons.prems(1) by auto
  qed
qed

lemma set-concat-map-sublist :
  assumes x ∈ set (concat (map f xs))
  and     set xs ⊆ set xs'
shows x ∈ set (concat (map f xs'))
using assms by (induction xs) (auto)

lemma set-concat-map-elem :
  assumes x ∈ set (concat (map f xs))
  shows ∃ x' ∈ set xs . x ∈ set (f x')
using assms by auto

lemma concat-replicate-length : length (concat (replicate n xs)) = n * (length xs)
  by (induction n; simp)
```

1.3 Enumerating Lists

```
fun lists-of-length :: 'a list ⇒ nat ⇒ 'a list list where
  lists-of-length T 0 = [[]] |
 lists-of-length T (Suc n) = concat (map (λ xs . map (λ x . x#xs) T ) (lists-of-length
```

```
T n))

lemma lists-of-length-containment :
  assumes set xs ⊆ set T
  and     length xs = n
shows xs ∈ set (lists-of-length T n)
using assms proof (induction xs arbitrary: n)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  then obtain k where n = Suc k
    by auto
  then have xs ∈ set (lists-of-length T k)
    using Cons by auto
  moreover have a ∈ set T
    using Cons by auto
  ultimately show ?case
    using ‹n = Suc k› by auto
qed


lemma lists-of-length-length :
  assumes xs ∈ set (lists-of-length T n)
  shows length xs = n
proof −
  have ∀ xs ∈ set (lists-of-length T n) . length xs = n
    by (induction n; simp)
  then show ?thesis using assms by blast
qed

lemma lists-of-length-elems :
  assumes xs ∈ set (lists-of-length T n)
  shows set xs ⊆ set T
proof −
  have ∀ xs ∈ set (lists-of-length T n) . set xs ⊆ set T
    by (induction n; simp)
  then show ?thesis using assms by blast
qed

lemma lists-of-length-list-set :
  set (lists-of-length xs k) = {xs' . length xs' = k ∧ set xs' ⊆ set xs}
  using lists-of-length-containment[of - xs k]
        lists-of-length-length[of - xs k]
        lists-of-length-elems[of - xs k]
  by blast
```

1.3.1 Enumerating List Subsets

fun *generate-selector-lists* :: *nat* ⇒ *bool list list* **where**
  generate-selector-lists k = *lists-of-length* [*False*,*True*] *k*

lemma *generate-selector-lists-set* :
  set (*generate-selector-lists k*) = {(*bs* :: *bool list*) . *length bs* = *k*}
  using *lists-of-length-list-set* **by** *auto*

lemma *selector-list-index-set*:
  assumes *length ms* = *length bs*
  shows *set* (*map fst* (*filter snd* (*zip ms bs*))) = { *ms* ! *i* | *i* . *i* < *length bs* ∧ *bs* ! *i*}
using *assms* **proof** (*induction bs arbitrary*: *ms rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc b bs*)
  let *?ms* = *butlast ms*
  let *?m* = *last ms*

  have *length ?ms* = *length bs* **using** *snoc.prems* **by** *auto*

  have *map fst* (*filter snd* (*zip ms* (*bs* @ [*b*])))
      = (*map fst* (*filter snd* (*zip ?ms bs*))) @ (*map fst* (*filter snd* (*zip* [*?m*] [*b*])))
    by (*metis* ‹*length* (*butlast ms*) = *length bs*› *append-eq-conv-conj filter-append length-0-conv*
      map-append snoc.prems snoc-eq-iff-butlast zip-append2)
  then have ∗: *set* (*map fst* (*filter snd* (*zip ms* (*bs* @ [*b*]))))
      = *set* (*map fst* (*filter snd* (*zip ?ms bs*))) ∪ *set* (*map fst* (*filter snd* (*zip* [*?m*] [*b*])))
    by *simp*

  have {*ms* ! *i* |*i*. *i* < *length* (*bs* @ [*b*]) ∧ (*bs* @ [*b*]) ! *i*}
      = {*ms* ! *i* |*i*. *i* ≤ (*length bs*) ∧ (*bs* @ [*b*]) ! *i*}
    by *auto*
  moreover have {*ms* ! *i* |*i*. *i* ≤ (*length bs*) ∧ (*bs* @ [*b*]) ! *i*}
      = {*ms* ! *i* |*i*. *i* < *length bs* ∧ (*bs* @ [*b*]) ! *i*}
      ∪ {*ms* ! *i* |*i*. *i* = *length bs* ∧ (*bs* @ [*b*]) ! *i*}
    by *fastforce*
  moreover have {*ms* ! *i* |*i*. *i* < *length bs* ∧ (*bs* @ [*b*]) ! *i*} = {*?ms* ! *i* |*i*. *i* < *length bs* ∧ *bs* ! *i*}
    using ‹*length ?ms* = *length bs*› **by** (*metis butlast-snoc nth-butlast*)
  ultimately have ∗∗: {*ms* ! *i* |*i*. *i* < *length* (*bs* @ [*b*]) ∧ (*bs* @ [*b*]) ! *i*}
      = {*?ms* ! *i* |*i*. *i* < *length bs* ∧ *bs* ! *i*}
      ∪ {*ms* ! *i* |*i*. *i* = *length bs* ∧ (*bs* @ [*b*]) ! *i*}
    by *simp*

```
  have set (map fst (filter snd (zip [?m] [b]))) = {ms ! i |i. i = length bs ∧ (bs @
[b]) ! i}
  proof (cases b)
    case True
    then have set (map fst (filter snd (zip [?m] [b]))) = {?m} by fastforce
    moreover have {ms ! i |i. i = length bs ∧ (bs @ [b]) ! i} = {?m}
    proof −
      have (bs @ [b]) ! length bs
        by (simp add: True)
      moreover have ms ! length bs = ?m
      by (metis last-conv-nth length-0-conv length-butlast snoc.prems snoc-eq-iff-butlast)

      ultimately show ?thesis by fastforce
    qed
    ultimately show ?thesis by auto
  next
    case False
    then show ?thesis by auto
  qed

  then have set (map fst (filter snd (zip (butlast ms) bs)))
              ∪ set (map fst (filter snd (zip [?m] [b])))
            = {butlast ms ! i |i. i < length bs ∧ bs ! i}
              ∪ {ms ! i |i. i = length bs ∧ (bs @ [b]) ! i}
    using snoc.IH[OF ‹length ?ms = length bs›] by blast

  then show ?case using ∗ ∗∗
    by simp
qed

lemma selector-list-ex :
  assumes set xs ⊆ set ms
  shows ∃ bs . length bs = length ms ∧ set xs = set (map fst (filter snd (zip ms
bs)))
using assms proof (induction xs rule: rev-induct)
  case Nil
  let ?bs = replicate (length ms) False
  have set [] = set (map fst (filter snd (zip ms ?bs)))
    by (metis filter-False in-set-zip length-replicate list.simps(8) nth-replicate)
  moreover have length ?bs = length ms by auto
  ultimately show ?case by blast
next
  case (snoc a xs)
  then have set xs ⊆ set ms and a ∈ set ms by auto
  then obtain bs where length bs = length ms and set xs = set (map fst (filter
snd (zip ms bs)))
    using snoc.IH by auto
```

from ‹$a \in$ *set ms*› **obtain** i **where** $i <$ *length ms* **and** $ms\ !\ i = a$
  by (*meson in-set-conv-nth*)

let *?bs* = *list-update bs i True*
have *length ms* = *length ?bs* **using** ‹*length bs* = *length ms*› **by** *auto*
have *length ?bs* = *length bs* **by** *auto*

have *set* (*map fst* (*filter snd* (*zip ms ?bs*))) = $\{ms\ !\ i\ |i.\ i <$ *length ?bs* $\wedge$ *?bs* ! $i\}$
  using *selector-list-index-set*[*OF* ‹*length ms* = *length ?bs*›] **by** *assumption*

have $\bigwedge j\ .\ j <$ *length ?bs* $\Longrightarrow j \neq i \Longrightarrow$ *?bs* ! j = *bs* ! j
  by *auto*
then have $\{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j \neq i \wedge bs\ !\ j\}$
    $= \{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j \neq i \wedge$ *?bs* ! $j\}$
  using ‹*length ?bs* = *length bs*› **by** *fastforce*

have $\{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j = i \wedge$ *?bs* ! $j\} = \{a\}$
  using ‹*length bs* = *length ms*› ‹$i <$ *length ms*› ‹$ms\ !\ i = a$› **by** *auto*
then have $\{ms\ !\ i\ |i.\ i <$ *length ?bs* $\wedge$ *?bs* ! $i\}$
    = *insert a* $\{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j \neq i \wedge$ *?bs* ! $j\}$
  by *fastforce*

have $\{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j = i \wedge bs\ !\ j\} \subseteq \{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j = i \wedge$ *?bs* ! $j\}$
  by (*simp add*: *Collect-mono*)
then have $\{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j = i \wedge bs\ !\ j\} \subseteq \{a\}$
  using ‹$\{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j = i \wedge$ *?bs* ! $j\} = \{a\}$›
  by *auto*
moreover have $\{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge bs\ !\ j\}$
    $= \{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j = i \wedge bs\ !\ j\}$
      $\cup \{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j \neq i \wedge bs\ !\ j\}$
  by *fastforce*

ultimately have $\{ms\ !\ i\ |i.\ i <$ *length ?bs* $\wedge$ *?bs* ! $i\}$
      = *insert a* $\{ms\ !\ i\ |i.\ i <$ *length bs* $\wedge bs\ !\ i\}$
  using ‹$\{ms\ !\ j\ |j.\ j <$ *length bs* $\wedge\ j \neq i \wedge bs\ !\ j\}$
    $= \{ms\ !\ j\ |j.\ j <$ *length ?bs* $\wedge\ j \neq i \wedge$ *?bs* ! $j\}$›
  using ‹$\{ms\ !\ ia\ |ia.\ ia <$ *length* (*bs*[*i* := *True*])
      $\wedge$ *bs*[*i* := *True*] ! *ia*$\}$
        = *insert a* $\{ms\ !\ j\ |j.\ j <$ *length* (*bs*[*i* := *True*])
          $\wedge\ j \neq i \wedge$ *bs*[*i* := *True*] ! $j\}$›
  by *auto*

moreover have *set* (*map fst* (*filter snd* (*zip ms bs*))) = $\{ms\ !\ i\ |i.\ i <$ *length bs* $\wedge$ *bs* ! $i\}$

using *selector-list-index-set*[*of ms bs*] ‹*length bs* = *length ms*› **by** *auto*

ultimately have *set* (*a*#*xs*) = *set* (*map fst* (*filter snd* (*zip ms ?bs*)))
using ‹*set* (*map fst* (*filter snd* (*zip ms ?bs*))) = {*ms* ! *i* |*i*. *i* < *length ?bs* ∧ *?bs* ! *i*}›
‹*set xs* = *set* (*map fst* (*filter snd* (*zip ms bs*)))›
by *auto*
then show *?case*
using ‹*length ms* = *length ?bs*›
by (*metis Un-commute insert-def list.set*(*1*) *list.simps*(*15*) *set-append singleton-conv*)
qed

1.3.2 Enumerating Choices from Lists of Lists

fun *generate-choices* :: (′*a* × (′*b list*)) *list* ⇒ (′*a* × ′*b option*) *list list* **where**
generate-choices [] = [[]] |
generate-choices (*xys*#*xyss*) =
concat (*map* (λ *xy*′ . *map* (λ *xys*′ . *xy*′ # *xys*′) (*generate-choices xyss*))
((*fst xys*, *None*) # (*map* (λ *y* . (*fst xys*, *Some y*)) (*snd xys*))))

lemma *concat-map-hd-tl-elem*:
assumes *hd cs* ∈ *set P1*
and *tl cs* ∈ *set P2*
and *length cs* > *0*
shows *cs* ∈ *set* (*concat* (*map* (λ *xy*′ . *map* (λ *xys*′ . *xy*′ # *xys*′) *P2*) *P1*))
proof −
have *hd cs* # *tl cs* = *cs* **using** *assms*(*3*) **by** *auto*
moreover have *hd cs* # *tl cs* ∈ *set* (*concat* (*map* (λ *xy*′ . *map* (λ *xys*′ . *xy*′ # *xys*′) *P2*) *P1*))
using *assms*(*1*,*2*) **by** *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *generate-choices-hd-tl* :
cs ∈ *set* (*generate-choices* (*xys*#*xyss*))
= (*length cs* = *length* (*xys*#*xyss*)
∧ *fst* (*hd cs*) = *fst xys*
∧ ((*snd* (*hd cs*) = *None* ∨ (*snd* (*hd cs*) ≠ *None* ∧ *the* (*snd* (*hd cs*)) ∈ *set* (*snd xys*))))
∧ (*tl cs* ∈ *set* (*generate-choices xyss*)))
proof (*induction xyss arbitrary*: *cs xys*)
case *Nil*
have (*cs* ∈ *set* (*generate-choices* [*xys*]))
= (*cs* ∈ *set* ([(*fst xys*, *None*)] # *map* (λ*y*. [(*fst xys*, *Some y*)]) (*snd xys*)))
unfolding *generate-choices.simps* **by** *auto*

moreover have ($cs \in set$ ([($fst\ xys$, $None$)] # map (λy. [($fst\ xys$, $Some\ y$)]) (snd xys)))
$\Longrightarrow$ ($length\ cs = length$ [xys] $\wedge$
fst ($hd\ cs$) = $fst\ xys$ $\wedge$
(snd ($hd\ cs$) = $None$ $\vee$ snd ($hd\ cs$) $\neq$ $None$ $\wedge$ the (snd ($hd\ cs$)) $\in$ set ($snd\ xys$)) $\wedge$
$tl\ cs \in set$ ($generate\text{-}choices$ []))
by *auto*
moreover have ($length\ cs = length$ [xys] $\wedge$
fst ($hd\ cs$) = $fst\ xys$ $\wedge$
(snd ($hd\ cs$) = $None$ $\vee$ snd ($hd\ cs$) $\neq$ $None$ $\wedge$ the (snd ($hd\ cs$)) $\in$ set ($snd\ xys$)) $\wedge$
$tl\ cs \in set$ ($generate\text{-}choices$ []))
$\Longrightarrow$ ($cs \in set$ ([($fst\ xys$, $None$)] # map (λy. [($fst\ xys$, $Some\ y$)]) (snd xys)))
unfolding *generate-choices.simps(1)*
proof −
assume *a1*: $length\ cs = length$ [xys]
$\wedge$ fst ($hd\ cs$) = $fst\ xys$
$\wedge$ (snd ($hd\ cs$) = $None$ $\vee$ snd ($hd\ cs$) $\neq$ $None$ $\wedge$ the (snd ($hd\ cs$)) $\in$ set ($snd\ xys$))
$\wedge$ $tl\ cs \in set$ [[]]
have *f2*: $\forall ps.\ ps = [] \vee ps = (hd\ ps{::}'a \times 'b\ option)$ # $tl\ ps$
by (*meson list.exhaust-sel*)
have *f3*: $cs \neq []$
using *a1* **by** *fastforce*
have snd ($hd\ cs$) = $None \longrightarrow$ ($fst\ xys$, $None$) = $hd\ cs$
using *a1* **by** (*metis prod.exhaust-sel*)
moreover
{ assume $hd\ cs$ # $tl\ cs \neq$ [($fst\ xys$, $Some$ (the (snd ($hd\ cs$))))]
then have snd ($hd\ cs$) = $None$
using *a1* **by** (*metis* (*no-types*) *length-0-conv length-tl list.sel(3)*
option.collapse prod.exhaust-sel) **}**
ultimately have $cs \in insert$ [($fst\ xys$, $None$)] ((λb. [($fst\ xys$, $Some\ b$)]) ‘ set ($snd\ xys$))
using *f3 f2 a1* **by** *fastforce*
then show *?thesis*
by *simp*
qed
ultimately show *?case* **by** *blast*
next
case (*Cons a xyss*)

have $length\ cs = length$ (xys#a#$xyss$)
$\Longrightarrow$ fst ($hd\ cs$) = $fst\ xys$
$\Longrightarrow$ (snd ($hd\ cs$) = $None$ $\vee$ (snd ($hd\ cs$) $\neq$ $None$ $\wedge$ the (snd ($hd\ cs$)) $\in$ set ($snd\ xys$)))
$\Longrightarrow$ ($tl\ cs \in set$ ($generate\text{-}choices$ (a#$xyss$)))
$\Longrightarrow$ $cs \in set$ ($generate\text{-}choices$ (xys#a#$xyss$))

```
  proof −
    assume length cs = length (xys#a#xyss)
      and fst (hd cs) = fst xys
      and (snd (hd cs) = None ∨ (snd (hd cs) ≠ None ∧ the (snd (hd cs)) ∈ set
(snd xys)))
      and (tl cs ∈ set (generate-choices (a#xyss)))
    then have length cs > 0 by auto

    have (hd cs) ∈ set ((fst xys, None) # (map (λ y . (fst xys, Some y)) (snd xys)))
      using ‹fst (hd cs) = fst xys›
            ‹(snd (hd cs) = None ∨ (snd (hd cs) ≠ None ∧ the (snd (hd cs)) ∈ set
(snd xys)))›
      by (metis (no-types, lifting) image-eqI list.set-intros(1) list.set-intros(2)
            option.collapse prod.collapse set-map)

    show cs ∈ set (generate-choices ((xys#(a#xyss))))
      using generate-choices.simps(2)[of xys a#xyss]
            concat-map-hd-tl-elem[OF ‹(hd cs) ∈ set ((fst xys, None) # (map (λ y .
(fst xys, Some y)) (snd xys)))›
                                    ‹(tl cs ∈ set (generate-choices (a#xyss)))›
                                    ‹length cs > 0›]
      by auto
  qed

  moreover have cs ∈ set (generate-choices (xys#a#xyss))
              ⟹ length cs = length (xys#a#xyss)
                ∧ fst (hd cs) = fst xys
                ∧ ((snd (hd cs) = None ∨ (snd (hd cs) ≠ None
                ∧ the (snd (hd cs)) ∈ set (snd xys))))
                ∧ (tl cs ∈ set (generate-choices (a#xyss)))
  proof −
    assume cs ∈ set (generate-choices (xys#a#xyss))
    then have p3: tl cs ∈ set (generate-choices (a#xyss))
      using generate-choices.simps(2)[of xys a#xyss] by fastforce
    then have length (tl cs) = length (a # xyss) using Cons.IH[of tl cs a] by simp
    then have p1: length cs = length (xys#a#xyss) by auto

    have p2 : fst (hd cs) = fst xys ∧ ((snd (hd cs) = None ∨ (snd (hd cs) ≠ None
                                ∧ the (snd (hd cs)) ∈ set (snd xys))))
     using ‹cs ∈ set (generate-choices (xys#a#xyss))› generate-choices.simps(2)[of
xys a#xyss]
      by fastforce

    show ?thesis using p1 p2 p3 by simp
  qed

  ultimately show ?case by blast
qed
```

lemma *list-append-idx-prop* :
  $(\forall\ i\ .\ (i < length\ xs \longrightarrow P\ (xs\ !\ i)))$
    $= (\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)\ !\ j)))$
proof –
  have $\bigwedge j\ .\ \forall i{<}length\ xs.\ P\ (xs\ !\ i) \Longrightarrow j < length\ (ys\ @\ xs)$
        $\Longrightarrow length\ ys \leq j \longrightarrow P\ ((ys\ @\ xs)\ !\ j)$
    by (*simp add*: *nth-append*)
  moreover have $\bigwedge i\ .\ (\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)$
$!\ j)))$
        $\Longrightarrow i < length\ xs \Longrightarrow P\ (xs\ !\ i)$
  proof –
    fix i **assume** $(\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)\ !$
$j)))$
      and $i < length\ xs$
    then have $P\ ((ys@xs)\ !\ (length\ ys + i))$
      by (*metis add-strict-left-mono le-add1 length-append*)
    moreover have $P\ (xs\ !\ i) = P\ ((ys@xs)\ !\ (length\ ys + i))$
      by *simp*
    ultimately show $P\ (xs\ !\ i)$ **by** *blast*
  qed
  ultimately show *?thesis* **by** *blast*
qed

lemma *list-append-idx-prop2* :
  assumes $length\ xs' = length\ xs$
    and $length\ ys' = length\ ys$
  shows $(\forall\ i\ .\ (i < length\ xs \longrightarrow P\ (xs\ !\ i)\ (xs'\ !\ i)))$
    $= (\forall\ j\ .\ ((j < length\ (ys@xs) \wedge j \geq length\ ys) \longrightarrow P\ ((ys@xs)\ !\ j)\ ((ys'@xs')$
$!\ j)))$
proof –
  have $\forall i{<}length\ xs.\ P\ (xs\ !\ i)\ (xs'\ !\ i) \Longrightarrow$
    $\forall j.\ j < length\ (ys\ @\ xs) \wedge length\ ys \leq j \longrightarrow P\ ((ys\ @\ xs)\ !\ j)\ ((ys'\ @\ xs')\ !\ j)$
    using *assms*
  proof –
    assume *a1*: $\forall i{<}length\ xs.\ P\ (xs\ !\ i)\ (xs'\ !\ i)$
    { fix $nn :: nat$
      have *ff1*: $\forall n\ na.\ (na{::}nat) + n - n = na$
        by *simp*
      have *ff2*: $\forall n\ na.\ (na{::}nat) \leq n + na$
        by *auto*
      then have *ff3*: $\forall as\ n.\ (ys'\ @\ as)\ !\ n = as\ !\ (n - length\ ys) \vee \neg\ length\ ys \leq$
n
      using *ff1* **by** (*metis* (*no-types*) *add.commute assms*(*2*) *eq-diff-iff nth-append-length-plus*)
      have *ff4*: $\forall n\ bs\ bsa.\ ((bsa\ @\ bs)\ !\ n{::}'b) = bs\ !\ (n - length\ bsa) \vee \neg\ length$
$bsa \leq n$
      using *ff2 ff1* **by** (*metis* (*no-types*) *add.commute eq-diff-iff nth-append-length-plus*)
      have $\forall n\ na\ nb.\ ((n{::}nat) + nb \leq na \vee \neg\ n \leq na - nb) \vee \neg\ nb \leq na$
        using *ff2 ff1* **by** (*metis le-diff-iff*)
      then have $(\neg\ nn < length\ (ys\ @\ xs) \vee \neg\ length\ ys \leq nn)$

$\vee$ P $((ys$ @ $xs)$! $nn)$ $((ys'$ @ $xs')$! $nn)$
using *ff4 ff3 a1* **by** (*metis add.commute length-append not-le*) **}**
then show *?thesis*
by *blast*
qed

moreover have $(\forall j.\ j < length\ (ys\ @\ xs) \wedge length\ ys \leq j \longrightarrow P\ ((ys\ @\ xs)\ !\ j)\ ((ys'\ @\ xs')\ !\ j))$
$\Longrightarrow \forall i{<}length\ xs.\ P\ (xs\ !\ i)\ (xs'\ !\ i)$
using *assms*
by (*metis le-add1 length-append nat-add-left-cancel-less nth-append-length-plus*)

ultimately show *?thesis* **by** *blast*
qed

lemma *generate-choices-idx* :
$cs \in set\ (generate\text{-}choices\ xyss)$
$= (length\ cs = length\ xyss$
$\wedge (\forall\ i < length\ cs\ .\ (fst\ (cs\ !\ i)) = (fst\ (xyss\ !\ i))$
$\wedge ((snd\ (cs\ !\ i)) = None$
$\vee ((snd\ (cs\ !\ i)) \neq None \wedge the\ (snd\ (cs\ !\ i)) \in set\ (snd\ (xyss\ !\ i))))))$
proof (*induction xyss arbitrary*: *cs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons xys xyss*)

have $cs \in set\ (generate\text{-}choices\ (xys\#xyss))$
$= (length\ cs = length\ (xys\#xyss)$
$\wedge fst\ (hd\ cs) = fst\ xys$
$\wedge ((snd\ (hd\ cs) = None \vee (snd\ (hd\ cs) \neq None \wedge the\ (snd\ (hd\ cs)) \in set\ (snd\ xys))))$
$\wedge (tl\ cs \in set\ (generate\text{-}choices\ xyss)))$
using *generate-choices-hd-tl* **by** *metis*

then have $cs \in set\ (generate\text{-}choices\ (xys\#xyss))$
$= (length\ cs = length\ (xys\#xyss)$
$\wedge fst\ (hd\ cs) = fst\ xys$
$\wedge ((snd\ (hd\ cs) = None \vee (snd\ (hd\ cs) \neq None \wedge the\ (snd\ (hd\ cs)) \in set\ (snd\ xys))))$
$\wedge (length\ (tl\ cs) = length\ xyss \wedge$
$(\forall i{<}length\ (tl\ cs).$
$fst\ (tl\ cs\ !\ i) = fst\ (xyss\ !\ i) \wedge$
$(snd\ (tl\ cs\ !\ i) = None$
$\vee\ snd\ (tl\ cs\ !\ i) \neq None \wedge the\ (snd\ (tl\ cs\ !\ i)) \in set\ (snd\ (xyss\ !\ i))))))$
using *Cons.IH*[*of tl cs*] **by** *blast*
then have $*$: $cs \in set\ (generate\text{-}choices\ (xys\#xyss))$
$= (length\ cs = length\ (xys\#xyss)$

$\wedge$ *fst* (*hd cs*) = *fst xys*
$\wedge$ ((*snd* (*hd cs*) = *None* $\vee$ (*snd* (*hd cs*) $\neq$ *None* $\wedge$ *the* (*snd* (*hd cs*)) $\in$ *set* (*snd xys*))))
$\wedge$ ($\forall$ *i*<*length* (*tl cs*).
fst (*tl cs* ! *i*) = *fst* (*xyss* ! *i*) $\wedge$
(*snd* (*tl cs* ! *i*) = *None*
$\vee$ *snd* (*tl cs* ! *i*) $\neq$ *None* $\wedge$ *the* (*snd* (*tl cs* ! *i*)) $\in$ *set* (*snd* (*xyss* ! *i*)))))
by *auto*

have *cs* $\in$ *set* (*generate-choices* (*xys*#*xyss*)) $\Longrightarrow$ (*length cs* = *length* (*xys* # *xyss*) $\wedge$
($\forall$ *i*<*length cs*.
fst (*cs* ! *i*) = *fst* ((*xys* # *xyss*) ! *i*) $\wedge$
(*snd* (*cs* ! *i*) = *None* $\vee$
snd (*cs* ! *i*) $\neq$ *None* $\wedge$ *the* (*snd* (*cs* ! *i*)) $\in$ *set* (*snd* ((*xys* # *xyss*) ! *i*)))))
proof −
assume *cs* $\in$ *set* (*generate-choices* (*xys*#*xyss*))
then have *p1*: *length cs* = *length* (*xys*#*xyss*)
and *p2*: *fst* (*hd cs*) = *fst xys*
and *p3*: ((*snd* (*hd cs*) = *None*
$\vee$ (*snd* (*hd cs*) $\neq$ *None* $\wedge$ *the* (*snd* (*hd cs*)) $\in$ *set* (*snd xys*))))
and *p4*: ($\forall$ *i*<*length* (*tl cs*).
fst (*tl cs* ! *i*) = *fst* (*xyss* ! *i*) $\wedge$
(*snd* (*tl cs* ! *i*) = *None*
$\vee$ *snd* (*tl cs* ! *i*) $\neq$ *None* $\wedge$ *the* (*snd* (*tl cs* ! *i*)) $\in$ *set* (*snd* (*xyss* ! *i*))))
using $*$ **by** *blast*+
then have *length xyss* = *length* (*tl cs*) **and** *length* (*xys* # *xyss*) = *length* ([*hd cs*] @ *tl cs*)
by *auto*

have [*hd cs*]@(*tl cs*) = *cs*
by (*metis* (*no-types*) *p1 append.left-neutral append-Cons length-greater-0-conv*

list.collapse list.simps(*3*))
then have *p4b*: ($\forall$ *i*<*length cs*. *i* > *0* $\longrightarrow$
(*fst* (*cs* ! *i*) = *fst* ((*xys*#*xyss*) ! *i*) $\wedge$
(*snd* (*cs* ! *i*) = *None*
$\vee$ *snd* (*cs* ! *i*) $\neq$ *None* $\wedge$ *the* (*snd* (*cs* ! *i*)) $\in$ *set* (*snd* ((*xys*#*xyss*) ! *i*)))))
using *p4 list-append-idx-prop2*[*of xyss tl cs xys*#*xyss* [*hd cs*]@(*tl cs*)
λ *x y* . *fst x* = *fst y*
$\wedge$ (*snd x* = *None*
$\vee$ *snd x* $\neq$ *None* $\wedge$ *the* (*snd x*) $\in$ *set* (*snd y*)),
OF ‹*length xyss* = *length* (*tl cs*)›
‹*length* (*xys* # *xyss*) = *length* ([*hd cs*] @ *tl cs*)›]

```
    by (metis (no-types, lifting) One-nat-def Suc-pred
          ‹length (xys # xyss) = length ([hd cs] @ tl cs)› ‹length xyss = length (tl
cs)›
          length-Cons list.size(3) not-less-eq nth-Cons-pos nth-append)

  have p4a :(fst (cs ! 0) = fst ((xys#xyss) ! 0) ∧ (snd (cs ! 0) = None
            ∨ snd (cs ! 0) ≠ None ∧ the (snd (cs ! 0)) ∈ set (snd ((xys#xyss) !
0))))
      using p1 p2 p3 by (metis hd-conv-nth length-greater-0-conv list.simps(3)
nth-Cons-0)

  show ?thesis using p1 p4a p4b by fastforce
 qed


 moreover have (length cs = length (xys # xyss) ∧
                (∀ i<length cs.
                    fst (cs ! i) = fst ((xys # xyss) ! i) ∧
                    (snd (cs ! i) = None ∨
                      snd (cs ! i) ≠ None ∧ the (snd (cs ! i)) ∈ set (snd ((xys #
xyss) ! i)))))
               ⟹ cs ∈ set (generate-choices (xys#xyss))
  using ∗
  by (metis (no-types, lifting) Nitpick.size-list-simp(2) Suc-mono hd-conv-nth
     length-greater-0-conv length-tl list.sel(3) list.simps(3) nth-Cons-0 nth-tl)

 ultimately show ?case by blast
qed
```

1.4 Finding the Index of the First Element of a List Satisfying a Property

```
fun find-index :: ('a ⇒ bool) ⇒ 'a list ⇒ nat option where
 find-index f []  = None |
 find-index f (x#xs) = (if f x
   then Some 0
   else (case find-index f xs of Some k ⇒ Some (Suc k) | None ⇒ None))

lemma find-index-index :
 assumes find-index f xs = Some k
 shows k < length xs and f (xs ! k) and ⋀ j . j < k ⟹ ¬ f (xs ! j)
proof −
 have (k < length xs) ∧ (f (xs ! k)) ∧ (∀ j < k . ¬ (f (xs ! j)))
   using assms proof (induction xs arbitrary: k)
   case Nil
   then show ?case by auto
 next
   case (Cons x xs)
```

```
    show ?case proof (cases f x)
      case True
      then show ?thesis using Cons.prems by auto
    next
      case False
      then have find-index f (x#xs)
                = (case find-index f xs of Some k ⇒ Some (Suc k) | None ⇒ None)
        by auto
      then have (case find-index f xs of Some k ⇒ Some (Suc k) | None ⇒ None)
= Some k
        using Cons.prems by auto
      then obtain k′ where find-index f xs = Some k′ and k = Suc k′
        by (metis option.case-eq-if option.collapse option.distinct(1) option.sel)

      have k < length (x # xs) ∧ f ((x # xs) ! k)
        using Cons.IH[OF ‹find-index f xs = Some k′›] ‹k = Suc k′›
        by auto
      moreover have (∀ j<k. ¬ f ((x # xs) ! j))
          using Cons.IH[OF ‹find-index f xs = Some k′›] ‹k = Suc k′› False
less-Suc-eq-0-disj
        by auto
      ultimately show ?thesis by presburger
    qed
  qed
  then show k < length xs and f (xs ! k) and ⋀ j . j < k ⟹ ¬ f (xs ! j) by
simp+
qed

lemma find-index-exhaustive :
  assumes ∃ x ∈ set xs . f x
  shows find-index f xs ≠ None
  using assms proof (induction xs)
case Nil
  then show ?case by auto
next
  case (Cons x xs)
  then show ?case by (cases f x; auto)
qed
```

1.5 List Distinctness from Sorting

```
lemma non-distinct-repetition-indices :
  assumes ¬ distinct xs
  shows ∃ i j . i < j ∧ j < length xs ∧ xs ! i = xs ! j
  by (metis assms distinct-conv-nth le-neq-implies-less not-le)

lemma non-distinct-repetition-indices-rev :
  assumes i < j and j < length xs and xs ! i = xs ! j
  shows ¬ distinct xs
```

```
  using assms nth-eq-iff-index-eq by fastforce

lemma ordered-list-distinct :
  fixes xs :: ('a::preorder) list
  assumes ⋀ i . Suc i < length xs ⟹ (xs ! i) < (xs ! (Suc i))
  shows distinct xs
proof −
  have ⋀ i j . i < j ⟹ j < length xs ⟹ (xs ! i) < (xs ! j)
  proof −
    fix i j assume i < j and j < length xs
    then show xs ! i < xs ! j
      using assms proof (induction xs arbitrary: i j rule: rev-induct)
      case Nil
      then show ?case by auto
    next
      case (snoc a xs)
      show ?case proof (cases j < length xs)
        case True
        show ?thesis using snoc.IH[OF snoc.prems(1) True] snoc.prems(3)
        proof −
          have f1: i < length xs
            using True less-trans snoc.prems(1) by blast
          have f2: ∀ is isa n. if n < length is then (is @ isa) ! n
                   = (is ! n::integer) else (is @ isa) ! n = isa ! (n − length is)
            by (meson nth-append)
          then have f3: (xs @ [a]) ! i = xs ! i
            using f1
            by (simp add: nth-append)
          have xs ! i < xs ! j
            using f2
           by (metis Suc-lessD ‹(⋀i. Suc i < length xs ⟹ xs ! i < xs ! Suc i) ⟹
xs ! i < xs ! j›
             butlast-snoc length-append-singleton less-SucI nth-butlast snoc.prems(3))

          then show ?thesis
            using f3 f2 True
            by (simp add: nth-append)
        qed
      next
        case False
        then have (xs @ [a]) ! j = a
          using snoc.prems(2)
          by (metis length-append-singleton less-SucE nth-append-length)

        consider j = 1 | j > 1
          using ‹i < j›
          by linarith
        then show ?thesis proof cases
          case 1
```

```
        then have i = 0 and j = Suc i using ‹i < j› by linarith+
        then show ?thesis
          using snoc.prems(3)
          using snoc.prems(2) by blast
      next
        case 2
        then consider i < j − 1 | i = j − 1 using ‹i < j› by linarith+
        then show ?thesis proof cases
          case 1

          have (⋀i. Suc i < length xs ⟹ xs ! i < xs ! Suc i) ⟹ xs ! i < xs ! (j
− 1)
            using snoc.IH[OF 1] snoc.prems(2) 2 by simp
          then have le1: (xs @ [a]) ! i < (xs @ [a]) ! (j −1)
            using snoc.prems(2)
                  by (metis 2 False One-nat-def Suc-diff-Suc Suc-lessD diff-zero
snoc.prems(3)
                        length-append-singleton less-SucE not-less-eq nth-append
snoc.prems(1))
          moreover have le2: (xs @ [a]) ! (j −1) < (xs @ [a]) ! j
            using snoc.prems(2,3) 2 less-trans
        by (metis (full-types) One-nat-def Suc-diff-Suc diff-zero less-numeral-extra(1))

          ultimately show ?thesis
            using less-trans by blast
        next
          case 2
          then have j = Suc i using ‹1 < j› by linarith
          then show ?thesis
            using snoc.prems(3)
            using snoc.prems(2) by blast
        qed
      qed
    qed
  qed
qed

  then show ?thesis
    by (metis less-asym non-distinct-repetition-indices)
qed



lemma ordered-list-distinct-rev :
  fixes xs :: ('a::preorder) list
  assumes ⋀ i . Suc i < length xs ⟹ (xs ! i) > (xs ! (Suc i))
  shows distinct xs
proof −
  have ⋀ i . Suc i < length (rev xs) ⟹ ((rev xs) ! i) < ((rev xs) ! (Suc i))
```

```
    using assms
  proof −
    fix i :: nat
    assume a1: Suc i < length (rev xs)
    obtain nn :: nat ⇒ nat ⇒ nat where
      ∀ x0 x1. (∃ v2. x1 = Suc v2 ∧ v2 < x0) = (x1 = Suc (nn x0 x1) ∧ nn x0 x1
< x0)
      by moura
    then have f2: ∀ n na. (¬ n < Suc na ∨ n = 0 ∨ n = Suc (nn na n) ∧ nn na
n < na)
                    ∧ (n < Suc na ∨ n ≠ 0 ∧ (∀ nb. n ≠ Suc nb ∨ ¬ nb < na))
      by (meson less-Suc-eq-0-disj)
    have f3: Suc (length xs − Suc (Suc i)) = length (rev xs) − Suc i
      using a1 by (simp add: Suc-diff-Suc)
    have i < length (rev xs)
      using a1 by (meson Suc-lessD)
    then have i < length xs
      by simp
    then show rev xs ! i < rev xs ! Suc i
     using f3 f2 a1 by (metis (no-types) assms diff-less length-rev not-less-iff-gr-or-eq
rev-nth)
  qed
  then have distinct (rev xs)
    using ordered-list-distinct[of rev xs] by blast
  then show ?thesis by auto
qed
```

1.6 Calculating Prefixes and Suffixes

```
fun suffixes :: 'a list ⇒ 'a list list where
  suffixes [] = [[]] |
  suffixes (x#xs) = (suffixes xs) @ [x#xs]

lemma suffixes-set :
  set (suffixes xs) = {zs . ∃ ys . ys@zs = xs}
proof (induction xs)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  then have ∗: set (suffixes (x#xs)) = {zs . ∃ ys . ys@zs = xs} ∪ {x#xs}
    by auto

  have {zs . ∃ ys . ys@zs = xs} = {zs . ∃ ys . x#ys@zs = x#xs}
    by force
  then have {zs . ∃ ys . ys@zs = xs} = {zs . ∃ ys . ys@zs = x#xs ∧ ys ≠ []}
    by (metis Cons-eq-append-conv list.distinct(1))
  moreover have {x#xs} = {zs . ∃ ys . ys@zs = x#xs ∧ ys = []}
    by force
```

```
  ultimately show ?case using ∗ by force
qed
```

```
lemma prefixes-set : set (prefixes xs) = {xs′ . ∃ xs′′ . xs′@xs′′ = xs}
proof (induction xs)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  moreover have prefixes (x#xs) = [] # map ((#) x) (prefixes xs)
    by auto
  ultimately have ∗: set (prefixes (x#xs)) = insert [] (((#) x) ‘ {xs′. ∃xs′′. xs′
@ xs′′ = xs})
    by auto
  also have . . . = {xs′ . ∃ xs′′ . xs′@xs′′ = (x#xs)}
  proof
    show insert [] ((#) x ‘ {xs′. ∃xs′′. xs′ @ xs′′ = xs}) ⊆ {xs′. ∃xs′′. xs′ @ xs′′ =
x # xs}
      by auto
    show {xs′. ∃xs′′. xs′ @ xs′′ = x # xs} ⊆ insert [] ((#) x ‘ {xs′. ∃xs′′. xs′ @
xs′′ = xs})
    proof
      fix y assume y ∈ {xs′. ∃xs′′. xs′ @ xs′′ = x # xs}
      then obtain y′ where y@y′ = x # xs
        by blast
      then show y ∈ insert [] ((#) x ‘ {xs′. ∃xs′′. xs′ @ xs′′ = xs})
        by (cases y; auto)
    qed
  qed
  finally show ?case .
qed
```

```
fun is-prefix :: ′a list ⇒ ′a list ⇒ bool where
  is-prefix [] - = True |
  is-prefix (x#xs) [] = False |
  is-prefix (x#xs) (y#ys) = (x = y ∧ is-prefix xs ys)

lemma is-prefix-prefix : is-prefix xs ys = (∃ xs′ . ys = xs@xs′)
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case by auto
next
```

```
  case (Cons x xs)
  show ?case proof (cases is-prefix (x#xs) ys)
    case True
    then show ?thesis using Cons.IH
      by (metis append-Cons is-prefix.simps(2) is-prefix.simps(3) neq-Nil-conv)
  next
    case False
    then show ?thesis
      using Cons.IH by auto
  qed
qed


fun add-prefixes :: 'a list list ⇒ 'a list list where
  add-prefixes xs = concat (map prefixes xs)


lemma add-prefixes-set : set (add-prefixes xs) = {xs' . ∃ xs'' . xs'@xs'' ∈ set xs}
proof −
  have set (add-prefixes xs) = {xs' . ∃ x ∈ set xs . xs' ∈ set (prefixes x)}
    unfolding add-prefixes.simps by auto
  also have ... = {xs' . ∃ xs'' . xs'@xs'' ∈ set xs}
  proof (induction xs)
    case Nil
    then show ?case using prefixes-set by auto
  next
    case (Cons a xs)
    then show ?case
    proof −
      have ⋀ xs' . xs' ∈ {xs'. ∃x∈set (a # xs). xs' ∈ set (prefixes x)}
            ⟷ xs' ∈ {xs'. ∃xs''. xs' @ xs'' ∈ set (a # xs)}
      proof −
        fix xs'
        show xs' ∈ {xs'. ∃x∈set (a # xs). xs' ∈ set (prefixes x)}
              ⟷ xs' ∈ {xs'. ∃xs''. xs' @ xs'' ∈ set (a # xs)}
          unfolding prefixes-set by force
      qed
      then show ?thesis by blast
    qed
  qed
  finally show ?thesis by blast
qed


lemma prefixes-set-ob :
  assumes xs ∈ set (prefixes xss)
  obtains xs' where xss = xs@xs'
  using assms unfolding prefixes-set
  by auto
```

lemma *prefixes-finite* : *finite* { $x \in$ *set* (*prefixes xs*) . *P x*}
  by (*metis Collect-mem-eq List.finite-set finite-Collect-conjI*)

lemma *prefixes-set-Cons-insert*: *set* (*prefixes* (w' @ [*xy*])) = *Set.insert* (w'@[*xy*]) (*set* (*prefixes* (w')))
  unfolding *prefixes-set*
proof (*induction* w' *arbitrary*: *xy rule*: *rev-induct*)
  case *Nil*
  then show *?case*
    by (*auto*; *simp add*: *append-eq-Cons-conv*)
  next
    case (*snoc x xs*)
    then show *?case*
      by (*auto*; *metis* (*no-types*, *opaque-lifting*) *butlast.simps*(*2*) *butlast-append butlast-snoc*)
  qed

lemma *prefixes-set-subset*:
  set (*prefixes xs*) $\subseteq$ *set* (*prefixes* (*xs*@*ys*))
  unfolding *prefixes-set* **by** *auto*

lemma *prefixes-prefix-subset* :
  assumes *xs* $\in$ *set* (*prefixes ys*)
  shows *set* (*prefixes xs*) $\subseteq$ *set* (*prefixes ys*)
  using *assms* **unfolding** *prefixes-set* **by** *auto*

lemma *prefixes-butlast-is-prefix* :
  butlast xs $\in$ *set* (*prefixes xs*)
  unfolding *prefixes-set*
  by (*metis* (*mono-tags*, *lifting*) *append-butlast-last-id butlast.simps*(*1*) *mem-Collect-eq self-append-conv2*)

lemma *prefixes-take-iff* :
  xs $\in$ *set* (*prefixes ys*) $\longleftrightarrow$ *take* (*length xs*) *ys* = *xs*
proof
  show *xs* $\in$ *set* (*prefixes ys*) $\Longrightarrow$ *take* (*length xs*) *ys* = *xs*
    unfolding *prefixes-set*
    by (*simp add*: *append-eq-conv-conj*)

  show *take* (*length xs*) *ys* = *xs* $\Longrightarrow$ *xs* $\in$ *set* (*prefixes ys*)
    unfolding *prefixes-set*
    by (*metis* (*mono-tags*, *lifting*) *append-take-drop-id mem-Collect-eq*)
qed

lemma *prefixes-set-Nil* : [] $\in$ *list.set* (*prefixes xs*)
  by (*metis append.left-neutral list.set-intros*(*1*) *prefixes.simps*(*1*) *prefixes-set-subset subset-iff*)

lemma *prefixes-prefixes* :
  assumes $ys \in list.set\ (prefixes\ xs)$
    $zs \in list.set\ (prefixes\ xs)$
  shows $ys \in list.set\ (prefixes\ zs) \vee zs \in list.set\ (prefixes\ ys)$
proof (*rule ccontr*)
  let $?ys = take\ (length\ ys)\ zs$
  let $?zs = take\ (length\ zs)\ ys$

  assume $\neg\ (ys \in list.set\ (prefixes\ zs) \vee zs \in list.set\ (prefixes\ ys))$
  then have $?ys \neq ys$ **and** $?zs \neq zs$
    using *prefixes-take-iff* **by** *blast+*
  moreover have $?ys = ys \vee ?zs = zs$
    using *assms*
    by (*metis linear min.commute prefixes-take-iff take-all-iff take-take*)
  ultimately show *False*
    by *simp*
qed

1.6.1 Pairs of Distinct Prefixes

fun *prefix-pairs* :: $'a\ list \Rightarrow ('a\ list \times 'a\ list)\ list$
  where *prefix-pairs* $[] = []$ |
    prefix-pairs xs = *prefix-pairs* (*butlast* xs) @ (*map* (λ ys. (ys,xs)) (*butlast* (*prefixes* xs)))

lemma *prefixes-butlast* :
  $set\ (butlast\ (prefixes\ xs)) = \{ys\ .\ \exists\ zs\ .\ ys@zs = xs \wedge zs \neq []\}$
proof (*induction length xs arbitrary*: xs)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc k*)

  then obtain $x\ xs'$ **where** $xs = x\#xs'$ **and** $k = length\ xs'$
    by (*metis length-Suc-conv*)

  then have $prefixes\ xs = []\ \#\ map\ ((\#)\ x)\ (prefixes\ xs')$
    by *auto*
  then have $butlast\ (prefixes\ xs) = []\ \#\ map\ ((\#)\ x)\ (butlast\ (prefixes\ xs'))$
    by (*simp add*: *map-butlast*)
  then have $set\ (butlast\ (prefixes\ xs)) = insert\ []\ (((\#)\ x)\ `\ \{ys\ .\ \exists\ zs\ .\ ys@zs = xs' \wedge zs \neq []\})$
    using *Suc.hyps*(*1*)[*OF* ‹$k = length\ xs'$›]
    by *auto*
  also have $\ldots = \{ys\ .\ \exists\ zs\ .\ ys@zs = (x\#xs') \wedge zs \neq []\}$
  proof
    show $insert\ []\ ((\#)\ x\ `\ \{ys.\ \exists zs.\ ys\ @\ zs = xs' \wedge zs \neq []\}) \subseteq \{ys.\ \exists zs.\ ys\ @\ zs$

$= x \# xs' \wedge zs \neq []\}$
by *auto*
show $\{ys.\ \exists zs.\ ys\ @\ zs = x \# xs' \wedge zs \neq []\} \subseteq insert\ []\ ((\#)\ x\ `\ \{ys.\ \exists zs.\ ys\ @\ zs = xs' \wedge zs \neq []\})$
proof
fix *ys* **assume** $ys \in \{ys.\ \exists zs.\ ys\ @\ zs = x \# xs' \wedge zs \neq []\}$
then show $ys \in insert\ []\ ((\#)\ x\ `\ \{ys.\ \exists zs.\ ys\ @\ zs = xs' \wedge zs \neq []\})$
by (*cases ys*; *auto*)
qed
qed
finally show *?case*
unfolding ‹$xs = x\#xs'$› **.**
qed

lemma *prefix-pairs-set* :
$set\ (prefix\text{-}pairs\ xs) = \{(zs,ys) \mid zs\ ys\ .\ \exists\ xs1\ xs2\ .\ zs@xs1 = ys \wedge ys@xs2 = xs \wedge xs1 \neq []\}$
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
have $prefix\text{-}pairs\ (xs\ @\ [x]) = prefix\text{-}pairs\ (butlast\ (xs\ @\ [x]))\ @\ (map\ (\lambda\ ys.\ (ys,(xs\ @\ [x])))\ (butlast\ (prefixes\ (xs\ @\ [x]))))$
by (*cases* $(xs\ @\ [x])$; *auto*)
then have $*$: $prefix\text{-}pairs\ (xs\ @\ [x]) = prefix\text{-}pairs\ xs\ @\ (map\ (\lambda\ ys.\ (ys,(xs\ @\ [x])))\ (butlast\ (prefixes\ (xs\ @\ [x]))))$
by *auto*

have $set\ (prefix\text{-}pairs\ xs) = \{(zs,\ ys) \mid zs\ ys.\ \exists xs1\ xs2.\ zs\ @\ xs1 = ys \wedge ys\ @\ xs2 = xs \wedge xs1 \neq []\}$
using *snoc.IH* **by** *assumption*
then have $set\ (prefix\text{-}pairs\ xs) = \{(zs,\ ys) \mid zs\ ys.\ \exists xs1\ xs2.\ zs\ @\ xs1 = ys \wedge ys\ @\ xs2\ @\ [x] = xs@[x] \wedge xs1 \neq []\}$
by *auto*
also have ... $= \{(zs,\ ys) \mid zs\ ys.\ \exists xs1\ xs2.\ zs\ @\ xs1 = ys \wedge ys\ @\ xs2 = xs\ @[x] \wedge xs1 \neq [] \wedge xs2 \neq []\}$
proof −
let $?P1 = \lambda\ zs\ ys\ .\ (\exists xs1\ xs2.\ zs\ @\ xs1 = ys \wedge ys\ @\ xs2\ @\ [x] = xs@[x] \wedge xs1 \neq [])$
let $?P2 = \lambda\ zs\ ys\ .\ (\exists xs1\ xs2.\ zs\ @\ xs1 = ys \wedge ys\ @\ xs2 = xs\ @[x] \wedge xs1 \neq [] \wedge xs2 \neq [])$

have $\bigwedge ys\ zs\ .\ ?P2\ zs\ ys \Longrightarrow ?P1\ zs\ ys$
by (*metis append-assoc butlast-append butlast-snoc*)
then have $\bigwedge ys\ zs\ .\ ?P1\ ys\ zs = ?P2\ ys\ zs$
by *blast*
then show *?thesis* **by** *force*

qed
finally have *set* (*prefix-pairs xs*) = {(*zs*, *ys*) |*zs ys*. ∃ *xs1 xs2*. *zs* @ *xs1* = *ys* ∧ *ys* @ *xs2* = *xs* @ [*x*] ∧ *xs1* ≠ [] ∧ *xs2* ≠ []}
by *assumption*

moreover have *set* (*map* (λ *ys*. (*ys*,(*xs* @ [*x*]))) (*butlast* (*prefixes* (*xs* @ [*x*])))) = {(*zs*, *ys*) |*zs ys*. ∃ *xs1 xs2*. *zs* @ *xs1* = *ys* ∧ *ys* @ *xs2* = *xs* @ [*x*] ∧ *xs1* ≠ [] ∧ *xs2* = []}
using *prefixes-butlast*[*of xs*@[*x*]] **by** *force*

ultimately show *?case* **using** ∗ **by** *force*
qed

lemma *prefix-pairs-set-alt* :
set (*prefix-pairs xs*) = {(*xs1*,*xs1*@*xs2*) | *xs1 xs2* . *xs2* ≠ [] ∧ (∃ *xs3* . *xs1*@*xs2*@*xs3* = *xs*)}
unfolding *prefix-pairs-set* **by** *auto*

lemma *prefixes-Cons* :
assumes (*x*#*xs*) ∈ *set* (*prefixes* (*y*#*ys*))
shows *x* = *y* **and** *xs* ∈ *set* (*prefixes ys*)
proof −
show *x* = *y*
by (*metis Cons-eq-appendI assms nth-Cons-0 prefixes-set-ob*)

show *xs* ∈ *set* (*prefixes ys*)
proof −
obtain *xs′ xs′′* **where** (*x*#*xs*) = *xs′* **and** (*y*#*ys*) = *xs′*@*xs′′*
by (*meson assms prefixes-set-ob*)
then have *xs′* = *x*#*tl xs′*
by *auto*
then have *xs* = *tl xs′*
using ‹(*x*#*xs*) = *xs′*› **by** *auto*
moreover have *ys* = (*tl xs′*)@*xs′′*
using ‹(*y*#*ys*) = *xs′*@*xs′′*› ‹*xs′* = *x*#*tl xs′*›
by (*metis append-Cons list.inject*)
ultimately show *?thesis*
unfolding *prefixes-set* **by** *blast*
qed
qed

lemma *prefixes-prepend* :
assumes *xs′* ∈ *set* (*prefixes xs*)
shows *ys*@*xs′* ∈ *set* (*prefixes* (*ys*@*xs*))
proof −
obtain *xs′′* **where** *xs* = *xs′*@*xs′′*
using *assms*
using *prefixes-set-ob* **by** *auto*
then have (*ys*@*xs*) = (*ys*@*xs′*)@*xs′′*

```
    by auto
  then show ?thesis
    unfolding prefixes-set by auto
qed


lemma prefixes-prefix-suffix-ob :
  assumes a ∈ set (prefixes (b@c))
  and     a ∉ set (prefixes b)
obtains c′ c′′ where c = c′@c′′
                 and a = b@c′
                 and c′ ≠ []
proof −
  have ∃ c′ c′′ . c = c′@c′′ ∧ a = b@c′ ∧ c′ ≠ []
    using assms
  proof (induction b arbitrary: a)
    case Nil
    then show ?case
      unfolding prefixes-set
      by fastforce
  next
    case (Cons x xs)
    show ?case proof (cases a)
      case Nil
      then show ?thesis
        by (metis Cons.prems(2) list.size(3) prefixes-take-iff take-eq-Nil)
    next
      case (Cons a′ as)
      then have a′ # as ∈ set (prefixes (x #(xs@c)))
        using Cons.prems(1) by auto

      have a′ = x and as ∈ set (prefixes (xs@c))
        using prefixes-Cons[OF ‹a′ # as ∈ set (prefixes (x #(xs@c)))›]
        by auto
      moreover have as ∉ set (prefixes xs)
         using ‹a ∉ set (prefixes (x # xs))› unfolding Cons ‹a′ = x› prefixes-set
by auto

      ultimately obtain c′ c′′ where c = c′@c′′
                                 and as = xs@c′
                                 and c′ ≠ []
        using Cons.IH by blast
      then have c = c′@c′′ and a = (x#xs)@c′ and c′ ≠ []
        unfolding Cons ‹a′ = x› by auto
      then show ?thesis
        using that by blast
    qed
  qed
  then show ?thesis using that by blast
```

qed

fun *list-ordered-pairs* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
list-ordered-pairs [] = [] |
list-ordered-pairs (*x*#*xs*) = (*map* (*Pair x*) *xs*) @ (*list-ordered-pairs xs*)

lemma *list-ordered-pairs-set-containment* :
assumes $x \in$ *list.set xs*
and $y \in$ *list.set xs*
and $x \neq y$
shows (*x*,*y*) ∈ *list.set* (*list-ordered-pairs xs*) ∨ (*y*,*x*) ∈ *list.set* (*list-ordered-pairs xs*)
using *assms* **by** (*induction xs*; *auto*)

1.7 Calculating Distinct Non-Reflexive Pairs over List Elements

fun *non-sym-dist-pairs'* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
non-sym-dist-pairs' [] = [] |
non-sym-dist-pairs' (*x*#*xs*) = (*map* (λ *y*. (*x*,*y*)) *xs*) @ *non-sym-dist-pairs' xs*

fun *non-sym-dist-pairs* :: *'a list* ⇒ (*'a* × *'a*) *list* **where**
non-sym-dist-pairs xs = *non-sym-dist-pairs'* (*remdups xs*)

lemma *non-sym-dist-pairs-subset* : *set* (*non-sym-dist-pairs xs*) ⊆ (*set xs*) × (*set xs*)
by (*induction xs*; *auto*)

lemma *non-sym-dist-pairs'-elems-distinct*:
assumes *distinct xs*
and (*x*,*y*) ∈ *set* (*non-sym-dist-pairs' xs*)
shows $x \in$ *set xs*
and $y \in$ *set xs*
and $x \neq y$
proof −
show $x \in$ *set xs* **and** $y \in$ *set xs*
using *non-sym-dist-pairs-subset assms*(*2*) **by** (*induction xs*; *auto*)+
show $x \neq y$
using *assms* **by** (*induction xs*; *auto*)
qed

lemma *non-sym-dist-pairs-elems-distinct*:
assumes (*x*,*y*) ∈ *set* (*non-sym-dist-pairs xs*)
shows $x \in$ *set xs*
and $y \in$ *set xs*
and $x \neq y$
using *non-sym-dist-pairs'-elems-distinct assms*
unfolding *non-sym-dist-pairs.simps* **by** *fastforce*+

```
lemma non-sym-dist-pairs-elems :
  assumes x ∈ set xs
  and     y ∈ set xs
  and     x ≠ y
shows (x,y) ∈ set (non-sym-dist-pairs xs) ∨ (y,x) ∈ set (non-sym-dist-pairs xs)
  using assms by (induction xs; auto)



lemma non-sym-dist-pairs′-elems-non-refl :
  assumes distinct xs
  and     (x,y) ∈ set (non-sym-dist-pairs′ xs)
shows (y,x) ∉ set (non-sym-dist-pairs′ xs)
  using assms
proof (induction xs arbitrary: x y)
  case Nil
  then show ?case by auto
next
  case (Cons z zs)
  then have distinct zs by auto

  have x ≠ y
    using non-sym-dist-pairs′-elems-distinct[OF Cons.prems] by simp

  consider (a) (x,y) ∈ set (map (Pair z) zs) |
           (b) (x,y) ∈ set (non-sym-dist-pairs′ zs)
   using ‹(x,y) ∈ set (non-sym-dist-pairs′ (z#zs))› unfolding non-sym-dist-pairs′.simps
by auto
  then show ?case proof cases
    case a
    then have x = z by auto
    then have (y,x) ∉ set (map (Pair z) zs)
      using ‹x ≠ y› by auto
    moreover have x ∉ set zs
      using ‹x = z› ‹distinct (z#zs)› by auto
    ultimately show ?thesis
      using ‹distinct zs› non-sym-dist-pairs′-elems-distinct(2) by fastforce
  next
    case b
    then have x ≠ z and y ≠ z
      using Cons.prems unfolding non-sym-dist-pairs′.simps
      by (meson distinct.simps(2) non-sym-dist-pairs′-elems-distinct(1,2))+

    then show ?thesis
      using Cons.IH[OF ‹distinct zs› b] by auto
  qed
qed
```

lemma *non-sym-dist-pairs-elems-non-refl* :
assumes $(x,y) \in$ *set* (*non-sym-dist-pairs xs*)
shows $(y,x) \notin$ *set* (*non-sym-dist-pairs xs*)
using *assms* **by** (*simp add*: *non-sym-dist-pairs′-elems-non-refl*)

lemma *non-sym-dist-pairs-set-iff* :
$(x,y) \in$ *set* (*non-sym-dist-pairs xs*)
$\longleftrightarrow (x \neq y \wedge x \in$ *set xs* $\wedge\ y \in$ *set xs* $\wedge\ (y,x) \notin$ *set* (*non-sym-dist-pairs xs*))
using *non-sym-dist-pairs-elems-non-refl*[*of x y xs*]
non-sym-dist-pairs-elems[*of x xs y*]
non-sym-dist-pairs-elems-distinct[*of x y xs*] **by** *blast*

1.8 Finite Linear Order From List Positions

fun *linear-order-from-list-position′* :: *′a list* ⇒ (*′a* × *′a*) *list* **where**
linear-order-from-list-position′ [] = [] |
linear-order-from-list-position′ (*x*#*xs*)
= (*x*,*x*) # (*map* (λ *y* . (*x*,*y*)) *xs*) @ (*linear-order-from-list-position′ xs*)

fun *linear-order-from-list-position* :: *′a list* ⇒ (*′a* × *′a*) *list* **where**
linear-order-from-list-position xs = *linear-order-from-list-position′* (*remdups xs*)

lemma *linear-order-from-list-position-set* :
set (*linear-order-from-list-position xs*)
= (*set* (*map* (λ *x* . (*x*,*x*)) *xs*)) ∪ *set* (*non-sym-dist-pairs xs*)
by (*induction xs*; *auto*)

lemma *linear-order-from-list-position-total*:
total-on (*set xs*) (*set* (*linear-order-from-list-position xs*))
unfolding *linear-order-from-list-position-set*
using *non-sym-dist-pairs-elems*[*of - xs*]
by (*meson UnI2 total-onI*)

lemma *linear-order-from-list-position-refl*:
refl-on (*set xs*) (*set* (*linear-order-from-list-position xs*))
proof (*rule refl-onI*)
show $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow (x,\ x) \in$ *set* (*linear-order-from-list-position xs*)
unfolding *linear-order-from-list-position-set*
using *non-sym-dist-pairs-subset*[*of xs*] **by** *auto*
qed

lemma *linear-order-from-list-position-antisym*:
antisym (*set* (*linear-order-from-list-position xs*))
proof

```
  fix x y assume (x, y) ∈ set (linear-order-from-list-position xs)
         and    (y, x) ∈ set (linear-order-from-list-position xs)
  then have (x, y) ∈ set (map (λx. (x, x)) xs) ∪ set (non-sym-dist-pairs xs)
      and  (y, x) ∈ set (map (λx. (x, x)) xs) ∪ set (non-sym-dist-pairs xs)
    unfolding linear-order-from-list-position-set by blast+
  then consider (a) (x, y) ∈ set (map (λx. (x, x)) xs) |
                (b) (x, y) ∈ set (non-sym-dist-pairs xs)
    by blast
  then show x = y
  proof cases
    case a
    then show ?thesis by auto
  next
    case b
    then have x ≠ y and (y,x) ∉ set (non-sym-dist-pairs xs)
      using non-sym-dist-pairs-set-iff[of x y xs] by simp+
    then have (y, x) ∉ set (map (λx. (x, x)) xs) ∪ set (non-sym-dist-pairs xs)
      by auto
    then show ?thesis
     using ‹(y, x) ∈ set (map (λx. (x, x)) xs) ∪ set (non-sym-dist-pairs xs)› by
blast
  qed
qed


lemma non-sym-dist-pairs′-indices :
  distinct xs ⟹ (x,y) ∈ set (non-sym-dist-pairs′ xs)
   ⟹ (∃ i j . xs ! i = x ∧ xs ! j = y ∧ i < j ∧ i < length xs ∧ j < length xs)
proof (induction xs)
  case Nil
  then show ?case by auto
next
  case (Cons a xs)
  show ?case proof (cases a = x)
    case True
    then have (a#xs) ! 0 = x and 0 < length (a#xs)
      by auto

    have y ∈ set xs
     using non-sym-dist-pairs′-elems-distinct(2,3)[OF Cons.prems(1,2)] True by
auto
    then obtain j where xs ! j = y and j < length xs
      by (meson in-set-conv-nth)
    then have (a#xs) ! (Suc j) = y and Suc j < length (a#xs)
      by auto

    then show ?thesis
      using ‹(a#xs) ! 0 = x› ‹0 < length (a#xs)› by blast
  next
```

```
    case False
    then have (x,y) ∈ set (non-sym-dist-pairs′ xs)
      using Cons.prems(2) by auto
    then show ?thesis
      using Cons.IH Cons.prems(1)
      by (metis Suc-mono distinct.simps(2) length-Cons nth-Cons-Suc)
  qed
qed


lemma non-sym-dist-pairs′-trans: distinct xs ⟹ trans (set (non-sym-dist-pairs′
xs))
proof
  fix x y z assume distinct xs
          and    (x, y) ∈ set (non-sym-dist-pairs′ xs)
          and    (y, z) ∈ set (non-sym-dist-pairs′ xs)

  obtain nx ny where xs ! nx = x and xs ! ny = y and nx < ny
              and nx < length xs and ny < length xs
   using non-sym-dist-pairs′-indices[OF ‹distinct xs› ‹(x, y) ∈ set (non-sym-dist-pairs′
xs)›]
    by blast

  obtain ny′ nz where xs ! ny′ = y and xs ! nz = z and ny′< nz
               and ny′ < length xs and nz < length xs
   using non-sym-dist-pairs′-indices[OF ‹distinct xs› ‹(y, z) ∈ set (non-sym-dist-pairs′
xs)›]
    by blast

  have ny′ = ny
    using ‹distinct xs› ‹xs ! ny = y› ‹xs ! ny′ = y› ‹ny < length xs› ‹ny′ < length
xs›
          nth-eq-iff-index-eq
    by metis
  then have nx < nz
    using ‹nx < ny› ‹ny′ < nz› by auto

  then have nx ≠ nz by simp
  then have x ≠ z
    using ‹distinct xs› ‹xs ! nx = x› ‹xs ! nz = z› ‹nx < length xs› ‹nz < length
xs›
          nth-eq-iff-index-eq
    by metis

  have remdups xs = xs
    using ‹distinct xs› by auto

  have ¬(z, x) ∈ set (non-sym-dist-pairs′ xs)
```

```
  proof
    assume (z, x) ∈ set (non-sym-dist-pairs′ xs)
    then obtain nz′ nx′ where xs ! nx′ = x and xs ! nz′ = z and nz′< nx′
                        and nx′ < length xs and nz′ < length xs
      using non-sym-dist-pairs′-indices[OF ‹distinct xs›, of z x] by metis

    have nx′ = nx
      using ‹distinct xs› ‹xs ! nx = x› ‹xs ! nx′ = x› ‹nx < length xs› ‹nx′ < length
xs›
            nth-eq-iff-index-eq
      by metis
    moreover have nz′ = nz
      using ‹distinct xs› ‹xs ! nz = z› ‹xs ! nz′ = z› ‹nz < length xs› ‹nz′ < length
xs›
            nth-eq-iff-index-eq
      by metis
    ultimately have nz < nx
      using ‹nz′< nx′› by auto
    then show False
      using ‹nx < nz› by simp
  qed
  then show (x, z) ∈ set (non-sym-dist-pairs′ xs)
      using non-sym-dist-pairs′-elems-distinct(1)[OF ‹distinct xs› ‹(x, y) ∈ set
(non-sym-dist-pairs′ xs)›]
              non-sym-dist-pairs′-elems-distinct(2)[OF ‹distinct xs› ‹(y, z) ∈ set
(non-sym-dist-pairs′ xs)›]
          ‹x ≠ z›
          non-sym-dist-pairs-elems[of x xs z]
    unfolding non-sym-dist-pairs.simps ‹remdups xs = xs›
    by blast
qed


lemma non-sym-dist-pairs-trans: trans (set (non-sym-dist-pairs xs))
  using non-sym-dist-pairs′-trans[of remdups xs, OF distinct-remdups]
  unfolding non-sym-dist-pairs.simps
  by assumption



lemma linear-order-from-list-position-trans: trans (set (linear-order-from-list-position
xs))
proof
  fix x y z assume (x, y) ∈ set (linear-order-from-list-position xs)
              and (y, z) ∈ set (linear-order-from-list-position xs)
  then consider (a) (x, y) ∈ set (map (λx. (x, x)) xs) ∧ (y, z) ∈ set (map (λx.
(x, x)) xs) |
              (b) (x, y) ∈ set (map (λx. (x, x)) xs) ∧ (y, z) ∈ set (non-sym-dist-pairs
xs) |
```

```
            (c) (x, y) ∈ set (non-sym-dist-pairs xs) ∧ (y, z) ∈ set (map (λx. (x,
x)) xs) |
          (d) (x, y) ∈ set (non-sym-dist-pairs xs) ∧ (y, z) ∈ set (non-sym-dist-pairs
xs)
    unfolding linear-order-from-list-position-set by blast+
  then show (x, z) ∈ set (linear-order-from-list-position xs)
  proof cases
    case a
    then show ?thesis unfolding linear-order-from-list-position-set by auto
  next
    case b
    then show ?thesis unfolding linear-order-from-list-position-set by auto
  next
    case c
    then show ?thesis unfolding linear-order-from-list-position-set by auto
  next
    case d
    then show ?thesis unfolding linear-order-from-list-position-set
                  using non-sym-dist-pairs-trans
                  by (metis UnI2 transE)
  qed
qed
```

1.9 Find And Remove in a Single Pass

```
fun find-remove' :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list ⇒ ('a × 'a list) option where
  find-remove' P [] - = None |
  find-remove' P (x#xs) prev = (if P x
      then Some (x,prev@xs)
      else find-remove' P xs (prev@[x]))

fun find-remove :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a × 'a list) option where
  find-remove P xs = find-remove' P xs []

lemma find-remove'-set :
  assumes find-remove' P xs prev = Some (x,xs')
shows P x
and   x ∈ set xs
and   xs' = prev@(remove1 x xs)
proof −
  have P x ∧ x ∈ set xs ∧ xs' = prev@(remove1 x xs)
    using assms proof (induction xs arbitrary: prev xs')
    case Nil
    then show ?case by auto
  next
    case (Cons x xs)
    show ?case proof (cases P x)
      case True
      then show ?thesis using Cons by auto
```

```
    next
      case False
      then show ?thesis using Cons by fastforce
    qed
  qed
  then show P x
      and   x ∈ set xs
      and   xs' = prev@(remove1 x xs)
    by blast+
qed

lemma find-remove'-set-rev :
  assumes x ∈ set xs
  and     P x
shows find-remove' P xs prev ≠ None
using assms(1) proof(induction xs arbitrary: prev)
  case Nil
  then show ?case by auto
next
  case (Cons x' xs)
  show ?case proof (cases P x)
    case True
    then show ?thesis using Cons by auto
  next
    case False
    then show ?thesis using Cons
      using assms(2) by auto
  qed
qed


lemma find-remove-None-iff :
  find-remove P xs = None ⟷ ¬ (∃ x . x ∈ set xs ∧ P x)
  unfolding find-remove.simps
  using find-remove'-set(1,2)
        find-remove'-set-rev
  by (metis old.prod.exhaust option.exhaust)

lemma find-remove-set :
  assumes find-remove P xs = Some (x,xs')
shows P x
and   x ∈ set xs
and   xs' = (remove1 x xs)
  using assms find-remove'-set[of P xs [] x xs'] by auto



fun find-remove-2' :: ('a⇒'b⇒bool) ⇒ 'a list ⇒ 'b list ⇒ 'a list ⇒ ('a × 'b × 'a
```

```
list) option
  where
  find-remove-2' P [] - - = None |
  find-remove-2' P (x#xs) ys prev = (case find (λy . P x y) ys of
      Some y ⇒ Some (x,y,prev@xs) |
      None   ⇒ find-remove-2' P xs ys (prev@[x]))

fun find-remove-2 :: ('a ⇒ 'b ⇒ bool) ⇒ 'a list ⇒ 'b list ⇒ ('a × 'b × 'a list)
option where
  find-remove-2 P xs ys = find-remove-2' P xs ys []


lemma find-remove-2'-set :
  assumes find-remove-2' P xs ys prev = Some (x,y,xs')
shows P x y
and   x ∈ set xs
and   y ∈ set ys
and   distinct (prev@xs) ⟹ set xs' = (set prev ∪ set xs) − {x}
and   distinct (prev@xs) ⟹ distinct xs'
and   xs' = prev@(remove1 x xs)
and   find (P x) ys = Some y
proof −
  have P x y
        ∧ x ∈ set xs
        ∧ y ∈ set ys
        ∧ (distinct (prev@xs) ⟶ set xs' = (set prev ∪ set xs) − {x})
        ∧ (distinct (prev@xs) ⟶ distinct xs')
        ∧ (xs' = prev@(remove1 x xs))
        ∧ find (P x) ys = Some y
    using assms
  proof (induction xs arbitrary: prev xs' x y)
    case Nil
    then show ?case by auto
  next
    case (Cons x' xs)
    then show ?case proof (cases find (λy . P x' y) ys)
      case None
        then have find-remove-2' P (x' # xs) ys prev = find-remove-2' P xs ys
(prev@[x'])
        using Cons.prems(1) by auto
      hence ∗: find-remove-2' P xs ys (prev@[x']) = Some (x, y, xs')
        using Cons.prems(1) by simp

      have x' ≠ x
        by (metis ∗ Cons.IH None find-from)
      moreover have distinct (prev @ x' # xs) ⟶ distinct ((x' # prev) @ xs)
        by auto
      ultimately show ?thesis using Cons.IH[OF ∗]
        by auto
```

```
    next
      case (Some y')
      then have find-remove-2' P (x' # xs) ys prev = Some (x',y',prev@xs)
        by auto
      then show ?thesis using Some
        using Cons.prems(1) find-condition find-set by fastforce
    qed
  qed
  then show P x y
      and   x ∈ set xs
      and   y ∈ set ys
      and   distinct (prev @ xs) ⟹ set xs' = (set prev ∪ set xs) − {x}
      and   distinct (prev@xs) ⟹ distinct xs'
      and   xs' = prev@(remove1 x xs)
      and   find (P x) ys = Some y
    by blast+
qed


lemma find-remove-2'-strengthening :
  assumes find-remove-2' P xs ys prev = Some (x,y,xs')
  and     P' x y
  and     ⋀ x' y' . P' x' y' ⟹ P x' y'
shows find-remove-2' P' xs ys prev = Some (x,y,xs')
  using assms proof (induction xs arbitrary: prev)
  case Nil
  then show ?case by auto
next
  case (Cons x' xs)
  then show ?case proof (cases find (λy . P x' y) ys)
    case None
    then show ?thesis using Cons
        by (metis (mono-tags, lifting) find-None-iff find-remove-2'.simps(2) op-
tion.simps(4))
  next
    case (Some a)
    then have x' = x and a = y
      using Cons.prems(1) unfolding find-remove-2'.simps by auto
    then have find (λy . P x y) ys = Some y
      using find-remove-2'-set[OF Cons.prems(1)] by auto
    then have find (λy . P' x y) ys = Some y
      using Cons.prems(3) proof (induction ys)
      case Nil
      then show ?case by auto
    next
      case (Cons y' ys)
      then show ?case
        by (metis assms(2) find.simps(2) option.inject)
```

```
    qed

    then show ?thesis
      using find-remove-2'-set(6)[OF Cons.prems(1)]
      unfolding ‹x' = x› find-remove-2'.simps by auto
  qed
qed

lemma find-remove-2-strengthening :
  assumes find-remove-2 P xs ys = Some (x,y,xs')
  and     P' x y
  and     ⋀ x' y' . P' x' y' ⟹ P x' y'
shows find-remove-2 P' xs ys = Some (x,y,xs')
  using assms find-remove-2'-strengthening
  by (metis find-remove-2.simps)



lemma find-remove-2'-prev-independence :
  assumes find-remove-2' P xs ys prev = Some (x,y,xs')
  shows ∃ xs'' . find-remove-2' P xs ys prev' = Some (x,y,xs'')
  using assms proof (induction xs arbitrary: prev prev' xs')
  case Nil
  then show ?case by auto
next
  case (Cons x' xs)
  show ?case proof (cases find (λy . P x' y) ys)
    case None
    then show ?thesis
      using Cons.IH Cons.prems by auto

  next
    case (Some a)
    then show ?thesis using Cons.prems unfolding find-remove-2'.simps
      by simp
  qed
qed


lemma find-remove-2'-filter :
  assumes find-remove-2' P (filter P' xs) ys prev = Some (x,y,xs')
  and     ⋀ x y . ¬ P' x ⟹ ¬ P x y
shows ∃ xs'' . find-remove-2' P xs ys prev = Some (x,y,xs'')
  using assms(1) proof (induction xs arbitrary: prev prev xs')
  case Nil
  then show ?case by auto
next
  case (Cons x' xs)
  then show ?case proof (cases P' x')
```

```
    case True
    then have *:find-remove-2' P (filter P' (x' # xs)) ys prev
              = find-remove-2' P (x' # filter P' xs) ys prev
      by auto

    show ?thesis proof (cases find (λy . P x' y) ys)
      case None
      then show ?thesis
       by (metis Cons.IH Cons.prems  find-remove-2'.simps(2) option.simps(4) *)
    next
      case (Some a)
      then have x' = x and a = y
        using Cons.prems
        unfolding * find-remove-2'.simps by auto

      show ?thesis
        using Some
        unfolding ‹x' = x› ‹a = y› find-remove-2'.simps
        by simp
    qed
  next
    case False
    then have find-remove-2' P (filter P' xs) ys prev = Some (x,y,xs')
      using Cons.prems by auto

    from False assms(2) have find (λy . P x' y) ys = None
      by (simp add: find-None-iff)
   then have find-remove-2' P (x'#xs) ys prev = find-remove-2' P xs ys (prev@[x'])
      by auto

    show ?thesis
     using Cons.IH[OF ‹find-remove-2' P (filter P' xs) ys prev = Some (x,y,xs')›]

       unfolding ‹find-remove-2' P (x'#xs) ys prev = find-remove-2' P xs ys
(prev@[x'])›
      using find-remove-2'-prev-independence by metis
  qed
qed


lemma find-remove-2-filter :
  assumes find-remove-2 P (filter P' xs) ys = Some (x,y,xs')
  and     ⋀ x y . ¬ P' x ⟹ ¬ P x y
shows ∃ xs'' . find-remove-2 P xs ys = Some (x,y,xs'')
  using assms by (simp add: find-remove-2'-filter)


lemma find-remove-2'-index :
  assumes find-remove-2' P xs ys prev = Some (x,y,xs')
```

```
  obtains i i' where i < length xs
                     xs ! i = x
                     ⋀ j . j < i ⟹ find (λy . P (xs ! j) y) ys = None
                     i' < length ys
                     ys ! i' = y
                     ⋀ j . j < i' ⟹ ¬ P (xs ! i) (ys ! j)
proof −
  have ∃ i i' . i < length xs
                ∧ xs ! i = x
                ∧ (∀ j < i . find (λy . P (xs ! j) y) ys = None)
                ∧ i' < length ys ∧ ys ! i' = y
                ∧ (∀ j < i' . ¬ P (xs ! i) (ys ! j))
    using assms
  proof (induction xs arbitrary: prev xs' x y)
    case Nil
    then show ?case by auto
  next
    case (Cons x' xs)
    then show ?case proof (cases find (λy . P x' y) ys)
      case None
       then have find-remove-2' P (x' # xs) ys prev = find-remove-2' P xs ys
(prev@[x'])
        using Cons.prems(1) by auto
      hence ∗: find-remove-2' P xs ys (prev@[x']) = Some (x, y, xs')
        using Cons.prems(1) by simp

      have x' ≠ x
        using find-remove-2'-set(1,3)[OF ∗] None unfolding find-None-iff
        by blast

      obtain i i' where i < length xs and xs ! i = x
                 and (∀ j < i . find (λy . P (xs ! j) y) ys = None) and i' < length
ys
                  and ys ! i' = y and (∀ j < i' . ¬ P (xs ! i) (ys ! j))
        using Cons.IH[OF ∗] by blast

      have Suc i < length (x'#xs)
        using ‹i < length xs› by auto
      moreover have (x'#xs) ! Suc i = x
        using ‹xs ! i = x› by auto
      moreover have (∀ j < Suc i . find (λy . P ((x'#xs) ! j) y) ys = None)
      proof −
       have ⋀ j . j > 0 ⟹ j < Suc i ⟹ find (λy . P ((x'#xs) ! j) y) ys = None
         using ‹(∀ j < i . find (λy . P (xs ! j) y) ys = None)› by auto
        then show ?thesis using None
         by (metis neq0-conv nth-Cons-0)
      qed
      moreover have (∀ j < i' . ¬ P ((x'#xs) ! Suc i) (ys ! j))
        using ‹(∀ j < i' . ¬ P (xs ! i) (ys ! j))›
```

```
      by simp

    ultimately show ?thesis
      using that ‹i' < length ys› ‹ys ! i' = y› by blast
  next
    case (Some y')
    then have x' = x and y' = y
      using Cons.prems by force+

    have 0 < length (x'#xs) ∧ (x'#xs) ! 0 = x'
         ∧ (∀ j < 0 . find (λy . P ((x'#xs) ! j) y) ys = None)
      by auto
    moreover obtain i' where i' < length ys and ys ! i' = y'
                        and (∀ j < i' . ¬ P ((x'#xs) ! 0) (ys ! j))
      using find-sort-index[OF Some] by auto
    ultimately show ?thesis
      unfolding ‹x' = x› ‹y' = y› by blast
  qed
 qed
 then show ?thesis using that by blast
qed

lemma find-remove-2-index :
 assumes find-remove-2 P xs ys = Some (x,y,xs')
 obtains i i' where i < length xs
                    xs ! i = x
                    ⋀ j . j < i ⟹ find (λy . P (xs ! j) y) ys = None
                    i' < length ys
                    ys ! i' = y
                    ⋀ j . j < i' ⟹ ¬ P (xs ! i) (ys ! j)
 using assms find-remove-2'-index[of P xs ys [] x y xs'] by auto


lemma find-remove-2'-set-rev :
 assumes x ∈ set xs
 and     y ∈ set ys
 and     P x y
shows find-remove-2' P xs ys prev ≠ None
using assms(1) proof(induction xs arbitrary: prev)
 case Nil
 then show ?case by auto
next
 case (Cons x' xs)
 then show ?case proof (cases find (λy . P x' y) ys)
   case None
   then have x ≠ x'
     using assms(2,3) by (metis find-None-iff)
   then have x ∈ set xs
     using Cons.prems by auto
```

```
    then show ?thesis
      using Cons.IH unfolding find-remove-2'.simps None by auto
  next
    case (Some a)
    then show ?thesis by auto
  qed
qed


lemma find-remove-2'-diff-prev-None :
  (find-remove-2' P xs ys prev = None ⟹ find-remove-2' P xs ys prev' = None)
proof (induction xs arbitrary: prev prev')
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  show ?case proof (cases find (λy . P x y) ys)
    case None
   then have find-remove-2' P (x#xs) ys prev = find-remove-2' P xs ys (prev@[x])

        and find-remove-2' P (x#xs) ys prev' = find-remove-2' P xs ys (prev'@[x])
      by auto
    then show ?thesis using Cons by auto
  next
    case (Some a)
    then show ?thesis using Cons by auto
  qed
qed

lemma find-remove-2'-diff-prev-Some :
  (find-remove-2' P xs ys prev = Some (x,y,xs')
    ⟹ ∃ xs'' . find-remove-2' P xs ys prev' = Some (x,y,xs''))
proof (induction xs arbitrary: prev prev')
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  show ?case proof (cases find (λy . P x y) ys)
    case None
   then have find-remove-2' P (x#xs) ys prev = find-remove-2' P xs ys (prev@[x])

        and find-remove-2' P (x#xs) ys prev' = find-remove-2' P xs ys (prev'@[x])
      by auto
    then show ?thesis using Cons by auto
  next
    case (Some a)
    then show ?thesis using Cons by auto
  qed
qed
```

lemma *find-remove-2-None-iff* :
find-remove-2 P xs ys = *None* $\longleftrightarrow$ $\neg$ ($\exists$ *x y* . *x* $\in$ *set xs* $\wedge$ *y* $\in$ *set ys* $\wedge$ *P x y*)
unfolding *find-remove-2.simps*
using *find-remove-2′-set*(*1*−*3*) *find-remove-2′-set-rev*
by (*metis old.prod.exhaust option.exhaust*)

lemma *find-remove-2-set* :
assumes *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′*)
shows *P x y*
and *x* $\in$ *set xs*
and *y* $\in$ *set ys*
and *distinct xs* $\Longrightarrow$ *set xs′* = (*set xs*) − {*x*}
and *distinct xs* $\Longrightarrow$ *distinct xs′*
and *xs′* = (*remove1 x xs*)
using *assms find-remove-2′-set*[*of P xs ys* [] *x y xs′*]
unfolding *find-remove-2.simps* **by** *auto*

lemma *find-remove-2-removeAll* :
assumes *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′*)
and *distinct xs*
shows *xs′* = *removeAll x xs*
using *find-remove-2-set*(*6*)[*OF assms*(*1*)]
by (*simp add*: *assms*(*2*) *distinct-remove1-removeAll*)

lemma *find-remove-2-length* :
assumes *find-remove-2 P xs ys* = *Some* (*x*,*y*,*xs′*)
shows *length xs′* = *length xs* − *1*
using *find-remove-2-set*(*2*,*6*)[*OF assms*]
by (*simp add*: *length-remove1*)

fun *separate-by* :: (*′a* ⇒ *bool*) ⇒ *′a list* ⇒ (*′a list* × *′a list*) **where**
separate-by P xs = (*filter P xs*, *filter* (λ *x* . ¬ *P x*) *xs*)

lemma *separate-by-code*[*code*] :
separate-by P xs = *foldr* (λ*x* (*prevPass*,*prevFail*) . *if P x then* (*x*#*prevPass*,*prevFail*)
else (*prevPass*,*x*#*prevFail*)) *xs* ([],[])
proof (*induction xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)

let *?f* = (λ*x* (*prevPass*,*prevFail*) . *if P x then* (*x*#*prevPass*,*prevFail*) *else* (*prevPass*,*x*#*prevFail*))

have (*filter P xs*, *filter* (λ *x* . ¬ *P x*) *xs*) = *foldr ?f xs* ([],[])

using *Cons.IH* **by** *auto*
moreover have *separate-by P* (*a*#*xs*) = *?f a* (*filter P xs*, *filter* (λ *x* . ¬ *P x*) *xs*)
by *auto*
ultimately show *?case*
by (*cases P a*; *auto*)
qed

fun *find-remove-2-all* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a list* ⇒ '*b list* ⇒ (('*a* × '*b*) *list* × '*a list*) **where**
find-remove-2-all P xs ys =
(*map* (λ *x* . (*x*, *the* (*find* (λ*y* . *P x y*) *ys*))) (*filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* ≠ *None*) *xs*)
,*filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* = *None*) *xs*)

fun *find-remove-2-all'* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a list* ⇒ '*b list* ⇒ (('*a* × '*b*) *list* × '*a list*) **where**
find-remove-2-all' P xs ys =
(*let* (*successesWithWitnesses*,*failures*) = *separate-by* (λ(*x*,*y*) . *y* ≠ *None*) (*map* (λ *x* . (*x*,*find* (λ*y* . *P x y*) *ys*)) *xs*)
in (*map* (λ (*x*,*y*) . (*x*, *the y*)) *successesWithWitnesses*, *map fst failures*))

lemma *find-remove-2-all-code*[*code*] :
find-remove-2-all P xs ys = *find-remove-2-all' P xs ys*
proof −
let *?s1* = *map* (λ *x* . (*x*, *the* (*find* (λ*y* . *P x y*) *ys*))) (*filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* ≠ *None*) *xs*)
let *?f1* = *filter* (λ *x* . *find* (λ*y* . *P x y*) *ys* = *None*) *xs*

let *?s2* = *map* (λ (*x*,*y*) . (*x*, *the y*)) (*filter* (λ(*x*,*y*) . *y* ≠ *None*) (*map* (λ *x* . (*x*,*find* (λ*y* . *P x y*) *ys*)) *xs*))
let *?f2* = *map fst* (*filter* (λ(*x*,*y*) . *y* = *None*) (*map* (λ *x* . (*x*,*find* (λ*y* . *P x y*) *ys*)) *xs*))

have *find-remove-2-all P xs ys* = (*?s1*,*?f1*)
by *simp*
moreover have *find-remove-2-all' P xs ys* = (*?s2*,*?f2*)
proof −
have ∀ *p*. (λ*pa*. ¬ (*case pa of* (*a*::'*a*, *x*::'*b option*) ⇒ *p x*)) = (λ(*a*, *z*). ¬ *p z*)
by *force*
then show *?thesis*
unfolding *find-remove-2-all'.simps Let-def separate-by.simps*
by *force*
qed
moreover have *?s1* = *?s2*
by (*induction xs*; *auto*)
moreover have *?f1* = *?f2*
by (*induction xs*; *auto*)
ultimately show *?thesis*

```
  by simp
qed
```

1.10 Set-Operations on Lists

```
fun pow-list :: 'a list ⇒ 'a list list where
 pow-list [] = [[]] |
 pow-list (x#xs) = (let pxs = pow-list xs in pxs @ map (λ ys . x#ys) pxs)


lemma pow-list-set :
 set (map set (pow-list xs)) = Pow (set xs)
proof (induction xs)
case Nil
 then show ?case by auto
next
 case (Cons x xs)

 moreover have Pow (set (x # xs)) = Pow (set xs) ∪ (image (insert x) (Pow
(set xs)))
  by (simp add: Pow-insert)

 moreover have set (map set (pow-list (x#xs)))
                = set (map set (pow-list xs)) ∪ (image (insert x) (set (map set
(pow-list xs))))
 proof −
  have ⋀ ys . ys ∈ set (map set (pow-list (x#xs)))
          ⟹ ys ∈ set (map set (pow-list xs)) ∪ (image (insert x) (set (map set
(pow-list xs))))
  proof −
   fix ys assume ys ∈ set (map set (pow-list (x#xs)))
   then consider (a) ys ∈ set (map set (pow-list xs)) |
                 (b) ys ∈ set (map set (map ((#) x) (pow-list xs)))
    unfolding pow-list.simps Let-def by auto
   then show ys ∈ set (map set (pow-list xs)) ∪ (image (insert x) (set (map set
(pow-list xs))))
    by (cases; auto)
  qed
  moreover have ⋀ ys . ys ∈ set (map set (pow-list xs))
                    ∪ (image (insert x) (set (map set (pow-list xs))))
                ⟹ ys ∈ set (map set (pow-list (x#xs)))
  proof −
   fix ys assume ys ∈ set (map set (pow-list xs))
                    ∪ (image (insert x) (set (map set (pow-list xs))))
   then consider (a) ys ∈ set (map set (pow-list xs)) |
                 (b) ys ∈ (image (insert x) (set (map set (pow-list xs))))
    by blast
   then show ys ∈ set (map set (pow-list (x#xs)))
    unfolding pow-list.simps Let-def by (cases; auto)
```

```
    qed
    ultimately show ?thesis by blast
  qed

  ultimately show ?case
    by auto
qed
```

1.10.1 Removing Subsets in a List of Sets

```
lemma remove1-length : x ∈ set xs ⟹ length (remove1 x xs) < length xs
  by (induction xs; auto)


function remove-subsets :: 'a set list ⇒ 'a set list where
  remove-subsets [] = [] |
  remove-subsets (x#xs) = (case find-remove (λ y . x ⊂ y) xs of
    Some (y',xs') ⇒ remove-subsets (y'# (filter (λ y . ¬(y ⊆ x)) xs')) |
    None          ⇒ x # (remove-subsets (filter (λ y . ¬(y ⊆ x)) xs)))
  by pat-completeness auto
termination
proof −
  have ⋀x xs. find-remove ((⊂) x) xs = None ⟹ (filter (λy. ¬ y ⊆ x) xs, x #
xs) ∈ measure length
   by (metis dual-order.trans impossible-Cons in-measure length-filter-le not-le-imp-less)
  moreover have (⋀(x :: 'a set) xs x2 xa y. find-remove ((⊂) x) xs = Some x2
⟹ (xa, y) = x2 ⟹ (xa # filter (λy. ¬ y ⊆ x) y, x # xs) ∈ measure length)
  proof −
    fix x :: 'a set
    fix xs y'xs' y' xs'
    assume find-remove ((⊂) x) xs = Some y'xs' and (y', xs') = y'xs'
    then have find-remove ((⊂) x) xs = Some (y',xs')
      by auto

    have length xs' = length xs − 1
      using find-remove-set(2,3)[OF ‹find-remove ((⊂) x) xs = Some (y',xs')›]
      by (simp add: length-remove1)
    then have length (y'#xs') = length xs
      using find-remove-set(2)[OF ‹find-remove ((⊂) x) xs = Some (y',xs')›]
      using remove1-length by fastforce

    have length (filter (λy. ¬ y ⊆ x) xs') ≤ length xs'
      by simp
    then have length (y' # filter (λy. ¬ y ⊆ x) xs') ≤ length xs' + 1
      by simp
    then have length (y' # filter (λy. ¬ y ⊆ x) xs') ≤ length xs
      unfolding ‹length (y'#xs') = length xs›[symmetric] by simp
    then show (y' # filter (λy. ¬ y ⊆ x) xs', x # xs) ∈ measure length
      by auto
```

```
  qed
  ultimately show ?thesis
    by (relation measure length; auto)
qed


lemma remove-subsets-set : set (remove-subsets xss) = {xs . xs ∈ set xss ∧ (∄ xs'
. xs' ∈ set xss ∧ xs ⊂ xs')}
proof (induction length xss arbitrary: xss rule: less-induct)
  case less

  show ?case proof (cases xss)

    case Nil
    then show ?thesis by auto
  next
    case (Cons x xss')

    show ?thesis proof (cases find-remove (λ y . x ⊂ y) xss')
      case None
      then have (∄ xs' . xs' ∈ set xss' ∧ x ⊂ xs')
        using find-remove-None-iff by metis

      have length (filter (λ y . ¬(y ⊆ x)) xss') < length xss
        using Cons
        by (meson dual-order.trans impossible-Cons leI length-filter-le)

      have remove-subsets (x#xss') = x # (remove-subsets (filter (λ y . ¬(y ⊆ x))
xss'))
        using None by auto
      then have set (remove-subsets (x#xss')) = insert x {xs ∈ set (filter (λy. ¬
y ⊆ x) xss'). ∄xs'. xs' ∈ set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'}
        using less[OF ‹length (filter (λ y . ¬(y ⊆ x)) xss') < length xss›]
        by auto
      also have ... = {xs . xs ∈ set (x#xss') ∧ (∄ xs' . xs' ∈ set (x#xss') ∧ xs ⊂
xs')}
      proof −
        have ⋀ xs . xs ∈ insert x {xs ∈ set (filter (λy. ¬ y ⊆ x) xss'). ∄xs'. xs' ∈
set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'}
              ⟹ xs ∈ {xs ∈ set (x # xss'). ∄xs'. xs' ∈ set (x # xss') ∧ xs ⊂ xs'}
        proof −
          fix xs assume xs ∈ insert x {xs ∈ set (filter (λy. ¬ y ⊆ x) xss'). ∄xs'. xs'
∈ set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'}
          then consider xs = x | xs ∈ set (filter (λy. ¬ y ⊆ x) xss') ∧ (∄xs'. xs' ∈
set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs')
            by blast
          then show xs ∈ {xs ∈ set (x # xss'). ∄xs'. xs' ∈ set (x # xss') ∧ xs ⊂
xs'}
            using ‹(∄ xs' . xs' ∈ set xss' ∧ x ⊂ xs')› by (cases; auto)
```

```
      qed
      moreover have ⋀ xs . xs ∈ {xs ∈ set (x # xss'). ∄xs'. xs' ∈ set (x # xss')
∧ xs ⊂ xs'}
                    ⟹ xs ∈ insert x {xs ∈ set (filter (λy. ¬ y ⊆ x) xss'). ∄xs'.
xs' ∈ set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'}
      proof −
        fix xs assume xs ∈ {xs ∈ set (x # xss'). ∄xs'. xs' ∈ set (x # xss') ∧ xs
⊂ xs'}
        then have xs ∈ set (x # xss') and ∄xs'. xs' ∈ set (x # xss') ∧ xs ⊂ xs'
          by blast+
        then consider xs = x | xs ∈ set xss' by auto
        then show xs ∈ insert x {xs ∈ set (filter (λy. ¬ y ⊆ x) xss'). ∄xs'. xs' ∈
set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'}
        proof cases
          case 1
          then show ?thesis by auto
        next
          case 2
          show ?thesis proof (cases xs ⊆ x)
            case True
            then show ?thesis
              using ‹∄xs'. xs' ∈ set (x # xss') ∧ xs ⊂ xs'› by auto
          next
            case False
            then have xs ∈ set (filter (λy. ¬ y ⊆ x) xss')
              using 2 by auto
            moreover have ∄xs'. xs' ∈ set (filter (λy. ¬ y ⊆ x) xss') ∧ xs ⊂ xs'
              using ‹∄xs'. xs' ∈ set (x # xss') ∧ xs ⊂ xs'› by auto
            ultimately show ?thesis by auto
          qed
        qed
      qed
      ultimately show ?thesis
        by (meson subset-antisym subset-eq)
    qed
    finally show ?thesis unfolding Cons[symmetric] by assumption
  next
    case (Some a)
    then obtain y' xs' where ∗: find-remove (λ y . x ⊂ y) xss' = Some (y',xs')
by force


    have length xs' = length xss' − 1
      using find-remove-set(2,3)[OF ∗]
      by (simp add: length-remove1)
    then have length (y'#xs') = length xss'
      using find-remove-set(2)[OF ∗]
      using remove1-length by fastforce
```

have *length* (*filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\leq$ *length xs'*
by *simp*
then have *length* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\leq$ *length xs'* + *1*
by *simp*
then have *length* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\leq$ *length xss'*
unfolding ‹*length* (*y'*#*xs'*) = *length xss'*›[*symmetric*] **by** *simp*
then have *length* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) < *length xss*
unfolding *Cons* **by** *auto*

have *remove-subsets* (*x*#*xss'*) = *remove-subsets* (*y'*# (*filter* (λ *y* . $\neg$(*y* $\subseteq$ *x*)) *xs'*))
using $*$ **by** *auto*
then have *set* (*remove-subsets* (*x*#*xss'*)) = {*xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*). $\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*}
using *less*[*OF* ‹*length* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) < *length xss*›]
by *auto*
also have ... = {*xs* . *xs* $\in$ *set* (*x*#*xss'*) $\wedge$ ($\nexists$ *xs'* . *xs'* $\in$ *set* (*x*#*xss'*) $\wedge$ *xs* $\subset$ *xs'*)}
proof −
have $\bigwedge$ *xs* . *xs* $\in$ {*xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*). $\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*}
$\Longrightarrow$ *xs* $\in$ {*xs* $\in$ *set* (*x* # *xss'*). $\nexists$ *xs'*. *xs'* $\in$ *set* (*x* # *xss'*) $\wedge$ *xs* $\subset$ *xs'*}
proof −
fix *xs* **assume** *xs* $\in$ {*xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*). $\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*}
then have *xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) **and** $\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*
by *blast*+

have *xs* $\in$ *set* (*x* # *xss'*)
using ‹*xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*)› *find-remove-set*(*2*,*3*)[*OF* $*$]
by *auto*
moreover have $\nexists$ *xs'*. *xs'* $\in$ *set* (*x* # *xss'*) $\wedge$ *xs* $\subset$ *xs'*
using ‹$\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*› *find-remove-set*[*OF* $*$]
by (*metis dual-order.strict-trans filter-list-set in-set-remove1 list.set-intros*(*1*) *list.set-intros*(*2*) *psubsetI set-ConsD*)
ultimately show *xs* $\in$ {*xs* $\in$ *set* (*x* # *xss'*). $\nexists$ *xs'*. *xs'* $\in$ *set* (*x* # *xss'*) $\wedge$ *xs* $\subset$ *xs'*}
by *blast*
qed
moreover have $\bigwedge$ *xs* . *xs* $\in$ {*xs* $\in$ *set* (*x* # *xss'*). $\nexists$ *xs'*. *xs'* $\in$ *set* (*x* # *xss'*) $\wedge$ *xs* $\subset$ *xs'*}
$\Longrightarrow$ *xs* $\in$ {*xs* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*). $\nexists$ *xs'a*. *xs'a* $\in$ *set* (*y'* # *filter* ($\lambda y.\ \neg\ y \subseteq x$) *xs'*) $\wedge$ *xs* $\subset$ *xs'a*}
proof −
fix *xs* **assume** *xs* $\in$ {*xs* $\in$ *set* (*x* # *xss'*). $\nexists$ *xs'*. *xs'* $\in$ *set* (*x* # *xss'*) $\wedge$ *xs*

$\subset xs'\}$
then have $xs \in set\ (x\ \#\ xss')$ **and** $\nexists xs'.\ xs' \in set\ (x\ \#\ xss') \wedge xs \subset xs'$
by *blast+*

then have $xs \in set\ (y'\ \#\ filter\ (\lambda y.\ \neg\ y \subseteq x)\ xs')$
using *find-remove-set*[*OF* $*$]
by (*metis filter-list-set in-set-remove1 list.set-intros*(*1*) *list.set-intros*(*2*) *psubsetI set-ConsD*)
moreover have $\nexists xs'a.\ xs'a \in set\ (y'\ \#\ filter\ (\lambda y.\ \neg\ y \subseteq x)\ xs') \wedge xs \subset xs'a$
using ‹$xs \in set\ (x\ \#\ xss')$› ‹$\nexists xs'.\ xs' \in set\ (x\ \#\ xss') \wedge xs \subset xs'$› *find-remove-set*[*OF* $*$]
by (*metis filter-is-subset list.set-intros*(*2*) *notin-set-remove1 set-ConsD subset-iff*)
ultimately show $xs \in \{xs \in set\ (y'\ \#\ filter\ (\lambda y.\ \neg\ y \subseteq x)\ xs').\ \nexists xs'a.\ xs'a \in set\ (y'\ \#\ filter\ (\lambda y.\ \neg\ y \subseteq x)\ xs') \wedge xs \subset xs'a\}$
by *blast*
qed
ultimately show *?thesis* **by** *blast*
qed
finally show *?thesis* **unfolding** *Cons* **by** *assumption*
qed
qed
qed

1.11 Linear Order on Sum

instantiation *sum* :: (*ord*,*ord*) *ord*
begin

fun *less-eq-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow bool$ **where**
less-eq-sum (*Inl a*) (*Inl b*) = $(a \leq b)$ |
less-eq-sum (*Inl a*) (*Inr b*) = *True* |
less-eq-sum (*Inr a*) (*Inl b*) = *False* |
less-eq-sum (*Inr a*) (*Inr b*) = $(a \leq b)$

fun *less-sum* :: $'a + 'b \Rightarrow 'a + 'b \Rightarrow bool$ **where**
less-sum a b = $(a \leq b \wedge a \neq b)$

instance by (*intro-classes*)
end

instantiation *sum* :: (*linorder*,*linorder*) *linorder*
begin

lemma *less-le-not-le-sum* :
fixes $x :: 'a + 'b$
and $y :: 'a + 'b$

shows $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
by (*cases x*; *cases y*; *auto*)

lemma *order-refl-sum* :
fixes $x :: 'a + 'b$
shows $x \leq x$
by (*cases x*; *auto*)

lemma *order-trans-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
fixes $z :: 'a + 'b$
shows $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
by (*cases x*; *cases y*; *cases z*; *auto*)

lemma *antisym-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
shows $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
by (*cases x*; *cases y*; *auto*)

lemma *linear-sum* :
fixes $x :: 'a + 'b$
fixes $y :: 'a + 'b$
shows $x \leq y \vee y \leq x$
by (*cases x*; *cases y*; *auto*)

instance
using *less-le-not-le-sum order-refl-sum order-trans-sum antisym-sum linear-sum*
by (*intro-classes*; *metis+*)
end

1.12 Removing Proper Prefixes

definition *remove-proper-prefixes* :: $'a$ *list set* $\Rightarrow$ $'a$ *list set* **where**
remove-proper-prefixes xs $= \{x\ .\ x \in xs \wedge (\nexists\ x'\ .\ x' \neq [] \wedge x@x' \in xs)\}$

lemma *remove-proper-prefixes-code*[*code*] :
remove-proper-prefixes (*set xs*) = *set* (*filter* (λx . ($\forall$ $y \in$ *set xs* . *is-prefix x y* $\longrightarrow$ $x = y$)) *xs*)
proof −

have $*$: *remove-proper-prefixes* (*set xs*) = *Set.filter* (λ *zs* . $\nexists$ *ys* . *ys* $\neq$ [] $\wedge$ *zs* @ *ys* $\in$ (*set xs*)) (*set xs*)
unfolding *remove-proper-prefixes-def* **by** *force*

have $\bigwedge$ *zs* . ($\nexists$ *ys* . *ys* $\neq$ [] $\wedge$ *zs* @ *ys* $\in$ (*set xs*)) = ($\forall$ *ys* $\in$ *set xs* . *is-prefix zs ys* $\longrightarrow$ *zs* = *ys*)

```
    unfolding is-prefix-prefix by auto

  then show ?thesis
    unfolding * filter-set by auto
qed
```

1.13 Underspecified List Representations of Sets

```
definition as-list-helper :: 'a set ⇒ 'a list where
  as-list-helper X = (SOME xs . set xs = X ∧ distinct xs)

lemma as-list-helper-props :
  assumes finite X
  shows set (as-list-helper X) = X
    and distinct (as-list-helper X)
  using finite-distinct-list[OF assms]
  using someI[of λ xs . set xs = X ∧ distinct xs]
  by (metis as-list-helper-def)+
```

1.14 Assigning indices to elements of a finite set

```
fun assign-indices :: ('a :: linorder) set ⇒ ('a ⇒ nat) where
  assign-indices xs = (λ x . the (find-index ((=)x) (sorted-list-of-set xs)))

lemma assign-indices-bij:
  assumes finite xs
  shows bij-betw (assign-indices xs) xs {..<card xs}
proof −

  have *:set (sorted-list-of-set xs) = xs
    by (simp add: assms)


  have ⋀x y . x∈xs ⟹ y∈xs ⟹ assign-indices xs x = assign-indices xs y ⟹ x
= y
  proof −
    fix x y assume x∈xs and y∈xs and assign-indices xs x = assign-indices xs y

    obtain i where find-index ((=)x) (sorted-list-of-set xs) = Some i
      using find-index-exhaustive[of sorted-list-of-set xs ((=) x)]
      using ‹x∈xs› unfolding *
      by blast
    then have assign-indices xs x = i
      by auto

    obtain j where find-index ((=)y) (sorted-list-of-set xs) = Some j
      using find-index-exhaustive[of sorted-list-of-set xs ((=) y)]
      using ‹y∈xs› unfolding *
      by blast
    then have assign-indices xs y = j
```

```
      by auto
    then have i = j
      using ‹assign-indices xs x = assign-indices xs y› ‹assign-indices xs x = i›
      by auto
    then have find-index ((=)y) (sorted-list-of-set xs) = Some i
      using ‹find-index ((=)y) (sorted-list-of-set xs) = Some j›
      by auto

    show x = y
     using find-index-index(2)[OF ‹find-index ((=)x) (sorted-list-of-set xs) = Some
i›]
     using find-index-index(2)[OF ‹find-index ((=)y) (sorted-list-of-set xs) = Some
i›]
      by auto
  qed
  moreover have (assign-indices xs) ` xs = {..<card xs}
  proof
    show assign-indices xs ` xs ⊆ {..<card xs}
    proof
      fix i assume i ∈ assign-indices xs ` xs
      then obtain x where x ∈ xs and i = assign-indices xs x
        by blast
      moreover obtain j where find-index ((=)x) (sorted-list-of-set xs) = Some j
        using find-index-exhaustive[of sorted-list-of-set xs ((=) x)]
        using ‹x∈xs› unfolding ∗
        by blast
      ultimately have find-index ((=)x) (sorted-list-of-set xs) = Some i
        by auto
      show i ∈ {..<card xs}
         using find-index-index(1)[OF ‹find-index ((=)x) (sorted-list-of-set xs) =
Some i›]
        by auto
    qed
    show {..<card xs} ⊆ assign-indices xs ` xs
    proof
      fix i assume i ∈ {..<card xs}
      then have i < length (sorted-list-of-set xs)
        by auto
      then have sorted-list-of-set xs ! i ∈ xs
        using ∗ nth-mem by blast
     then obtain j where find-index ((=) (sorted-list-of-set xs ! i)) (sorted-list-of-set
xs) = Some j
        using find-index-exhaustive[of sorted-list-of-set xs ((=) (sorted-list-of-set xs
! i))]
        unfolding ∗
        by blast
      have i = j
         using find-index-index(1,2)[OF ‹find-index ((=) (sorted-list-of-set xs ! i))
(sorted-list-of-set xs) = Some j›]
```

using ‹i < *length* (*sorted-list-of-set xs*)› *distinct-sorted-list-of-set nth-eq-iff-index-eq* **by** *blast*
then show $i \in$ *assign-indices xs* ‘ *xs*
using ‹*find-index* ((=) (*sorted-list-of-set xs* ! i)) (*sorted-list-of-set xs*) = *Some j*›
by (*metis* ‹*sorted-list-of-set xs* ! $i \in$ *xs*› *assign-indices.elims image-iff option.sel*)
qed
qed
ultimately show *?thesis*
unfolding *bij-betw-def inj-on-def* **by** *blast*
qed

1.15 Other Lemmata

lemma *foldr-elem-check*:
assumes *list.set xs* $\subseteq$ *A*
shows *foldr* (λ *x y* . *if x* $\notin$ *A then y else f x y*) *xs v* = *foldr f xs v*
using *assms* **by** (*induction xs*; *auto*)

lemma *foldl-elem-check*:
assumes *list.set xs* $\subseteq$ *A*
shows *foldl* (λ *y x* . *if x* $\notin$ *A then y else f y x*) *v xs* = *foldl f v xs*
using *assms* **by** (*induction xs rule*: *rev-induct*; *auto*)

lemma *foldr-length-helper* :
assumes *length xs* = *length ys*
shows *foldr* (λ- *x* . *f x*) *xs b* = *foldr* (λ*a x* . *f x*) *ys b*
using *assms* **by** (*induction xs ys rule*: *list-induct2*; *auto*)

lemma *list-append-subset3* : *set xs1* $\subseteq$ *set ys1* $\Longrightarrow$ *set xs2* $\subseteq$ *set ys2* $\Longrightarrow$ *set xs3* $\subseteq$ *set ys3* $\Longrightarrow$ *set* (*xs1*@*xs2*@*xs3*) $\subseteq$ *set*(*ys1*@*ys2*@*ys3*) **by** *auto*

lemma *subset-filter* : *set xs* $\subseteq$ *set ys* $\Longrightarrow$ *set xs* = *set* (*filter* (λ *x* . *x* $\in$ *set xs*) *ys*)
by *auto*

lemma *map-filter-elem* :
assumes *y* $\in$ *set* (*List.map-filter f xs*)
obtains *x* **where** *x* $\in$ *set xs*
and *f x* = *Some y*
using *assms* **unfolding** *List.map-filter-def*
by *auto*

lemma *filter-length-weakening* :
assumes $\bigwedge$ *q* . *f1 q* $\Longrightarrow$ *f2 q*
shows *length* (*filter f1 p*) $\leq$ *length* (*filter f2 p*)
proof (*induction p*)
case *Nil*
then show *?case* **by** *auto*

```
next
  case (Cons a p)
  then show ?case using assms by (cases f1 a; auto)
qed

lemma max-length-elem :
  fixes xs :: 'a list set
  assumes finite xs
  and     xs ≠ {}
shows ∃ x ∈ xs . ¬(∃ y ∈ xs . length y > length x)
using assms proof (induction xs)
  case empty
  then show ?case by auto
next
  case (insert x F)
  then show ?case proof (cases F = {})
    case True
    then show ?thesis by blast
  next
    case False
    then obtain y where y ∈ F and ¬(∃ y' ∈ F . length y' > length y)
      using insert.IH by blast
   then show ?thesis using dual-order.strict-trans by (cases length x > length y;
auto)
  qed
qed

lemma min-length-elem :
  fixes xs :: 'a list set
  assumes finite xs
  and     xs ≠ {}
shows ∃ x ∈ xs . ¬(∃ y ∈ xs . length y < length x)
using assms proof (induction xs)
  case empty
  then show ?case by auto
next
  case (insert x F)
  then show ?case proof (cases F = {})
    case True
    then show ?thesis by blast
  next
    case False
    then obtain y where y ∈ F and ¬(∃ y' ∈ F . length y' < length y)
      using insert.IH by blast
   then show ?thesis using dual-order.strict-trans by (cases length x < length y;
auto)
  qed
qed
```

lemma *list-property-from-index-property* :
  assumes $\bigwedge$ *i* . *i* < *length xs* $\Longrightarrow$ *P* (*xs* ! *i*)
  shows $\bigwedge$ *x* . *x* $\in$ *set xs* $\Longrightarrow$ *P x*
  by (*metis assms in-set-conv-nth*)

lemma *list-distinct-prefix* :
  assumes $\bigwedge$ *i* . *i* < *length xs* $\Longrightarrow$ *xs* ! *i* $\notin$ *set* (*take i xs*)
  shows *distinct xs*
proof −
  have $\bigwedge$ *j* . *distinct* (*take j xs*)
  proof −
    fix *j*
    show *distinct* (*take j xs*)
    proof (*induction j*)
      case *0*
      then show *?case* **by** *auto*
    next
      case (*Suc j*)
      then show *?case* **proof** (*cases Suc j* $\leq$ *length xs*)
        case *True*
        then have *take* (*Suc j*) *xs* = (*take j xs*) @ [*xs* ! *j*]
          by (*simp add*: *Suc-le-eq take-Suc-conv-app-nth*)
        then show *?thesis* **using** *Suc.IH assms*[*of j*] *True* **by** *auto*
      next
        case *False*
        then have *take* (*Suc j*) *xs* = *take j xs* **by** *auto*
        then show *?thesis* **using** *Suc.IH* **by** *auto*
      qed
    qed
  qed
  then have *distinct* (*take* (*length xs*) *xs*)
    by *blast*
  then show *?thesis* **by** *auto*
qed

lemma *concat-pair-set* :
  set (*concat* (*map* (λ*x*. *map* (*Pair x*) *ys*) *xs*)) = {*xy* . *fst xy* $\in$ *set xs* $\wedge$ *snd xy* $\in$ *set ys*}
  by *auto*

lemma *list-set-sym* :
  set (*x*@*y*) = *set* (*y*@*x*) **by** *auto*

lemma *list-contains-last-take* :
  assumes *x* $\in$ *set xs*
  shows $\exists$ *i* . *0* < *i* $\wedge$ *i* $\leq$ *length xs* $\wedge$ *last* (*take i xs*) = *x*

by (*metis Suc-leI assms hd-drop-conv-nth in-set-conv-nth last-snoc take-hd-drop zero-less-Suc*)

lemma *take-last-index* :
assumes $i < length\ xs$
shows $last\ (take\ (Suc\ i)\ xs) = xs\ !\ i$
by (*simp add*: *assms take-Suc-conv-app-nth*)

lemma *integer-singleton-least* :
assumes $\{x\ .\ P\ x\} = \{a::integer\}$
shows $a = (LEAST\ x\ .\ P\ x)$
by (*metis Collect-empty-eq Least-equality assms insert-not-empty mem-Collect-eq order-refl singletonD*)

lemma *sort-list-split* :
$\forall\ x \in set\ (take\ i\ (sort\ xs))\ .\ \forall\ y \in set\ (drop\ i\ (sort\ xs))\ .\ x \leq y$
using *sorted-append* **by** *fastforce*

lemma *set-map-subset* :
assumes $x \in set\ xs$
and $t \in set\ (map\ f\ [x])$
shows $t \in set\ (map\ f\ xs)$
using *assms* **by** *auto*

lemma *rev-induct2*[*consumes 1*, *case-names Nil snoc*]:
assumes $length\ xs = length\ ys$
and $P\ []\ []$
and $(\bigwedge x\ xs\ y\ ys.\ length\ xs = length\ ys \Longrightarrow P\ xs\ ys \Longrightarrow P\ (xs@[x])\ (ys@[y]))$
shows $P\ xs\ ys$
using *assms* **proof** (*induct xs arbitrary*: *ys rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case* **proof** (*cases ys*)
case *Nil*
then show *?thesis*
using *snoc.prems*(*1*) **by** *auto*
next
case (*Cons a list*)
then show *?thesis*
by (*metis append-butlast-last-id diff-Suc-1 length-append-singleton list.distinct*(*1*) *snoc.hyps snoc.prems*)
qed
qed

lemma *finite-set-min-param-ex* :

assumes *finite XS*
and $\bigwedge x . x \in XS \Longrightarrow \exists\ k . \forall\ k' . k \leq k' \longrightarrow P\ x\ k'$
shows $\exists\ (k{::}nat) . \forall\ x \in XS . P\ x\ k$
proof −
obtain *f* **where** *f-def* : $\bigwedge x . x \in XS \Longrightarrow \forall\ k' . (f\ x) \leq k' \longrightarrow P\ x\ k'$
using *assms*(*2*) **by** *meson*
let *?k* = *Max* (*image f XS*)
have $\forall\ x \in XS . P\ x\ ?k$
using *f-def* **by** (*simp add*: *assms*(*1*))
then show *?thesis* **by** *blast*
qed

fun *list-max* :: *nat list* ⇒ *nat* **where**
list-max [] = *0* |
list-max xs = *Max* (*set xs*)

lemma *list-max-is-max* : $q \in set\ xs \Longrightarrow q \leq list\text{-}max\ xs$
by (*metis List.finite-set Max-ge length-greater-0-conv length-pos-if-in-set list-max.elims*)

lemma *list-prefix-subset* : $\exists\ ys . ts = xs@ys \Longrightarrow set\ xs \subseteq set\ ts$ **by** *auto*
lemma *list-map-set-prop* : $x \in set\ (map\ f\ xs) \Longrightarrow \forall\ y . P\ (f\ y) \Longrightarrow P\ x$ **by** *auto*
lemma *list-concat-non-elem* : $x \notin set\ xs \Longrightarrow x \notin set\ ys \Longrightarrow x \notin set\ (xs@ys)$ **by** *auto*
lemma *list-prefix-elem* : $x \in set\ (xs@ys) \Longrightarrow x \notin set\ ys \Longrightarrow x \in set\ xs$ **by** *auto*
lemma *list-map-source-elem* : $x \in set\ (map\ f\ xs) \Longrightarrow \exists\ x' \in set\ xs . x = f\ x'$ **by** *auto*

lemma *maximal-set-cover* :
fixes *X* :: *'a set set*
assumes *finite X*
and $S \in X$
shows $\exists\ S' \in X . S \subseteq S' \wedge (\forall\ S'' \in X . \neg(S' \subset S''))$
proof (*rule ccontr*)
assume $\neg\ (\exists S' \in X.\ S \subseteq S' \wedge (\forall S'' \in X.\ \neg\ S' \subset S''))$
then have $*$: $\bigwedge T . T \in X \Longrightarrow S \subseteq T \Longrightarrow \exists\ T' \in X . T \subset T'$
by *auto*

have $\bigwedge k . \exists\ ss . (length\ ss = Suc\ k) \wedge (hd\ ss = S) \wedge (\forall\ i < k . ss\ !\ i \subset ss\ !\ (Suc\ i)) \wedge (set\ ss \subseteq X)$
proof −
fix *k* **show** $\exists\ ss . (length\ ss = Suc\ k) \wedge (hd\ ss = S) \wedge (\forall\ i < k . ss\ !\ i \subset ss\ !\ (Suc\ i)) \wedge (set\ ss \subseteq X)$
proof (*induction k*)
case *0*
have $length\ [S] = Suc\ 0 \wedge hd\ [S] = S \wedge (\forall\ i < 0 . [S]\ !\ i \subset [S]\ !\ (Suc\ i)) \wedge (set\ [S] \subseteq X)$ **using** *assms*(*2*) **by** *auto*
then show *?case* **by** *blast*

```
  next
    case (Suc k)
    then obtain ss where length ss = Suc k
                  and hd ss = S
                  and (∀ i<k. ss ! i ⊂ ss ! Suc i)
                  and set ss ⊆ X
      by blast
    then have ss ! k ∈ X
      by auto
    moreover have S ⊆ (ss ! k)
    proof −
      have ⋀ i . i < Suc k ⟹ S ⊆ (ss ! i)
      proof −
        fix i assume i < Suc k
        then show S ⊆ (ss ! i)
        proof (induction i)
          case 0
          then show ?case using ‹hd ss = S› ‹length ss = Suc k›
            by (metis hd-conv-nth list.size(3) nat.distinct(1) order-refl)
        next
          case (Suc i)
          then have S ⊆ ss ! i and i < k by auto
         then have ss ! i ⊂ ss ! Suc i using ‹(∀ i<k. ss ! i ⊂ ss ! Suc i)› by blast
          then show ?case using ‹S ⊆ ss ! i› by auto
        qed
      qed
      then show ?thesis using ‹length ss = Suc k› by auto
    qed
    ultimately obtain T′ where T′ ∈ X and ss ! k ⊂ T′
      using ∗ by meson

    let ?ss = ss@[T′]

    have length ?ss = Suc (Suc k)
      using ‹length ss = Suc k› by auto
    moreover have hd ?ss = S
        using ‹hd ss = S› by (metis ‹length ss = Suc k› hd-append list.size(3)
nat.distinct(1))
    moreover have (∀ i < Suc k. ?ss ! i ⊂ ?ss ! Suc i)
      using ‹(∀ i<k. ss ! i ⊂ ss ! Suc i)› ‹ss ! k ⊂ T′›
        by (metis Suc-lessI ‹length ss = Suc k› diff-Suc-1 less-SucE nth-append
nth-append-length)
    moreover have set ?ss ⊆ X
      using ‹set ss ⊆ X› ‹T′ ∈ X› by auto
    ultimately show ?case by blast
  qed
 qed

 then obtain ss where (length ss = Suc (card X))
```

```
                and (hd ss = S)
                and (∀ i < card X . ss ! i ⊂ ss ! (Suc i))
                and (set ss ⊆ X)
    by blast
  then have (∀ i < length ss − 1 . ss ! i ⊂ ss ! (Suc i))
    by auto

  have **: ⋀ i (ss :: 'a set list) . (∀ i < length ss − 1 . ss ! i ⊂ ss ! (Suc i)) ⟹
i < length ss ⟹ ∀ s ∈ set (take i ss) . s ⊂ ss ! i
  proof −
    fix i
    fix ss :: 'a set list
    assume i < length ss and (∀ i < length ss − 1 . ss ! i ⊂ ss ! (Suc i))
    then show ∀ s ∈ set (take i ss) . s ⊂ ss ! i
    proof (induction i)
      case 0
      then show ?case by auto
    next
      case (Suc i)
      then have ∀ s∈set (take i ss). s ⊂ ss ! i by auto
      then have ∀ s∈set (take i ss). s ⊂ ss ! (Suc i) using Suc.prems
       by (metis One-nat-def Suc-diff-Suc Suc-lessE diff-zero dual-order.strict-trans
nat.inject zero-less-Suc)
      moreover have ss ! i ⊂ ss ! (Suc i) using Suc.prems by auto
      moreover have (take (Suc i) ss) = (take i ss)@[ss ! i] using Suc.prems(1)
        by (simp add: take-Suc-conv-app-nth)
      ultimately show ?case by auto
    qed
  qed

  have distinct ss
    using ‹(∀ i < length ss − 1 . ss ! i ⊂ ss ! (Suc i))›
  proof (induction ss rule: rev-induct)
    case Nil
    then show ?case by auto
  next
    case (snoc a ss)
    from snoc.prems have ∀ i<length ss − 1. ss ! i ⊂ ss ! Suc i
       by (metis Suc-lessD diff-Suc-1 diff-Suc-eq-diff-pred length-append-singleton
nth-append zero-less-diff)
    then have distinct ss
      using snoc.IH by auto
    moreover have a ∉ set ss
      using **[OF snoc.prems, of length (ss @ [a]) − 1] by auto
    ultimately show ?case by auto
  qed

  then have card (set ss) = Suc (card X)
    using ‹(length ss = Suc (card X))› by (simp add: distinct-card)
```

then show *False*
using ‹*set ss* ⊆ *X*› ‹*finite X*› **by** (*metis Suc-n-not-le-n card-mono*)
qed

lemma *map-set* :
assumes $x \in$ *set xs*
shows *f x* $\in$ *set* (*map f xs*) **using** *assms* **by** *auto*

lemma *maximal-distinct-prefix* :
assumes ¬ *distinct xs*
obtains *n* **where** *distinct* (*take* (*Suc n*) *xs*)
and ¬ (*distinct* (*take* (*Suc* (*Suc n*)) *xs*))
using *assms* **proof** (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)

show *?case* **proof** (*cases distinct xs*)
case *True*
then have *distinct* (*take* (*length xs*) (*xs*@[*x*])) **by** *auto*
moreover have¬ (*distinct* (*take* (*Suc* (*length xs*)) (*xs*@[*x*]))) **using** *snoc.prems*(*2*) **by** *auto*
ultimately show *?thesis* **using** *that* **by** (*metis Suc-pred distinct-singleton length-greater-0-conv self-append-conv2 snoc.prems*(*1*) *snoc.prems*(*2*))
next
case *False*

then show *?thesis* **using** *snoc.IH that*
by (*metis Suc-mono butlast-snoc length-append-singleton less-SucI linorder-not-le snoc.prems*(*1*) *take-all take-butlast*)
qed
qed

lemma *distinct-not-in-prefix* :
assumes $\bigwedge$ *i* . ($\bigwedge$ *x* . *x* $\in$ *set* (*take i xs*) $\Longrightarrow$ *xs* ! *i* $\neq$ *x*)
shows *distinct xs*
using *assms list-distinct-prefix* **by** *blast*

lemma *list-index-fun-gt* : $\bigwedge$ *xs* (*f*::'*a* $\Rightarrow$ *nat*) *i j* .
($\bigwedge$ *i* . *Suc i* < *length xs* $\Longrightarrow$ *f* (*xs* ! *i*) > *f* (*xs* ! (*Suc i*)))
$\Longrightarrow$ *j* < *i*
$\Longrightarrow$ *i* < *length xs*
$\Longrightarrow$ *f* (*xs* ! *j*) > *f* (*xs* ! *i*)

```
proof −
  fix xs::'a list
  fix f::'a ⇒ nat
  fix i j
  assume (⋀ i . Suc i < length xs ⟹ f (xs ! i) > f (xs ! (Suc i)))
    and j < i
    and i < length xs
  then show f (xs ! j) > f (xs ! i)
  proof (induction i − j arbitrary: i j)
    case 0
    then show ?case by auto
  next
    case (Suc x)
    then show ?case
    proof −
      have f1: ∀ n. ¬ Suc n < length xs ∨ f (xs ! Suc n) < f (xs ! n)
        using Suc.prems(1) by presburger
      have f2: ∀ n na. ¬ n < na ∨ Suc n ≤ na
        using Suc-leI by satx
      have x = i − Suc j
        by (metis Suc.hyps(2) Suc.prems(2) Suc-diff-Suc nat.simps(1))
      then have ¬ Suc j < i ∨ f (xs ! i) < f (xs ! Suc j)
        using f1 Suc.hyps(1) Suc.prems(3) by blast
      then show ?thesis
      using f2 f1 by (metis Suc.prems(2) Suc.prems(3) leI le-less-trans not-less-iff-gr-or-eq)
    qed
  qed
qed

lemma finite-set-elem-maximal-extension-ex :
  assumes xs ∈ S
  and     finite S
shows ∃ ys . xs@ys ∈ S ∧ ¬ (∃ zs . zs ≠ [] ∧ xs@ys@zs ∈ S)
using ‹finite S› ‹xs ∈ S› proof (induction S arbitrary: xs)
  case empty
  then show ?case by auto
next
  case (insert x S)

  consider (a) ∃ ys . x = xs@ys ∧ ¬ (∃ zs . zs ≠ [] ∧ xs@ys@zs ∈ (insert x S)) |
           (b) ¬(∃ ys . x = xs@ys ∧ ¬ (∃ zs . zs ≠ [] ∧ xs@ys@zs ∈ (insert x S)))
    by blast
  then show ?case proof cases
    case a
    then show ?thesis by auto
  next
    case b
    then show ?thesis proof (cases ∃ vs . vs ≠ [] ∧ xs@vs ∈ S)
      case True
```

```
      then obtain vs where vs ≠ [] and xs@vs ∈ S
        by blast

      have ∃ ys. xs @ (vs @ ys) ∈ S ∧ (∄ zs. zs ≠ [] ∧ xs @ (vs @ ys) @ zs ∈ S)
        using insert.IH[OF ‹xs@vs ∈ S›] by auto
      then have ∃ ys. xs @ (vs @ ys) ∈ S ∧ (∄ zs. zs ≠ [] ∧ xs @ (vs @ ys) @ zs ∈
(insert x S))
        using b
        unfolding append.assoc append-is-Nil-conv append-self-conv insert-iff
        by (metis append.assoc append-Nil2 append-is-Nil-conv same-append-eq)
      then show ?thesis by blast
    next
      case False
      then show ?thesis using insert.prems
        by (metis append-is-Nil-conv append-self-conv insertE same-append-eq)
    qed
  qed
qed


lemma list-index-split-set:
  assumes i < length xs
shows set xs = set ((xs ! i) # ((take i xs) @ (drop (Suc i) xs)))
using assms proof (induction xs arbitrary: i)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  then show ?case proof (cases i)
    case 0
    then show ?thesis by auto
  next
    case (Suc j)
    then have j < length xs using Cons.prems by auto
     then have set xs = set ((xs ! j) # ((take j xs) @ (drop (Suc j) xs))) using
Cons.IH[of j] by blast

    have ∗: take (Suc j) (x#xs) = x#(take j xs) by auto
    have ∗∗: drop (Suc (Suc j)) (x#xs) = (drop (Suc j) xs) by auto
    have ∗∗∗: (x # xs) ! Suc j = xs ! j by auto

    show ?thesis
      using ‹set xs = set ((xs ! j) # ((take j xs) @ (drop (Suc j) xs)))›
      unfolding Suc ∗ ∗∗ ∗∗∗ by auto
  qed
qed


lemma max-by-foldr :
```

assumes $x \in set\ xs$
shows $f\ x < Suc\ (foldr\ (\lambda\ x'\ m\ .\ max\ (f\ x')\ m)\ xs\ 0)$
using *assms* **by** (*induction xs*; *auto*)

lemma *Max-elem* : $finite\ (xs :: 'a\ set) \Longrightarrow xs \neq \{\} \Longrightarrow \exists\ x \in xs\ .\ Max\ (image\ (f :: 'a \Rightarrow nat)\ xs) = f\ x$
by (*metis* (*mono-tags*, *opaque-lifting*) *Max-in empty-is-image finite-imageI imageE*)

lemma *card-union-of-singletons* :
assumes $\bigwedge S\ .\ S \in SS \Longrightarrow (\exists\ t\ .\ S = \{t\})$
shows $card\ (\bigcup\ SS) = card\ SS$
proof −
let $?f = \lambda\ x\ .\ \{x\}$
have $bij\text{-}betw\ ?f\ (\bigcup\ SS)\ SS$
unfolding *bij-betw-def inj-on-def* **using** *assms* **by** *fastforce*
then show *?thesis*
using *bij-betw-same-card* **by** *blast*
qed

lemma *card-union-of-distinct* :
assumes $\bigwedge S1\ S2\ .\ S1 \in SS \Longrightarrow S2 \in SS \Longrightarrow S1 = S2 \vee f\ S1 \cap f\ S2 = \{\}$
and $finite\ SS$
and $\bigwedge S\ .\ S \in SS \Longrightarrow f\ S \neq \{\}$
shows $card\ (image\ f\ SS) = card\ SS$
proof −
from *assms*(*2*) **have** $\forall\ S1 \in SS\ .\ \forall\ S2 \in SS\ .\ S1 = S2 \vee f\ S1 \cap f\ S2 = \{\}$
$\Longrightarrow \forall\ S \in SS\ .\ f\ S \neq \{\} \Longrightarrow ?thesis$
proof (*induction SS*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert x F*)
then have $\neg\ (\exists\ y \in F\ .\ f\ y = f\ x)$
by *auto*
then have $f\ x \notin image\ f\ F$
by *auto*
then have $card\ (image\ f\ (insert\ x\ F)) = Suc\ (card\ (image\ f\ F))$
using *insert* **by** *auto*
moreover have $card\ (f\ `\ F) = card\ F$
using *insert* **by** *auto*
moreover have $card\ (insert\ x\ F) = Suc\ (card\ F)$
using *insert* **by** *auto*
ultimately show *?case*
by *simp*
qed
then show *?thesis*
using *assms* **by** *simp*

qed

lemma *take-le* :
assumes $i \leq$ *length xs*
shows *take i* (*xs@ys*) = *take i xs*
by (*simp add*: *assms less-imp-le-nat*)

lemma *butlast-take-le* :
assumes $i \leq$ *length* (*butlast xs*)
shows *take i* (*butlast xs*) = *take i xs*
using *take-le*[*OF assms*, *of* [*last xs*]]
by (*metis append-butlast-last-id butlast.simps*(*1*))

lemma *distinct-union-union-card* :
assumes *finite xs*
and $\bigwedge$ *x1 x2 y1 y2* . *x1* $\neq$ *x2* $\Longrightarrow$ *x1* $\in$ *xs* $\Longrightarrow$ *x2* $\in$ *xs* $\Longrightarrow$ *y1* $\in$ *f x1* $\Longrightarrow$ *y2* $\in$ *f x2* $\Longrightarrow$ *g y1* $\cap$ *g y2* = {}
and $\bigwedge$ *x1 y1 y2* . *y1* $\in$ *f x1* $\Longrightarrow$ *y2* $\in$ *f x1* $\Longrightarrow$ *y1* $\neq$ *y2* $\Longrightarrow$ *g y1* $\cap$ *g y2* = {}
and $\bigwedge$ *x1* . *finite* (*f x1*)
and $\bigwedge$ *y1* . *finite* (*g y1*)
and $\bigwedge$ *y1* . *g y1* $\subseteq$ *zs*
and *finite zs*
shows ($\sum$ *x* $\in$ *xs* . *card* ($\bigcup$ *y* $\in$ *f x* . *g y*)) $\leq$ *card zs*
proof −
have ($\sum$ *x* $\in$ *xs* . *card* ($\bigcup$ *y* $\in$ *f x* . *g y*)) = *card* ($\bigcup$ *x* $\in$ *xs* . ($\bigcup$ *y* $\in$ *f x* . *g y*))
using *assms*(*1*,*2*) **proof** *induction*
case *empty*
then show *?case* **by** *auto*
next
case (*insert x xs*)
then have ($\bigwedge$*x1 x2*. *x1* $\in$ *xs* $\Longrightarrow$ *x2* $\in$ *xs* $\Longrightarrow$ *x1* $\neq$ *x2* $\Longrightarrow$ $\bigcup$ (*g* ‘ *f x1*) $\cap$ $\bigcup$ (*g* ‘ *f x2*) = {}) **and** *x* $\in$ *insert x xs* **by** *blast*+
then have ($\sum$*x*$\in$*xs*. *card* ($\bigcup$ (*g* ‘ *f x*))) = *card* ($\bigcup$*x*$\in$*xs*. $\bigcup$ (*g* ‘ *f x*)) **using** *insert.IH* **by** *blast*

moreover have ($\sum$*x*$\in$(*insert x xs*). *card* ($\bigcup$ (*g* ‘ *f x*))) = ($\sum$*x*$\in$*xs*. *card* ($\bigcup$ (*g* ‘ *f x*))) + *card* ($\bigcup$ (*g* ‘ *f x*))
using *insert.hyps* **by** *auto*

moreover have *card* ($\bigcup$*x*$\in$(*insert x xs*). $\bigcup$ (*g* ‘ *f x*)) = *card* ($\bigcup$*x*$\in$*xs*. $\bigcup$ (*g* ‘ *f x*)) + *card* ($\bigcup$ (*g* ‘ *f x*))
proof −
have (($\bigcup$*x*$\in$*xs*. $\bigcup$ (*g* ‘ *f x*)) $\cup$ $\bigcup$ (*g* ‘ *f x*)) = ($\bigcup$*x*$\in$(*insert x xs*). $\bigcup$ (*g* ‘ *f x*))
by *blast*

```
    have *: (⋃x∈xs. ⋃ (g ‘ f x)) ∩ (⋃ (g ‘ f x)) = {}
    proof (rule ccontr)
      assume (⋃x∈xs. ⋃ (g ‘ f x)) ∩ ⋃ (g ‘ f x) ≠ {}
       then obtain z where z ∈ ⋃ (g ‘ f x) and z ∈ (⋃x∈xs. ⋃ (g ‘ f x)) by
blast
      then obtain x′ where x′ ∈ xs and z ∈ ⋃ (g ‘ f x′) by blast
      then have x′ ≠ x and x′ ∈ insert x xs using insert.hyps by blast+

      have ⋃ (g ‘ f x′) ∩ ⋃ (g ‘ f x) = {}
        using insert.prems[OF ‹x′ ≠ x› ‹x′ ∈ insert x xs› ‹x ∈ insert x xs› ]
        by blast
      then show False
        using ‹z ∈ ⋃ (g ‘ f x′)› ‹z ∈ ⋃ (g ‘ f x)› by blast
    qed
    have **: finite (⋃ (g ‘ f x))
      using assms(4) assms(5) by blast
    have ***: finite (⋃x∈xs. ⋃ (g ‘ f x))
      by (simp add: assms(4) assms(5) insert.hyps(1))

    have card ((⋃x∈xs. ⋃ (g ‘ f x)) ∪ ⋃ (g ‘ f x)) = card (⋃x∈xs. ⋃ (g ‘ f x))
+ card (⋃ (g ‘ f x))
      using card-Un-disjoint[OF *** ** *] by simp


    then show ?thesis
      unfolding ‹((⋃x∈xs. ⋃ (g ‘ f x)) ∪ ⋃ (g ‘ f x)) = (⋃x∈(insert x xs). ⋃
(g ‘ f x))› by assumption
  qed

  ultimately show ?case by linarith
 qed

 moreover have card (⋃ x ∈ xs . (⋃ y ∈ f x . g y)) ≤ card zs
 proof −
  have (⋃ x ∈ xs . (⋃ y ∈ f x . g y)) ⊆ zs
    using assms(6) by (simp add: UN-least)
  moreover have finite (⋃ x ∈ xs . (⋃ y ∈ f x . g y))
    by (simp add: assms(1) assms(4) assms(5))
  ultimately show ?thesis
    using assms(7)
    by (simp add: card-mono)
 qed

 ultimately show ?thesis
  by linarith
qed


lemma set-concat-elem :
```

```
  assumes x ∈ set (concat xss)
  obtains xs where xs ∈ set xss and x ∈ set xs
  using assms by auto

lemma set-map-elem :
  assumes y ∈ set (map f xs)
  obtains x where y = f x and x ∈ set xs
  using assms by auto

lemma finite-snd-helper:
  assumes finite xs
  shows finite {z. ((q, p), z) ∈ xs}
proof −
  have {z. ((q, p), z) ∈ xs} ⊆ (λ((a,b),c) . c) ‘ xs
  proof
    fix x assume x ∈ {z. ((q, p), z) ∈ xs}
    then have ((q,p),x) ∈ xs by auto
    then show x ∈ (λ((a,b),c) . c) ‘ xs by force
  qed
  then show ?thesis using assms
    using finite-surj by blast
qed

lemma fold-dual : fold (λ x (a1,a2) . (g1 x a1, g2 x a2)) xs (a1,a2) = (fold g1
xs a1, fold g2 xs a2)
  by (induction xs arbitrary: a1 a2; auto)

lemma recursion-renaming-helper :
  assumes f1 = (λx . if P x then x else f1 (Suc x))
  and     f2 = (λx . if P x then x else f2 (Suc x))
  and     ⋀ x . x ≥ k ⟹ P x
shows f1 = f2
proof
  fix x
  show f1 x = f2 x
  proof (induction k − x arbitrary: x)
    case 0
    then have x ≥ k
      by auto
    then show ?case
      using assms(3) by (simp add: assms(1,2))
  next
    case (Suc k′)
    show ?case proof (cases P x)
      case True
      then show ?thesis by (simp add: assms(1,2))
    next
      case False
      moreover have f1 (Suc x) = f2 (Suc x)
```

```
      using Suc.hyps(1)[of Suc x] Suc.hyps(2) by auto
    ultimately show ?thesis by (simp add: assms(1,2))
  qed
 qed
qed


lemma minimal-fixpoint-helper :
 assumes f = (λx . if P x then x else f (Suc x))
 and     ⋀ x . x ≥ k ⟹ P x
shows P (f x)
 and ⋀ x' . x' ≥ x ⟹ x' < f x ⟹ ¬ P x'
proof −
 have P (f x) ∧ (∀ x' . x' ≥ x ⟶ x' < f x ⟶ ¬ P x')
 proof (induction k−x arbitrary: x)
  case 0
  then have P x
    using assms(2) by auto
  moreover have f x = x
    using calculation by (simp add: assms(1))
  ultimately show ?case
    using assms(1) by auto
 next
  case (Suc k')
  then have P (f (Suc x)) and ⋀ x' . x' ≥ Suc x ⟹ x' < f (Suc x) ⟹ ¬ P x'
    by force+

  show ?case proof (cases P x)
    case True
    then have f x = x
      by (simp add: assms(1))
    show ?thesis
      using True unfolding ‹f x = x› by auto
  next
    case False
    then have f x = f (Suc x)
      by (simp add: assms(1))
    then have P (f x)
      using ‹P (f (Suc x))› by simp
    moreover have (∀ x'≥x. x' < f x ⟶ ¬ P x')
      using ‹⋀ x' . x' ≥ Suc x ⟹ x' < f (Suc x) ⟹ ¬ P x'› False ‹f x = f
(Suc x)›
      by (metis Suc-leI le-neq-implies-less)
    ultimately show ?thesis
      by blast
  qed
 qed
 then show P (f x) and ⋀ x' . x' ≥ x ⟹ x' < f x ⟹ ¬ P x'
  by blast+
```

```
qed

lemma map-set-index-helper :
  assumes xs ≠ []
  shows set (map f xs) = (λi . f (xs ! i)) ‘ {.. (length xs − 1)}
using assms proof (induction xs rule: rev-induct)
  case Nil
  then show ?case by auto
next
  case (snoc x xs)
  show ?case proof (cases xs = [])
    case True
    show ?thesis
      using snoc.prems unfolding True by auto
  next
    case False

    have {..length (xs@[x]) − 1} = insert (length (xs@[x]) − 1) {..length xs − 1}
      by force
    moreover have ((λi. f ((xs@[x]) ! i)) (length (xs@[x]) − 1)) = f x
      by auto
    moreover have ((λi. f ((xs@[x]) ! i)) ‘ {..length xs − 1}) = ((λi. f (xs ! i)) ‘
{..length xs − 1})
    proof −
      have ⋀ i . i < length xs ⟹ f ((xs@[x]) ! i) = f (xs ! i)
        by (simp add: nth-append)
      moreover have ⋀ i . i ∈ {..length xs − 1} ⟹ i < length xs
        using False
        by (metis Suc-pred′ atMost-iff length-greater-0-conv less-Suc-eq-le)
      ultimately show ?thesis
        by (meson image-cong)
    qed
    ultimately have (λi. f ((xs@[x]) ! i)) ‘ {..length (xs@[x]) − 1} = insert (f x)
((λi. f (xs ! i)) ‘ {..length xs − 1})
      by auto
    moreover have set (map f (xs@[x])) = insert (f x) (set (map f xs))
      by auto
    moreover have set (map f xs) = (λi. f (xs ! i)) ‘ {..length xs − 1}
      using snoc.IH False by auto
    ultimately show ?thesis
      by force
  qed
qed

lemma partition-helper :
  assumes finite X
  and     X ≠ {}
  and     ⋀ x . x ∈ X ⟹ p x ⊆ X
  and     ⋀ x . x ∈ X ⟹ p x ≠ {}
```

and $\bigwedge x\ y\ .\ x \in X \Longrightarrow y \in X \Longrightarrow p\ x = p\ y \vee p\ x \cap p\ y = \{\}$
and $(\bigcup\ x \in X\ .\ p\ x) = X$
obtains l::*nat* **and** p' **where**
p' ‘ $\{..l\} = p$ ‘ X
$\bigwedge i\ j\ .\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow p'\ i \cap p'\ j = \{\}$
card $(p$ ‘ $X) = Suc\ l$
proof −
let *?P* = *as-list-helper* ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*)

have *?P* ≠ []
using *assms*(*1*) *assms*(*2*)
by (*metis as-list-helper-props*(*1*) *finite-imageI image-is-empty set-empty*)

define *l* **where** *l*: *l* = *length ?P* − *1*
define *p′* **where** *p′*: *p′* = (λ *x* . *set* (*?P* ! *x*))

have *finite* ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*)
using *assms*(*1*)
by *simp*

have *set* ‘ ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*) = *p* ‘ *X*
proof −
have *set* ‘ ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*) = ((λ*x*. *set* (*as-list-helper* (*p x*))) ‘ *X*)
by *auto*
also have . . . = *p* ‘ *X*
by (*metis* (*no-types*, *lifting*) *as-list-helper-props*(*1*) *assms*(*1*) *assms*(*6*) *finite-UN image-cong*)
finally show *?thesis* **.**
qed
moreover have *set ?P* = (λ*x*. *as-list-helper* (*p x*)) ‘ *X*
by (*simp add*: *as-list-helper-props*(*1*) *assms*(*1*))
ultimately have *set* ‘ (*set ?P*) = *p* ‘ *X*
by *auto*
moreover have (*p′* ‘ $\{..l\}$) = *set* (*map set ?P*)
using *map-set-index-helper*[*OF* ‹*?P* ≠ []›]
proof −
have (λ*n*. *set* (*as-list-helper* ((λ*n*. *as-list-helper* (*p n*)) ‘ *X*) ! *n*)) ‘ $\{..l\}$ = *p′* ‘ $\{..l\}$
using *p′* **by** *force*
then show *?thesis*
by (*metis* ‹⋀*f*. *set* (*map f* (*as-list-helper* ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*))) = (λ*i*. *f* (*as-list-helper* ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*) ! *i*)) ‘ {..*length* (*as-list-helper* ((λ*x*. *as-list-helper* (*p x*)) ‘ *X*)) − *1*}› *l*)
qed
ultimately have *p1*: *p′* ‘ $\{..l\}$ = *p* ‘ *X*
by (*metis list.set-map*)

moreover have *p2*: $\bigwedge i\ j\ .\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow p'\ i \cap p'\ j = \{\}$
proof −
fix *i j* **assume** $i \leq l\ j \leq l\ i \neq j$
moreover define *PX* **where** *PX*: $PX = ((\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X)$
ultimately have $i < \textit{length}\ (\textit{as-list-helper}\ PX)$ **and** $j < \textit{length}\ (\textit{as-list-helper}\ PX)$
unfolding *l* **by** *auto*
then have $?P\ !\ i \neq ?P\ !\ j$
using ‹$i \neq j$› **unfolding** *PX*
using *as-list-helper-props*(*2*)[*OF* ‹*finite* $((\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X)$›]
using *nth-eq-iff-index-eq* **by** *blast*
moreover obtain *xi* **where** $xi \in X$ **and** $*$:$?P\ !\ i = \textit{as-list-helper}\ (p\ xi)$
by (*metis* (*no-types*, *lifting*) *PX* ‹$i < \textit{length}\ (\textit{as-list-helper}\ PX)$› ‹*set* (*as-list-helper* $((\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X)) = (\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X$› *image-iff nth-mem*)
moreover obtain *xj* **where** $xj \in X$ **and** $**$:$?P\ !\ j = \textit{as-list-helper}\ (p\ xj)$
by (*metis* (*no-types*, *lifting*) *PX* ‹$j < \textit{length}\ (\textit{as-list-helper}\ PX)$› ‹*set* (*as-list-helper* $((\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X)) = (\lambda x.\ \textit{as-list-helper}\ (p\ x))\ `\ X$› *image-iff nth-mem*)
ultimately have $p\ xi \neq p\ xj$
by *metis*
then have $p'\ i \neq p'\ j$
unfolding p'
by (*metis* $*$ $**$ ‹$xi \in X$› ‹$xj \in X$› *as-list-helper-props*(*1*) *assms*(*1*) *assms*(*3*) *infinite-super*)
then show $p'\ i \cap p'\ j = \{\}$
using *assms*(*5*)
by (*metis* $*$ $**$ ‹$xi \in X$› ‹$xj \in X$› *as-list-helper-props*(*1*) *assms*(*1*) *assms*(*3*) *finite-subset* p')
qed
moreover have $\textit{card}\ (p\ `\ X) = \textit{Suc}\ l$
proof −
have $\bigwedge i\ .\ i \in \{..l\} \Longrightarrow p'\ i \neq \{\}$
using *p1 assms* (*4*)
by (*metis imageE imageI*)
then show *?thesis*
unfolding *p1*[*symmetric*]
by (*metis atMost-iff card-atMost card-union-of-distinct finite-atMost p2*)
qed
ultimately show *?thesis*
using *that*[*of* $p'\ l$]
by *blast*
qed

lemma *take-diff* :
assumes $i \leq \textit{length}\ xs$
and $j \leq \textit{length}\ xs$
and $i \neq j$
shows $\textit{take}\ i\ xs \neq \textit{take}\ j\ xs$
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *length-take min.commute min.order-iff*)

lemma *image-inj-card-helper* :
  assumes *finite X*
  and $\bigwedge a\ b\ .\ a \in X \Longrightarrow b \in X \Longrightarrow a \neq b \Longrightarrow f\ a \neq f\ b$
shows *card* $(f\ `\ X) = card\ X$
using *assms* **proof** (*induction X*)
  case *empty*
  then show *?case* **by** *auto*
next
  case (*insert x X*)
  then have $f\ x \notin f\ `\ X$
    by (*metis imageE insertCI*)
  then have *card* $(f\ `\ (insert\ x\ X)) = Suc\ (card\ X)$
    using *insert.IH insert.hyps*(*1*) *insert.prems* **by** *auto*
  moreover have *card* $(insert\ x\ X) = Suc\ (card\ X)$
    by (*meson card-insert-if insert.hyps*(*1*) *insert.hyps*(*2*))
  ultimately show *?case*
    by *auto*
qed

lemma *sum-image-inj-card-helper* :
  fixes $l :: nat$
  assumes $\bigwedge i\ .\ i \leq l \Longrightarrow finite\ (I\ i)$
  and $\bigwedge i\ j\ .\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow I\ i \cap I\ j = \{\}$
shows $(\sum\ i \in \{..l\}\ .\ (card\ (I\ i))) = card\ (\bigcup\ i \in \{..l\}\ .\ I\ i)$
  using *assms* **proof** (*induction l*)
case *0*
  then show *?case* **by** *auto*
next
  case (*Suc l*)
  then have $(\sum i{\leq}l.\ card\ (I\ i)) = card\ (\bigcup\ (I\ `\ \{..l\}))$
    using *le-Suc-eq* **by** *presburger*
  moreover have $(\sum i{\leq}Suc\ l.\ card\ (I\ i)) = card\ (I\ (Suc\ l)) + (\sum i{\leq}l.\ card\ (I\ i))$
    by *auto*
  moreover have $card\ (\bigcup\ (I\ `\ \{..Suc\ l\})) = card\ (I\ (Suc\ l)) + card\ (\bigcup\ (I\ `\ \{..l\}))$
    using *Suc.prems*(*2*)
    by (*simp add*: *Suc.prems*(*1*) *card-UN-disjoint*)
  ultimately show *?case*
    by *auto*
qed

lemma *Min-elem* : $finite\ (xs :: 'a\ set) \Longrightarrow xs \neq \{\} \Longrightarrow \exists\ x \in xs\ .\ Min\ (image\ (f :: 'a \Rightarrow nat)\ xs) = f\ x$
  by (*metis* (*mono-tags*, *opaque-lifting*) *Min-in empty-is-image finite-imageI imageE*)

lemma *finite-subset-mapping-limit* :
  fixes $f :: nat \Rightarrow 'a\ set$
  assumes $finite\ (f\ 0)$

```
  and     ⋀ i j . i ≤ j ⟹ f j ⊆ f i
obtains k where ⋀ k' . k ≤ k' ⟹ f k' = f k
proof (cases f 0 = {})
  case True
  then show ?thesis
    using assms(2) that by fastforce
next
  case False
  then have (f ` UNIV) ≠ {}
    by auto

  have ∃ k . ∀ k' . k ≤ k' ⟶ f k' = f k
  proof (rule ccontr)
    assume ∄k. ∀k'≥k. f k' = f k
    then have ⋀ k . ∃ k' . k' > k ∧ f k' ⊂ f k
      using assms(2)
      by (metis dual-order.order-iff-strict)

    have f ` UNIV ⊆ Pow (f 0)
      using assms(2)
      by (simp add: image-subset-iff)
    moreover have finite (Pow (f 0))
      using assms(1) by simp
    ultimately have finite (f ` UNIV)
      using finite-subset by auto


    obtain x where x ∈ f ` UNIV and ⋀ x' . x' ∈ f ` UNIV ⟹ card x ≤ card x'
      using Min-elem[OF ‹finite (f ` UNIV)› ‹(f ` UNIV) ≠ {}›, of card]
     by (metis (mono-tags, lifting) Min.boundedE ‹finite (range f)› ‹range f ≠ {}›
ball-imageD finite-imageI image-is-empty order-refl)

    obtain k where f k = x
      using ‹x ∈ f ` UNIV› by blast
    then obtain k' where f k' ⊂ x
      using ‹⋀ k . ∃ k' . k' > k ∧ f k' ⊂ f k› by blast
    moreover have ⋀ k . finite (f k)
      by (meson assms(1) assms(2) infinite-super le0)
    ultimately have card (f k') < card x
      using ‹f k = x› by (metis psubset-card-mono)
    then show False
      using ‹⋀ x' . x' ∈ f ` UNIV ⟹ card x ≤ card x'›
      by (simp add: less-le-not-le)
  qed
  then show ?thesis
    using that by blast
qed

lemma finite-card-less-witnesses :
```

```
  assumes finite A
  and     card (g ‘ A) < card (f ‘ A)
obtains a b where a ∈ A and b ∈ A and f a ≠ f b and g a = g b
proof −
  have ∃ a b . a ∈ A ∧ b ∈ A ∧ f a ≠ f b ∧ g a = g b
    using assms proof (induction A)
    case empty
    then show ?case by auto
  next
    case (insert x F)
    show ?case proof (cases card (g ‘ F) < card (f ‘ F))
      case True
      then show ?thesis using insert.IH by blast
    next
      case False

      have finite (g ‘ F) and finite (f ‘ F)
        using insert.hyps(1) by auto
      have card (g ‘ insert x F) = (if g x ∈ g ‘ F then card (g ‘ F) else Suc (card
(g ‘ F)))
        using card-insert-if[OF ‹finite (g ‘ F)›]
        by simp
      moreover have card (f ‘ insert x F) = (if f x ∈ f ‘ F then card (f ‘ F) else
Suc (card (f ‘ F)))
        using card-insert-if[OF ‹finite (f ‘ F)›]
        by simp
      ultimately have card (g ‘ F) = card (f ‘ F)
        using insert.prems False
        by (metis Suc-lessD not-less-less-Suc-eq)
      then have card (g ‘ insert x F) = card (g ‘ F)
        using insert.prems
        by (metis Suc-lessD ‹card (f ‘ insert x F) = (if f x ∈ f ‘ F then card (f ‘ F)
else Suc (card (f ‘ F)))› ‹card (g ‘ insert x F) = (if g x ∈ g ‘ F then card (g ‘ F)
else Suc (card (g ‘ F)))› less-not-refl3)

      then obtain y where y ∈ F and g x = g y
        using ‹finite F›
        by (metis ‹card (g ‘ insert x F) = (if g x ∈ g ‘ F then card (g ‘ F) else Suc
(card (g ‘ F)))› imageE lessI less-irrefl-nat)

      have card (f ‘ insert x F) > card (f ‘ F)
        using ‹card (g ‘ F) = card (f ‘ F)› ‹card (g ‘ insert x F) = card (g ‘ F)›
insert.prems by presburger
      then have f x ≠ f y
        using ‹y ∈ F›
        by (metis ‹card (f ‘ insert x F) = (if f x ∈ f ‘ F then card (f ‘ F) else Suc
(card (f ‘ F)))› image-eqI less-irrefl-nat)

      then show ?thesis
```

```
      using ‹y ∈ F› ‹g x = g y› by blast
    qed
  qed
  then show ?thesis
    using that by blast
qed

lemma monotone-function-with-limit-witness-helper :
  fixes f :: nat ⇒ nat
  assumes ⋀ i j . i ≤ j ⟹ f i ≤ f j
  and     ⋀ i j m . i < j ⟹ f i = f j ⟹ j ≤ m ⟹ f i = f m
  and     ⋀ i . f i ≤ k
obtains x where f (Suc x) = f x and x ≤ k − f 0
proof −
  have ⋀ i . f (Suc i) ≥ f 0 + Suc i ∨ (f (Suc i) < f 0 + Suc i ∧ f i = f (Suc i))
  proof −
    fix i
    show f (Suc i) ≥ f 0 + Suc i ∨ (f (Suc i) < f 0 + Suc i ∧ f i = f (Suc i))
    proof (induction i)
      case 0
      then show ?case using assms(1)
        by (metis add.commute add.left-neutral add-Suc-shift le0 le-antisym lessI
not-less-eq-eq)
    next
      case (Suc i)
      then show ?case
      proof −
        have ∀ n. n ≤ Suc n
          by simp
        then show ?thesis
            by (metis Suc add-Suc-right assms(1) assms(2) le-antisym not-less
not-less-eq-eq order-trans-rules(23))
      qed
    qed
  qed

  have ∃ x . f (Suc x) = f x ∧ x ≤ k − f 0
  using assms(3) proof (induction k)
    case 0
    then show ?case by auto
  next
    case (Suc k)

    consider f 0 + Suc k ≤ f (Suc k) | f (Suc k) < f 0 + Suc k ∧ f k = f (Suc k)
      using ‹⋀ i . f (Suc i) ≥ f 0 + Suc i ∨ (f (Suc i) < f 0 + Suc i ∧ f i = f
(Suc i))›[of k]
      by blast

    then show ?case proof cases
```

```
      case 1
      then have f (Suc (Suc k)) = f (Suc k)
        using Suc.prems[of Suc (Suc k)] assms(1)[of Suc k Suc (Suc k)]
        by auto
      then show ?thesis
          by (metis 1 Suc.prems add.commute add-diff-cancel-left' add-increasing2
le-add2 le-add-same-cancel2 le-antisym)
    next
      case 2
      then have f (Suc k) < f 0 + Suc k and f k = f (Suc k)
        by auto
      then show ?thesis
        by (metis Suc.prems ‹⋀i. f 0 + Suc i ≤ f (Suc i) ∨ f (Suc i) < f 0 + Suc
i ∧ f i = f (Suc i)› add-Suc-right add-diff-cancel-left' le0 le-Suc-ex nat-arith.rule0
not-less-eq-eq)
    qed
  qed

  then show ?thesis
    using that by blast
qed

lemma different-lists-shared-prefix :
  assumes xs ≠ xs'
obtains i where take i xs = take i xs'
           and take (Suc i) xs ≠ take (Suc i) xs'
proof −
  have ∃ i . take i xs = take i xs' ∧ take (Suc i) xs ≠ take (Suc i) xs'
  proof (rule ccontr)
    assume ∄ i. take i xs = take i xs' ∧ take (Suc i) xs ≠ take (Suc i) xs'

    have ⋀ i . take i xs = take i xs'
    proof −
      fix i show take i xs = take i xs'
      proof (induction i)
        case 0
        then show ?case by auto
      next
        case (Suc i)
        then show ?case
          using ‹∄ i. take i xs = take i xs' ∧ take (Suc i) xs ≠ take (Suc i) xs'› by
blast
      qed
    qed

    have xs = xs'
      by (simp add: ‹⋀i. take i xs = take i xs'› take-equalityI)
    then show False
      using assms by simp
```

```
  qed
  then show ?thesis using that by blast
qed

lemma foldr-funion-fempty : foldr (|∪|) xs fempty = ffUnion (fset-of-list xs)
  by (induction xs; auto)

lemma foldr-funion-fsingleton : foldr (|∪|) xs x = ffUnion (fset-of-list (x#xs))
  by (induction xs; auto)

lemma foldl-funion-fempty : foldl (|∪|) fempty xs = ffUnion (fset-of-list xs)
  by (induction xs rule: rev-induct; auto)

lemma foldl-funion-fsingleton : foldl (|∪|) x xs = ffUnion (fset-of-list (x#xs))
  by (induction xs rule: rev-induct; auto)

lemma ffUnion-fmember-ob : x |∈| ffUnion XS ⟹ ∃ X . X |∈| XS ∧ x |∈| X
  by (induction XS; auto)


lemma filter-not-all-length :
  filter P xs ≠ [] ⟹ length (filter (λ x . ¬ P x) xs) < length xs
  by (metis filter-False length-filter-less)

lemma foldr-funion-fmember : B |⊆| (foldr (|∪|) A B)
  by (induction A; auto)

lemma prefix-free-set-maximal-list-ob :
  assumes finite xs
  and     x ∈ xs
obtains x' where x@x' ∈ xs and ∄ y' . y' ≠ [] ∧ (x@x')@y' ∈ xs
proof −

  let ?xs = {x' . x@x' ∈ xs}
  let ?x' = arg-max length (λ x . x ∈ ?xs)


  have ⋀y. y ∈ ?xs ⟹ length y < Suc (Max (length ‘ xs))
  proof −
    fix y assume y ∈ ?xs
    then have x@y ∈ xs
      by blast
    moreover have ⋀y. y ∈ xs ⟹ length y < Suc (Max (length ‘ xs))
      using assms(1)
      by (simp add: le-imp-less-Suc)
    ultimately show length y < Suc (Max (length ‘ xs))
      by fastforce
  qed
  moreover have [] ∈ ?xs
```

```
    using assms(2) by auto
  ultimately have ?x' ∈ ?xs and (∀ x' . x' ∈ ?xs ⟶ length x' ≤ length ?x')
    using arg-max-nat-lemma[of (λ x . x ∈ ?xs) [] length Suc (Max (length ‘ xs))]
    by blast+

  have ∄ y' . y' ≠ [] ∧ (x@?x')@y' ∈ xs
  proof
    assume ∃ y' . y' ≠ [] ∧ (x@?x')@y' ∈ xs
    then obtain y' where y' ≠ [] ∧ x@(?x'@y')∈ xs
      by auto
    then have (?x'@y') ∈ ?xs and length (?x'@y') > length ?x'
      by auto
    then show False
      using ‹(∀ x' . x' ∈ ?xs ⟶ length x' ≤ length ?x')›
      by auto
  qed

  then show ?thesis
    using that using ‹?x' ∈ ?xs› by blast
qed

lemma map-upds-map-set-left :
  assumes [map f xs [↦] xs] q = Some x
  shows x ∈ set xs and q = f x
proof −
  have x ∈ set xs ∧ q = f x
  using assms proof (induction xs rule: rev-induct)
    case Nil
    then show ?case by auto
  next
    case (snoc x' xs)
    show ?case proof (cases f x' = q)
      case True
      then have x = x'
        using snoc.prems by (induction xs; auto)
      then show ?thesis
        using True by auto
    next
      case False
      then have [map f (xs @ [x']) [↦] xs @ [x']] q = [map f (xs) [↦] xs] q
        by (induction xs; auto)
      then show ?thesis
        using snoc by auto
    qed
  qed
  then show x ∈ set xs and q = f x
    by auto
qed
```

lemma *map-upds-map-set-right* :
  assumes $x \in$ *set xs*
  shows [*xs* [$\mapsto$] *map f xs*] *x* = *Some* (*f x*)
using *assms* **proof** (*induction xs rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc x′ xs*)
  show *?case* **proof** (*cases x=x′*)
    case *True*
    then show *?thesis*
      by (*induction xs*; *auto*)
  next
    case *False*
    then have [*xs* @ [*x′*] [$\mapsto$] *map f* (*xs* @ [*x′*])] *x* = [*xs* [$\mapsto$] *map f xs*] *x*
      by (*induction xs*; *auto*)
    then show *?thesis*
      using *snoc False* **by** *auto*
  qed
qed

lemma *map-upds-overwrite* :
  assumes $x \in$ *set xs*
      and *length xs* = *length ys*
  shows (*m*(*xs*[$\mapsto$]*ys*)) *x* = [*xs*[$\mapsto$]*ys*] *x*
  using *assms*(*2*,*1*) **by** (*induction xs ys rule*: *rev-induct2*; *auto*)

lemma *ran-dom-the-eq* : (λ*k* . *the* (*m k*)) ‘ *dom m* = *ran m*
  unfolding *ran-def dom-def* **by** *force*

lemma *map-pair-fst* :
  map fst (*map* (λ*x* . (*x*,*f x*)) *xs*) = *xs*
  by (*induction xs*; *auto*)

lemma *map-of-map-pair-entry*: *map-of* (*map* (λ*k*. (*k*, *f k*)) *xs*) *x* = (*if x* $\in$ *list.set xs then Some* (*f x*) *else None*)
  by (*induction xs*; *auto*)

lemma *map-filter-alt-def* :
  List.map-filter f1′ xs = *map the* (*filter* (λ*x* . *x* $\neq$ *None*) (*map f1′ xs*))
  by (*induction xs*; *unfold map-filter-simps*; *auto*)

lemma *map-filter-Nil* :
  List.map-filter f1′ xs = [] $\longleftrightarrow$ ($\forall$ *x* $\in$ *list.set xs* . *f1′ x* = *None*)
  unfolding *map-filter-alt-def* **by** (*induction xs*; *auto*)

lemma *sorted-list-of-set-set*: *set* ((*sorted-list-of-set* ∘ *set*) *xs*) = *set xs*
  by *auto*

fun *mapping-of* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *mapping* **where**
  mapping-of kvs = *foldl* (λ*m kv* . *Mapping.update* (*fst kv*) (*snd kv*) *m*) *Mapping.empty kvs*

lemma *mapping-of-map-of* :
  assumes *distinct* (*map fst kvs*)
  shows *Mapping.lookup* (*mapping-of kvs*) = *map-of kvs*
proof
  show ⋀*x*. *Mapping.lookup* (*mapping-of kvs*) *x* = *map-of kvs x*
    using *assms*
  proof (*induction kvs rule*: *rev-induct*)
    case *Nil*
    then show *?case* **by** *auto*
  next
    case (*snoc xy xs*)

    have ∗:*map-of* (*xs* @ [*xy*]) = *map-of* (*xy*#*xs*)
      using *snoc.prems map-of-inject-set*[*of xs* @ [*xy*] *xy*#*xs*, *OF snoc.prems*]
      by *simp*

    show *?case*
      using *snoc* **unfolding** ∗
      by (*cases x* = *fst xy*; *auto*)
  qed
qed

lemma *map-pair-fst-helper* :
  map fst (*map* (λ (*x1*,*x2*) . ((*x1*,*x2*), *f x1 x2*)) *xs*) = *xs*
  using *map-pair-fst*[*of* λ (*x1*,*x2*) . *f x1 x2 xs*]
  by (*metis* (*no-types*, *lifting*) *map-eq-conv prod.collapse split-beta*)

end

2 Refinements for Utilities

Introduces program refinement for *Util.thy*.

theory *Util-Refined*
imports *Util Containers.Containers*
begin

2.1 New Code Equations for *set-as-map*

lemma *set-as-map-refined* [*code*]:
  fixes *t* :: ('*a* :: *ccompare* × '*c* :: *ccompare*) *set-rbt*

and *xs*:: ('*b* :: *ceq* × '*d* :: *ceq*) *set-dlist*
shows *set-as-map* (*DList-set xs*) = (*case ID CEQ*(('*b* × '*d*)) *of*
Some - ⇒ *Mapping.lookup* (*DList-Set.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m* (*x*) *of*
None ⇒ *Mapping.update* (*x*) {*z*} *m* |
Some zs ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)
xs
Mapping.empty) |
None ⇒ *Code.abort* (*STR* ''*set-as-map RBT-set*: *ccompare* = *None*'')
(λ-. *set-as-map* (*DList-set xs*)))
(**is** *?C2*)
and *set-as-map* (*RBT-set t*) = (*case ID CCOMPARE*(('*a* × '*c*)) *of*
Some - ⇒ *Mapping.lookup* (*RBT-Set2.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m* (*x*) *of*
None ⇒ *Mapping.update* (*x*) {*z*} *m* |
Some zs ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)
t
Mapping.empty) |
None ⇒ *Code.abort* (*STR* ''*set-as-map RBT-set*: *ccompare* = *None*'')
(λ-. *set-as-map* (*RBT-set t*)))
(**is** *?C1*)
proof −
show *?C1*
proof (*cases ID CCOMPARE*(('*a* × '*c*)))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

let *?f'* = (λ *t'* . (*RBT-Set2.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
t'
Mapping.empty))

let *?f* = λ *xs* . (*fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
xs Mapping.empty)
have ⋀ *xs* :: ('*a* × '*c*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof −
fix *xs* :: ('*a* × '*c*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*

then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

have ∗: (*?f* (*xs*@[(*x*,*z*)])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k*′ . *if k*′ = *k then Some v else Mapping.lookup m k*′)
by (*metis lookup-update*′)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x*′ . *if x*′ = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x*′)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z*′ . (*x*,*z*′) ∈ *set* (*xs*@[(*x*,*z*)])) **and** {*z*′ . (*x*,*z*′) ∈ *set* (*xs*@[(*x*,*z*)])} = {*z*}
by *fastforce*+
then have *m2*: (λ *x*′ . *if* (∃ *z*′ . (*x*′,*z*′) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z*′ . (*x*′,*z*′) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x*′ . *if x*′ = *x then Some* {*z*} *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x*′)
by *force*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** ∗ **by** *auto*
have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k*′ . *if k*′ = *k then Some v else Mapping.lookup m k*′)

by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x′*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have (∃ *z′* . (*x*,*z′*) ∈ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have (∃ *z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) **by** *fastforce*

have {*z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) ∈ *set xs*} = *Some zs*
using ‹(λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) ∈ *set xs*} = *zs* **by** *auto*
then show *?thesis* **by** *auto*
qed

have ⋀ *a* . (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*) *a*
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*) *a*
proof −
fix *a* **show** (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*) *a*
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*) *a*
using ‹{*z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*› ‹(∃ *z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)]))›
by (*cases a* = *x*; *auto*)
qed

then have *m2*: (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*)
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
qed

```
      qed
    qed
   then have Mapping.lookup (?f' t) = (λ x . if (∃ z . (x,z) ∈ set (RBT-Set2.keys
t)) then Some {z . (x,z) ∈ set (RBT-Set2.keys t)} else None)
      unfolding fold-conv-fold-keys by metis
    moreover have set (RBT-Set2.keys t) = (RBT-set t)
      using Some by (simp add: RBT-set-conv-keys)
    ultimately have Mapping.lookup (?f' t) = (λ x . if (∃ z . (x,z) ∈ (RBT-set
t)) then Some {z . (x,z) ∈ (RBT-set t)} else None)
      by force


    then show ?thesis
      using Some unfolding set-as-map-def by simp
  qed

  show ?C2
  proof (cases ID CEQ(('b × 'd)))
    case None
    then show ?thesis by auto
  next
    case (Some a)

    let ?f' = (λ t' . (DList-Set.fold (λ (x,z) m . case Mapping.lookup m x of
                                        None ⇒ Mapping.update x {z} m |
                                       Some zs ⇒ Mapping.update x (Set.insert z
zs) m)
                        t'
                        Mapping.empty))

    let ?f = λ xs . (fold (λ (x,z) m . case Mapping.lookup m x of
                                        None ⇒ Mapping.update x {z} m |
                                       Some zs ⇒ Mapping.update x (Set.insert z
zs) m)
                       xs Mapping.empty)
    have ∗: ⋀ xs :: ('b × 'd) list . Mapping.lookup (?f xs) = (λ x . if (∃ z . (x,z)
∈ set xs) then Some {z . (x,z) ∈ set xs} else None)
    proof −
      fix xs :: ('b × 'd) list
      show Mapping.lookup (?f xs) = (λ x . if (∃ z . (x,z) ∈ set xs) then Some {z
. (x,z) ∈ set xs} else None)
      proof (induction xs rule: rev-induct)
        case Nil
        then show ?case
          by (simp add: Mapping.empty.abs-eq Mapping.lookup.abs-eq)
      next
        case (snoc xz xs)
        then obtain x z where xz = (x,z)
          by (metis (mono-tags, opaque-lifting) surj-pair)
```

have $*$: (*?f* (*xs*@[(*x*,*z*)])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have $**$: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** $*$ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x′*)
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) **and** {*z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} = {*z*}
by *fastforce*+
then have *m2*: (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x′* . *if x′* = *x then Some* {*z*} *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*)
by *force*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have $**$: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** $*$ **by** *auto*
have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x′*)
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) **by** *fastforce*

have {*z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) $\in$ *set xs*} = *Some zs*
using ‹(λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) $\in$ *set xs*} = *zs* **by** *auto*
then show *?thesis* **by** *auto*
qed

have $\bigwedge$ *a* . (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x'*) *a*
proof −
fix *a* **show** (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x'*) *a*
using ‹{*z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*› ‹($\exists$ *z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)]))›
by (*cases a* = *x*; *auto*)
qed

then have *m2*: (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x'*)
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
qed
qed
qed

have *ID CEQ*('*b* × '*d*) $\neq$ *None*
using *Some* **by** *auto*
then have $**$: $\bigwedge$ *x* . *x* $\in$ *set* (*list-of-dlist xs*) = (*x* $\in$ (*DList-set xs*))

by (*simp add*: *Collect-member DList-set-def*)

have *Mapping.lookup* (*?f' xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ (*DList-set xs*)) *then* *Some* {*z* . (*x*,*z*) ∈ (*DList-set xs*)} *else None*)
using ∗[*of* (*list-of-dlist xs*)]
unfolding *DList-Set.fold.rep-eq* ∗∗ **by** *assumption*
then show *?thesis* **unfolding** *set-as-map-def* **using** *Some* **by** *simp*
qed
qed

end

3 Underlying FSM Representation

This theory contains the underlying datatype for (possibly not well-formed) finite state machines.

theory *FSM-Impl*
imports *Util Datatype-Order-Generator.Order-Generator HOL−Library.FSet*
begin

A finite state machine (FSM) is represented using its classical definition:

datatype (*'state*, *'input*, *'output*) *fsm-impl* = *FSMI* (*initial* : *'state*)
(*states* : *'state set*)
(*inputs* : *'input set*)
(*outputs* : *'output set*)
(*transitions* : (*'state* × *'input* × *'output* × *'state*) *set*)

3.1 Types for Transitions and Paths

type-synonym (*'a*,*'b*,*'c*) *transition* = (*'a* × *'b* × *'c* × *'a*)
type-synonym (*'a*,*'b*,*'c*) *path* = (*'a*,*'b*,*'c*) *transition list*

abbreviation *t-source* (*a* :: (*'a*,*'b*,*'c*) *transition*) ≡ *fst a*
abbreviation *t-input* (*a* :: (*'a*,*'b*,*'c*) *transition*) ≡ *fst* (*snd a*)
abbreviation *t-output* (*a* :: (*'a*,*'b*,*'c*) *transition*) ≡ *fst* (*snd* (*snd a*))
abbreviation *t-target* (*a* :: (*'a*,*'b*,*'c*) *transition*) ≡ *snd* (*snd* (*snd a*))

3.2 Basic Algorithms on FSM

3.2.1 Reading FSMs from Lists

fun *fsm-impl-from-list* :: *'a* ⇒
(*'a*,*'b*,*'c*) *transition list* ⇒
(*'a*, *'b*, *'c*) *fsm-impl*
where
fsm-impl-from-list q [] = *FSMI q* {*q*} {} {} {} |
fsm-impl-from-list q (*t*#*ts*) =

(*let ts′ = set (t#ts)*
in FSMI (t-source t)
((image t-source ts′) ∪ (image t-target ts′))
(image t-input ts′)
(image t-output ts′)
(ts′))

fun *fsm-impl-from-list′ :: ′a ⇒ (′a,′b,′c) transition list ⇒ (′a, ′b, ′c) fsm-impl*
where
fsm-impl-from-list′ q [] = FSMI q {q} {} {} {} |
fsm-impl-from-list′ q (t#ts) = (let tsr = (remdups (t#ts))
in FSMI (t-source t)
(set (remdups ((map t-source tsr) @ (map t-target tsr))))
(set (remdups (map t-input tsr)))
(set (remdups (map t-output tsr)))
(set tsr))

lemma *fsm-impl-from-list-code*[*code*] :
fsm-impl-from-list q ts = fsm-impl-from-list′ q ts
by (*cases ts*; *auto*)

3.2.2 Changing the initial State

fun *from-FSMI :: (′a,′b,′c) fsm-impl ⇒ ′a ⇒ (′a,′b,′c) fsm-impl* **where**
from-FSMI M q = (if q ∈ states M then FSMI q (states M) (inputs M) (outputs M) (transitions M) else M)

3.2.3 Product Construction

fun *product :: (′a,′b,′c) fsm-impl ⇒ (′d,′b,′c) fsm-impl ⇒ (′a × ′d,′b,′c) fsm-impl*
where
product A B = FSMI ((initial A, initial B))
((states A) × (states B))
(inputs A ∪ inputs B)
(outputs A ∪ outputs B)
{((qA,qB),x,y,(qA′,qB′)) | qA qB x y qA′ qB′ . (qA,x,y,qA′) ∈ transitions A ∧ (qB,x,y,qB′) ∈ transitions B}

lemma *product-code-naive*[*code*] :
product A B = FSMI ((initial A, initial B))
((states A) × (states B))
(inputs A ∪ inputs B)
(outputs A ∪ outputs B)
(image (λ((qA,x,y,qA′), (qB,x′,y′,qB′)) . ((qA,qB),x,y,(qA′,qB′)))
(Set.filter (λ((qA,x,y,qA′), (qB,x′,y′,qB′)) . x = x′ ∧ y = y′) (⋃(image (λ tA . image (λ tB . (tA,tB)) (transitions B)) (transitions A)))))
(**is** *?P1 = ?P2*)
proof −

have $(\bigcup(image\ (\lambda\ tA\ .\ image\ (\lambda\ tB\ .\ (tA,tB))\ (transitions\ B))\ (transitions\ A)))$
$= \{(tA,tB) \mid tA\ tB\ .\ tA \in transitions\ A \wedge tB \in transitions\ B\}$
by *auto*
then have $(Set.filter\ (\lambda((qA,x,y,qA'),\ (qB,x',y',qB'))\ .\ x = x' \wedge y = y')\ (\bigcup(image$
$(\lambda\ tA\ .\ image\ (\lambda\ tB\ .\ (tA,tB))\ (transitions\ B))\ (transitions\ A)))) = \{((qA,x,y,qA'),(qB,x,y,qB'))$
$\mid qA\ qB\ x\ y\ qA'\ qB'\ .\ (qA,x,y,qA') \in transitions\ A \wedge (qB,x,y,qB') \in transitions\ B\}$
by *auto*
then have $image\ (\lambda((qA,x,y,qA'),\ (qB,x',y',qB'))\ .\ ((qA,qB),x,y,(qA',qB')))\ (Set.filter$
$(\lambda((qA,x,y,qA'),\ (qB,x',y',qB'))\ .\ x = x' \wedge y = y')\ (\bigcup(image\ (\lambda\ tA\ .\ image\ (\lambda\ tB$
$.\ (tA,tB))\ (transitions\ B))\ (transitions\ A))))$
$= image\ (\lambda((qA,x,y,qA'),\ (qB,x',y',qB'))\ .\ ((qA,qB),x,y,(qA',qB')))$
$\{((qA,x,y,qA'),(qB,x,y,qB')) \mid qA\ qB\ x\ y\ qA'\ qB'\ .\ (qA,x,y,qA') \in transitions\ A \wedge$
$(qB,x,y,qB') \in transitions\ B\}$
by *auto*
moreover have $image\ (\lambda((qA,x,y,qA'),\ (qB,x',y',qB'))\ .\ ((qA,qB),x,y,(qA',qB')))$
$\{((qA,x,y,qA'),(qB,x,y,qB')) \mid qA\ qB\ x\ y\ qA'\ qB'\ .\ (qA,x,y,qA') \in transitions\ A \wedge$
$(qB,x,y,qB') \in transitions\ B\} = \{((qA,qB),x,y,(qA',qB')) \mid qA\ qB\ x\ y\ qA'\ qB'\ .$
$(qA,x,y,qA') \in transitions\ A \wedge (qB,x,y,qB') \in transitions\ B\}$
by *force*
ultimately have $transitions\ ?P1 = transitions\ ?P2$
unfolding *product.simps* **by** *auto*
moreover have $initial\ ?P1 = initial\ ?P2$ **by** *auto*
moreover have $states\ ?P1 = states\ ?P2$ **by** *auto*
moreover have $inputs\ ?P1 = inputs\ ?P2$ **by** *auto*
moreover have $outputs\ ?P1 = outputs\ ?P2$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

3.2.4 Filtering Transitions

fun $filter\text{-}transitions :: ('a,'b,'c)\ fsm\text{-}impl \Rightarrow (('a,'b,'c)\ transition \Rightarrow bool) \Rightarrow ('a,'b,'c)$
$fsm\text{-}impl$ **where**
$filter\text{-}transitions\ M\ P = FSMI\ (initial\ M)$
$(states\ M)$
$(inputs\ M)$
$(outputs\ M)$
$(Set.filter\ P\ (transitions\ M))$

3.2.5 Filtering States

fun $filter\text{-}states :: ('a,'b,'c)\ fsm\text{-}impl \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a,'b,'c)\ fsm\text{-}impl$ **where**
$filter\text{-}states\ M\ P = (if\ P\ (initial\ M)\ then\ FSMI\ (initial\ M)$
$(Set.filter\ P\ (states\ M))$
$(inputs\ M)$
$(outputs\ M)$
$(Set.filter\ (\lambda\ t\ .\ P\ (t\text{-}source\ t) \wedge P\ (t\text{-}target$
$t))\ (transitions\ M))$
$else\ M)$

3.2.6 Initial Singleton FSMI (For Trivial Preamble)

fun *initial-singleton* :: $('a,'b,'c)$ *fsm-impl* $\Rightarrow$ $('a,'b,'c)$ *fsm-impl* **where**
initial-singleton M = *FSMI* (*initial M*)
{*initial M*}
(*inputs M*)
(*outputs M*)
{}

3.2.7 Canonical Separator

abbreviation *shift-Inl t* $\equiv$ (*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*))

definition *shifted-transitions* :: $(('a \times 'a) \times 'b \times 'c \times ('a \times 'a))$ *set* $\Rightarrow$ $((('a \times 'a) + 'd) \times 'b \times 'c \times (('a \times 'a) + 'd))$ *set* **where**
shifted-transitions ts = *image shift-Inl ts*

definition *distinguishing-transitions* :: $(('a \times 'b) \Rightarrow 'c$ *set*$) \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'a)$ *set* $\Rightarrow 'b$ *set* $\Rightarrow$ $((('a \times 'a) + 'a) \times 'b \times 'c \times (('a \times 'a) + 'a))$ *set* **where**
distinguishing-transitions f q1 q2 stateSet inputSet = $\bigcup$ (*Set.image* (λ((*q1'*,*q2'*),*x*)
.
(*image* (λ*y* . (*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr q1*)) (*f* (*q1'*,*x*) − *f* (*q2'*,*x*)))
$\cup$ (*image* (λ*y* . (*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr q2*)) (*f* (*q2'*,*x*) − *f* (*q1'*,*x*))))
(*stateSet* $\times$ *inputSet*))

fun *canonical-separator'* :: $('a,'b,'c)$ *fsm-impl* $\Rightarrow$ $(('a \times 'a),'b,'c)$ *fsm-impl* $\Rightarrow 'a \Rightarrow 'a \Rightarrow (('a \times 'a) + 'a,'b,'c)$ *fsm-impl* **where**
canonical-separator' M P q1 q2 = (*if initial P* = (*q1*,*q2*)
then
(*let f'* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*)) (*transitions M*));
f = (λ*qx* . (*case f' qx of Some yqs* $\Rightarrow$ *yqs* | *None* $\Rightarrow$ {}));
shifted-transitions' = *shifted-transitions* (*transitions P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states P*) (*inputs P*);
ts = *shifted-transitions'* $\cup$ *distinguishing-transitions-lr*
in

FSMI (*Inl* (*q1*,*q2*))
((*image Inl* (*states P*)) $\cup$ {*Inr q1*, *Inr q2*})
(*inputs M* $\cup$ *inputs P*)
(*outputs M* $\cup$ *outputs P*)
(*ts*))
else FSMI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {})

lemma *h-out-impl-helper*: (λ (*q*,*x*) . {*y* . $\exists$ *q'* . (*q*,*x*,*y*,*q'*) $\in$ *A*}) = (λ*qx* . (*case*

(*set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*)) *A*)) *qx of Some yqs* $\Rightarrow$ *yqs* | *None* $\Rightarrow$ {}))
proof
fix *qx*
show (λ (*q*,*x*) . {*y* . $\exists$ *q'* . (*q*,*x*,*y*,*q'*) $\in$ *A*}) *qx* = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*)) *A*)) *qx of Some yqs* $\Rightarrow$ *yqs* | *None* $\Rightarrow$ {})) *qx*
proof −
obtain *q x* **where** *qx* = (*q*,*x*) **using** *prod.exhaust* **by** *metis*
have $**$: $\bigwedge$ *z* . ((*q*, *x*), *z*) $\in$ (λ(*q*, *x*, *y*, *q'*). ((*q*, *x*), *y*)) ‘ *A* = (*z* $\in$ {*y* . $\exists$ *q'* . (*q*,*x*,*y*,*q'*) $\in$ *A*})
by *force*
show *?thesis* **unfolding** ‹*qx* = (*q*,*x*)› *case-prod-conv set-as-map-def*
unfolding $**$ **by** *auto*
qed
qed

lemma *canonical-separator'-simps* :
initial (*canonical-separator' M P q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator' M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then* (*image Inl* (*states P*)) $\cup$ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
inputs (*canonical-separator' M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then inputs M* $\cup$ *inputs P else* {})
outputs (*canonical-separator' M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then outputs M* $\cup$ *outputs P else* {})
transitions (*canonical-separator' M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then shifted-transitions* (*transitions P*) $\cup$ *distinguishing-transitions* (λ (*q*,*x*) . {*y* . $\exists$ *q'* . (*q*,*x*,*y*,*q'*) $\in$ *transitions M*}) *q1 q2* (*states P*) (*inputs P*) *else* {})
unfolding *h-out-impl-helper* **by** (*simp add*: *Let-def*)+

3.2.8 Generalised Canonical Separator

A variation on the state separator that uses states *L* and *R* instead of *Inr q1* and *Inr q2* to indicate targets of transitions in the canonical separator that are available only for the left or right component of a state pair

Note: this definition of a canonical separator might serve as a way to avoid recalculation of state separators for different pairs of states, but is currently not fully implemented

datatype *LR* = *Left* | *Right*
derive *linorder LR*

definition *distinguishing-transitions-LR* :: (('*a* $\times$ '*b*) $\Rightarrow$ '*c set*) $\Rightarrow$ ('*a* $\times$ '*a*) *set* $\Rightarrow$ '*b set* $\Rightarrow$ ((('*a* $\times$ '*a*) + *LR*) $\times$ '*b* $\times$ '*c* $\times$ (('*a* $\times$ '*a*) + *LR*)) *set* **where**
distinguishing-transitions-LR f stateSet inputSet = $\bigcup$ (*Set.image* (λ((*q1'*,*q2'*),*x*) .
(*image* (λ*y* . (*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr Left*)) (*f* (*q1'*,*x*) − *f* (*q2'*,*x*)))
$\cup$ (*image* (λ*y* . (*Inl* (*q1'*,*q2'*),*x*,*y*,*Inr Right*)) (*f* (*q2'*,*x*) − *f* (*q1'*,*x*))))

(*stateSet* × *inputSet*))

fun *canonical-separator-complete′* :: (′*a*,′*b*,′*c*) *fsm-impl* ⇒ ((′*a* × ′*a*) + *LR*,′*b*,′*c*) *fsm-impl* **where**
canonical-separator-complete′ M =
(*let P* = *product M M*;
f′ = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions M*));
f = (λ*qx* . (*case f′ qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions′ = *shifted-transitions* (*transitions P*);
distinguishing-transitions-lr = *distinguishing-transitions-LR f* (*states P*) (*inputs P*);
ts = *shifted-transitions′* ∪ *distinguishing-transitions-lr*
in
FSMI (*Inl* (*initial P*))
((*image Inl* (*states P*)) ∪ {*Inr Left*, *Inr Right*})
(*inputs M* ∪ *inputs P*)
(*outputs M* ∪ *outputs P*)
ts)

3.2.9 Adding Transitions

fun *add-transitions* :: (′*a*,′*b*,′*c*) *fsm-impl* ⇒ (′*a*,′*b*,′*c*) *transition set* ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
add-transitions M ts = (*if* (∀ *t* ∈ *ts* . *t-source t* ∈ *states M* ∧ *t-input t* ∈ *inputs M* ∧ *t-output t* ∈ *outputs M* ∧ *t-target t* ∈ *states M*)
then FSMI (*initial M*)
(*states M*)
(*inputs M*)
(*outputs M*)
((*transitions M*) ∪ *ts*)
else M)

3.2.10 Creating an FSMI without transitions

fun *create-unconnected-FSMI* :: ′*a* ⇒ ′*a set* ⇒ ′*b set* ⇒ ′*c set* ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
create-unconnected-FSMI q ns ins outs = (*if* (*finite ns* ∧ *finite ins* ∧ *finite outs*)
then FSMI q (*insert q ns*) *ins outs* {}
else FSMI q {*q*} {} {} {})

fun *create-unconnected-fsm-from-lists* :: ′*a* ⇒ ′*a list* ⇒ ′*b list* ⇒ ′*c list* ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
create-unconnected-fsm-from-lists q ns ins outs = *FSMI q* (*insert q* (*set ns*)) (*set ins*) (*set outs*) {}

fun *create-unconnected-fsm-from-fsets* :: ′*a* ⇒ ′*a fset* ⇒ ′*b fset* ⇒ ′*c fset* ⇒ (′*a*,′*b*,′*c*) *fsm-impl* **where**
create-unconnected-fsm-from-fsets q ns ins outs = *FSMI q* (*insert q* (*fset ns*)) (*fset ins*) (*fset outs*) {}

fun *create-fsm-from-sets* :: *'a* ⇒ *'a set* ⇒ *'b set* ⇒ *'c set* ⇒ (*'a*,*'b*,*'c*) *transition set* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**
create-fsm-from-sets q qs ins outs ts = (*if q* ∈ *qs* ∧ *finite qs* ∧ *finite ins* ∧ *finite outs*
then add-transitions (*FSMI q qs ins outs* {}) *ts*
else FSMI q {*q*} {} {} {})

3.3 Transition Function h

Function *h* represents the classical view of the transition relation of an FSM *M* as a function: given a state *q* and an input *x*, (*h M*) (*q*,*x*) returns all possibly reactions (*y*,*q'*) of *M* in state *q* to *x*, where *y* is the produced output and *q'* the target state of the reaction transition.

fun *h* :: (*'state*, *'input*, *'output*) *fsm-impl* ⇒ (*'state* × *'input*) ⇒ (*'output* × *'state*) *set* **where**
h M (*q*,*x*) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions M* }

fun *h-obs* :: (*'a*,*'b*,*'c*) *fsm-impl* ⇒ *'a* ⇒ *'b* ⇒ *'c* ⇒ *'a option* **where**
h-obs M q x y = (*let*
tgts = *snd* ‘ *Set.filter* (λ (*y'*,*q'*) . *y'* = *y*) (*h M* (*q*,*x*))
in if card tgts = *1*
then Some (*the-elem tgts*)
else None)

lemma *h-code*[*code*] :
h M (*q*,*x*) = (*let m* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*transitions M*))
in (*case m* (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
unfolding *set-as-map-def* **by** *force*

3.4 Extending FSMs by single elements

fun *add-transition* :: (*'a*,*'b*,*'c*) *fsm-impl* ⇒
(*'a*,*'b*,*'c*) *transition* ⇒
(*'a*,*'b*,*'c*) *fsm-impl*
where
add-transition M t =
(*if t-source t* ∈ *states M* ∧ *t-input t* ∈ *inputs M* ∧
t-output t ∈ *outputs M* ∧ *t-target t* ∈ *states M*
then FSMI (*initial M*)
(*states M*)
(*inputs M*)
(*outputs M*)
(*insert t* (*transitions M*))
else M)

fun *add-state* :: (*'a*,*'b*,*'c*) *fsm-impl* ⇒ *'a* ⇒ (*'a*,*'b*,*'c*) *fsm-impl* **where**

add-state M q = *FSMI* (*initial M*) (*insert q* (*states M*)) (*inputs M*) (*outputs M*) (*transitions M*)

fun *add-input* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ '*b* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-input M x = *FSMI* (*initial M*) (*states M*) (*insert x* (*inputs M*)) (*outputs M*) (*transitions M*)

fun *add-output* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ '*c* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-output M y = *FSMI* (*initial M*) (*states M*) (*inputs M*) (*insert y* (*outputs M*)) (*transitions M*)

fun *add-transition-with-components* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *fsm-impl* **where**
add-transition-with-components M t = *add-transition* (*add-state* (*add-state* (*add-input* (*add-output M* (*t-output t*)) (*t-input t*)) (*t-source t*)) (*t-target t*)) *t*

3.5 Renaming elements

fun *rename-states* :: ('*a*,'*b*,'*c*) *fsm-impl* ⇒ ('*a* ⇒ '*d*) ⇒ ('*d*,'*b*,'*c*) *fsm-impl* **where**
rename-states M f = *FSMI* (*f* (*initial M*))
(*f* ‘ *states M*)
(*inputs M*)
(*outputs M*)
((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions M*)

end

4 Finite State Machines

This theory defines well-formed finite state machines and introduces various closely related notions, as well as a selection of basic properties and definitions.

theory *FSM*
imports *FSM-Impl HOL−Library.Quotient-Type HOL−Library.Product-Lexorder*
begin

4.1 Well-formed Finite State Machines

A value of type *fsm-impl* constitutes a well-formed FSM if its contained sets are finite and the initial state and the components of each transition are contained in their respective sets.

abbreviation(*input*) *well-formed-fsm* (*M* :: ('*state*, '*input*, '*output*) *fsm-impl*)
≡ (*initial M* ∈ *states M*
∧ *finite* (*states M*)
∧ *finite* (*inputs M*)

$\wedge$ *finite* (*outputs M*)
$\wedge$ *finite* (*transitions M*)
$\wedge$ ($\forall$ *t* $\in$ *transitions M* . *t-source t* $\in$ *states M* $\wedge$
t-input t $\in$ *inputs M* $\wedge$
t-target t $\in$ *states M* $\wedge$
t-output t $\in$ *outputs M*))

typedef ('*state*, '*input*, '*output*) *fsm* =
{ *M* :: ('*state*, '*input*, '*output*) *fsm-impl* . *well-formed-fsm M*}
morphisms *fsm-impl-of-fsm Abs-fsm*
proof −
obtain *q* :: '*state* **where** *True* **by** *blast*
have *well-formed-fsm* (*FSMI q* {*q*} {} {} {}) **by** *auto*
then show *?thesis* **by** *blast*
qed

setup-lifting *type-definition-fsm*

lift-definition *initial* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state* **is** *FSM-Impl.initial*
done
lift-definition *states* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state set* **is** *FSM-Impl.states*
done
lift-definition *inputs* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*input set* **is** *FSM-Impl.inputs*
done
lift-definition *outputs* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*output set* **is** *FSM-Impl.outputs*
done
lift-definition *transitions* ::
('*state*, '*input*, '*output*) *fsm* ⇒ ('*state* × '*input* × '*output* × '*state*) *set*
is *FSM-Impl.transitions* **done**

lift-definition *fsm-from-list* :: '*a* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ ('*a*, '*b*, '*c*) *fsm*
is *FSM-Impl.fsm-impl-from-list*
proof −
fix *q* :: '*a*
fix *ts* :: ('*a*,'*b*,'*c*) *transition list*
show *well-formed-fsm* (*fsm-impl-from-list q ts*)
by (*induction ts*; *auto*)
qed

lemma *fsm-initial*[*intro*]: *initial M* $\in$ *states M*
by (*transfer*; *blast*)
lemma *fsm-states-finite*: *finite* (*states M*)
by (*transfer*; *blast*)
lemma *fsm-inputs-finite*: *finite* (*inputs M*)
by (*transfer*; *blast*)
lemma *fsm-outputs-finite*: *finite* (*outputs M*)

by (*transfer*; *blast*)
lemma *fsm-transitions-finite*: *finite* (*transitions M*)
by (*transfer*; *blast*)
lemma *fsm-transition-source*[*intro*]: $\bigwedge$ *t* . *t* $\in$ (*transitions M*) $\Longrightarrow$ *t-source t* $\in$ *states M*
by (*transfer*; *blast*)
lemma *fsm-transition-target*[*intro*]: $\bigwedge$ *t* . *t* $\in$ (*transitions M*) $\Longrightarrow$ *t-target t* $\in$ *states M*
by (*transfer*; *blast*)
lemma *fsm-transition-input*[*intro*]: $\bigwedge$ *t* . *t* $\in$ (*transitions M*) $\Longrightarrow$ *t-input t* $\in$ *inputs M*
by (*transfer*; *blast*)
lemma *fsm-transition-output*[*intro*]: $\bigwedge$ *t* . *t* $\in$ (*transitions M*) $\Longrightarrow$ *t-output t* $\in$ *outputs M*
by (*transfer*; *blast*)

instantiation *fsm* :: (*type*,*type*,*type*) *equal*
begin
definition *equal-fsm* :: ('*a*, '*b*, '*c*) *fsm* $\Rightarrow$ ('*a*, '*b*, '*c*) *fsm* $\Rightarrow$ *bool* **where**
equal-fsm x y = (*initial x* = *initial y* $\wedge$ *states x* = *states y* $\wedge$ *inputs x* = *inputs y* $\wedge$ *outputs x* = *outputs y* $\wedge$ *transitions x* = *transitions y*)

instance
apply (*intro-classes*)
unfolding *equal-fsm-def*
apply *transfer*
using *fsm-impl.expand* **by** *auto*
end

4.1.1 Example FSMs

definition *m-ex-H* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-H = *fsm-from-list 1* [(*1*,*0*,*0*,*2*),
(*1*,*0*,*1*,*4*),
(*1*,*1*,*1*,*4*),
(*2*,*0*,*0*,*2*),
(*2*,*1*,*1*,*4*),
(*3*,*0*,*1*,*4*),
(*3*,*1*,*0*,*1*),
(*3*,*1*,*1*,*3*),
(*4*,*0*,*0*,*3*),
(*4*,*1*,*0*,*1*)]

definition *m-ex-9* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-9 = *fsm-from-list 0* [(*0*,*0*,*2*,*2*),
(*0*,*0*,*3*,*2*),
(*0*,*1*,*0*,*3*),

(0,1,1,3),
(1,0,3,2),
(1,1,1,3),
(2,0,2,2),
(2,1,3,3),
(3,0,2,2),
(3,1,0,2),
(3,1,1,1)]

definition *m-ex-DR* :: (*integer*,*integer*,*integer*) *fsm* **where**
m-ex-DR = *fsm-from-list 0* [(0,0,0,100),
(100,0,0,101),
(100,0,1,101),
(101,0,0,102),
(101,0,1,102),
(102,0,0,103),
(102,0,1,103),
(103,0,0,104),
(103,0,1,104),
(104,0,0,100),
(104,0,1,100),
(104,1,0,400),
(0,0,2,200),
(200,0,2,201),
(201,0,2,202),
(202,0,2,203),
(203,0,2,200),
(203,1,0,400),
(0,1,0,300),
(100,1,0,300),
(101,1,0,300),
(102,1,0,300),
(103,1,0,300),
(200,1,0,300),
(201,1,0,300),
(202,1,0,300),
(300,0,0,300),
(300,1,0,300),
(400,0,0,300),
(400,1,0,300)]

4.2 Transition Function h and related functions

lift-definition *h* :: (*'state*, *'input*, *'output*) *fsm* ⇒ (*'state* × *'input*) ⇒ (*'output* × *'state*) *set*
is *FSM-Impl.h* **.**

lemma *h-simps*[*simp*]: *FSM.h M* (*q*,*x*) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) ∈ *transitions M* }
by (*transfer*; *auto*)

lift-definition *h-obs* :: ('*state*, '*input*, '*output*) *fsm* ⇒ '*state* ⇒ '*input* ⇒ '*output* ⇒ '*state option*
  is *FSM-Impl.h-obs* **.**

lemma *h-obs-simps*[*simp*]: *FSM.h-obs M q x y* = (*let*
    tgts = *snd* ‘ *Set.filter* (λ (*y*′,*q*′) . *y*′ = *y*) (*h M* (*q*,*x*))
  in if card tgts = *1*
    then Some (*the-elem tgts*)
    else None)
  by (*transfer*; *auto*)

fun *defined-inputs*′ :: (('*a* ×'*b*) ⇒ ('*c*×'*a*) *set*) ⇒ '*b set* ⇒ '*a* ⇒ '*b set* **where**
  defined-inputs′ *hM iM q* = {*x* ∈ *iM* . *hM* (*q*,*x*) ≠ {}}

fun *defined-inputs* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*b set* **where**
  defined-inputs M q = *defined-inputs*′ (*h M*) (*inputs M*) *q*

lemma *defined-inputs-set* : *defined-inputs M q* = {*x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {}
}
  by *auto*

fun *transitions-from*′ :: (('*a* ×'*b*) ⇒ ('*c*×'*a*) *set*) ⇒ '*b set* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *transition set* **where**
  transitions-from′ *hM iM q* = ⋃(*image* (λ*x* . *image* (λ(*y*,*q*′) . (*q*,*x*,*y*,*q*′)) (*hM* (*q*,*x*))) *iM*)

fun *transitions-from* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *transition set* **where**
  transitions-from M q = *transitions-from*′ (*h M*) (*inputs M*) *q*

lemma *transitions-from-set* :
  assumes *q* ∈ *states M*
  shows *transitions-from M q* = {*t* ∈ *transitions M* . *t-source t* = *q*}
proof −
  have ⋀ *t* . *t* ∈ *transitions-from M q* ⟹ *t* ∈ *transitions M* ∧ *t-source t* = *q* **by** *auto*
  moreover have ⋀ *t* . *t* ∈ *transitions M* ⟹ *t-source t* = *q* ⟹ *t* ∈ *transitions-from M q*
  proof −
    fix *t* **assume** *t* ∈ *transitions M* **and** *t-source t* = *q*
    then have (*t-output t*, *t-target t*) ∈ *h M* (*q*,*t-input t*) **and** *t-input t* ∈ *inputs M* **by** *auto*
    then have *t-input t* ∈ *defined-inputs*′ (*h M*) (*inputs M*) *q*
      unfolding *defined-inputs*′*.simps* ‹*t-source t* = *q*› **by** *blast*

    have (*q*, *t-input t*, *t-output t*, *t-target t*) ∈ *transitions M*
      using ‹*t-source t* = *q*› ‹*t* ∈ *transitions M*› **by** *auto*
    then have (*q*, *t-input t*, *t-output t*, *t-target t*) ∈ (λ(*y*, *q*′). (*q*, *t-input t*, *y*, *q*′))

‘ *h M* (*q*, *t-input t*)
using ‹(*t-output t*, *t-target t*) ∈ *h M* (*q*,*t-input t*)›
unfolding *h.simps*
by (*metis* (*no-types*, *lifting*) *image-iff prod.case-eq-if surjective-pairing*)
then have *t* ∈ (λ(*y*, *q′*). (*q*, *t-input t*, *y*, *q′*)) ‘ *h M* (*q*, *t-input t*)
using ‹*t-source t* = *q*› **by** (*metis prod.collapse*)
then show *t* ∈ *transitions-from M q*

unfolding *transitions-from.simps transitions-from′.simps*
using ‹*t-input t* ∈ *defined-inputs′* (*h M*) (*inputs M*) *q*›
using ‹*t-input t* ∈ *FSM.inputs M*› **by** *blast*
qed
ultimately show *?thesis* **by** *blast*
qed

fun *h-from* :: (*′state*, *′input*, *′output*) *fsm* ⇒ *′state* ⇒ (*′input* × *′output* × *′state*) *set* **where**
h-from M q = { (*x*,*y*,*q′*) . (*q*,*x*,*y*,*q′*) ∈ *transitions M* }

lemma *h-from*[*code*] : *h-from M q* = (*let m* = *set-as-map* (*transitions M*)
in (*case m q of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
unfolding *set-as-map-def* **by** *force*

fun *h-out* :: (*′a*,*′b*,*′c*) *fsm* ⇒ (*′a* × *′b*) ⇒ *′c set* **where**
h-out M (*q*,*x*) = {*y* . ∃ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*}

lemma *h-out-code*[*code*]:
h-out M = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions M*))) *qx of*
Some yqs ⇒ *yqs* |
None ⇒ {}))
proof −

let *?f* = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions M*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}))

have ⋀ *qx* . (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions M*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {})) *qx* = (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ (*transitions M*)}) *qx*
unfolding *set-as-map-def* **by** *auto*

moreover have ⋀ *qx* . (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ (*transitions M*)}) *qx* = (λ *qx* . {*y* | *y* . ∃ *q′* . (*fst qx*, *snd qx*, *y*, *q′*) ∈ (*transitions M*)}) *qx*
by *force*

ultimately have *?f* = (λ *qx* . {*y* | *y* . ∃ *q′* . (*fst qx*, *snd qx*, *y*, *q′*) ∈ (*transitions M*)})
by *blast*
then have *?f* = (λ (*q*,*x*) . {*y* | *y* . ∃ *q′* . (*q*, *x*, *y*, *q′*) ∈ (*transitions M*)}) **by** *force*

then show *?thesis* **by** *force*
qed

lemma *h-out-alt-def* :
h-out M (*q*,*x*) = {*t-output t* | *t* . *t* ∈ *transitions M* ∧ *t-source t* = *q* ∧ *t-input t* = *x*}
unfolding *h-out.simps*
by *auto*

4.3 Size

instantiation *fsm* :: (*type*,*type*,*type*) *size*
begin

definition *size* **where** [*simp*, *code*]: *size* (*m*::(*′a*, *′b*, *′c*) *fsm*) = *card* (*states m*)

instance ..
end

lemma *fsm-size-Suc* :
size M > *0*
unfolding *FSM.size-def*
using *fsm-states-finite*[*of M*] *fsm-initial*[*of M*]
using *card-gt-0-iff* **by** *blast*

4.4 Paths

inductive *path* :: (*′state*, *′input*, *′output*) *fsm* ⇒ *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *bool*
where
nil[*intro*!] : *q* ∈ *states M* ⟹ *path M q* [] |
cons[*intro*!] : *t* ∈ *transitions M* ⟹ *path M* (*t-target t*) *ts* ⟹ *path M* (*t-source t*) (*t*#*ts*)

inductive-cases *path-nil-elim*[*elim*!]: *path M q* []
inductive-cases *path-cons-elim*[*elim*!]: *path M q* (*t*#*ts*)

fun *visited-states* :: *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *′state list* **where**
visited-states q p = (*q* # *map t-target p*)

fun *target* :: *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *′state* **where**
target q p = *last* (*visited-states q p*)

lemma *target-nil* [*simp*] : *target q* [] = *q* **by** *auto*
lemma *target-snoc* [*simp*] : *target q* (*p*@[*t*]) = *t-target t* **by** *auto*

lemma *path-begin-state* :
  assumes *path M q p*
  shows *q* ∈ *states M*
  using *assms* **by** (*cases*; *auto*)

lemma *path-append*[*intro*!] :
  assumes *path M q p1*
    and *path M* (*target q p1*) *p2*
  shows *path M q* (*p1*@*p2*)
  using *assms* **by** (*induct p1 arbitrary*: *p2*; *auto*)

lemma *path-target-is-state* :
  assumes *path M q p*
  shows *target q p* ∈ *states M*
using *assms* **by** (*induct p*; *auto*)

lemma *path-suffix* :
  assumes *path M q* (*p1*@*p2*)
  shows *path M* (*target q p1*) *p2*
using *assms* **by** (*induction p1 arbitrary*: *q*; *auto*)

lemma *path-prefix* :
  assumes *path M q* (*p1*@*p2*)
  shows *path M q p1*
using *assms* **by** (*induction p1 arbitrary*: *q*; *auto*; (*metis path-begin-state*))

lemma *path-append-elim*[*elim*!] :
  assumes *path M q* (*p1*@*p2*)
  obtains *path M q p1*
    and *path M* (*target q p1*) *p2*
  by (*meson assms path-prefix path-suffix*)

lemma *path-append-target*:
  target q (*p1*@*p2*) = *target* (*target q p1*) *p2*
  by (*induction p1*) (*simp*+)

lemma *path-append-target-hd* :
  assumes *length p* > *0*
  shows *target q p* = *target* (*t-target* (*hd p*)) (*tl p*)
using *assms* **by** (*induction p*) (*simp*+)

lemma *path-transitions* :
  assumes *path M q p*
  shows *set p* ⊆ *transitions M*
  using *assms* **by** (*induct p arbitrary*: *q*; *fastforce*)

lemma *path-append-transition*[*intro*!] :
  assumes *path M q p*
  and $t \in$ *transitions M*
  and *t-source t* = *target q p*
shows *path M q* (*p*@[*t*])
 by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *cons fsm-transition-target nil path-append*)

lemma *path-append-transition-elim*[*elim*!] :
  assumes *path M q* (*p*@[*t*])
shows *path M q p*
and $t \in$ *transitions M*
and *t-source t* = *target q p*
  using *assms* **by** *auto*

lemma *path-prepend-t* : *path M q′ p* $\Longrightarrow$ (*q*,*x*,*y*,*q′*) $\in$ *transitions M* $\Longrightarrow$ *path M q* ((*q*,*x*,*y*,*q′*)#*p*)
  by (*metis* (*mono-tags*, *lifting*) *fst-conv path.intros*(*2*) *prod.sel*(*2*))

lemma *path-target-append* : *target q1 p1* = *q2* $\Longrightarrow$ *target q2 p2* = *q3* $\Longrightarrow$ *target q1* (*p1*@*p2*) = *q3*
  by *auto*

lemma *single-transition-path* : $t \in$ *transitions M* $\Longrightarrow$ *path M* (*t-source t*) [*t*] **by** *auto*

lemma *path-source-target-index* :
  assumes *Suc i* < *length p*
  and *path M q p*
shows *t-target* (*p* ! *i*) = *t-source* (*p* ! (*Suc i*))
  using *assms* **proof** (*induction p rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc t ps*)
  then have *path M q ps* **and** *t-source t* = *target q ps* **and** $t \in$ *transitions M* **by** *auto*

  show *?case* **proof** (*cases Suc i* < *length ps*)
    case *True*
    then have *t-target* (*ps* ! *i*) = *t-source* (*ps* ! *Suc i*)
      using *snoc.IH* ‹*path M q ps*› **by** *auto*
    then show *?thesis*
      by (*simp add*: *Suc-lessD True nth-append*)
  next
    case *False*
    then have *Suc i* = *length ps*
      using *snoc.prems*(*1*) **by** *auto*
    then have (*ps* @ [*t*]) ! *Suc i* = *t*

```
      by auto

    show ?thesis proof (cases ps = [])
      case True
      then show ?thesis using ‹Suc i = length ps› by auto
    next
      case False
      then have target q ps = t-target (last ps)
        unfolding target.simps visited-states.simps
        by (simp add: last-map)
      then have target q ps = t-target (ps ! i)
        using ‹Suc i = length ps›
        by (metis False diff-Suc-1 last-conv-nth)
      then show ?thesis
        using ‹t-source t = target q ps›
        by (metis ‹(ps @ [t]) ! Suc i = t› ‹Suc i = length ps› lessI nth-append)
    qed
  qed
qed

lemma paths-finite : finite { p . path M q p ∧ length p ≤ k }
proof −
  have { p . path M q p ∧ length p ≤ k } ⊆ {xs . set xs ⊆ transitions M ∧ length
xs ≤ k}
    by (metis (no-types, lifting) Collect-mono path-transitions)
  then show finite { p . path M q p ∧ length p ≤ k }
    using finite-lists-length-le[OF fsm-transitions-finite[of M], of k]
    by (metis (mono-tags) finite-subset)
qed

lemma visited-states-prefix :
  assumes q′ ∈ set (visited-states q p)
  shows  ∃ p1 p2 . p = p1@p2 ∧ target q p1 = q′
using assms proof (induction p arbitrary: q)
  case Nil
  then show ?case by auto
next
  case (Cons a p)
  then show ?case
  proof (cases q′ ∈ set (visited-states (t-target a) p))
    case True
    then obtain p1 p2 where p = p1 @ p2 ∧ target (t-target a) p1 = q′
      using Cons.IH by blast
    then have (a#p) = (a#p1)@p2 ∧ target q (a#p1) = q′
      by auto
    then show ?thesis by blast
  next
    case False
    then have q′ = q
```

```
    using Cons.prems by auto
  then show ?thesis
    by auto
 qed
qed

lemma visited-states-are-states :
 assumes path M q1 p
 shows set (visited-states q1 p) ⊆ states M
 by (metis assms path-prefix path-target-is-state subsetI visited-states-prefix)

lemma transition-subset-path :
 assumes transitions A ⊆ transitions B
 and path A q p
 and q ∈ states B
shows path B q p
using assms(2) proof (induction p rule: rev-induct)
 case Nil
 show ?case using assms(3) by auto
next
 case (snoc t p)
 then show ?case using assms(1) path-suffix
  by fastforce
qed
```

4.4.1 Paths of fixed length

```
fun paths-of-length' :: ('a,'b,'c) path ⇒ 'a ⇒ (('a ×'b) ⇒ ('c×'a) set) ⇒ 'b set ⇒
nat ⇒ ('a,'b,'c) path set
 where
 paths-of-length' prev q hM iM 0 = {prev} |
 paths-of-length' prev q hM iM (Suc k) =
  (let hF = transitions-from' hM iM q
   in ⋃ (image (λ t . paths-of-length' (prev@[t]) (t-target t) hM iM k) hF))

fun paths-of-length :: ('a,'b,'c) fsm ⇒ 'a ⇒ nat ⇒ ('a,'b,'c) path set where
 paths-of-length M q k = paths-of-length' [] q (h M) (inputs M) k
```

4.4.2 Paths up to fixed length

```
fun paths-up-to-length' :: ('a,'b,'c) path ⇒ 'a ⇒ (('a ×'b) ⇒ (('c×'a) set)) ⇒ 'b
set ⇒ nat ⇒ ('a,'b,'c) path set
 where
 paths-up-to-length' prev q hM iM 0 = {prev} |
 paths-up-to-length' prev q hM iM (Suc k) =
  (let hF = transitions-from' hM iM q
   in insert prev (⋃ (image (λ t . paths-up-to-length' (prev@[t]) (t-target t) hM
iM k) hF)))

fun paths-up-to-length :: ('a,'b,'c) fsm ⇒ 'a ⇒ nat ⇒ ('a,'b,'c) path set where
```

paths-up-to-length M q k = *paths-up-to-length′* [] *q* (*h M*) (*inputs M*) *k*

lemma *paths-up-to-length′-set* :
assumes $q \in$ *states M*
and *path M q prev*
shows *paths-up-to-length′ prev* (*target q prev*) (*h M*) (*inputs M*) *k*
= {(*prev*@*p*) | *p* . *path M* (*target q prev*) *p* ∧ *length p* ≤ *k*}
using *assms*(*2*) **proof** (*induction k arbitrary*: *prev*)
case *0*
show *?case* **unfolding** *paths-up-to-length′.simps* **using** *path-target-is-state*[*OF 0.prems*(*1*)] **by** *auto*
next
case (*Suc k*)

have ⋀ *p* . *p* ∈ *paths-up-to-length′ prev* (*target q prev*) (*h M*) (*inputs M*) (*Suc k*)
⟹ *p* ∈ {(*prev*@*p*) | *p* . *path M* (*target q prev*) *p* ∧ *length p* ≤ *Suc k*}
proof −
fix *p* **assume** *p* ∈ *paths-up-to-length′ prev* (*target q prev*) (*h M*) (*inputs M*) (*Suc k*)
then show *p* ∈ {(*prev*@*p*) | *p* . *path M* (*target q prev*) *p* ∧ *length p* ≤ *Suc k*}
proof (*cases p* = *prev*)
case *True*
show *?thesis* **using** *path-target-is-state*[*OF Suc.prems*(*1*)] **unfolding** *True* **by** (*simp add*: *nil*)
next
case *False*
then have *p* ∈ (⋃ (*image* (λ*t*. *paths-up-to-length′* (*prev*@[*t*]) (*t-target t*) (*h M*) (*inputs M*) *k*)
(*transitions-from′* (*h M*) (*inputs M*) (*target q prev*))))
using ‹*p* ∈ *paths-up-to-length′ prev* (*target q prev*) (*h M*) (*inputs M*) (*Suc k*)›
unfolding *paths-up-to-length′.simps Let-def* **by** *blast*
then obtain *t* **where** *t* ∈ ⋃(*image* (λ*x* . *image* (λ(*y*,*q′*) . ((*target q prev*),*x*,*y*,*q′*))
(*h M* ((*target q prev*),*x*))) (*inputs M*))
and *p* ∈ *paths-up-to-length′* (*prev*@[*t*]) (*t-target t*) (*h M*) (*inputs M*) *k*
unfolding *transitions-from′.simps* **by** *blast*

have *t* ∈ *transitions M* **and** *t-source t* = (*target q prev*)
using ‹*t* ∈ ⋃(*image* (λ*x* . *image* (λ(*y*,*q′*) . ((*target q prev*),*x*,*y*,*q′*))
(*h M* ((*target q prev*),*x*))) (*inputs M*))› **by** *auto*
then have *path M q* (*prev*@[*t*])
using *Suc.prems*(*1*) **using** *path-append-transition* **by** *simp*

have (*target q* (*prev* @ [*t*])) = *t-target t* **by** *auto*

```
    show ?thesis
      using ‹p ∈ paths-up-to-length′ (prev@[t]) (t-target t) (h M) (inputs M) k›
      using Suc.IH[OF ‹path M q (prev@[t])›]
      unfolding ‹(target q (prev @ [t])) = t-target t›
      using ‹path M q (prev @ [t])› by auto
  qed
 qed

 moreover have ⋀ p . p ∈ {(prev@p) | p . path M (target q prev) p ∧ length p
≤ Suc k}
                ⟹ p ∈ paths-up-to-length′ prev (target q prev) (h M) (inputs M)
(Suc k)
 proof −
  fix p assume p ∈ {(prev@p) | p . path M (target q prev) p ∧ length p ≤ Suc k}
  then obtain p′ where p = prev@p′
                 and   path M (target q prev) p′
                 and   length p′ ≤ Suc k
    by blast

  have prev@p′ ∈ paths-up-to-length′ prev (target q prev) (h M) (inputs M) (Suc
k)
  proof (cases p′)
    case Nil
    then show ?thesis by auto
  next
    case (Cons t p″)

    then have t ∈ transitions M and t-source t = (target q prev)
      using ‹path M (target q prev) p′› by auto
    then have path M q (prev@[t])
      using Suc.prems(1) using path-append-transition by simp

    have (target q (prev @ [t])) = t-target t by auto

    have length p″ ≤ k using ‹length p′ ≤ Suc k› Cons by auto
    moreover have path M (target q (prev@[t])) p″
      using ‹path M (target q prev) p′› unfolding Cons
      by auto
      ultimately have p ∈ paths-up-to-length′ (prev @ [t]) (t-target t) (h M)
(FSM.inputs M) k
      using Suc.IH[OF ‹path M q (prev@[t])›]
     unfolding ‹(target q (prev @ [t])) = t-target t› ‹p = prev@p′› Cons by simp
     then have prev@t#p″ ∈ paths-up-to-length′ (prev @ [t]) (t-target t) (h M)
(FSM.inputs M) k
      unfolding ‹p = prev@p′› Cons by auto

    have t ∈ (λ(y, q′). (t-source t, t-input t, y, q′)) ‘
                        {(y, q′). (t-source t, t-input t, y, q′) ∈ FSM.transitions M}
```

using ‹$t \in$ *transitions M*›
by (*metis* (*no-types*, *lifting*) *case-prodI mem-Collect-eq pair-imageI surjective-pairing*)
then have $t \in$ *transitions-from'* (*h M*) (*inputs M*) (*target q prev*)
unfolding *transitions-from'.simps*
using *fsm-transition-input*[*OF* ‹$t \in$ *transitions M*›]
unfolding ‹*t-source t* = (*target q prev*)›[*symmetric*] *h-simps*
by *blast*

then show *?thesis*
using ‹*prev* @ *t* # *p''* $\in$ *paths-up-to-length'* (*prev*@[*t*]) (*t-target t*) (*h M*) (*FSM.inputs M*) *k*›
unfolding ‹*p* = *prev*@*p'*› *Cons paths-up-to-length'.simps Let-def* **by** *blast*
qed
then show *p* $\in$ *paths-up-to-length' prev* (*target q prev*) (*h M*) (*inputs M*) (*Suc k*)
unfolding ‹*p* = *prev*@*p'*› **by** *assumption*
qed

ultimately show *?case* **by** *blast*
qed

lemma *paths-up-to-length-set* :
assumes *q* $\in$ *states M*
shows *paths-up-to-length M q k* = {*p* . *path M q p* $\wedge$ *length p* $\leq$ *k*}
unfolding *paths-up-to-length.simps*
using *paths-up-to-length'-set*[*OF assms nil*[*OF assms*], *of k*] **by** *auto*

4.4.3 Calculating Acyclic Paths

fun *acyclic-paths-up-to-length'* :: ('*a*,'*b*,'*c*) *path* $\Rightarrow$ '*a* $\Rightarrow$ ('*a* $\Rightarrow$ (('*b*$\times$'*c*$\times$'*a*) *set*)) $\Rightarrow$ '*a set* $\Rightarrow$ *nat* $\Rightarrow$ ('*a*,'*b*,'*c*) *path set*
where
acyclic-paths-up-to-length' prev q hF visitedStates 0 = {*prev*} |
acyclic-paths-up-to-length' prev q hF visitedStates (*Suc k*) =
(*let tF* = *Set.filter* (λ (*x*,*y*,*q'*) . *q'* $\notin$ *visitedStates*) (*hF q*)
in (*insert prev* ($\bigcup$ (*image* (λ (*x*,*y*,*q'*) . *acyclic-paths-up-to-length'* (*prev*@[(*q*,*x*,*y*,*q'*)]) *q' hF* (*insert q' visitedStates*) *k*) *tF*))))

fun *p-source* :: '*a* $\Rightarrow$ ('*a*,'*b*,'*c*) *path* $\Rightarrow$ '*a*
where *p-source q p* = *hd* (*visited-states q p*)

lemma *acyclic-paths-up-to-length'-prev* :
p' $\in$ *acyclic-paths-up-to-length'* (*prev*@*prev'*) *q hF visitedStates k* $\Longrightarrow$ $\exists$ *p''* . *p'* = *prev*@*p''*
by (*induction k arbitrary*: *p' q visitedStates prev'*; *auto*)

lemma *acyclic-paths-up-to-length′-set* :
assumes *path M* (*p-source q prev*) *prev*
and $\bigwedge$ *q′* . *hF q′* = {(*x*,*y*,*q″*) | *x y q″* . (*q′*,*x*,*y*,*q″*) ∈ *transitions M*}
and *distinct* (*visited-states* (*p-source q prev*) *prev*)
and *visitedStates* = *set* (*visited-states* (*p-source q prev*) *prev*)
shows *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates k*
= { *prev*@*p* | *p* . *path M* (*p-source q prev*) (*prev*@*p*)
∧ *length p* ≤ *k*
∧ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*)) }
using *assms* **proof** (*induction k arbitrary*: *q hF prev visitedStates*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)

let *?tgt* = (*target* (*p-source q prev*) *prev*)

have $\bigwedge$ *p* . (*prev*@*p*) ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
⟹ *path M* (*p-source q prev*) (*prev*@*p*)
∧ *length p* ≤ *Suc k*
∧ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*))
proof −
fix *p* **assume** (*prev*@*p*) ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
then consider (*a*) (*prev*@*p*) = *prev* |
(*b*) (*prev*@*p*) ∈ (⋃ (*image* (λ (*x*,*y*,*q′*) . *acyclic-paths-up-to-length′* (*prev*@[(*?tgt*,*x*,*y*,*q′*)]) *q′ hF* (*insert q′ visitedStates*) *k*)
(*Set.filter* (λ (*x*,*y*,*q′*) . *q′* ∉ *visitedStates*) (*hF* (*target* (*p-source q prev*) *prev*)))))
by *auto*
then show *path M* (*p-source q prev*) (*prev*@*p*) ∧ *length p* ≤ *Suc k* ∧ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*))
proof (*cases*)
case *a*
then show *?thesis* **using** *Suc.prems*(*1*,*3*) **by** *auto*
next
case *b*
then obtain *x y q′* **where** ∗: (*x*,*y*,*q′*) ∈ *Set.filter* (λ (*x*,*y*,*q′*) . *q′* ∉ *visitedStates*) (*hF ?tgt*)
and ∗∗:(*prev*@*p*) ∈ *acyclic-paths-up-to-length′* (*prev*@[(*?tgt*,*x*,*y*,*q′*)]) *q′ hF* (*insert q′ visitedStates*) *k*
by *auto*

let *?t* = (*?tgt*,*x*,*y*,*q′*)

from ∗ **have** *?t* ∈ *transitions M* **and** *q′* ∉ *visitedStates*
using *Suc.prems*(*2*)[*of ?tgt*] **by** *simp*+

moreover have *t-source ?t* = *target* (*p-source q prev*) *prev*
by *simp*
moreover have *p-source* (*p-source q prev*) (*prev*@[*?t*]) = *p-source q prev*
by *auto*
ultimately have *p1*: *path M* (*p-source* (*p-source q prev*) (*prev*@[*?t*])) (*prev*@[*?t*])

using *Suc.prems*(*1*)
by (*simp add*: *path-append-transition*)

have $q' \notin$ *set* (*visited-states* (*p-source q prev*) *prev*)
using ‹$q' \notin$ *visitedStates*› *Suc.prems*(*4*) **by** *auto*
then have *p2*: *distinct* (*visited-states* (*p-source* (*p-source q prev*) (*prev*@[*?t*])) (*prev*@[*?t*]))
using *Suc.prems*(*3*) **by** *auto*

have *p3*: (*insert* q' *visitedStates*)
= *set* (*visited-states* (*p-source* (*p-source q prev*) (*prev*@[*?t*])) (*prev*@[*?t*]))
using *Suc.prems*(*4*) **by** *auto*

have ∗∗∗: (*target* (*p-source* (*p-source q prev*) (*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]))
(*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]))
= q'
by *auto*

show *?thesis*
using *Suc.IH*[*OF p1 Suc.prems*(*2*) *p2 p3*] ∗∗
unfolding ∗∗∗
unfolding ‹*p-source* (*p-source q prev*) (*prev*@[*?t*]) = *p-source q prev*›
proof −
assume *acyclic-paths-up-to-length'* (*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) q' *hF* (*insert* q' *visitedStates*) *k*
= {(*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *p* |*p*.
path M (*p-source q prev*) ((*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *p*)
∧ *length p* ≤ *k*
∧ *distinct* (*visited-states* (*p-source q prev*) ((*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *p*))}
then have ∃ *ps*. *prev* @ *p* = (*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *ps*
∧ *path M* (*p-source q prev*) ((*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *ps*)
∧ *length ps* ≤ *k*
∧ *distinct* (*visited-states* (*p-source q prev*) ((*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) @ *ps*))
using ‹*prev* @ *p* ∈ *acyclic-paths-up-to-length'* (*prev* @ [(*target* (*p-source q prev*) *prev*, *x*, *y*, q')]) q' *hF* (*insert* q' *visitedStates*) *k*›

by *blast*
then show *?thesis*
by (*metis* (*no-types*) *Suc-le-mono append.assoc append.right-neutral append-Cons length-Cons same-append-eq*)
qed
qed
qed
moreover have $\bigwedge$ *p′* . *p′* ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
⟹ ∃ *p″* . *p′* = *prev*@*p″*
using *acyclic-paths-up-to-length′-prev*[*of - prev* [] *target* (*p-source q prev*) *prev hF visitedStates Suc k*]
by *force*
ultimately have *fwd*: $\bigwedge$ *p′* . *p′* ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
⟹ *p′* ∈ { *prev*@*p* | *p* . *path M* (*p-source q prev*) (*prev*@*p*)
∧ *length p* ≤ *Suc k*
∧ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*)) }
by *blast*

have $\bigwedge$ *p* . *path M* (*p-source q prev*) (*prev*@*p*)
⟹ *length p* ≤ *Suc k*
⟹ *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*))
⟹ (*prev*@*p*) ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
proof −
fix *p* **assume** *path M* (*p-source q prev*) (*prev*@*p*)
and *length p* ≤ *Suc k*
and *distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*))

show (*prev*@*p*) ∈ *acyclic-paths-up-to-length′ prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
proof (*cases p*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*Cons t p′*)

then have *t-source t* = *target* (*p-source q* (*prev*)) (*prev*) **and** *t* ∈ *transitions M*
using ‹*path M* (*p-source q prev*) (*prev*@*p*)› **by** *auto*

have *path M* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])@*p′*)
and *path M* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*]))
using *Cons* ‹*path M* (*p-source q prev*) (*prev*@*p*)› **by** *auto*
have *length p′* ≤ *k*
using *Cons* ‹*length p* ≤ *Suc k*› **by** *auto*
have *distinct* (*visited-states* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])@*p′*))

and *distinct* (*visited-states* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])))
using *Cons* ‹*distinct* (*visited-states* (*p-source q prev*) (*prev*@*p*))› **by** *auto*
then have *t-target t* ∉ *visitedStates*
using *Suc.prems*(*4*) **by** *auto*

let *?vN* = *insert* (*t-target t*) *visitedStates*
have *?vN* = *set* (*visited-states* (*p-source q* (*prev* @ [*t*])) (*prev* @ [*t*]))
using *Suc.prems*(*4*) **by** *auto*

have *prev*@*p* = *prev*@([*t*]@*p*′)
using *Cons* **by** *auto*

have (*prev*@[*t*])@*p*′ ∈ *acyclic-paths-up-to-length*′ (*prev* @ [*t*]) (*target* (*p-source q* (*prev* @ [*t*])) (*prev* @ [*t*])) *hF* (*insert* (*t-target t*) *visitedStates*) *k*
using *Suc.IH*[*of q prev*@[*t*], *OF* ‹*path M* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*]))› *Suc.prems*(*2*)
‹*distinct* (*visited-states* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])))›
‹*?vN* = *set* (*visited-states* (*p-source q* (*prev* @ [*t*])) (*prev* @ [*t*]))›]
using ‹*path M* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])@*p*′)›
‹*length p*′ ≤ *k*›
‹*distinct* (*visited-states* (*p-source q* (*prev*@[*t*])) ((*prev*@[*t*])@*p*′))›
by *force*

then have (*prev*@[*t*])@*p*′ ∈ *acyclic-paths-up-to-length*′ (*prev*@[*t*]) (*t-target t*) *hF ?vN k*
by *auto*
moreover have (*t-input t*,*t-output t*, *t-target t*) ∈ *Set.filter* (λ (*x*,*y*,*q*′) . *q*′ ∉ *visitedStates*) (*hF* (*t-source t*))
using *Suc.prems*(*2*)[*of t-source t*] ‹*t* ∈ *transitions M*› ‹*t-target t* ∉ *visitedStates*›
proof −
have ∃ *b c a*. *snd t* = (*b*, *c*, *a*) ∧ (*t-source t*, *b*, *c*, *a*) ∈ *FSM.transitions M*
by (*metis* (*no-types*) ‹*t* ∈ *FSM.transitions M*› *prod.collapse*)
then show *?thesis*
using ‹*hF* (*t-source t*) = {(*x*, *y*, *q*″) |*x y q*″. (*t-source t*, *x*, *y*, *q*″) ∈ *FSM.transitions M*}›
‹*t-target t* ∉ *visitedStates*›
by *fastforce*
qed
ultimately have ∃ (*x*,*y*,*q*′) ∈ (*Set.filter* (λ (*x*,*y*,*q*′) . *q*′ ∉ *visitedStates*) (*hF* (*target* (*p-source q prev*) *prev*))) .
(*prev*@[*t*])@*p*′ ∈ (*acyclic-paths-up-to-length*′ (*prev*@[((*target* (*p-source q prev*) *prev*),*x*,*y*,*q*′)]) *q*′ *hF* (*insert q*′ *visitedStates*) *k*)
unfolding ‹*t-source t* = *target* (*p-source q* (*prev*)) (*prev*)›
by (*metis* (*no-types*, *lifting*) ‹*t-source t* = *target* (*p-source q prev*) *prev*› *case-prodI prod.collapse*)

then show *?thesis* **unfolding** ‹*prev@p* = *prev@[t]@p'*›
unfolding *acyclic-paths-up-to-length'.simps Let-def* **by** *force*
qed
qed
then have *rev*: $\bigwedge$ *p'* . *p'* $\in$ {*prev@p* | *p* . *path M* (*p-source q prev*) (*prev@p*)
$\wedge$ *length p* $\leq$ *Suc k*
$\wedge$ *distinct* (*visited-states* (*p-source q prev*) (*prev@p*))}
$\Longrightarrow$ *p'* $\in$ *acyclic-paths-up-to-length' prev* (*target* (*p-source q prev*) *prev*) *hF visitedStates* (*Suc k*)
by *blast*

show *?case*
using *fwd rev* **by** *blast*
qed

fun *acyclic-paths-up-to-length* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ *nat* $\Rightarrow$ ('*a*,'*b*,'*c*) *path set*
where
acyclic-paths-up-to-length M q k = {*p*. *path M q p* $\wedge$ *length p* $\leq$ *k* $\wedge$ *distinct* (*visited-states q p*)}

lemma *acyclic-paths-up-to-length-code*[*code*] :
acyclic-paths-up-to-length M q k = (*if q* $\in$ *states M*
then acyclic-paths-up-to-length' [] *q* (*m2f* (*set-as-map* (*transitions M*))) {*q*} *k*
else {})
proof (*cases q* $\in$ *states M*)
case *False*
then have *acyclic-paths-up-to-length M q k* = {}
using *path-begin-state* **by** *fastforce*
then show *?thesis* **using** *False* **by** *auto*
next
case *True*
then have $*$: *path M* (*p-source q* []) [] **by** *auto*
have $**$: ($\bigwedge$*q'*. (*m2f* (*set-as-map* (*transitions M*))) *q'* = {(*x*, *y*, *q''*) |*x y q''*. (*q'*, *x*, *y*, *q''*) $\in$ *FSM.transitions M*})
unfolding *set-as-map-def* **by** *auto*
have $***$: *distinct* (*visited-states* (*p-source q* []) [])
by *auto*
have $****$: {*q*} = *set* (*visited-states* (*p-source q* []) [])
by *auto*

show *?thesis*
using *acyclic-paths-up-to-length'-set*[*OF* $*$ $**$ $***$ $****$, *of k*]
using *True* **by** *auto*
qed

lemma *path-map-target* : *target* (*f4 q*) (*map* (λ *t* . (*f1* (*t-source t*), *f2* (*t-input t*),

f3 (*t-output t*), *f4* (*t-target t*))) *p*) = *f4* (*target q p*)
  by (*induction p*; *auto*)

lemma *path-length-sum* :
  assumes *path M q p*
  shows *length p* = ($\sum$ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *p*))
  using *assms*
proof (*induction p rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc x xs*)
  then have *length xs* = ($\sum$ *q*∈*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) *xs*))
    by *auto*

  have ∗: *t-target x* ∈ *states M*
    using ‹*path M q* (*xs* @ [*x*])› **by** *auto*
  then have ∗∗: *length* (*filter* (λ*t*. *t-target t* = *t-target x*) (*xs* @ [*x*]))
              = *Suc* (*length* (*filter* (λ*t*. *t-target t* = *t-target x*) *xs*))
    by *auto*

  have ⋀ *q* . *q* ∈ *states M* ⟹ *q* ≠ *t-target x*
      ⟹ *length* (*filter* (λ*t*. *t-target t* = *q*) (*xs* @ [*x*])) = *length* (*filter* (λ*t*. *t-target t* = *q*) *xs*)
    by *simp*
  then have ∗∗∗: ($\sum$ *q*∈*states M* − {*t-target x*}. *length* (*filter* (λ*t*. *t-target t* = *q*) (*xs* @ [*x*])))
              = ($\sum$ *q*∈*states M* − {*t-target x*}. *length* (*filter* (λ*t*. *t-target t* = *q*) *xs*))
    using *fsm-states-finite*[*of M*]
    by (*metis* (*no-types*, *lifting*) *DiffE insertCI sum.cong*)

  have ($\sum$ *q*∈*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) (*xs* @ [*x*])))
        = ($\sum$ *q*∈*states M* − {*t-target x*}. *length* (*filter* (λ*t*. *t-target t* = *q*) (*xs* @ [*x*])))
          + (*length* (*filter* (λ*t*. *t-target t* = *t-target x*) (*xs* @ [*x*])))
    using ∗ *fsm-states-finite*[*of M*]
  proof −
    have ($\sum$ *a*∈*insert* (*t-target x*) (*states M*). *length* (*filter* (λ*p*. *t-target p* = *a*) (*xs* @ [*x*])))
          = ($\sum$ *a*∈*states M*. *length* (*filter* (λ*p*. *t-target p* = *a*) (*xs* @ [*x*])))
      by (*simp add*: ‹*t-target x* ∈ *states M*› *insert-absorb*)
    then show *?thesis*
      by (*simp add*: ‹*finite* (*states M*)› *sum.insert-remove*)
  qed
  moreover have ($\sum$ *q*∈*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) *xs*))
              = ($\sum$ *q*∈*states M* − {*t-target x*}. *length* (*filter* (λ*t*. *t-target t* = *q*) *xs*))

$+ (length\ (filter\ (\lambda t.\ t\text{-}target\ t = t\text{-}target\ x)\ xs))$
using $*$ *fsm-states-finite*[*of M*]
proof −
have $(\sum a{\in}insert\ (t\text{-}target\ x)\ (states\ M).\ length\ (filter\ (\lambda p.\ t\text{-}target\ p = a)\ xs))$

$= (\sum a{\in}states\ M.\ length\ (filter\ (\lambda p.\ t\text{-}target\ p = a)\ xs))$
by (*simp add*: ‹*t-target x* ∈ *states M*› *insert-absorb*)
then show *?thesis*
by (*simp add*: ‹*finite* (*states M*)› *sum.insert-remove*)
qed

ultimately have $(\sum q{\in}states\ M.\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ (xs\ @\ [x])))$
$= Suc\ (\sum q{\in}states\ M.\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ xs))$
using $**$ $***$ **by** *auto*

then show *?case*
by (*simp add*: ‹*length xs* = $(\sum q{\in}states\ M.\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ xs))$›)
qed

lemma *path-loop-cut* :
assumes *path M q p*
and *t-target* (*p* ! *i*) = *t-target* (*p* ! *j*)
and *i* < *j*
and *j* < *length p*
shows *path M q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*))
and *target q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) = *target q p*
and *length* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) < *length p*
and *path M* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*))
and *target* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) = (*target q* (*take* (*Suc i*) *p*))
proof −

have *p* = (*take* (*Suc j*) *p*) @ (*drop* (*Suc j*) *p*)
by *auto*
also have ... = ((*take* (*Suc i*) (*take* (*Suc j*) *p*)) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*))) @ (*drop* (*Suc j*) *p*)
by (*metis append-take-drop-id*)
also have ... = ((*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*))) @ (*drop* (*Suc j*) *p*)
using ‹*i* < *j*› **by** *simp*
finally have *p* = (*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) @ (*drop* (*Suc j*) *p*)
by *simp*

then have *path M q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) @ (*drop* (*Suc j*) *p*))
and *path M q* (((*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*))) @ (*drop*

(*Suc j*) *p*))
using ‹*path M q p*› **by** *auto*

have *path M q* (*take* (*Suc i*) *p*) **and** *path M* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*) @ *drop* (*Suc j*) *p*)
using *path-append-elim*[*OF* ‹*path M q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) @ (*drop* (*Suc j*) *p*))›]
by *blast*+

have ∗: (*take* (*Suc i*) *p* @ *drop* (*Suc i*) (*take* (*Suc j*) *p*)) = (*take* (*Suc j*) *p*)
using ‹*i* < *j*› *append-take-drop-id*
by (*metis* ‹(*take* (*Suc i*) (*take* (*Suc j*) *p*) @ *drop* (*Suc i*) (*take* (*Suc j*) *p*)) @ *drop* (*Suc j*) *p* = (*take* (*Suc i*) *p* @ *drop* (*Suc i*) (*take* (*Suc j*) *p*)) @ *drop* (*Suc j*) *p*› *append-same-eq*)

have *path M q* (*take* (*Suc j*) *p*) **and** *path M* (*target q* (*take* (*Suc j*) *p*)) (*drop* (*Suc j*) *p*)
using *path-append-elim*[*OF* ‹*path M q* (((*take* (*Suc i*) *p*) @ (*drop* (*Suc i*) (*take* (*Suc j*) *p*))) @ (*drop* (*Suc j*) *p*))›]
unfolding ∗
by *blast*+

have ∗∗: (*target q* (*take* (*Suc j*) *p*)) = (*target q* (*take* (*Suc i*) *p*))
proof −
have *p* ! *i* = *last* (*take* (*Suc i*) *p*)
by (*metis Suc-lessD assms*(*3*) *assms*(*4*) *less-trans-Suc take-last-index*)
moreover have *p* ! *j* = *last* (*take* (*Suc j*) *p*)
by (*simp add*: *assms*(*4*) *take-last-index*)
ultimately show *?thesis*
using *assms*(*2*) **unfolding** ∗ *target.simps visited-states.simps*
by (*simp add*: *last-map*)
qed

show *path M q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*))
using ‹*path M q* (*take* (*Suc i*) *p*)› ‹*path M* (*target q* (*take* (*Suc j*) *p*)) (*drop* (*Suc j*) *p*)› **unfolding** ∗∗ **by** *auto*

show *target q* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) = *target q p*
by (*metis* ∗∗ *append-take-drop-id path-append-target*)

show *length* ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*)) < *length p*
proof −
have ∗∗∗: *length p* = *length* ((*take* (*Suc j*) *p*) @ (*drop* (*Suc j*) *p*))
by *auto*

have *length* (*take* (*Suc i*) *p*) < *length* (*take* (*Suc j*) *p*)
using *assms*(*3*,*4*)
by (*simp add*: *min-absorb2*)

have *scheme*: $\bigwedge$ *a b c* . *length a* < *length b* $\Longrightarrow$ *length* (*a*@*c*) < *length* (*b*@*c*)
by *auto*

show *?thesis*
unfolding ∗∗∗ **using** *scheme*[*OF* ‹*length* (*take* (*Suc i*) *p*) < *length* (*take* (*Suc j*) *p*)›, *of* (*drop* (*Suc j*) *p*)]
by *assumption*
qed

show *path M* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*))
using ‹*path M* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*) @ *drop* (*Suc j*) *p*)› **by** *blast*

show *target* (*target q* (*take* (*Suc i*) *p*)) (*drop* (*Suc i*) (*take* (*Suc j*) *p*)) = (*target q* (*take* (*Suc i*) *p*))
by (*metis* ∗ ∗∗ *path-append-target*)
qed

lemma *path-prefix-take* :
assumes *path M q p*
shows *path M q* (*take i p*)
proof −
have *p* = (*take i p*)@(*drop i p*) **by** *auto*
then have *path M q* ((*take i p*)@(*drop i p*)) **using** *assms* **by** *auto*
then show *?thesis*
by *blast*
qed

4.5 Acyclic Paths

lemma *cyclic-path-loop* :
assumes *path M q p*
and ¬ *distinct* (*visited-states q p*)
shows ∃ *p1 p2 p3* . *p* = *p1*@*p2*@*p3* ∧ *p2* ≠ [] ∧ *target q p1* = *target q* (*p1*@*p2*)
using *assms* **proof** (*induction p arbitrary*: *q*)
case (*nil M q*)
then show *?case* **by** *auto*
next
case (*cons t M ts*)
then show *?case*
proof (*cases distinct* (*visited-states* (*t-target t*) *ts*))
case *True*
then have *q* ∈ *set* (*visited-states* (*t-target t*) *ts*)
using *cons.prems* **by** *simp*
then obtain *p2 p3* **where** *ts* = *p2*@*p3* **and** *target* (*t-target t*) *p2* = *q*
using *visited-states-prefix*[*of q t-target t ts*] **by** *blast*
then have (*t*#*ts*) = []@(*t*#*p2*)@*p3* ∧ (*t*#*p2*) ≠ [] ∧ *target q* [] = *target q*

```
([]@(t#p2))
      using cons.hyps by auto
    then show ?thesis by blast
  next
    case False
    then obtain p1 p2 p3 where ts = p1@p2@p3 and p2 ≠ []
                          and target (t-target t) p1 = target (t-target t) (p1@p2)
      using cons.IH by blast
     then have t#ts = (t#p1)@p2@p3 ∧ p2 ≠ [] ∧ target q (t#p1) = target q
((t#p1)@p2)
      by simp
    then show ?thesis by blast
  qed
qed


lemma cyclic-path-pumping :
  assumes path M (initial M) p
      and ¬ distinct (visited-states (initial M) p)
  shows ∃ p . path M (initial M) p ∧ length p ≥ n
proof −
  from assms obtain p1 p2 p3 where p = p1 @ p2 @ p3 and p2 ≠ []
                            and target (initial M) p1 = target (initial M) (p1 @ p2)
    using cyclic-path-loop[of M initial M p] by blast
  then have path M (target (initial M) p1) p3
    using path-suffix[of M initial M p1@p2 p3] ‹path M (initial M) p› by auto

  have path M (initial M) p1
    using path-prefix[of M initial M p1 p2@p3] ‹path M (initial M) p› ‹p = p1 @
p2 @ p3›
    by auto
  have path M (initial M) ((p1@p2)@p3)
    using ‹path M (initial M) p› ‹p = p1 @ p2 @ p3›
    by auto
  have path M (target (initial M) p1) p2
     using path-suffix[of M initial M p1 p2, OF path-prefix[of M initial M p1@p2
p3, OF ‹path M (initial M) ((p1@p2)@p3)›]]
    by assumption
  have target (target (initial M) p1) p2 = (target (initial M) p1)
   using path-append-target ‹target (initial M) p1 = target (initial M) (p1 @ p2)›

    by auto

  have path M (initial M) (p1 @ (concat (replicate n p2)) @ p3)
  proof (induction n)
    case 0
    then show ?case
      using path-append[OF ‹path M (initial M) p1› ‹path M (target (initial M)
p1) p3›]
```

by *auto*
next
case (*Suc n*)
then show *?case*
using ‹*path M* (*target* (*initial M*) *p1*) *p2*› ‹*target* (*target* (*initial M*) *p1*) *p2* = *target* (*initial M*) *p1*›
by *auto*
qed
moreover have *length* (*p1* @ (*concat* (*replicate n p2*)) @ *p3*) $\geq$ *n*
proof −
have *length* (*concat* (*replicate n p2*)) = *n* $*$ (*length p2*)
using *concat-replicate-length* **by** *metis*
moreover have *length p2* > *0*
using ‹*p2* $\neq$ []› **by** *auto*
ultimately have *length* (*concat* (*replicate n p2*)) $\geq$ *n*
by (*simp add*: *Suc-leI*)
then show *?thesis* **by** *auto*
qed
ultimately show $\exists$ *p* . *path M* (*initial M*) *p* $\wedge$ *length p* $\geq$ *n* **by** *blast*
qed

lemma *cyclic-path-shortening* :
assumes *path M q p*
and $\neg$ *distinct* (*visited-states q p*)
shows $\exists$ *p′* . *path M q p′* $\wedge$ *target q p′* = *target q p* $\wedge$ *length p′* < *length p*
proof −
obtain *p1 p2 p3* **where** $*$: *p* = *p1*@*p2*@*p3* $\wedge$ *p2* $\neq$ [] $\wedge$ *target q p1* = *target q* (*p1*@*p2*)
using *cyclic-path-loop*[*OF assms*] **by** *blast*
then have *path M q* (*p1*@*p3*)
using *assms*(*1*) **by** *force*
moreover have *target q* (*p1*@*p3*) = *target q p*
by (*metis* (*full-types*) $*$ *path-append-target*)
moreover have *length* (*p1*@*p3*) < *length p*
using $*$ **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *acyclic-path-from-cyclic-path* :
assumes *path M q p*
and $\neg$ *distinct* (*visited-states q p*)
obtains *p′* **where** *path M q p′* **and** *target q p* = *target q p′* **and** *distinct* (*visited-states q p′*)
proof −

let *?paths* = {*p′* . (*path M q p′* $\wedge$ *target q p′* = *target q p* $\wedge$ *length p′* $\leq$ *length p*)}

```
  let ?minPath = arg-min length (λ io . io ∈ ?paths)

  have ?paths ≠ empty
    using assms(1) by auto
  moreover have finite ?paths
    using paths-finite[of M q length p]
    by (metis (no-types, lifting) Collect-mono rev-finite-subset)
  ultimately have minPath-def : ?minPath ∈ ?paths ∧ (∀ p′ ∈ ?paths . length
?minPath ≤ length p′)
    by (meson arg-min-nat-lemma equals0I)
  then have path M q ?minPath and target q ?minPath = target q p
    by auto

  moreover have distinct (visited-states q ?minPath)
  proof (rule ccontr)
    assume ¬ distinct (visited-states q ?minPath)
   have ∃ p′ . path M q p′ ∧ target q p′ = target q p ∧ length p′ < length ?minPath

    using cyclic-path-shortening[OF ‹path M q ?minPath› ‹¬ distinct (visited-states
q ?minPath)›] minPath-def
          ‹target q ?minPath= target q p› by auto
    then show False
      using minPath-def using arg-min-nat-le dual-order.strict-trans1 by auto
  qed

  ultimately show ?thesis
    by (simp add: that)
qed


lemma acyclic-path-length-limit :
  assumes path M q p
  and     distinct (visited-states q p)
shows length p < size M
proof (rule ccontr)
  assume ∗: ¬ length p < size M
  then have length p ≥ card (states M)
    using size-def by auto
  then have length (visited-states q p) > card (states M)
    by auto
  moreover have set (visited-states q p) ⊆ states M
    by (metis assms(1) path-prefix path-target-is-state subsetI visited-states-prefix)
  ultimately have ¬ distinct (visited-states q p)
    using distinct-card[OF assms(2)]
    using List.finite-set[of visited-states q p]
    by (metis card-mono fsm-states-finite leD)
  then show False using assms(2) by blast
qed
```

4.6 Reachable States

definition *reachable* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ *bool* **where**
reachable M q = ($\exists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* = *q*)

definition *reachable-states* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ *set* **where**
reachable-states M = {*target* (*initial M*) *p* | *p* . *path M* (*initial M*) *p* }

abbreviation *size-r M* $\equiv$ *card* (*reachable-states M*)

lemma *acyclic-paths-set* :
acyclic-paths-up-to-length M q (*size M* − *1*) = {*p* . *path M q p* $\wedge$ *distinct* (*visited-states q p*)}
unfolding *acyclic-paths-up-to-length.simps* **using** *acyclic-path-length-limit*[*of M q*]
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred cyclic-path-shortening leD list.size*(*3*)
not-less-eq-eq not-less-zero path.intros(*1*) *path-begin-state*)

lemma *reachable-states-code*[*code*] :
reachable-states M = *image* (*target* (*initial M*)) (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
proof −
have $\bigwedge$ *q'* . *q'* $\in$ *reachable-states M*
$\Longrightarrow$ *q'* $\in$ *image* (*target* (*initial M*)) (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
proof −
fix *q'* **assume** *q'* $\in$ *reachable-states M*
then obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *q'*
unfolding *reachable-states-def* **by** *blast*

obtain *p'* **where** *path M* (*initial M*) *p'* **and** *target* (*initial M*) *p'* = *q'*
and *distinct* (*visited-states* (*initial M*) *p'*)
proof (*cases distinct* (*visited-states* (*initial M*) *p*))
case *True*
then show *?thesis* **using** ‹*path M* (*initial M*) *p*› ‹*target* (*initial M*) *p* = *q'*› *that* **by** *auto*
next
case *False*
then show *?thesis*
using *acyclic-path-from-cyclic-path*[*OF* ‹*path M* (*initial M*) *p*›]
unfolding ‹*target* (*initial M*) *p* = *q'*› **using** *that* **by** *blast*
qed
then show *q'* $\in$ *image* (*target* (*initial M*)) (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
unfolding *acyclic-paths-set* **by** *force*
qed
moreover have $\bigwedge$ *q'* . *q'* $\in$ *image* (*target* (*initial M*)) (*acyclic-paths-up-to-length*

M (*initial M*) (*size M* − *1*))
$\Longrightarrow q' \in$ *reachable-states M*
unfolding *reachable-states-def acyclic-paths-set* **by** *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *reachable-states-intro*[*intro*!] :
assumes *path M* (*initial M*) *p*
shows *target* (*initial M*) *p* ∈ *reachable-states M*
using *assms* **unfolding** *reachable-states-def* **by** *auto*

lemma *reachable-states-initial* :
initial M ∈ *reachable-states M*
unfolding *reachable-states-def* **by** *auto*

lemma *reachable-states-next* :
assumes *q* ∈ *reachable-states M* **and** *t* ∈ *transitions M* **and** *t-source t* = *q*
shows *t-target t* ∈ *reachable-states M*
proof −
from ‹*q* ∈ *reachable-states M*› **obtain** *p* **where** ∗ :*path M* (*initial M*) *p*
and ∗∗:*target* (*initial M*) *p* = *q*
unfolding *reachable-states-def* **by** *auto*

then have *path M* (*initial M*) (*p*@[*t*]) **using** *assms*(*2*,*3*) *path-append-transition*
by *metis*
moreover have *target* (*initial M*) (*p*@[*t*]) = *t-target t* **by** *auto*
ultimately show *?thesis*
unfolding *reachable-states-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
qed

lemma *reachable-states-path* :
assumes *q* ∈ *reachable-states M*
and *path M q p*
and *t* ∈ *set p*
shows *t-source t* ∈ *reachable-states M*
using *assms* **unfolding** *reachable-states-def* **proof** (*induction p arbitrary*: *q*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons t′ p′*)
then show *?case* **proof** (*cases t* = *t′*)
case *True*
then show *?thesis* **using** *Cons.prems*(*1*,*2*) **by** *force*

next
case *False* **then show** *?thesis* **using** *Cons*
by (*metis* (*mono-tags*, *lifting*) *path-cons-elim reachable-states-def reachable-states-next*
set-ConsD)
qed
qed

lemma *reachable-states-initial-or-target* :
assumes $q \in$ *reachable-states M*
shows $q =$ *initial M* $\vee$ ($\exists$ $t \in$ *transitions M* . *t-source t* $\in$ *reachable-states M* $\wedge$ *t-target t* $= q$)
proof −
obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* $= q$
using *assms* **unfolding** *reachable-states-def* **by** *auto*

show *?thesis* **proof** (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis* **using** ‹*path M* (*initial M*) *p*› ‹*target* (*initial M*) *p* $= q$› **by** *auto*
next
case (*snoc p′ t*)

have $t \in$ *transitions M*
using ‹*path M* (*initial M*) *p*› **unfolding** *snoc* **by** *auto*
moreover have *t-target t* $= q$
using ‹*target* (*initial M*) *p* $= q$› **unfolding** *snoc* **by** *auto*
moreover have *t-source t* $\in$ *reachable-states M*
using ‹*path M* (*initial M*) *p*› **unfolding** *snoc*
by (*metis append-is-Nil-conv last-in-set last-snoc not-Cons-self2 reachable-states-initial reachable-states-path*)

ultimately show *?thesis*
by *blast*
qed
qed

lemma *reachable-state-is-state* :
$q \in$ *reachable-states M* $\Longrightarrow q \in$ *states M*
unfolding *reachable-states-def* **using** *path-target-is-state* **by** *fastforce*

lemma *reachable-states-finite* : *finite* (*reachable-states M*)
using *fsm-states-finite*[*of M*] *reachable-state-is-state*[*of - M*]
by (*meson finite-subset subset-eq*)

4.7 Language

abbreviation *p-io* (*p* :: ('*state*,'*input*,'*output*) *path*) ≡ *map* (λ *t* . (*t-input t*, *t-output t*)) *p*

fun *language-state-for-input* :: ('*state*,'*input*,'*output*) *fsm* ⇒ '*state* ⇒ '*input list* ⇒ ('*input* × '*output*) *list set* **where**
language-state-for-input M q xs = {*p-io p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *xs*}

fun LS_{in} :: ('*state*,'*input*,'*output*) *fsm* ⇒ '*state* ⇒ '*input list set* ⇒ ('*input* × '*output*) *list set* **where**
LS_{in} *M q xss* = {*p-io p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) ∈ *xss*}

abbreviation(*input*) L_{in} *M* ≡ LS_{in} *M* (*initial M*)

lemma *language-state-for-input-inputs* :
assumes *io* ∈ *language-state-for-input M q xs*
shows *map fst io* = *xs*
using *assms* **by** *auto*

lemma *language-state-for-inputs-inputs* :
assumes *io* ∈ LS_{in} *M q xss*
shows *map fst io* ∈ *xss* **using** *assms* **by** *auto*

fun *LS* :: ('*state*,'*input*,'*output*) *fsm* ⇒ '*state* ⇒ ('*input* × '*output*) *list set* **where**
LS M q = { *p-io p* | *p* . *path M q p* }

abbreviation *L M* ≡ *LS M* (*initial M*)

lemma *language-state-containment* :
assumes *path M q p*
and *p-io p* = *io*
shows *io* ∈ *LS M q*
using *assms* **by** *auto*

lemma *language-prefix* :
assumes *io1@io2* ∈ *LS M q*
shows *io1* ∈ *LS M q*
proof −
obtain *p* **where** *path M q p* **and** *p-io p* = *io1@io2*
using *assms* **by** *auto*
let *?tp* = *take* (*length io1*) *p*
have *path M q ?tp*
by (*metis* (*no-types*) ‹*path M q p*› *append-take-drop-id path-prefix*)
moreover have *p-io ?tp* = *io1*
using ‹*p-io p* = *io1@io2*› **by** (*metis append-eq-conv-conj take-map*)
ultimately show *?thesis*
by *force*

qed

lemma *language-contains-empty-sequence* : $[] \in L\ M$
  by *auto*

lemma *language-state-split* :
  assumes $io1\ @\ io2 \in LS\ M\ q1$
  obtains $p1\ p2$ **where** $path\ M\ q1\ p1$
    and $path\ M\ (target\ q1\ p1)\ p2$
    and $p\text{-}io\ p1 = io1$
    and $p\text{-}io\ p2 = io2$
proof −
  obtain $p12$ **where** $path\ M\ q1\ p12$ **and** $p\text{-}io\ p12 = io1\ @\ io2$
    using *assms* **unfolding** *LS.simps* **by** *auto*

  let $?p1 = take\ (length\ io1)\ p12$
  let $?p2 = drop\ (length\ io1)\ p12$

  have $p12 = ?p1\ @\ ?p2$
    by *auto*
  then have $path\ M\ q1\ (?p1\ @\ ?p2)$
    using ‹$path\ M\ q1\ p12$› **by** *auto*

  have $path\ M\ q1\ ?p1$ **and** $path\ M\ (target\ q1\ ?p1)\ ?p2$
    using *path-append-elim*[*OF* ‹$path\ M\ q1\ (?p1\ @\ ?p2)$›] **by** *blast+*
  moreover have $p\text{-}io\ ?p1 = io1$
    using ‹$p12 = ?p1\ @\ ?p2$› ‹$p\text{-}io\ p12 = io1\ @\ io2$›
    by (*metis append-eq-conv-conj take-map*)
  moreover have $p\text{-}io\ ?p2 = io2$
    using ‹$p12 = ?p1\ @\ ?p2$› ‹$p\text{-}io\ p12 = io1\ @\ io2$›
    by (*metis* (*no-types*) ‹$p\text{-}io\ p12 = io1\ @\ io2$› *append-eq-conv-conj drop-map*)
  ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *language-initial-path-append-transition* :
  assumes $ios\ @\ [io] \in L\ M$
  obtains $p\ t$ **where** $path\ M\ (initial\ M)\ (p@[t])$ **and** $p\text{-}io\ (p@[t]) = ios\ @\ [io]$
proof −
  obtain pt **where** $path\ M\ (initial\ M)\ pt$ **and** $p\text{-}io\ pt = ios\ @\ [io]$
    using *assms* **unfolding** *LS.simps* **by** *auto*
  then have $pt \neq []$
    by *auto*
  then obtain $p\ t$ **where** $pt = p\ @\ [t]$
    using *rev-exhaust* **by** *blast*
  then have $path\ M\ (initial\ M)\ (p@[t])$ **and** $p\text{-}io\ (p@[t]) = ios\ @\ [io]$
    using ‹$path\ M\ (initial\ M)\ pt$› ‹$p\text{-}io\ pt = ios\ @\ [io]$› **by** *auto*
  then show *?thesis* **using** *that* **by** *simp*

qed

lemma *language-path-append-transition* :
  assumes *ios* @ [*io*] ∈ *LS M q*
  obtains *p t* **where** *path M q* (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *ios* @ [*io*]
proof −
  obtain *pt* **where** *path M q pt* **and** *p-io pt* = *ios* @ [*io*]
    using *assms* **unfolding** *LS.simps* **by** *auto*
  then have *pt* ≠ []
    by *auto*
  then obtain *p t* **where** *pt* = *p* @ [*t*]
    using *rev-exhaust* **by** *blast*
  then have *path M q* (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *ios* @ [*io*]
    using ‹*path M q pt*› ‹*p-io pt* = *ios* @ [*io*]› **by** *auto*
  then show *?thesis* **using** *that* **by** *simp*
qed

lemma *language-split* :
  assumes *io1*@*io2* ∈ *L M*
  obtains *p1 p2* **where** *path M* (*initial M*) (*p1*@*p2*) **and** *p-io p1* = *io1* **and** *p-io p2* = *io2*
proof −
  from *assms* **obtain** *p* **where** *path M* (*initial M*) *p* **and** *p-io p* = *io1* @ *io2*
    by *auto*

  let *?p1* = *take* (*length io1*) *p*
  let *?p2* = *drop* (*length io1*) *p*

  have *path M* (*initial M*) (*?p1*@*?p2*)
    using ‹*path M* (*initial M*) *p*› **by** *simp*
  moreover have *p-io ?p1* = *io1*
    using ‹*p-io p* = *io1* @ *io2*›
    by (*metis append-eq-conv-conj take-map*)
  moreover have *p-io ?p2* = *io2*
    using ‹*p-io p* = *io1* @ *io2*›
    by (*metis append-eq-conv-conj drop-map*)
  ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *language-io* :
  assumes *io* ∈ *LS M q*
  and (*x*,*y*) ∈ *set io*
shows *x* ∈ (*inputs M*)
and *y* ∈ *outputs M*
proof −
  obtain *p* **where** *path M q p* **and** *p-io p* = *io*

```
    using ‹io ∈ LS M q› by auto
  then obtain t where t ∈ set p and t-input t = x and t-output t = y
    using ‹(x,y) ∈ set io› by auto

  have t ∈ transitions M
    using ‹path M q p› ‹t ∈ set p›
    by (induction p; auto)

  show x ∈ (inputs M)
    using ‹t ∈ transitions M› ‹t-input t = x› by auto

  show y ∈ outputs M
    using ‹t ∈ transitions M› ‹t-output t = y› by auto
qed


lemma path-io-split :
  assumes path M q p
  and     p-io p = io1@io2
shows path M q (take (length io1) p)
and   p-io (take (length io1) p) = io1
and   path M (target q (take (length io1) p)) (drop (length io1) p)
and   p-io (drop (length io1) p) = io2
proof −
  have length io1 ≤ length p
    using ‹p-io p = io1@io2›
    unfolding length-map[of (λ t . (t-input t, t-output t)), symmetric]
    by auto

  have p = (take (length io1) p)@(drop (length io1) p)
    by simp
  then have ∗: path M q ((take (length io1) p)@(drop (length io1) p))
    using ‹path M q p› by auto

  show path M q (take (length io1) p)
      and  path M (target q (take (length io1) p)) (drop (length io1) p)
    using path-append-elim[OF ∗] by blast+

  show p-io (take (length io1) p) = io1
    using ‹p = (take (length io1) p)@(drop (length io1) p)› ‹p-io p = io1@io2›
    by (metis append-eq-conv-conj take-map)

  show p-io (drop (length io1) p) = io2
    using ‹p = (take (length io1) p)@(drop (length io1) p)› ‹p-io p = io1@io2›
    by (metis append-eq-conv-conj drop-map)
qed


lemma language-intro :
```

assumes *path M q p*
shows *p-io p* ∈ *LS M q*
using *assms* **unfolding** *LS.simps* **by** *auto*

lemma *language-prefix-append* :
assumes *io1* @ (*p-io p*) ∈ *L M*
shows *io1* @ *p-io* (*take i p*) ∈ *L M*
proof −
fix *i*
have *p-io p* = (*p-io* (*take i p*)) @ (*p-io* (*drop i p*))
by (*metis append-take-drop-id map-append*)
then have (*io1* @ (*p-io* (*take i p*))) @ (*p-io* (*drop i p*)) ∈ *L M*
using ‹*io1* @ *p-io p* ∈ *L M*› **by** *auto*
show *io1* @ *p-io* (*take i p*) ∈ *L M*
using *language-prefix*[*OF* ‹(*io1* @ (*p-io* (*take i p*))) @ (*p-io* (*drop i p*)) ∈ *L M*›]
by *assumption*
qed

lemma *language-finite*: *finite* {*io* . *io* ∈ *L M* ∧ *length io* ≤ *k*}
proof −
have {*io* . *io* ∈ *L M* ∧ *length io* ≤ *k*} ⊆ *p-io* ‘ {*p*. *path M* (*FSM.initial M*) *p* ∧ *length p* ≤ *k*}
by *auto*
then show *?thesis*
using *paths-finite*[*of M initial M k*]
using *finite-surj* **by** *auto*
qed

lemma *LS-prepend-transition* :
assumes *t* ∈ *transitions M*
and *io* ∈ *LS M* (*t-target t*)
shows (*t-input t*, *t-output t*) # *io* ∈ *LS M* (*t-source t*)
proof −
obtain *p* **where** *path M* (*t-target t*) *p* **and** *p-io p* = *io*
using *assms*(*2*) **by** *auto*
then have *path M* (*t-source t*) (*t*#*p*) **and** *p-io* (*t*#*p*) = (*t-input t*, *t-output t*) # *io*
using *assms*(*1*) **by** *auto*
then show *?thesis*
unfolding *LS.simps*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
qed

lemma *language-empty-IO* :
assumes *inputs M* = {} ∨ *outputs M* = {}
shows *L M* = {[]}

proof −
  consider *inputs M* = {} | *outputs M* = {} **using** *assms* **by** *blast*
  then show *?thesis* **proof** *cases*
    case *1*

    show *L M* = {[]}
      using *language-io*(*1*)[*of - M initial M*] **unfolding** *1*
    by (*metis* (*no-types*, *opaque-lifting*) *ex-in-conv is-singletonI′ is-singleton-the-elem language-contains-empty-sequence set-empty2 singleton-iff surj-pair*)
  next
    case *2*
    show *L M* = {[]}
      using *language-io*(*2*)[*of - M initial M*] **unfolding** *2*
    by (*metis* (*no-types*, *opaque-lifting*) *ex-in-conv is-singletonI′ is-singleton-the-elem language-contains-empty-sequence set-empty2 singleton-iff surj-pair*)
  qed
qed

lemma *language-equivalence-from-isomorphism-helper* :
  assumes *bij-betw f* (*states M1*) (*states M2*)
  and *f* (*initial M1*) = *initial M2*
  and ⋀ *q x y q′* . *q* ∈ *states M1* ⟹ *q′* ∈ *states M1* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q′*) ∈ *transitions M2*
  and *q* ∈ *states M1*
shows *LS M1 q* ⊆ *LS M2* (*f q*)
proof
  fix *io* **assume** *io* ∈ *LS M1 q*

  then obtain *p* **where** *path M1 q p* **and** *p-io p* = *io*
    by *auto*

  let *?f* = λ(*q*,*x*,*y*,*q′*) . (*f q*,*x*,*y*,*f q′*)
  let *?p* = *map ?f p*

  have *f q* ∈ *states M2*
    using *assms*(*1*,*4*)
    using *bij-betwE* **by** *auto*

  have *path M2* (*f q*) *?p*
  using ‹*path M1 q p*› **proof** (*induction p rule*: *rev-induct*)
    case *Nil*
    show *?case* **using** ‹*f q* ∈ *states M2*› **by** *auto*
  next
    case (*snoc a p*)
    then have *path M2* (*f q*) (*map ?f p*)
      by *auto*

    have *target* (*f q*) (*map ?f p*) = *f* (*target q p*)
      using ‹*f* (*initial M1*) = *initial M2*› *assms*(*2*)

by (*induction p*; *auto*)
then have *t-source* (*?f a*) = *target* (*f q*) (*map ?f p*)
by (*metis* (*no-types*, *lifting*) *case-prod-beta′ fst-conv path-append-transition-elim*(*3*) *snoc.prems*)

have *a* ∈ *transitions M1*
using *snoc.prems* **by** *auto*
then have *?f a* ∈ *transitions M2*
by (*metis* (*mono-tags*, *lifting*) *assms*(*3*) *case-prod-beta fsm-transition-source fsm-transition-target surjective-pairing*)

have *map ?f* (*p*@[*a*]) = (*map ?f p*)@[*?f a*]
by *auto*

show *?case*
unfolding ‹*map ?f* (*p*@[*a*]) = (*map ?f p*)@[*?f a*]›
using *path-append-transition*[*OF* ‹*path M2* (*f q*) (*map ?f p*)› ‹*?f a* ∈ *transitions M2*› ‹*t-source* (*?f a*) = *target* (*f q*) (*map ?f p*)›]
by *assumption*
qed
moreover have *p-io ?p* = *io*
using ‹*p-io p* = *io*›
by (*induction p*; *auto*)
ultimately show *io* ∈ *LS M2* (*f q*)
using *language-state-containment* **by** *fastforce*
qed

lemma *language-equivalence-from-isomorphism* :
assumes *bij-betw f* (*states M1*) (*states M2*)
and *f* (*initial M1*) = *initial M2*
and ⋀ *q x y q′* . *q* ∈ *states M1* ⟹ *q′* ∈ *states M1* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q′*) ∈ *transitions M2*
and *q* ∈ *states M1*
shows *LS M1 q* = *LS M2* (*f q*)
proof
show *LS M1 q* ⊆ *LS M2* (*f q*)
using *language-equivalence-from-isomorphism-helper*[*OF assms*] **.**

have *f q* ∈ *states M2*
using *assms*(*1*,*4*)
using *bij-betwE* **by** *auto*
have (*inv-into* (*FSM.states M1*) *f* (*f q*)) = *q*
by (*meson assms*(*1*) *assms*(*4*) *bij-betw-imp-inj-on inv-into-f-f*)

have *bij-betw* (*inv-into* (*states M1*) *f*) (*states M2*) (*states M1*)
using *bij-betw-inv-into*[*OF assms*(*1*)] **.**
moreover have (*inv-into* (*states M1*) *f*) (*initial M2*) = (*initial M1*)

using *assms*(*1*,*2*)
by (*metis bij-betw-inv-into-left fsm-initial*)
moreover have $\bigwedge$ *q x y q′* . *q* ∈ *states M2* ⟹ *q′* ∈ *states M2* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M2* ⟷ ((*inv-into* (*states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*states M1*) *f*) *q′*) ∈ *transitions M1*
proof
fix *q x y q′* **assume** *q* ∈ *states M2* **and** *q′* ∈ *states M2*

show (*q*,*x*,*y*,*q′*) ∈ *transitions M2* ⟹ ((*inv-into* (*states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*states M1*) *f*) *q′*) ∈ *transitions M1*
proof −
assume *a1*: (*q*, *x*, *y*, *q′*) ∈ *FSM.transitions M2*
have *f2*: ∀ *f B A*. ¬ *bij-betw f B A* ∨ (∀ *b*. (*b*::*′b*) ∉ *B* ∨ (*f b*::*′a*) ∈ *A*)
using *bij-betwE* **by** *blast*
then have *f3*: *inv-into* (*states M1*) *f q* ∈ *states M1*
using ‹*q* ∈ *states M2*› *calculation*(*1*) **by** *blast*
have *inv-into* (*states M1*) *f q′* ∈ *states M1*
using *f2* ‹*q′* ∈ *states M2*› *calculation*(*1*) **by** *blast*
then show *?thesis*
using *f3 a1* ‹*q* ∈ *states M2*› ‹*q′* ∈ *states M2*› *assms*(*1*) *assms*(*3*) *bij-betw-inv-into-right* **by** *fastforce*
qed

show ((*inv-into* (*states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*states M1*) *f*) *q′*) ∈ *transitions M1* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M2*
proof −
assume *a1*: (*inv-into* (*states M1*) *f q*, *x*, *y*, *inv-into* (*states M1*) *f q′*) ∈ *FSM.transitions M1*
have *f2*: ∀ *f B A*. ¬ *bij-betw f B A* ∨ (∀ *b*. (*b*::*′b*) ∉ *B* ∨ (*f b*::*′a*) ∈ *A*)
by (*metis* (*full-types*) *bij-betwE*)
then have *f3*: *inv-into* (*states M1*) *f q′* ∈ *states M1*
using ‹*q′* ∈ *states M2*› *calculation*(*1*) **by** *blast*
have *inv-into* (*states M1*) *f q* ∈ *states M1*
using *f2* ‹*q* ∈ *states M2*› *calculation*(*1*) **by** *blast*
then show *?thesis*
using *f3 a1* ‹*q* ∈ *states M2*› ‹*q′* ∈ *states M2*› *assms*(*1*) *assms*(*3*) *bij-betw-inv-into-right* **by** *fastforce*
qed
qed
ultimately show *LS M2* (*f q*) ⊆ *LS M1 q*
using *language-equivalence-from-isomorphism-helper*[*of* (*inv-into* (*states M1*) *f*) *M2 M1*, *OF* - - - ‹*f q* ∈ *states M2*›]
unfolding ‹(*inv-into* (*FSM.states M1*) *f* (*f q*)) = *q*›
by *blast*
qed

lemma *language-equivalence-from-isomorphism-helper-reachable* :

assumes *bij-betw f* (*reachable-states M1*) (*reachable-states M2*)
and *f* (*initial M1*) = *initial M2*
and $\bigwedge$ *q x y q′* . *q* ∈ *reachable-states M1* ⟹ *q′* ∈ *reachable-states M1* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q′*) ∈ *transitions M2*
shows *L M1* ⊆ *L M2*
proof
fix *io* **assume** *io* ∈ *L M1*

then obtain *p* **where** *path M1* (*initial M1*) *p* **and** *p-io p* = *io*
by *auto*

let *?f* = λ(*q*,*x*,*y*,*q′*) . (*f q*,*x*,*y*,*f q′*)
let *?p* = *map ?f p*

have *path M2* (*initial M2*) *?p*
using ‹*path M1* (*initial M1*) *p*› **proof** (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a p*)
then have *path M2* (*initial M2*) (*map ?f p*)
by *auto*

have *target* (*initial M2*) (*map ?f p*) = *f* (*target* (*initial M1*) *p*)
using ‹*f* (*initial M1*) = *initial M2*› *assms*(*2*)
by (*induction p*; *auto*)
then have *t-source* (*?f a*) = *target* (*initial M2*) (*map ?f p*)
by (*metis* (*no-types*, *lifting*) *case-prod-beta′ fst-conv path-append-transition-elim*(*3*) *snoc.prems*)

have *t-source a* ∈ *reachable-states M1*
using ‹*path M1* (*FSM.initial M1*) (*p* @ [*a*])›
by (*metis path-append-transition-elim*(*3*) *path-prefix reachable-states-intro*)
have *t-target a* ∈ *reachable-states M1*
using ‹*path M1* (*FSM.initial M1*) (*p* @ [*a*])›
by (*meson* ‹*t-source a* ∈ *reachable-states M1*› *path-append-transition-elim*(*2*) *reachable-states-next*)

have *a* ∈ *transitions M1*
using *snoc.prems* **by** *auto*
then have *?f a* ∈ *transitions M2*
using *assms*(*3*)[*OF* ‹*t-source a* ∈ *reachable-states M1*› ‹*t-target a* ∈ *reachable-states M1*›]
by (*metis* (*mono-tags*, *lifting*) *prod.case-eq-if prod.collapse*)

have *map ?f* (*p*@[*a*]) = (*map ?f p*)@[*?f a*]
by *auto*

show *?case*
unfolding ‹*map ?f* (*p*@[*a*]) = (*map ?f p*)@[*?f a*]›
using *path-append-transition*[*OF* ‹*path M2* (*initial M2*) (*map ?f p*)› ‹*?f a* ∈ *transitions M2*› ‹*t-source* (*?f a*) = *target* (*initial M2*) (*map ?f p*)›]
by *assumption*
qed
moreover have *p-io ?p* = *io*
using ‹*p-io p* = *io*›
by (*induction p*; *auto*)
ultimately show *io* ∈ *L M2*
using *language-state-containment* **by** *fastforce*
qed

lemma *language-equivalence-from-isomorphism-reachable* :
assumes *bij-betw f* (*reachable-states M1*) (*reachable-states M2*)
and *f* (*initial M1*) = *initial M2*
and ⋀ *q x y q′* . *q* ∈ *reachable-states M1* ⟹ *q′* ∈ *reachable-states M1* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M1* ⟷ (*f q*,*x*,*y*,*f q′*) ∈ *transitions M2*
shows *L M1* = *L M2*
proof
show *L M1* ⊆ *L M2*
using *language-equivalence-from-isomorphism-helper-reachable*[*OF assms*] **.**

have *bij-betw* (*inv-into* (*reachable-states M1*) *f*) (*reachable-states M2*) (*reachable-states M1*)
using *bij-betw-inv-into*[*OF assms*(*1*)] **.**
moreover have (*inv-into* (*reachable-states M1*) *f*) (*initial M2*) = (*initial M1*)
using *assms*(*1*,*2*) *reachable-states-initial*
by (*metis bij-betw-inv-into-left*)
moreover have ⋀ *q x y q′* . *q* ∈ *reachable-states M2* ⟹ *q′* ∈ *reachable-states M2* ⟹ (*q*,*x*,*y*,*q′*) ∈ *transitions M2* ⟷ ((*inv-into* (*reachable-states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*reachable-states M1*) *f*) *q′*) ∈ *transitions M1*
proof
fix *q x y q′* **assume** *q* ∈ *reachable-states M2* **and** *q′* ∈ *reachable-states M2*

show (*q*,*x*,*y*,*q′*) ∈ *transitions M2* ⟹ ((*inv-into* (*reachable-states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*reachable-states M1*) *f*) *q′*) ∈ *transitions M1*
proof −
assume *a1*: (*q*, *x*, *y*, *q′*) ∈ *FSM.transitions M2*
have *f2*: ∀ *f B A*. ¬ *bij-betw f B A* ∨ (∀ *b*. (*b*::′*b*) ∉ *B* ∨ (*f b*::′*a*) ∈ *A*)
using *bij-betwE* **by** *blast*
then have *f3*: *inv-into* (*FSM.reachable-states M1*) *f q* ∈ *FSM.reachable-states M1*
using ‹*q* ∈ *FSM.reachable-states M2*› *calculation*(*1*) **by** *blast*
have *inv-into* (*FSM.reachable-states M1*) *f q′* ∈ *FSM.reachable-states M1*
using *f2* ‹*q′* ∈ *FSM.reachable-states M2*› *calculation*(*1*) **by** *blast*
then show *?thesis*

using *f3 a1* ‹$q \in$ *FSM.reachable-states M2*› ‹$q' \in$ *FSM.reachable-states M2*› *assms*(*1*) *assms*(*3*) *bij-betw-inv-into-right* **by** *fastforce*
qed

show ((*inv-into* (*reachable-states M1*) *f*) *q*,*x*,*y*,(*inv-into* (*reachable-states M1*) *f*) *q*′) $\in$ *transitions M1* $\Longrightarrow$ (*q*,*x*,*y*,*q*′) $\in$ *transitions M2*
proof −
assume *a1*: (*inv-into* (*FSM.reachable-states M1*) *f q*, *x*, *y*, *inv-into* (*FSM.reachable-states M1*) *f q*′) $\in$ *FSM.transitions M1*
have *f2*: $\forall$ *f B A*. $\neg$ *bij-betw f B A* $\vee$ ($\forall$ *b*. (*b*::′*b*) $\notin$ *B* $\vee$ (*f b*::′*a*) $\in$ *A*)
by (*metis* (*full-types*) *bij-betwE*)
then have *f3*: *inv-into* (*FSM.reachable-states M1*) *f q*′ $\in$ *FSM.reachable-states M1*
using ‹$q' \in$ *FSM.reachable-states M2*› *calculation*(*1*) **by** *blast*
have *inv-into* (*FSM.reachable-states M1*) *f q* $\in$ *FSM.reachable-states M1*
using *f2* ‹$q \in$ *FSM.reachable-states M2*› *calculation*(*1*) **by** *blast*
then show *?thesis*
using *f3 a1* ‹$q \in$ *FSM.reachable-states M2*› ‹$q' \in$ *FSM.reachable-states M2*› *assms*(*1*) *assms*(*3*) *bij-betw-inv-into-right* **by** *fastforce*
qed
qed
ultimately show *L M2* $\subseteq$ *L M1*
using *language-equivalence-from-isomorphism-helper-reachable*[*of* (*inv-into* (*reachable-states M1*) *f*) *M2 M1*]
by *blast*
qed

lemma *language-empty-io* :
assumes *inputs M* = {} $\vee$ *outputs M* = {}
shows *L M* = {[]}
proof −
have *transitions M* = {}
using *assms fsm-transition-input fsm-transition-output*
by *auto*
then have $\bigwedge$ *p* . *path M* (*initial M*) *p* $\Longrightarrow$ *p* = []
by (*metis empty-iff path.cases*)
then show *?thesis*
unfolding *LS.simps*
by *blast*
qed

4.8 Basic FSM Properties

4.8.1 Completely Specified

fun *completely-specified* :: (′*a*,′*b*,′*c*) *fsm* $\Rightarrow$ *bool* **where**
completely-specified M = ($\forall$ *q* $\in$ *states M* . $\forall$ *x* $\in$ *inputs M* . $\exists$ *t* $\in$ *transitions M* . *t-source t* = *q* $\wedge$ *t-input t* = *x*)

lemma *completely-specified-alt-def* :
completely-specified M = (∀ *q* ∈ *states M* . ∀ *x* ∈ *inputs M* . ∃ *q′ y* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*)
by *force*

lemma *completely-specified-alt-def-h* :
completely-specified M = (∀ *q* ∈ *states M* . ∀ *x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {})
by *force*

fun *completely-specified-state* :: (*′a*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ *bool* **where**
completely-specified-state M q = (∀ *x* ∈ *inputs M* . ∃ *t* ∈ *transitions M* . *t-source t* = *q* ∧ *t-input t* = *x*)

lemma *completely-specified-states* :
completely-specified M = (∀ *q* ∈ *states M* . *completely-specified-state M q*)
unfolding *completely-specified.simps completely-specified-state.simps* **by** *force*

lemma *completely-specified-state-alt-def-h* :
completely-specified-state M q = (∀ *x* ∈ *inputs M* . *h M* (*q*,*x*) ≠ {})
by *force*

lemma *completely-specified-path-extension* :
assumes *completely-specified M*
and *q* ∈ *states M*
and *path M q p*
and *x* ∈ (*inputs M*)
obtains *t* **where** *t* ∈ *transitions M* **and** *t-input t* = *x* **and** *t-source t* = *target q p*
proof −
have *target q p* ∈ *states M*
using *path-target-is-state* ‹*path M q p*› **by** *metis*
then obtain *t* **where** *t* ∈ *transitions M* **and** *t-input t* = *x* **and** *t-source t* = *target q p*
using ‹*completely-specified M*› ‹*x* ∈ (*inputs M*)›
unfolding *completely-specified.simps* **by** *blast*
then show *?thesis* **using** *that* **by** *blast*
qed

lemma *completely-specified-language-extension* :
assumes *completely-specified M*
and *q* ∈ *states M*
and *io* ∈ *LS M q*
and *x* ∈ (*inputs M*)
obtains *y* **where** *io*@[(*x*,*y*)] ∈ *LS M q*
proof −
obtain *p* **where** *path M q p* **and** *p-io p* = *io*

using ‹*io* ∈ *LS M q*› **by** *auto*

moreover obtain *t* **where** *t* ∈ *transitions M* **and** *t-input t* = *x* **and** *t-source t* = *target q p*
using *completely-specified-path-extension*[*OF assms*(*1*,*2*) ‹*path M q p*› *assms*(*4*)] **by** *blast*

ultimately have *path M q* (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *io*@[(*x*,*t-output t*)]
by (*simp add*: *path-append-transition*)+

then have *io*@[(*x*,*t-output t*)] ∈ *LS M q*
using *language-state-containment*[*of M q p*@[*t*] *io*@[(*x*,*t-output t*)]] **by** *auto*
then show *?thesis* **using** *that* **by** *blast*
qed

lemma *path-of-length-ex* :
assumes *completely-specified M*
and *q* ∈ *states M*
and *inputs M* ≠ {}
shows ∃ *p* . *path M q p* ∧ *length p* = *k*
using *assms*(*2*) **proof** (*induction k arbitrary*: *q*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)

obtain *t* **where** *t-source t* = *q* **and** *t* ∈ *transitions M*
by (*meson Suc.prems assms*(*1*) *assms*(*3*) *completely-specified.simps equals0I*)
then have *t-target t* ∈ *states M*
using *fsm-transition-target* **by** *blast*
then obtain *p* **where** *path M* (*t-target t*) *p* ∧ *length p* = *k*
using *Suc.IH* **by** *blast*
then show *?case*
using ‹*t-source t* = *q*› ‹*t* ∈ *transitions M*›
by *auto*
qed

4.8.2 Deterministic

fun *deterministic* :: (*'a*,*'b*,*'c*) *fsm* ⇒ *bool* **where**
deterministic M = (∀ *t1* ∈ *transitions M* .
∀ *t2* ∈ *transitions M* .
(*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2*)
⟶ (*t-output t1* = *t-output t2* ∧ *t-target t1* = *t-target t2*))

lemma *deterministic-alt-def* :
deterministic M = (∀ *q1 x y' y''* *q1' q1''* . (*q1*,*x*,*y'*,*q1'*) ∈ *transitions M* ∧ (*q1*,*x*,*y''*,*q1''*) ∈ *transitions M* ⟶ *y'* = *y''* ∧ *q1'* = *q1''*)

by *auto*

lemma *deterministic-alt-def-h* :
deterministic M = (∀ *q1 x yq yq′* . (*yq* ∈ *h M* (*q1*,*x*) ∧ *yq′* ∈ *h M* (*q1*,*x*)) ⟶ *yq* = *yq′*)
by *auto*

4.8.3 Observable

fun *observable* :: (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**
observable M = (∀ *t1* ∈ *transitions M* .
∀ *t2* ∈ *transitions M* .
(*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
⟶ *t-target t1* = *t-target t2*)

lemma *observable-alt-def* :
observable M = (∀ *q1 x y q1′ q1′′* . (*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q1*,*x*,*y*,*q1′′*) ∈ *transitions M* ⟶ *q1′* = *q1′′*)
by *auto*

lemma *observable-alt-def-h* :
observable M = (∀ *q1 x yq yq′* . (*yq* ∈ *h M* (*q1*,*x*) ∧ *yq′* ∈ *h M* (*q1*,*x*)) ⟶ *fst yq* = *fst yq′* ⟶ *snd yq* = *snd yq′*)
by *auto*

lemma *language-append-path-ob* :
assumes *io*@[(*x*,*y*)] ∈ *L M*
obtains *p t* **where** *path M* (*initial M*) (*p*@[*t*]) **and** *p-io p* = *io* **and** *t-input t* = *x* **and** *t-output t* = *y*
proof −
obtain *p p2* **where** *path M* (*initial M*) *p* **and** *path M* (*target* (*initial M*) *p*) *p2* **and** *p-io p* = *io* **and** *p-io p2* = [(*x*,*y*)]
using *language-state-split*[*OF assms*] **by** *blast*

obtain *t* **where** *p2* = [*t*] **and** *t-input t* = *x* **and** *t-output t* = *y*
using ‹*p-io p2* = [(*x*,*y*)]› **by** *auto*

have *path M* (*initial M*) (*p*@[*t*])
using ‹*path M* (*initial M*) *p*› ‹*path M* (*target* (*initial M*) *p*) *p2*› **unfolding** ‹*p2* = [*t*]› **by** *auto*
then show *?thesis* **using** *that*[*OF* - ‹*p-io p* = *io*› ‹*t-input t* = *x*› ‹*t-output t* = *y*›]
by *simp*
qed

4.8.4 Single Input

fun *single-input* :: (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**

single-input M = (∀ *t1* ∈ *transitions M* .
∀ *t2* ∈ *transitions M* .
t-source t1 = *t-source t2* ⟶ *t-input t1* = *t-input t2*)

lemma *single-input-alt-def* :
single-input M = (∀ *q1 x x′ y y′ q1′ q1′′* . (*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q1*,*x′*,*y′*,*q1′′*) ∈ *transitions M* ⟶ *x* = *x′*)
by *fastforce*

lemma *single-input-alt-def-h* :
single-input M = (∀ *q x x′* . (*h M* (*q*,*x*) ≠ {} ∧ *h M* (*q*,*x′*) ≠ {}) ⟶ *x* = *x′*)
by *force*

4.8.5 Output Complete

fun *output-complete* :: (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**
output-complete M = (∀ *t* ∈ *transitions M* .
∀ *y* ∈ *outputs M* .
∃ *t′* ∈ *transitions M* . *t-source t* = *t-source t′* ∧
t-input t = *t-input t′* ∧
t-output t′ = *y*)

lemma *output-complete-alt-def* :
output-complete M = (∀ *q x* . (∃ *y q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*) ⟶ (∀ *y* ∈ (*outputs M*) . ∃ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*))
by *force*

lemma *output-complete-alt-def-h* :
output-complete M = (∀ *q x* . *h M* (*q*,*x*) ≠ {} ⟶ (∀ *y* ∈ *outputs M* . ∃ *q′* . (*y*,*q′*) ∈ *h M* (*q*,*x*)))
by *force*

4.8.6 Acyclic

fun *acyclic* :: (′*a*,′*b*,′*c*) *fsm* ⇒ *bool* **where**
acyclic M = (∀ *p* . *path M* (*initial M*) *p* ⟶ *distinct* (*visited-states* (*initial M*) *p*))

lemma *visited-states-length* : *length* (*visited-states q p*) = *Suc* (*length p*) **by** *auto*

lemma *visited-states-take* :
(*take* (*Suc n*) (*visited-states q p*)) = (*visited-states q* (*take n p*))
proof (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case* **by** (*cases n* ≤ *length xs*; *auto*)

qed

lemma *acyclic-code*[*code*] :
  acyclic M = (¬(∃ *p* ∈ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)) .
        ∃ *t* ∈ *transitions M* . *t-source t* = *target* (*initial M*) *p* ∧
          t-target t ∈ *set* (*visited-states* (*initial M*) *p*)))
proof −
  have (∃ *p* ∈ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)) .
      ∃ *t* ∈ *transitions M* . *t-source t* = *target* (*initial M*) *p* ∧
        t-target t ∈ *set* (*visited-states* (*initial M*) *p*))
    ⟹ ¬ *FSM.acyclic M*
  proof −
    assume (∃ *p* ∈ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)) .
      ∃ *t* ∈ *transitions M* . *t-source t* = *target* (*initial M*) *p* ∧
        t-target t ∈ *set* (*visited-states* (*initial M*) *p*))
    then obtain *p t* **where** *path M* (*initial M*) *p*
      and *distinct* (*visited-states* (*initial M*) *p*)
      and *t* ∈ *transitions M*
      and *t-source t* = *target* (*initial M*) *p*
      and *t-target t* ∈ *set* (*visited-states* (*initial M*) *p*)
      unfolding *acyclic-paths-set* **by** *blast*
    then have *path M* (*initial M*) (*p*@[*t*])
      by (*simp add*: *path-append-transition*)
    moreover have ¬ (*distinct* (*visited-states* (*initial M*) (*p*@[*t*])))
      using ‹*t-target t* ∈ *set* (*visited-states* (*initial M*) *p*)› **by** *auto*
    ultimately show ¬ *FSM.acyclic M*
      by (*meson acyclic.elims*(*2*))
  qed
  moreover have ¬ *FSM.acyclic M* ⟹
      (∃ *p* ∈ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)) .
        ∃ *t* ∈ *transitions M* . *t-source t* = *target* (*initial M*) *p* ∧
          t-target t ∈ *set* (*visited-states* (*initial M*) *p*))
  proof −
    assume ¬ *FSM.acyclic M*
    then obtain *p* **where** *path M* (*initial M*) *p*
      and ¬ *distinct* (*visited-states* (*initial M*) *p*)
      by *auto*
    then obtain *n* **where** *distinct* (*take* (*Suc n*) (*visited-states* (*initial M*) *p*))
      and ¬ *distinct* (*take* (*Suc* (*Suc n*)) (*visited-states* (*initial M*) *p*))
      using *maximal-distinct-prefix* **by** *blast*
    then have *distinct* (*visited-states* (*initial M*) (*take n p*))
      and ¬ *distinct* (*visited-states* (*initial M*)(*take* (*Suc n*) *p*))
      unfolding *visited-states-take* **by** *simp*+

    then obtain *p′ t′* **where** ∗: *take n p* = *p′*
      and ∗∗: *take* (*Suc n*) *p* = *p′* @ [*t′*]
      by (*metis Suc-less-eq* ‹¬ *distinct* (*visited-states* (*FSM.initial M*) *p*)›

le-imp-less-Suc not-less-eq-eq take-all take-hd-drop)

have $***$: *visited-states* (*FSM.initial M*) (p' @ [t']) = (*visited-states* (*FSM.initial M*) p')@[*t-target* t']
by *auto*

have *path M* (*initial M*) p'
using $*$ ‹*path M* (*initial M*) *p*›
by (*metis append-take-drop-id path-prefix*)
then have $p' \in$ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
using ‹*distinct* (*visited-states* (*initial M*) (*take n p*))›
unfolding $*$ *acyclic-paths-set* **by** *blast*
moreover have $t' \in$ *transitions M* $\wedge$ *t-source* t' = *target* (*initial M*) p'
using $*$ $**$ ‹*path M* (*initial M*) *p*›
by (*metis append-take-drop-id path-append-elim path-cons-elim*)

moreover have *t-target* $t' \in$ *set* (*visited-states* (*initial M*) p')
using ‹*distinct* (*visited-states* (*initial M*) (*take n p*))›
‹¬ *distinct* (*visited-states* (*initial M*)(*take* (*Suc n*) *p*))›
unfolding $*$ $**$ $***$ **by** *auto*
ultimately show (∃ $p \in$ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)) .
∃ $t \in$ *transitions M* . *t-source t* = *target* (*initial M*) *p* $\wedge$
t-target t $\in$ *set* (*visited-states* (*initial M*) *p*))
by *blast*
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *acyclic-alt-def* : *acyclic M* = *finite* (*L M*)
proof
show *acyclic M* ⟹ *finite* (*L M*)
proof −
assume *acyclic M*
then have { *p* . *path M* (*initial M*) *p*} ⊆ (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
unfolding *acyclic-paths-set* **by** *auto*
moreover have *finite* (*acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*))
unfolding *acyclic-paths-up-to-length.simps* **using** *paths-finite*[*of M initial M size M* − *1*]
by (*metis* (*mono-tags*, *lifting*) *Collect-cong* ‹*FSM.acyclic M*› *acyclic.elims*(*2*))

ultimately have *finite* { *p* . *path M* (*initial M*) *p*}
using *finite-subset* **by** *blast*
then show *finite* (*L M*)
unfolding *LS.simps* **by** *auto*

```
  qed

  show finite (L M) ⟹ acyclic M
  proof (rule ccontr)
    assume finite (L M)
    assume ¬ acyclic M

    obtain max-io-len where ∀ io ∈ L M . length io < max-io-len
      using finite-maxlen[OF ‹finite (L M)›] by blast
    then have ⋀ p . path M (initial M) p ⟹ length p < max-io-len
    proof −
      fix p assume path M (initial M) p
      show length p < max-io-len
      proof (rule ccontr)
        assume ¬ length p < max-io-len
        then have ¬ length (p-io p) < max-io-len by auto
        moreover have p-io p ∈ L M
          unfolding LS.simps using ‹path M (initial M) p› by blast
        ultimately show False
          using ‹∀ io ∈ L M . length io < max-io-len› by blast
      qed
    qed

    obtain p where path M (initial M) p and ¬ distinct (visited-states (initial M)
p)
      using ‹¬ acyclic M› unfolding acyclic.simps by blast
    then obtain pL where path M (initial M) pL and max-io-len ≤ length pL
      using cyclic-path-pumping[of M p max-io-len] by blast
    then show False
      using ‹⋀ p . path M (initial M) p ⟹ length p < max-io-len›
      using not-le by blast
  qed
qed


lemma acyclic-finite-paths-from-reachable-state :
  assumes acyclic M
  and     path M (initial M) p
  and     target (initial M) p = q
    shows finite {p . path M q p}
proof −
  from assms have { p . path M (initial M) p} ⊆ (acyclic-paths-up-to-length M
(initial M) (size M − 1))
    unfolding acyclic-paths-set by auto
  moreover have finite (acyclic-paths-up-to-length M (initial M) (size M − 1))
     unfolding acyclic-paths-up-to-length.simps using paths-finite[of M initial M
size M − 1]
    by (metis (mono-tags, lifting) Collect-cong ‹FSM.acyclic M› acyclic.elims(2))
  ultimately have finite { p . path M (initial M) p}
```

```
    using finite-subset by blast

  show finite {p . path M q p}
  proof (cases q ∈ states M)
    case True

    have image (λp' . p@p') {p' . path M q p'} ⊆ {p' . path M (initial M) p'}
    proof
      fix x assume x ∈ image (λp' . p@p') {p' . path M q p'}
      then obtain p' where x = p@p' and p' ∈ {p' . path M q p'}
        by blast
      then have path M q p' by auto
      then have path M (initial M) (p@p')
        using path-append[OF ‹path M (initial M) p›] ‹target (initial M) p = q›
by auto
      then show x ∈ {p' . path M (initial M) p'} using ‹x = p@p'› by blast
    qed

    then have finite (image (λp' . p@p') {p' . path M q p'})
      using ‹finite { p . path M (initial M) p}› finite-subset by auto
    show ?thesis using finite-imageD[OF ‹finite (image (λp' . p@p') {p' . path M
q p'})›]
      by (meson inj-onI same-append-eq)
  next
    case False
    then show ?thesis
      by (meson not-finite-existsD path-begin-state)
  qed
qed


lemma acyclic-paths-from-reachable-states :
  assumes acyclic M
  and     path M (initial M) p'
  and     target (initial M) p' = q
  and     path M q p
shows distinct (visited-states q p)
proof −
  have path M (initial M) (p'@p)
    using assms(2,3,4) path-append by metis
  then have distinct (visited-states (initial M) (p'@p))
    using assms(1) unfolding acyclic.simps by blast
  then have distinct (initial M # (map t-target p') @ map t-target p)
    by auto
  moreover have initial M # (map t-target p') @ map t-target p
                = (butlast (initial M # map t-target p')) @ ((last (initial M # map
t-target p')) # map t-target p)
    by auto
  ultimately have distinct ((last (initial M # map t-target p')) # map t-target p)
```

by *auto*
then show *?thesis*
using ‹*target* (*initial M*) *p′* = *q*› **unfolding** *visited-states.simps target.simps* **by** *simp*
qed

definition *LS-acyclic* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ (′*b* × ′*c*) *list set* **where**
LS-acyclic M q = {*p-io p* | *p* . *path M q p* ∧ *distinct* (*visited-states q p*)}

lemma *LS-acyclic-code*[*code*] :
LS-acyclic M q = *image p-io* (*acyclic-paths-up-to-length M q* (*size M* − *1*))
unfolding *acyclic-paths-set LS-acyclic-def* **by** *blast*

lemma *LS-from-LS-acyclic* :
assumes *acyclic M*
shows *L M* = *LS-acyclic M* (*initial M*)
proof −
obtain *pps* :: ((′*b* × ′*c*) *list* ⇒ *bool*) ⇒ ((′*b* × ′*c*) *list* ⇒ *bool*) ⇒ (′*b* × ′*c*) *list* **where**
f1: ∀ *p pa*. (¬ *p* (*pps pa p*)) = *pa* (*pps pa p*) ∨ *Collect p* = *Collect pa*
by (*metis* (*no-types*) *Collect-cong*)
have ∀ *ps*. ¬ *path M* (*FSM.initial M*) *ps* ∨ *distinct* (*visited-states* (*FSM.initial M*) *ps*)
using *acyclic.simps assms* **by** *blast*
then have (∄ *ps*. *pps* (λ*ps*. ∃ *psa*. *ps* = *p-io psa* ∧ *path M* (*FSM.initial M*) *psa*)
(λ*ps*. ∃ *psa*. *ps* = *p-io psa* ∧ *path M* (*FSM.initial M*) *psa*
∧ *distinct* (*visited-states* (*FSM.initial M*) *psa*))
= *p-io ps* ∧ *path M* (*FSM.initial M*) *ps* ∧ *distinct* (*visited-states* (*FSM.initial M*) *ps*))
≠ (∃ *ps*. *pps* (λ*ps*. ∃ *psa*. *ps* = *p-io psa* ∧ *path M* (*FSM.initial M*) *psa*)
(λ*ps*. ∃ *psa*. *ps* = *p-io psa* ∧ *path M* (*FSM.initial M*) *psa*
∧ *distinct* (*visited-states* (*FSM.initial M*) *psa*))
= *p-io ps* ∧ *path M* (*FSM.initial M*) *ps*)
by *blast*
then have {*p-io ps* |*ps*. *path M* (*FSM.initial M*) *ps* ∧ *distinct* (*visited-states* (*FSM.initial M*) *ps*)}
= {*p-io ps* |*ps*. *path M* (*FSM.initial M*) *ps*}
using *f1*
by (*meson* ‹∀ *ps*. ¬ *path M* (*FSM.initial M*) *ps* ∨ *distinct* (*visited-states* (*FSM.initial M*) *ps*)›)
then show *?thesis*
by (*simp add*: *LS-acyclic-def*)
qed

lemma *cyclic-cycle* :
assumes ¬ *acyclic M*
shows ∃ *q p* . *path M q p* ∧ *p* ≠ [] ∧ *target q p* = *q*

```
proof −
  from ‹¬ acyclic M› obtain p t where path M (initial M) (p@[t])
                                  and ¬distinct (visited-states (initial M) (p@[t]))
   by (metis (no-types, opaque-lifting) Nil-is-append-conv acyclic.simps append-take-drop-id

        maximal-distinct-prefix rev-exhaust visited-states-take)


  show ?thesis
  proof (cases initial M ∈ set (map t-target (p@[t])))
    case True
    then obtain i where last (take i (map t-target (p@[t]))) = initial M
                  and i ≤ length (map t-target (p@[t])) and 0 < i
      using list-contains-last-take by metis

    let ?p = take i (p@[t])
    have path M (initial M) (?p@(drop i (p@[t])))
      using ‹path M (initial M) (p@[t])›
      by (metis append-take-drop-id)
    then have path M (initial M) ?p by auto
    moreover have ?p ≠ [] using ‹0 < i› by auto
    moreover have target (initial M) ?p = initial M
      using ‹last (take i (map t-target (p@[t]))) = initial M›
      unfolding target.simps visited-states.simps
     by (metis (no-types, lifting) calculation(2) last-ConsR list.map-disc-iff take-map)

    ultimately show ?thesis by blast
  next
    case False
    then have ¬ distinct (map t-target (p@[t]))
      using ‹¬distinct (visited-states (initial M) (p@[t]))›
      unfolding visited-states.simps
      by auto
    then obtain i j where i < j and j < length (map t-target (p@[t]))
                  and (map t-target (p@[t])) ! i = (map t-target (p@[t])) ! j
      using non-distinct-repetition-indices by blast

    let ?pre-i = take (Suc i) (p@[t])
    let ?p = take ((Suc j)−(Suc i)) (drop (Suc i) (p@[t]))
    let ?post-j = drop ((Suc j)−(Suc i)) (drop (Suc i) (p@[t]))

    have p@[t] = ?pre-i @ ?p @ ?post-j
      using ‹i < j› ‹j < length (map t-target (p@[t]))›
      by (metis append-take-drop-id)
    then have path M (target (initial M) ?pre-i) ?p
      using ‹path M (initial M) (p@[t])›
      by (metis path-prefix path-suffix)

    have ?p ≠ []
```

```
      using ‹i < j› ‹j < length (map t-target (p@[t]))› by auto

    have i < length (map t-target (p@[t]))
      using ‹i < j› ‹j < length (map t-target (p@[t]))› by auto
    have (target (initial M) ?pre-i) = (map t-target (p@[t])) ! i
      unfolding target.simps visited-states.simps
      using take-last-index[OF ‹i < length (map t-target (p@[t]))›]
      by (metis (no-types, lifting) ‹i < length (map t-target (p @ [t]))›
          last-ConsR snoc-eq-iff-butlast take-Suc-conv-app-nth take-map)

    have ?pre-i@?p = take (Suc j) (p@[t])
    by (metis (no-types) ‹i < j› add-Suc add-diff-cancel-left' less-SucI less-imp-Suc-add
take-add)
     moreover have (target (initial M) (take (Suc j) (p@[t]))) = (map t-target
(p@[t])) ! j
      unfolding target.simps visited-states.simps
      using take-last-index[OF ‹j < length (map t-target (p@[t]))›]
      by (metis (no-types, lifting) ‹j < length (map t-target (p @ [t]))›
           last-ConsR snoc-eq-iff-butlast take-Suc-conv-app-nth take-map)
    ultimately have (target (initial M) (?pre-i@?p)) = (map t-target (p@[t])) ! j
      by auto
    then have (target (initial M) (?pre-i@?p)) = (map t-target (p@[t])) ! i
      using ‹(map t-target (p@[t])) ! i = (map t-target (p@[t])) ! j› by simp
    moreover have (target (initial M) (?pre-i@?p)) = (target (target (initial M)
?pre-i) ?p)
      unfolding target.simps visited-states.simps last.simps by auto
   ultimately have (target (target (initial M) ?pre-i) ?p) = (map t-target (p@[t]))
! i
      by auto
    then have (target (target (initial M) ?pre-i) ?p) = (target (initial M) ?pre-i)
      using ‹(target (initial M) ?pre-i) = (map t-target (p@[t])) ! i› by auto

    show ?thesis
      using ‹path M (target (initial M) ?pre-i) ?p› ‹?p ≠ []›
           ‹(target (target (initial M) ?pre-i) ?p) = (target (initial M) ?pre-i)›
      by blast
  qed
qed


lemma cyclic-cycle-rev :
  fixes M :: ('a,'b,'c) fsm
  assumes path M (initial M) p'
  and     target (initial M) p' = q
  and     path M q p
  and     p ≠ []
  and     target q p = q
shows ¬ acyclic M
  using assms unfolding acyclic.simps target.simps visited-states.simps
```

using *distinct.simps*(*2*) **by** *fastforce*

lemma *acyclic-initial* :
assumes *acyclic M*
shows $\neg$ ($\exists$ *t* $\in$ *transitions M* . *t-target t* = *initial M* $\wedge$
($\exists$ *p* . *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p* =
t-source t))
by (*metis append-Cons assms cyclic-cycle-rev list.distinct*(*1*) *path.simps*
path-append path-append-transition-elim(*3*) *single-transition-path*)

lemma *cyclic-path-shift* :
assumes *path M q p*
and *target q p* = *q*
shows *path M* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*))
and *target* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*)) = (*target q* (*take i p*))
proof −
show *path M* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*))
by (*metis append-take-drop-id assms*(*1*) *assms*(*2*) *path-append path-append-elim*
path-append-target)
show *target* (*target q* (*take i p*)) ((*drop i p*) @ (*take i p*)) = (*target q* (*take i p*))
by (*metis append-take-drop-id assms*(*2*) *path-append-target*)
qed

lemma *cyclic-path-transition-states-property* :
assumes $\exists$ *t* $\in$ *set p* . *P* (*t-source t*)
and $\forall$ *t* $\in$ *set p* . *P* (*t-source t*) $\longrightarrow$ *P* (*t-target t*)
and *path M q p*
and *target q p* = *q*
shows $\forall$ *t* $\in$ *set p* . *P* (*t-source t*)
and $\forall$ *t* $\in$ *set p* . *P* (*t-target t*)
proof −
obtain *t0* **where** *t0* $\in$ *set p* **and** *P* (*t-source t0*)
using *assms*(*1*) **by** *blast*
then obtain *i* **where** *i* < *length p* **and** *p* ! *i* = *t0*
by (*meson in-set-conv-nth*)

let *?p* = (*drop i p* @ *take i p*)
have *path M* (*target q* (*take i p*)) *?p*
using *cyclic-path-shift*(*1*)[*OF assms*(*3*,*4*), *of i*] **by** *assumption*

have *set ?p* = *set p*
proof −
have *set ?p* = *set* (*take i p* @ *drop i p*)
using *list-set-sym* **by** *metis*
then show *?thesis* **by** *auto*
qed
then have $\bigwedge$ *t* . *t* $\in$ *set ?p* $\Longrightarrow$ *P* (*t-source t*) $\Longrightarrow$ *P* (*t-target t*)
using *assms*(*2*) **by** *blast*

```
  have ⋀ j . j < length ?p ⟹ P (t-source (?p ! j))
  proof −
    fix j assume j < length ?p
    then show P (t-source (?p ! j))
    proof (induction j)
      case 0
      then show ?case
        using ‹p ! i = t0› ‹P (t-source t0)›
       by (metis ‹i < length p› drop-eq-Nil hd-append2 hd-conv-nth hd-drop-conv-nth
leD
              length-greater-0-conv)
    next
      case (Suc j)
      then have P (t-source (?p ! j))
        by auto
      then have P (t-target (?p ! j))
        using Suc.prems ‹⋀ t . t ∈ set ?p ⟹ P (t-source t) ⟹ P (t-target t)›[of
?p ! j]
        using Suc-lessD nth-mem by blast
      moreover have t-target (?p ! j) = t-source (?p ! (Suc j))
        using path-source-target-index[OF Suc.prems ‹path M (target q (take i p))
?p›]
        by assumption
      ultimately show ?case
        using ‹⋀ t . t ∈ set ?p ⟹ P (t-source t) ⟹ P (t-target t)›[of ?p ! j]
        by simp
    qed
  qed
  then have ∀ t ∈ set ?p . P (t-source t)
    by (metis in-set-conv-nth)
  then show ∀ t ∈ set p . P (t-source t)
    using ‹set ?p = set p› by blast
  then show ∀ t ∈ set p . P (t-target t)
    using assms(2) by blast
qed


lemma cycle-incoming-transition-ex :
  assumes path M q p
  and     p ≠ []
  and     target q p = q
  and     t ∈ set p
shows ∃ tI ∈ set p . t-target tI = t-source t
proof −
  obtain i where i < length p and p ! i = t
    using assms(4) by (meson in-set-conv-nth)

  let ?p = (drop i p @ take i p)
```

```
  have path M (target q (take i p)) ?p
  and  target (target q (take i p)) ?p = target q (take i p)
    using cyclic-path-shift[OF assms(1,3), of i] by linarith+

  have p = (take i p @ drop i p) by auto
  then have path M (target q (take i p)) (drop i p)
    using path-suffix assms(1) by metis
  moreover have t = hd (drop i p)
    using ‹i < length p› ‹p ! i = t›
    by (simp add: hd-drop-conv-nth)
  ultimately have path M (target q (take i p)) [t]
   by (metis ‹i < length p› append-take-drop-id assms(1) path-append-elim take-hd-drop)
  then have t-source t = (target q (take i p))
    by auto
  moreover have t-target (last ?p) = (target q (take i p))
    using ‹path M (target q (take i p)) ?p› ‹target (target q (take i p)) ?p = target
q (take i p)›
          assms(2)
    unfolding target.simps visited-states.simps last.simps
   by (metis (no-types, lifting) ‹p = take i p @ drop i p› append-is-Nil-conv last-map

          list.map-disc-iff)


  moreover have set ?p = set p
  proof −
    have set ?p = set (take i p @ drop i p)
      using list-set-sym by metis
    then show ?thesis by auto
  qed

  ultimately show ?thesis
    by (metis ‹i < length p› append-is-Nil-conv drop-eq-Nil last-in-set leD)
qed


lemma acyclic-paths-finite :
  finite {p . path M q p ∧ distinct (visited-states q p) }
proof −
  have ⋀ p . path M q p ⟹ distinct (visited-states q p) ⟹ distinct p
  proof −
    fix p assume path M q p and distinct (visited-states q p)
    then have distinct (map t-target p) by auto
    then show distinct p by (simp add: distinct-map)
  qed

  then show ?thesis
    using finite-subset-distinct[OF fsm-transitions-finite, of M]  path-transitions[of
M q]
```

by (*metis* (*no-types*, *lifting*) *infinite-super mem-Collect-eq path-transitions subsetI*)
qed

lemma *acyclic-no-self-loop* :
assumes *acyclic M*
and $q \in$ *reachable-states M*
shows $\neg$ ($\exists$ *x y* . (*q*,*x*,*y*,*q*) $\in$ *transitions M*)
proof
assume $\exists$ *x y*. (*q*, *x*, *y*, *q*) $\in$ *FSM.transitions M*
then obtain *x y* **where** (*q*, *x*, *y*, *q*) $\in$ *FSM.transitions M* **by** *blast*
moreover obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *q*
using *assms*(*2*) **unfolding** *reachable-states-def* **by** *blast*
ultimately have *path M* (*initial M*) (*p*@[(*q*,*x*,*y*,*q*)])
by (*simp add*: *path-append-transition*)
moreover have $\neg$ (*distinct* (*visited-states* (*initial M*) (*p*@[(*q*,*x*,*y*,*q*)])))
using ‹*target* (*initial M*) *p* = *q*› **unfolding** *visited-states.simps target.simps*
by (*cases p rule*: *rev-cases*; *auto*)
ultimately show *False*
using *assms*(*1*) **unfolding** *acyclic.simps*
by *meson*
qed

4.8.7 Deadlock States

fun *deadlock-state* :: (*'a*,*'b*,*'c*) *fsm* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
deadlock-state M q = ($\neg$($\exists$ *t* $\in$ *transitions M* . *t-source t* = *q*))

lemma *deadlock-state-alt-def* : *deadlock-state M q* = (*LS M q* $\subseteq$ {[]})
proof
show *deadlock-state M q* $\Longrightarrow$ *LS M q* $\subseteq$ {[]}
proof −
assume *deadlock-state M q*
moreover have $\bigwedge$ *p* . *deadlock-state M q* $\Longrightarrow$ *path M q p* $\Longrightarrow$ *p* = []
unfolding *deadlock-state.simps* **by** (*metis path.cases*)
ultimately show *LS M q* $\subseteq$ {[]}
unfolding *LS.simps* **by** *blast*
qed
show *LS M q* $\subseteq$ {[]} $\Longrightarrow$ *deadlock-state M q*
unfolding *LS.simps deadlock-state.simps* **using** *path.cases*[*of M q*] **by** *blast*
qed

lemma *deadlock-state-alt-def-h* : *deadlock-state M q* = ($\forall$ *x* $\in$ *inputs M* . *h M* (*q*,*x*) = {})
unfolding *deadlock-state.simps h.simps*
using *fsm-transition-input* **by** *force*

lemma *acyclic-deadlock-reachable* :
  assumes *acyclic M*
  shows $\exists$ *q* $\in$ *reachable-states M* . *deadlock-state M q*
proof (*rule ccontr*)
  assume $\neg$ ($\exists$ *q*$\in$*reachable-states M*. *deadlock-state M q*)
  then have $*$: $\bigwedge$ *q* . *q* $\in$ *reachable-states M* $\Longrightarrow$ ($\exists$ *t* $\in$ *transitions M* . *t-source t* = *q*)
    unfolding *deadlock-state.simps* **by** *blast*

  let *?p* = *arg-max-on length* {*p*. *path M* (*initial M*) *p*}

  have *finite* {*p*. *path M* (*initial M*) *p*}
   by (*metis Collect-cong acyclic-finite-paths-from-reachable-state assms eq-Nil-appendI fsm-initial*
        nil path-append path-append-elim)

  moreover have {*p*. *path M* (*initial M*) *p*} $\neq$ {}
    by *auto*
  ultimately obtain *p* **where** *path M* (*initial M*) *p*
                and $\bigwedge$ *p'* . *path M* (*initial M*) *p'* $\Longrightarrow$ *length p'* $\leq$ *length p*
    using *max-length-elem*
    by (*metis mem-Collect-eq not-le-imp-less*)

  then obtain *t* **where** *t* $\in$ *transitions M* **and** *t-source t* = *target* (*initial M*) *p*
    using $*$[*of target* (*initial M*) *p*] **unfolding** *reachable-states-def*
    by *blast*

  then have *path M* (*initial M*) (*p*@[*t*])
    using ‹*path M* (*initial M*) *p*›
    by (*simp add*: *path-append-transition*)

  then show *False*
    using ‹$\bigwedge$ *p'* . *path M* (*initial M*) *p'* $\Longrightarrow$ *length p'* $\leq$ *length p*›
    by (*metis impossible-Cons length-rotate1 rotate1.simps*(*2*))
qed

lemma *deadlock-prefix* :
  assumes *path M q p*
  and *t* $\in$ *set* (*butlast p*)
shows $\neg$ (*deadlock-state M* (*t-target t*))
  using *assms* **proof** (*induction p rule*: *rev-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*snoc t' p'*)

  show *?case* **proof** (*cases t* $\in$ *set* (*butlast p'*))
    case *True*

```
    show ?thesis
      using snoc.IH[OF - True] snoc.prems(1)
      by blast
  next
    case False
    then have p' = (butlast p')@[t]
      using snoc.prems(2) by (metis append-butlast-last-id append-self-conv2 but-
last-snoc
                              in-set-butlast-appendI list-prefix-elem set-ConsD)
    then have path M q ((butlast p'@[t])@[t'])
      using snoc.prems(1)
      by auto

    have t' ∈ transitions M and t-source t' = target q (butlast p'@[t])
      using path-suffix[OF ‹path M q ((butlast p'@[t])@[t'])›]
      by auto
    then have t' ∈ transitions M ∧ t-source t' = t-target t
      unfolding target.simps visited-states.simps by auto
    then show ?thesis
      unfolding deadlock-state.simps using ‹t' ∈ transitions M› by blast
  qed
qed


lemma states-initial-deadlock :
  assumes deadlock-state M (initial M)
  shows reachable-states M = {initial M}

proof −
  have ⋀ q . q ∈ reachable-states M ⟹ q = initial M
  proof −
    fix q assume q ∈ reachable-states M
    then obtain p where path M (initial M) p and target (initial M) p = q
      unfolding reachable-states-def by auto

    show q = initial M proof (cases p)
      case Nil
      then show ?thesis using ‹target (initial M) p = q› by auto
    next
      case (Cons t p')
        then have False using assms ‹path M (initial M) p› unfolding dead-
lock-state.simps
        by auto
      then show ?thesis by simp
    qed
  qed
  then show ?thesis
    using reachable-states-initial[of M] by blast
qed
```

4.8.8 Other

fun *completed-path* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('a,'b,'c)$ *path* $\Rightarrow$ *bool* **where**
completed-path M q p = *deadlock-state M* (*target q p*)

fun *minimal* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
minimal M = ($\forall$ *q* $\in$ *states M* . $\forall$ *q'* $\in$ *states M* . *q* $\neq$ *q'* $\longrightarrow$ *LS M q* $\neq$ *LS M q'*)

lemma *minimal-alt-def* : *minimal M* = ($\forall$ *q q'* . *q* $\in$ *states M* $\longrightarrow$ *q'* $\in$ *states M* $\longrightarrow$ *LS M q* = *LS M q'* $\longrightarrow$ *q* = *q'*)
by *auto*

definition *retains-outputs-for-states-and-inputs* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
retains-outputs-for-states-and-inputs M S
= ($\forall$ *tS* $\in$ *transitions S* .
$\forall$ *tM* $\in$ *transitions M* .
(*t-source tS* = *t-source tM* $\wedge$ *t-input tS* = *t-input tM*) $\longrightarrow$ *tM* $\in$ *transitions S*)

4.9 IO Targets and Observability

fun *paths-for-io'* :: $(('a \times 'b) \Rightarrow ('c \times 'a)$ *set*$) \Rightarrow ('b \times 'c)$ *list* $\Rightarrow$ $'a$ $\Rightarrow$ $('a,'b,'c)$ *path* $\Rightarrow$ $('a,'b,'c)$ *path set* **where**
paths-for-io' f [] *q prev* = {*prev*} |
paths-for-io' f ((*x*,*y*)#*io*) *q prev* = $\bigcup$(*image* (λ*yq'* . *paths-for-io' f io* (*snd yq'*) (*prev*@[(*q*,*x*,*y*,(*snd yq'*))])) (*Set.filter* (λ*yq'* . *fst yq'* = *y*) (*f* (*q*,*x*))))

lemma *paths-for-io'-set* :
assumes *q* $\in$ *states M*
shows *paths-for-io'* (*h M*) *io q prev* = {*prev*@*p* | *p* . *path M q p* $\wedge$ *p-io p* = *io*}
using *assms* **proof** (*induction io arbitrary*: *q prev*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons xy io*)
obtain *x y* **where** *xy* = (*x*,*y*)
by (*meson surj-pair*)

let *?UN* = $\bigcup$(*image* (λ*yq'* . *paths-for-io'* (*h M*) *io* (*snd yq'*) (*prev*@[(*q*,*x*,*y*,(*snd yq'*))]))
(*Set.filter* (λ*yq'* . *fst yq'* = *y*) (*h M* (*q*,*x*))))

have *?UN* = {*prev*@*p* | *p* . *path M q p* $\wedge$ *p-io p* = (*x*,*y*)#*io*}
proof
have $\bigwedge$ *p* . *p* $\in$ *?UN* $\Longrightarrow$ *p* $\in$ {*prev*@*p* | *p* . *path M q p* $\wedge$ *p-io p* = (*x*,*y*)#*io*}
proof −
fix *p* **assume** *p* $\in$ *?UN*
then obtain *q'* **where** (*y*,*q'*) $\in$ (*Set.filter* (λ*yq'* . *fst yq'* = *y*) (*h M* (*q*,*x*)))

and $p \in$ *paths-for-io'* $(h\ M)$ *io* q' $(prev@[(q,x,y,q')])$
by *auto*

from ‹$(y,q') \in$ (*Set.filter* $(\lambda yq'$. *fst* $yq' = y)$ $(h\ M\ (q,x)))$› **have** $q' \in$ *states* M
and $(q,x,y,q') \in$ *transitions* M
using *fsm-transition-target* **unfolding** *h.simps* **by** *auto*

have $p \in \{(prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p\ |p.\ path\ M\ q'\ p \wedge p\text{-}io\ p = io\}$
using ‹$p \in$ *paths-for-io'* $(h\ M)$ *io* q' $(prev@[(q,x,y,q')])$›
unfolding *Cons.IH*[*OF* ‹$q' \in$ *states* M›] **by** *assumption*
moreover have $\{(prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p\ |p.\ path\ M\ q'\ p \wedge p\text{-}io\ p = io\}$
$\subseteq \{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\}$
using ‹$(q,x,y,q') \in$ *transitions* M›
using *cons* **by** *force*
ultimately show $p \in \{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\}$
by *blast*
qed
then show *?UN* $\subseteq \{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\}$
by *blast*

have $\bigwedge p\ .\ p \in \{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\} \Longrightarrow p \in$ *?UN*
proof −
fix *pp* **assume** $pp \in \{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\}$
then obtain p **where** $pp = prev@p$ **and** *path* $M\ q\ p$ **and** $p\text{-}io\ p = (x,y)\#io$
by *fastforce*
then obtain $t\ p'$ **where** $p = t\#p'$ **and** *path* $M\ q\ (t\#p')$ **and** $p\text{-}io\ (t\#p') = (x,y)\#io$
and $p\text{-}io\ p' = io$
by (*metis* (*no-types*, *lifting*) *map-eq-Cons-D*)
then have *path* M (*t-target* t) p' **and** *t-source* $t = q$ **and** *t-input* $t = x$
and *t-output* $t = y$ **and** *t-target* $t \in$ *states* M
and $t \in$ *transitions* M
by *auto*

have $(y,$*t-target* $t) \in$ *Set.filter* $(\lambda yq'.$ *fst* $yq' = y)$ $(h\ M\ (q,\ x))$
using ‹$t \in$ *transitions* M› ‹*t-output* $t = y$› ‹*t-input* $t = x$› ‹*t-source* $t = q$›
unfolding *h.simps*
by *auto*
moreover have $(prev@p) \in$ *paths-for-io'* $(h\ M)$ *io* (*snd* $(y,$*t-target* $t))$ $(prev\ @\ [(q,\ x,\ y,\ snd\ (y,$*t-target* $t))])$
using *Cons.IH*[*OF* ‹*t-target* $t \in$ *states* M›, *of* $prev@[(q,\ x,\ y,$ *t-target* $t)]]$
using ‹$p = t\ \#\ p'$› ‹*p-io* $p' = io$› ‹*path* M (*t-target* t) p'› ‹*t-input* $t = x$› ‹*t-output* $t = y$› ‹*t-source* $t = q$›
by *auto*

ultimately show $pp \in$ *?UN* **unfolding** ‹$pp = prev@p$›
by *blast*
qed

then show $\{prev@p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = (x,y)\#io\} \subseteq ?UN$
by (*meson subsetI*)
qed

then show *?case*
by (*simp add*: ‹$xy = (x,\ y)$›)
qed

definition *paths-for-io* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $('a,'b,'c)$ *path set*
where
paths-for-io M q io $= \{p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = io\}$

lemma *paths-for-io-set-code*[*code*] :
paths-for-io M q io = (*if* $q \in$ *states M then paths-for-io'* (*h M*) *io q* [] *else* {})
using *paths-for-io'-set*[*of q M io* []]
unfolding *paths-for-io-def*
proof −
have {[] @ *ps* |*ps. path M q ps* ∧ *p-io ps* = *io*} = (*if q* ∈ *FSM.states M then paths-for-io'* (*h M*) *io q* [] *else* {})
⟶ {*ps. path M q ps* ∧ *p-io ps* = *io*} = (*if q* ∈ *FSM.states M then paths-for-io'* (*h M*) *io q* [] *else* {})
by *auto*
moreover
{ assume {[] @ *ps* |*ps. path M q ps* ∧ *p-io ps* = *io*} ≠ (*if q* ∈ *FSM.states M then paths-for-io'* (*h M*) *io q* [] *else* {})
then have *q* ∉ *FSM.states M*
using ‹*q* ∈ *FSM.states M* ⟹ *paths-for-io'* (*h M*) *io q* [] = {[] @ *p* |*p. path M q p* ∧ *p-io p* = *io*}› **by** *force*
then have {*ps. path M q ps* ∧ *p-io ps* = *io*} = (*if q* ∈ *FSM.states M then paths-for-io'* (*h M*) *io q* [] *else* {})
using *path-begin-state* **by** *force* **}**
ultimately show {*ps. path M q ps* ∧ *p-io ps* = *io*} = (*if q* ∈ *FSM.states M then paths-for-io'* (*h M*) *io q* [] *else* {})
by *linarith*
qed

fun *io-targets* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ *set* **where**
io-targets M io q $= \{target\ q\ p \mid p\ .\ path\ M\ q\ p \wedge p\text{-}io\ p = io\}$

lemma *io-targets-code*[*code*] : *io-targets M io q* = *image* (*target q*) (*paths-for-io M q io*)
unfolding *io-targets.simps paths-for-io-def* **by** *blast*

lemma *io-targets-states* :
io-targets M io q ⊆ *states M*
using *path-target-is-state* **by** *fastforce*

```
lemma observable-transition-unique :
  assumes observable M
      and t ∈ transitions M
    shows ∃! t′ ∈ transitions M . t-source t′ = t-source t ∧
                                  t-input t′ = t-input t ∧
                                  t-output t′ = t-output t
  by (metis assms observable.elims(2) prod.expand)

lemma observable-path-unique :
  assumes observable M
  and     path M q p
  and     path M q p′
  and     p-io p = p-io p′
shows p = p′
proof −
  have length p = length p′
    using assms(4) map-eq-imp-length-eq by blast
  then show ?thesis
    using ‹p-io p = p-io p′› ‹path M q p› ‹path M q p′›
  proof (induction p p′ arbitrary: q rule: list-induct2)
    case Nil
    then show ?case by auto
  next
    case (Cons x xs y ys)
   then have ∗: x ∈ transitions M ∧ y ∈ transitions M ∧ t-source x = t-source y
                                  ∧ t-input x = t-input y ∧ t-output x = t-output y
      by auto
    then have t-target x = t-target y
      using assms(1) observable.elims(2) by blast
    then have x = y
      by (simp add: ∗ prod.expand)


    have p-io xs = p-io ys
      using Cons by auto

    moreover have path M (t-target x) xs
      using Cons by auto
    moreover have path M (t-target x) ys
      using Cons ‹t-target x = t-target y› by auto
    ultimately have xs = ys
      using Cons by auto

    then show ?case
      using ‹x = y› by simp
  qed
```

qed

lemma *observable-io-targets* :
  assumes *observable M*
  and $io \in LS\ M\ q$
obtains q'
where *io-targets M io q* $= \{q'\}$
proof −

  obtain *p* **where** *path M q p* **and** *p-io p* = *io*
    using *assms*(*2*) **by** *auto*
  then have *target q p* ∈ *io-targets M io q*
    by *auto*

  have ∃ q' . *io-targets M io q* $= \{q'\}$
  proof (*rule ccontr*)
    assume ¬(∃ q'. *io-targets M io q* $= \{q'\}$)
    then have ∃ q' . $q' \neq$ *target q p* ∧ $q' \in$ *io-targets M io q*
    proof −
      have ¬ *io-targets M io q* ⊆ {*target q p*}
        using ‹¬(∃ q'. *io-targets M io q* $= \{q'\}$)› ‹*target q p* ∈ *io-targets M io q*› **by** *blast*
      then show *?thesis*
        by *blast*
    qed
    then obtain q' **where** $q' \neq$ *target q p* **and** $q' \in$ *io-targets M io q*
      by *blast*
    then obtain p' **where** *path M q* p' **and** *target q* $p' = q'$ **and** *p-io* p' = *io*
      by *auto*
    then have *p-io p* = *p-io* p'
      using ‹*p-io p* = *io*› **by** *simp*
    then have $p = p'$
      using *observable-path-unique*[*OF assms*(*1*) ‹*path M q p*› ‹*path M q* p'›] **by** *simp*
    then show *False*
      using ‹$q' \neq$ *target q p*› ‹*target q* $p' = q'$› **by** *auto*
  qed

  then show *?thesis* **using** *that* **by** *blast*
qed

lemma *observable-path-io-target* :
  assumes *observable M*
  and *path M q p*
shows *io-targets M* (*p-io p*) *q* = {*target q p*}
  using *observable-io-targets*[*OF assms*(*1*) *language-state-containment*[*OF assms*(*2*)], *of p-io p*]

```
      singletonD[of target q p]
  unfolding io-targets.simps
proof −
  assume a1: ⋀a. target q p ∈ {a} ⟹ target q p = a
  assume ⋀thesis. ⟦p-io p = p-io p; ⋀q'. {target q pa |pa. path M q pa ∧ p-io pa
= p-io p} = {q'} ⟹ thesis⟧ ⟹ thesis
  then obtain aa :: 'a where ⋀b. {target q ps |ps. path M q ps ∧ p-io ps = p-io
p} = {aa} ∨ b
    by meson
  then show {target q ps |ps. path M q ps ∧ p-io ps = p-io p} = {target q p}
    using a1 assms(2) by blast
qed


lemma completely-specified-io-targets :
  assumes completely-specified M
  shows ∀ q ∈ io-targets M io (initial M) . ∀ x ∈ (inputs M) . ∃ t ∈ transitions
M . t-source t = q ∧ t-input t = x
  by (meson assms completely-specified.elims(2) io-targets-states subsetD)



lemma observable-path-language-step :
  assumes observable M
      and path M q p
      and ¬ (∃ t∈transitions M.
              t-source t = target q p ∧
              t-input t = x ∧ t-output t = y)
    shows (p-io p)@[(x,y)] ∉ LS M q
using assms proof (induction p rule: rev-induct)
  case Nil
  show ?case proof
    assume p-io [] @ [(x, y)] ∈ LS M q
    then obtain p' where path M q p' and p-io p' = [(x,y)] unfolding LS.simps
      by force
    then obtain t where p' = [t] by blast

    have t∈transitions M and t-source t = target q []
      using ‹path M q p'› ‹p' = [t]› by auto
    moreover have t-input t = x ∧ t-output t = y
      using ‹p-io p' = [(x,y)]› ‹p' = [t]› by auto
    ultimately show False
      using Nil.prems(3) by blast
  qed
next
  case (snoc t p)

  from ‹path M q (p @ [t])› have path M q p and t-source t = target q p
                                  and t ∈ transitions M
```

by *auto*

show *?case* **proof**
assume *p-io* $(p$ @ $[t])$ @ $[(x, y)] \in$ *LS M q*
then obtain p' **where** *path M q* p' **and** *p-io* p' = *p-io* $(p$ @ $[t])$ @ $[(x, y)]$
by *auto*
then obtain p'' t' t'' **where** $p' = p''@[t']@[t'']$
by (*metis* (*no-types*, *lifting*) *append.assoc map-butlast map-is-Nil-conv snoc-eq-iff-butlast*)
then have *path M q* p''
using ‹*path M q* p'› **by** *blast*

have *p-io* p'' = *p-io p*
using ‹$p' = p''@[t']@[t'']$› ‹*p-io* p' = *p-io* $(p$ @ $[t])$ @ $[(x, y)]$› **by** *auto*
then have $p'' = p$
using *observable-path-unique*[*OF assms*(*1*) ‹*path M q* p''› ‹*path M q p*›] **by** *blast*

have *t-source* t' = *target q* p'' **and** $t' \in$ *transitions M*
using ‹*path M q* p'› ‹$p' = p''@[t']@[t'']$› **by** *auto*
then have *t-source* t' = *t-source t*
using ‹$p'' = p$› ‹*t-source t* = *target q p*› **by** *auto*
moreover have *t-input* t' = *t-input t* $\wedge$ *t-output* t' = *t-output t*
using ‹*p-io* p' = *p-io* $(p$ @ $[t])$ @ $[(x, y)]$› ‹$p' = p''@[t']@[t'']$› ‹$p'' = p$› **by** *auto*
ultimately have $t' = t$
using ‹$t \in$ *transitions M*› ‹$t' \in$ *transitions M*› *assms*(*1*) **unfolding** *observable.simps*
by (*meson prod.expand*)

have $t'' \in$ *transitions M* **and** *t-source* t'' = *target q* $(p@[t])$
using ‹*path M q* p'› ‹$p' = p''@[t']@[t'']$› ‹$p'' = p$› ‹$t' = t$› **by** *auto*
moreover have *t-input* $t'' = x \wedge$ *t-output* $t'' = y$
using ‹*p-io* p' = *p-io* $(p$ @ $[t])$ @ $[(x, y)]$› ‹$p' = p''@[t']@[t'']$› **by** *auto*
ultimately show *False*
using *snoc.prems*(*3*) **by** *blast*
qed
qed

lemma *observable-io-targets-language* :
assumes *io1* @ *io2* $\in$ *LS M q1*
and *observable M*
and *q2* $\in$ *io-targets M io1 q1*
shows *io2* $\in$ *LS M q2*
proof −
obtain *p1 p2* **where** *path M q1 p1* **and** *path M* (*target q1 p1*) *p2*
and *p-io p1* = *io1* **and** *p-io p2* = *io2*
using *language-state-split*[*OF assms*(*1*)] **by** *blast*
then have *io1* $\in$ *LS M q1* **and** *io2* $\in$ *LS M* (*target q1 p1*)

by *auto*

have *target q1 p1* ∈ *io-targets M io1 q1*
using ‹*path M q1 p1*› ‹*p-io p1* = *io1*›
unfolding *io-targets.simps* **by** *blast*
then have *target q1 p1* = *q2*
using *observable-io-targets*[*OF assms*(*2*) ‹*io1* ∈ *LS M q1*›]
by (*metis assms*(*3*) *singletonD*)
then show *?thesis*
using ‹*io2* ∈ *LS M* (*target q1 p1*)› **by** *auto*
qed

lemma *io-targets-language-append* :
assumes *q1* ∈ *io-targets M io1 q*
and *io2* ∈ *LS M q1*
shows *io1*@*io2* ∈ *LS M q*
proof −
obtain *p1* **where** *path M q p1* **and** *p-io p1* = *io1* **and** *target q p1* = *q1*
using *assms*(*1*) **by** *auto*
moreover obtain *p2* **where** *path M q1 p2* **and** *p-io p2* = *io2*
using *assms*(*2*) **by** *auto*
ultimately have *path M q* (*p1*@*p2*) **and** *p-io* (*p1*@*p2*) = *io1*@*io2*
by *auto*
then show *?thesis*
using *language-state-containment*[*of M q p1*@*p2 io1*@*io2*] **by** *simp*
qed

lemma *io-targets-next* :
assumes *t* ∈ *transitions M*
shows *io-targets M io* (*t-target t*) ⊆ *io-targets M* (*p-io* [*t*] @ *io*) (*t-source t*)
unfolding *io-targets.simps*
proof
fix *q* **assume** *q* ∈ {*target* (*t-target t*) *p* |*p*. *path M* (*t-target t*) *p* ∧ *p-io p* = *io*}
then obtain *p* **where** *path M* (*t-target t*) *p* ∧ *p-io p* = *io* ∧ *target* (*t-target t*) *p* = *q*
by *auto*
then have *path M* (*t-source t*) (*t*#*p*) ∧ *p-io* (*t*#*p*) = *p-io* [*t*] @ *io* ∧ *target* (*t-source t*) (*t*#*p*) = *q*
using *FSM.path.cons*[*OF assms*] **by** *auto*
then show *q* ∈ {*target* (*t-source t*) *p* |*p*. *path M* (*t-source t*) *p* ∧ *p-io p* = *p-io* [*t*] @ *io*}
by *blast*
qed

lemma *observable-io-targets-next* :
assumes *observable M*

```
  and     t ∈ transitions M
shows io-targets M (p-io [t] @ io) (t-source t) = io-targets M io (t-target t)
proof
  show io-targets M (p-io [t] @ io) (t-source t) ⊆ io-targets M io (t-target t)
  proof
    fix q assume q ∈ io-targets M (p-io [t] @ io) (t-source t)
    then obtain p where q = target (t-source t) p
                  and path M (t-source t) p
                  and p-io p = p-io [t] @ io
      unfolding io-targets.simps by blast
    then have q = t-target (last p) unfolding target.simps visited-states.simps
      using last-map by auto

    obtain t′ p′ where p = t′ # p′
      using ‹p-io p = p-io [t] @ io› by auto
    then have t′ ∈ transitions M and t-source t′ = t-source t
      using ‹path M (t-source t) p› by auto
    moreover have t-input t′ = t-input t and t-output t′ = t-output t
      using ‹p = t′ # p′› ‹p-io p = p-io [t] @ io› by auto
    ultimately have t′ = t
      using ‹t ∈ transitions M› ‹observable M› unfolding observable.simps
      by (meson prod.expand)

    then have path M (t-target t) p′
      using ‹path M (t-source t) p› ‹p = t′ # p′› by auto
    moreover have p-io p′ = io
      using ‹p-io p = p-io [t] @ io› ‹p = t′ # p′› by auto
    moreover have q = target (t-target t) p′
      using ‹q = target (t-source t) p› ‹p = t′ # p′› ‹t′ = t› by auto
    ultimately show q ∈ io-targets M io (t-target t)
      by auto
  qed

  show io-targets M io (t-target t) ⊆ io-targets M (p-io [t] @ io) (t-source t)
    using io-targets-next[OF assms(2)] by assumption
qed


lemma observable-language-target :
  assumes observable M
  and     q ∈ io-targets M io1 (initial M)
  and     t ∈ io-targets T io1 (initial T)
  and     L T ⊆ L M
shows LS T t ⊆ LS M q
proof
  fix io2 assume io2 ∈ LS T t
  then obtain pT2 where path T t pT2 and p-io pT2 = io2
    by auto
```

obtain *pT1* **where** *path T* (*initial T*) *pT1* **and** *p-io pT1* = *io1* **and** *target* (*initial T*) *pT1* = *t*
using ‹*t* ∈ *io-targets T io1* (*initial T*)› **by** *auto*
then have *path T* (*initial T*) (*pT1*@*pT2*)
using ‹*path T t pT2*› **using** *path-append* **by** *metis*
moreover have *p-io* (*pT1*@*pT2*) = *io1*@*io2*
using ‹*p-io pT1* = *io1*› ‹*p-io pT2* = *io2*› **by** *auto*
ultimately have *io1*@*io2* ∈ *L T*
using *language-state-containment*[*of T*] **by** *auto*
then have *io1*@*io2* ∈ *L M*
using ‹*L T* ⊆ *L M*› **by** *blast*
then obtain *pM* **where** *path M* (*initial M*) *pM* **and** *p-io pM* = *io1*@*io2*
by *auto*

let *?pM1* = *take* (*length io1*) *pM*
let *?pM2* = *drop* (*length io1*) *pM*

have *path M* (*initial M*) (*?pM1*@*?pM2*)
using ‹*path M* (*initial M*) *pM*› **by** *auto*
then have *path M* (*initial M*) *?pM1* **and** *path M* (*target* (*initial M*) *?pM1*) *?pM2*
by *blast*+

have *p-io ?pM1* = *io1*
using ‹*p-io pM* = *io1*@*io2*›
by (*metis append-eq-conv-conj take-map*)
have *p-io ?pM2* = *io2*
using ‹*p-io pM* = *io1*@*io2*›
by (*metis append-eq-conv-conj drop-map*)

obtain *pM1* **where** *path M* (*initial M*) *pM1* **and** *p-io pM1* = *io1* **and** *target* (*initial M*) *pM1* = *q*
using ‹*q* ∈ *io-targets M io1* (*initial M*)› **by** *auto*

have *pM1* = *?pM1*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*initial M*) *pM1*› ‹*path M* (*initial M*) *?pM1*›]
unfolding ‹*p-io pM1* = *io1*› ‹*p-io ?pM1* = *io1*› **by** *simp*

then have *path M q ?pM2*
using ‹*path M* (*target* (*initial M*) *?pM1*) *?pM2*› ‹*target* (*initial M*) *pM1* = *q*› **by** *auto*
then show *io2* ∈ *LS M q*
using *language-state-containment*[*OF* - ‹*p-io ?pM2* = *io2*›, *of M*] **by** *auto*
qed

lemma *observable-language-target-failure* :

assumes *observable M*
and $q \in$ *io-targets M io1* (*initial M*)
and $t \in$ *io-targets T io1* (*initial T*)
and $\neg$ *LS T t* $\subseteq$ *LS M q*
shows $\neg$ *L T* $\subseteq$ *L M*
using *observable-language-target*[*OF assms*(*1*,*2*,*3*)] *assms*(*4*) **by** *blast*

lemma *language-path-append-transition-observable* :
assumes (*p-io p*) @ [(*x*,*y*)] $\in$ *LS M q*
and *path M q p*
and *observable M*
obtains *t* **where** *path M q* (*p*@[*t*]) **and** *t-input t* = *x* **and** *t-output t* = *y*
proof −
obtain *p′ t* **where** *path M q* (*p′*@[*t*]) **and** *p-io* (*p′*@[*t*]) = (*p-io p*) @ [(*x*,*y*)]
using *language-path-append-transition*[*OF assms*(*1*)] **by** *blast*
then have *path M q p′* **and** *p-io p′* = *p-io p* **and** *t-input t* = *x* **and** *t-output t* = *y*
by *auto*

have *p′* = *p*
using *observable-path-unique*[*OF assms*(*3*) ‹*path M q p′*› ‹*path M q p*› ‹*p-io p′* = *p-io p*›] **by** *assumption*
then have *path M q* (*p*@[*t*])
using ‹*path M q* (*p′*@[*t*])› **by** *auto*
then show *?thesis* **using** *that* ‹*t-input t* = *x*› ‹*t-output t* = *y*› **by** *metis*
qed

lemma *language-io-target-append* :
assumes *q′* $\in$ *io-targets M io1 q*
and *io2* $\in$ *LS M q′*
shows (*io1*@*io2*) $\in$ *LS M q*
proof −
obtain *p2* **where** *path M q′ p2* **and** *p-io p2* = *io2*
using *assms*(*2*) **by** *auto*

moreover obtain *p1* **where** *q′* = *target q p1* **and** *path M q p1* **and** *p-io p1* = *io1*
using *assms*(*1*) **by** *auto*

ultimately show *?thesis* **unfolding** *LS.simps*
by (*metis* (*mono-tags*, *lifting*) *map-append mem-Collect-eq path-append*)
qed

lemma *observable-path-suffix* :
assumes (*p-io p*)@*io* $\in$ *LS M q*
and *path M q p*

and *observable M*
obtains p' **where** *path M* (*target q p*) p' **and** *p-io* p' = *io*
proof −
obtain *p1 p2* **where** *path M q p1* **and** *path M* (*target q p1*) *p2* **and** *p-io p1* = *p-io p* **and** *p-io p2* = *io*
using *language-state-split*[*OF assms*(*1*)] **by** *blast*

have *p1* = *p*
using *observable-path-unique*[*OF assms*(*3*,*2*) ‹*path M q p1*› ‹*p-io p1* = *p-io p*›[*symmetric*]]
by *simp*

show *?thesis* **using** *that*[*of p2*] ‹*path M* (*target q p1*) *p2*› ‹*p-io p2* = *io*› **unfolding** ‹*p1* = *p*›
by *blast*
qed

lemma *io-targets-finite* :
finite (*io-targets M io q*)
proof −
have (*io-targets M io q*) ⊆ {*target q p* | *p* . *path M q p* ∧ *length p* ≤ *length io*}
unfolding *io-targets.simps length-map*[*of* (λ *t* . (*t-input t*, *t-output t*)), *symmetric*] **by** *force*
moreover have *finite* {*target q p* | *p* . *path M q p* ∧ *length p* ≤ *length io*}
using *paths-finite*[*of M q length io*]
by *simp*
ultimately show *?thesis*
using *rev-finite-subset* **by** *blast*
qed

lemma *language-next-transition-ob* :
assumes (*x*,*y*)#*ios* ∈ *LS M q*
obtains *t* **where** *t-source t* = *q*
and *t* ∈ *transitions M*
and *t-input t* = *x*
and *t-output t* = *y*
and *ios* ∈ *LS M* (*t-target t*)
proof −
obtain *p* **where** *path M q p* **and** *p-io p* = (*x*,*y*)#*ios*
using *assms* **unfolding** *LS.simps mem-Collect-eq*
by (*metis* (*no-types*, *lifting*))
then obtain *t* p' **where** *p* = *t*#p'
by *blast*

have *t-source t* = *q*
and *t* ∈ *transitions M*
and *path M* (*t-target t*) p'
using ‹*path M q p*› **unfolding** ‹*p* = *t*#p'› **by** *auto*

moreover have *t-input t* = *x*
and *t-output t* = *y*
and *p-io p′* = *ios*
using ‹*p-io p* = (*x*,*y*)#*ios*› **unfolding** ‹*p* = *t*#*p′*› **by** *auto*
ultimately show *?thesis* **using** *that*[*of t*] **by** *auto*
qed

lemma *h-observable-card* :
assumes *observable M*
shows *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) ≤ *1*
and *finite* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)))
proof −
have *snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)) = {*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*}
unfolding *h.simps* **by** *force*
moreover have {*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {} ∨ (∃ *q′* . {*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {*q′*})
using *assms* **unfolding** *observable-alt-def* **by** *blast*
ultimately show *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) ≤ *1*
and *finite* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)))
by *auto*
qed

lemma *h-obs-None* :
assumes *observable M*
shows (*h-obs M q x y* = *None*) = (∄ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*)
proof
show (*h-obs M q x y* = *None*) ⟹ (∄ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*)
proof −
assume *h-obs M q x y* = *None*
then have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) ≠ *1*
by *auto*
then have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = *0*
using *h-observable-card*(*1*)[*OF assms*, *of y q x*] **by** *presburger*
then have (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = {}
using *h-observable-card*(*2*)[*OF assms*, *of y q x*] *card-0-eq*[*of* (*snd* ‘ *Set.filter* (λ(*y′*, *q′*). *y′* = *y*) (*h M* (*q*, *x*)))] **by** *blast*
then show *?thesis*
unfolding *h.simps* **by** *force*
qed
show (∄ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*) ⟹ (*h-obs M q x y* = *None*)
proof −
assume (∄ *q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*)
then have *snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)) = {}
unfolding *h.simps* **by** *force*
then have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = *0*
by (*simp add*: *card-eq-0-iff*)
then show *?thesis*
unfolding *h-obs-simps Let-def* ‹*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))

= {}›
    by *auto*
  qed
qed

lemma *h-obs-Some* :
  assumes *observable M*
  shows (*h-obs M q x y = Some q′*) = ({*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {*q′*})
proof
  have ∗: *snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)) = {*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*}
    unfolding *h.simps* **by** *force*

  show *h-obs M q x y = Some q′* ⟹ ({*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {*q′*})
  proof −
    assume *h-obs M q x y = Some q′*
    then have (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) ≠ {}
      by (*auto simp add*: *card-1-singleton-iff split*: *if-splits*)
    then have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) > *0*
      unfolding *h-simps* **using** *fsm-transitions-finite*[*of M*]
      by (*metis assms card-0-eq h-observable-card*(*2*) *h-simps neq0-conv*)
    moreover have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) ≤ *1*
      using *assms* **unfolding** *observable-alt-def h-simps*
      by (*metis assms h-observable-card*(*1*) *h-simps*)
    ultimately have *card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = *1*
      by *auto*
    then have (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = {*q′*}
      using ‹*h-obs M q x y = Some q′*› **unfolding** *h-obs-simps Let-def*
      by (*metis card-1-singletonE option.inject the-elem-eq*)
    then show *?thesis*
      using ∗ **unfolding** *h.simps* **by** *blast*
  qed

  show ({*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {*q′*}) ⟹ (*h-obs M q x y = Some q′*)
  proof −
    assume ({*q′* . (*q*,*x*,*y*,*q′*) ∈ *transitions M*} = {*q′*})
    then have *snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*)) = {*q′*}
      unfolding *h.simps* **by** *force*
    then show *?thesis*
      unfolding *Let-def*
      by *simp*
  qed
qed

lemma *h-obs-state* :
  assumes *h-obs M q x y = Some q′*
  shows *q′* ∈ *states M*
proof (*cases card* (*snd* ‘ *Set.filter* (λ (*y′*,*q′*) . *y′* = *y*) (*h M* (*q*,*x*))) = *1*)
  case *True*

```
  then have (snd ‘ Set.filter (λ (y′,q′) . y′ = y) (h M (q,x))) = {q′}
    using ‹h-obs M q x y = Some q′› unfolding h-obs-simps Let-def
    by (metis card-1-singletonE option.inject the-elem-eq)
  then have (q,x,y,q′) ∈ transitions M
    unfolding h-simps by auto
  then show ?thesis
    by (metis fsm-transition-target snd-conv)
next
  case False
  then have h-obs M q x y = None
    using False unfolding h-obs-simps Let-def by auto
  then show ?thesis using assms by auto
qed


fun after :: (′a,′b,′c) fsm ⇒ ′a ⇒ (′b × ′c) list ⇒ ′a where
  after M q [] = q |
  after M q ((x,y)#io) = after M (the (h-obs M q x y)) io


abbreviation after-initial M io ≡ after M (initial M) io

lemma after-path :
  assumes observable M
  and     path M q p
shows after M q (p-io p) = target q p
using assms(2) proof (induction p arbitrary: q rule: list.induct)
  case Nil
  then show ?case by auto
next
  case (Cons t p)
  then have t ∈ transitions M and path M (t-target t) p and t-source t = q
    by auto

  have ⋀ q′ . (q, t-input t, t-output t, q′) ∈ FSM.transitions M ⟹ q′ = t-target t
    using observable-transition-unique[OF assms(1) ‹t ∈ transitions M›] ‹t ∈ tran-
sitions M›
    using ‹t-source t = q› assms(1) by auto
  then have ({q′. (q, t-input t, t-output t, q′) ∈ FSM.transitions M} = {t-target
t})
    using ‹t ∈ transitions M› ‹t-source t = q› by auto
  then have (h-obs M q (t-input t) (t-output t)) = Some (t-target t)
    using h-obs-Some[OF assms(1), of q t-input t t-output t t-target t]
    by blast
  then have after M q (p-io (t#p)) = after M (t-target t) (p-io p)
    by auto
  moreover have target (t-target t) p = target q (t#p)
    using ‹t-source t = q› by auto
  ultimately show ?case
```

```
    using Cons.IH[OF ‹path M (t-target t) p›]
    by simp
qed

lemma observable-after-path :
  assumes observable M
  and     io ∈ LS M q
obtains p where path M q p
           and p-io p = io
           and target q p = after M q io
  using after-path[OF assms(1)]
  using assms(2) by auto

lemma h-obs-from-LS :
  assumes observable M
  and     [(x,y)] ∈ LS M q
obtains q′ where h-obs M q x y = Some q′
  using assms(2) h-obs-None[OF assms(1), of q x y] by force

lemma after-h-obs :
  assumes observable M
  and     h-obs M q x y = Some q′
shows after M q [(x,y)] = q′
proof −
  have path M q [(q,x,y,q′)]
    using assms(2) unfolding h-obs-Some[OF assms(1)]
    using single-transition-path by fastforce
  then show ?thesis
    using assms(2) after-path[OF assms(1), of q [(q,x,y,q′)]] by auto
qed

lemma after-h-obs-prepend :
  assumes observable M
  and     h-obs M q x y = Some q′
  and     io ∈ LS M q′
shows after M q ((x,y)#io) = after M q′ io
proof −
  obtain p where path M q′ p and p-io p = io
    using assms(3) by auto
  then have after M q′ io = target q′ p
    using after-path[OF assms(1)]
    by blast

  have path M q ((q,x,y,q′)#p)
   using assms(2) path-prepend-t[OF ‹path M q′ p›, of q x y] unfolding h-obs-Some[OF
assms(1)] by auto
  moreover have p-io ((q,x,y,q′)#p) = (x,y)#io
    using ‹p-io p = io› by auto
  ultimately have after M q ((x,y)#io) = target q ((q,x,y,q′)#p)
```

using *after-path*[*OF assms*(*1*), *of q* (*q*,*x*,*y*,*q*′)#*p*] **by** *simp*
moreover have *target q* ((*q*,*x*,*y*,*q*′)#*p*) = *target q*′ *p*
by *auto*
ultimately show *?thesis*
using ‹*after M q*′ *io* = *target q*′ *p*› **by** *simp*
qed

lemma *after-split* :
assumes *observable M*
and $\alpha@\gamma \in LS\ M\ q$
shows *after M* (*after M q* α) γ = *after M q* (α @ γ)
proof −
obtain *p1 p2* **where** *path M q p1* **and** *path M* (*target q p1*) *p2* **and** *p-io p1* = α **and** *p-io p2* = γ
using *language-state-split*[*OF assms*(*2*)]
by *blast*
then have *path M q* (*p1*@*p2*) **and** *p-io* (*p1*@*p2*) = (α @ γ)
by *auto*
then have *after M q* (α @ γ) = *target q* (*p1*@*p2*)
using *assms*(*1*)
by (*metis* (*mono-tags*, *lifting*) *after-path*)
moreover have *after M q* α = *target q p1*
using ‹*path M q p1*› ‹*p-io p1* = α› *assms*(*1*)
by (*metis* (*mono-tags*, *lifting*) *after-path*)
moreover have *after M* (*target q p1*) γ = *target* (*target q p1*) *p2*
using ‹*path M* (*target q p1*) *p2*› ‹*p-io p2* = γ› *assms*(*1*)
by (*metis* (*mono-tags*, *lifting*) *after-path*)
moreover have *target* (*target q p1*) *p2* = *target q* (*p1*@*p2*)
by *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *after-io-targets* :
assumes *observable M*
and $io \in LS\ M\ q$
shows *after M q io* = *the-elem* (*io-targets M io q*)
proof −
have *after M q io* ∈ *io-targets M io q*
using *after-path*[*OF assms*(*1*)] *assms*(*2*)
unfolding *io-targets.simps LS.simps*
by *blast*
then show *?thesis*
using *observable-io-targets*[*OF assms*]
by (*metis singletonD the-elem-eq*)
qed

lemma *after-language-subset* :
assumes *observable M*
and $\alpha@\gamma \in L\ M$
and $\beta \in LS\ M\ (after\text{-}initial\ M\ (\alpha@\gamma))$
shows $\gamma@\beta \in LS\ M\ (after\text{-}initial\ M\ \alpha)$
by (*metis after-io-targets after-split assms*(*1*) *assms*(*2*) *assms*(*3*) *language-io-target-append language-prefix observable-io-targets observable-io-targets-language singletonI the-elem-eq*)

lemma *after-language-append-iff* :
assumes *observable M*
and $\alpha@\gamma \in L\ M$
shows $\beta \in LS\ M\ (after\text{-}initial\ M\ (\alpha@\gamma)) = (\gamma@\beta \in LS\ M\ (after\text{-}initial\ M\ \alpha))$
by (*metis after-io-targets after-language-subset after-split assms*(*1*) *assms*(*2*) *language-prefix observable-io-targets observable-io-targets-language singletonI the-elem-eq*)

lemma *h-obs-language-iff* :
assumes *observable M*
shows $(x,y)\#io \in LS\ M\ q = (\exists\ q'\ .\ h\text{-}obs\ M\ q\ x\ y = Some\ q' \wedge io \in LS\ M\ q')$
(**is** *?P1* = *?P2*)
proof
show *?P1* $\Longrightarrow$ *?P2*
proof −
assume *?P1*
then obtain *t p* **where** $t \in transitions\ M$
and *path M* (*t-target t*) *p*
and *t-input t* = *x*
and *t-output t* = *y*
and *t-source t* = *q*
and *p-io p* = *io*
by *auto*
then have (*q*,*x*,*y*,*t-target t*) $\in$ *transitions M*
by *auto*
then have *h-obs M q x y* = *Some* (*t-target t*)
unfolding *h-obs-Some*[*OF assms*]
using *assms* **by** *auto*
moreover have *io* $\in$ *LS M* (*t-target t*)
using ‹*path M* (*t-target t*) *p*› ‹*p-io p* = *io*›
by *auto*
ultimately show *?P2*
by *blast*
qed
show *?P2* $\Longrightarrow$ *?P1*
unfolding *h-obs-Some*[*OF assms*] **using** *LS-prepend-transition*[**where** *io*=*io* **and** *M*=*M*]
by (*metis fst-conv mem-Collect-eq singletonI snd-conv*)

qed

lemma *after-language-iff* :
  assumes *observable M*
  and $\alpha \in LS\ M\ q$
shows $(\gamma \in LS\ M\ (after\ M\ q\ \alpha)) = (\alpha@\gamma \in LS\ M\ q)$
  by (*metis after-io-targets assms*(*1*) *assms*(*2*) *language-io-target-append observable-io-targets observable-io-targets-language singletonI the-elem-eq*)

lemma *language-maximal-contained-prefix-ob* :
  assumes $io \notin LS\ M\ q$
  and $q \in states\ M$
  and *observable M*
obtains $io'\ x\ y\ io''$ **where** $io = io'@[(x,y)]@io''$
                and $io' \in LS\ M\ q$
                and $io'@[(x,y)] \notin LS\ M\ q$
proof −
  have $\exists\ io'\ x\ y\ io''\ .\ io = io'@[(x,y)]@io'' \wedge io' \in LS\ M\ q \wedge io'@[(x,y)] \notin LS\ M\ q$
    using *assms*(*1*,*2*) **proof** (*induction io arbitrary*: *q*)
    case *Nil*
    then show *?case* **by** *auto*
  next
    case (*Cons xy io*)

    obtain *x y* **where** $xy = (x,y)$
      by *fastforce*

    show *?case* **proof** (*cases h-obs M q x y*)
      case *None*
      then have $[]@[(x,y)] \notin LS\ M\ q$
        unfolding *h-obs-None*[*OF assms*(*3*)] **by** *auto*
      moreover have $[] \in LS\ M\ q$
        using *Cons.prems* **by** *auto*
      moreover have $(x,y)\#io = []@[(x,y)]@io$
        using *Cons.prems*
        unfolding ‹$xy = (x,y)$› **by** *auto*
      ultimately show *?thesis*
        unfolding ‹$xy = (x,y)$› **by** *blast*
    next
      case (*Some q'*)
      then have $io \notin LS\ M\ q'$
        using *h-obs-language-iff*[*OF assms*(*3*), *of x y io q*] *Cons.prems*(*1*)
        unfolding ‹$xy = (x,y)$›
        by *auto*
      then obtain $io'\ x'\ y'\ io''$ **where** $io = io'@[(x',y')]@io''$
                and $io' \in LS\ M\ q'$
                and $io'@[(x',y')] \notin LS\ M\ q'$

```
      using Cons.IH[OF - h-obs-state[OF Some]]
      by blast

    have xy#io = (xy#io')@[(x',y')]@io''
      using ‹io = io'@[(x',y')]@io''› by auto
    moreover have (xy#io') ∈ LS M q
      using ‹io' ∈ LS M q'› Some
      unfolding ‹xy = (x,y)› h-obs-language-iff[OF assms(3)]
      by blast
    moreover have (xy#io')@[(x',y')] ∉ LS M q
      using ‹io'@[(x',y')] ∉ LS M q'› Some h-obs-language-iff[OF assms(3), of x
y io'@[(x',y')] q]
      unfolding ‹xy = (x,y)›
      by auto
    ultimately show ?thesis
      by blast
  qed
 qed
 then show ?thesis
  using that by blast
qed

lemma after-is-state :
 assumes observable M
 assumes io ∈ LS M q
 shows FSM.after M q io ∈ states M
 using assms
 by (metis observable-after-path path-target-is-state)

lemma after-reachable-initial :
 assumes observable M
 and     io ∈ L M
shows after-initial M io ∈ reachable-states M
proof −
 obtain p where path M (initial M) p and p-io p = io
  using assms(2) by auto
 then have after-initial M io = target (initial M) p
  using after-path[OF assms(1)]
  by blast
 then show ?thesis
  unfolding reachable-states-def using ‹path M (initial M) p› by blast
qed

lemma after-transition :
 assumes observable M
 and     (q,x,y,q') ∈ transitions M
shows after M q [(x,y)] = q'
 using after-path[OF assms(1) single-transition-path[OF assms(2)]]
 by auto
```

lemma *after-transition-exhaust* :
  assumes *observable M*
  and $t \in$ *transitions M*
shows *t-target t* = *after M* (*t-source t*) [(*t-input t*, *t-output t*)]
  using *after-transition*[*OF assms*(*1*)] *assms*(*2*)
  by (*metis surjective-pairing*)

lemma *after-reachable* :
  assumes *observable M*
  and *io* $\in$ *LS M q*
  and *q* $\in$ *reachable-states M*
shows *after M q io* $\in$ *reachable-states M*
proof −
  obtain *p* **where** *path M q p* **and** *p-io p* = *io*
    using *assms*(*2*) **by** *auto*
  then have *after M q io* = *target q p*
    using *after-path*[*OF assms*(*1*)] **by** *force*

  obtain *p′* **where** *path M* (*initial M*) *p′* **and** *target* (*initial M*) *p′* = *q*
    using *assms*(*3*) **unfolding** *reachable-states-def* **by** *blast*

  then have *path M* (*initial M*) (*p′*@*p*)
    using ‹*path M q p*› **by** *auto*
  moreover have *after M q io* = *target* (*initial M*) (*p′*@*p*)
    using ‹*target* (*initial M*) *p′* = *q*›
    unfolding ‹*after M q io* = *target q p*›
    by *auto*
  ultimately show *?thesis*
    unfolding *reachable-states-def* **by** *blast*
qed

lemma *observable-after-language-append* :
  assumes *observable M*
  and *io1* $\in$ *LS M q*
  and *io2* $\in$ *LS M* (*after M q io1*)
shows *io1*@*io2* $\in$ *LS M q*
  using *observable-after-path*[*OF assms*(*1*,*2*)] *assms*(*3*)
proof −
  assume *a1*: $\bigwedge$*thesis*. ($\bigwedge$*p*. ⟦*path M q p*; *p-io p* = *io1*; *target q p* = *after M q io1*⟧ $\Longrightarrow$ *thesis*) $\Longrightarrow$ *thesis*
  have $\exists$ *ps*. *io2* = *p-io ps* $\wedge$ *path M* (*after M q io1*) *ps*
    using ‹*io2* $\in$ *LS M* (*after M q io1*)› **by** *auto*
  moreover
  { assume ($\exists$ *ps*. *io2* = *p-io ps* $\wedge$ *path M* (*after M q io1*) *ps*) $\wedge$ ($\forall$ *ps*. *io1* @ *io2* $\neq$ *p-io ps* $\vee$ $\neg$ *path M q ps*)
    then have *io1* @ *io2* $\in$ {*p-io ps* |*ps*. *path M q ps*}
      using *a1* **by** (*metis* (*lifting*) *map-append path-append*) **}**
  ultimately show *?thesis*

by *auto*
qed

lemma *observable-after-language-none* :
assumes *observable M*
and *io1* $\in$ *LS M q*
and *io2* $\notin$ *LS M* (*after M q io1*)
shows *io1@io2* $\notin$ *LS M q*
using *after-path*[*OF assms*(*1*)] *language-state-split*[*of io1 io2 M q*]
by (*metis* (*mono-tags*, *lifting*) *assms*(*3*) *language-intro*)

lemma *observable-after-eq* :
assumes *observable M*
and *after M q io1* = *after M q io2*
and *io1* $\in$ *LS M q*
and *io2* $\in$ *LS M q*
shows *io1@io* $\in$ *LS M q* $\longleftrightarrow$ *io2@io* $\in$ *LS M q*
using *observable-after-language-append*[*OF assms*(*1*,*3*), *of io*]
observable-after-language-append[*OF assms*(*1*,*4*), *of io*]
assms(*2*)
by (*metis assms*(*1*) *language-prefix observable-after-language-none*)

lemma *observable-after-target* :
assumes *observable M*
and *io @ io'* $\in$ *LS M q*
and *path M* (*FSM.after M q io*) *p*
and *p-io p* = *io'*
shows *target* (*FSM.after M q io*) *p* = (*FSM.after M q* (*io @ io'*))
proof −
obtain *p'* **where** *path M q p'* **and** *p-io p'* = *io @ io'*
using ‹*io @ io'* $\in$ *LS M q*› **by** *auto*

then have *path M q* (*take* (*length io*) *p'*)
and *p-io* (*take* (*length io*) *p'*) = *io*
and *path M* (*target q* (*take* (*length io*) *p'*)) (*drop* (*length io*) *p'*)
and *p-io* (*drop* (*length io*) *p'*) = *io'*
using *path-io-split*[*of M q p' io io'*]
by *auto*
then have *FSM.after M q io* = *target q* (*take* (*length io*) *p'*)
using *after-path assms*(*1*) **by** *fastforce*
then have *p* = (*drop* (*length io*) *p'*)
using ‹*path M* (*target q* (*take* (*length io*) *p'*)) (*drop* (*length io*) *p'*)› ‹*p-io* (*drop* (*length io*) *p'*) = *io'*›
assms(*3*,*4*)
observable-path-unique[*OF* ‹*observable M*›]
by *force*

```
  have (FSM.after M q (io @ io')) = target q p'
    using after-path[OF ‹observable M› ‹path M q p'›] unfolding ‹p-io p' = io @
io'› .
  moreover have target (FSM.after M q io) p = target q p'
    using ‹FSM.after M q io = target q (take (length io) p')›
    by (metis ‹p = drop (length io) p'› append-take-drop-id path-append-target)
  ultimately show ?thesis
    by simp
qed


fun is-in-language :: ('a,'b,'c) fsm ⇒ 'a ⇒ ('b ×'c) list ⇒ bool where
  is-in-language M q [] = True |
  is-in-language M q ((x,y)#io) = (case h-obs M q x y of
    None ⇒ False |
    Some q' ⇒ is-in-language M q' io)

lemma is-in-language-iff :
  assumes observable M
  and     q ∈ states M
  shows is-in-language M q io ⟷ io ∈ LS M q
using assms(2) proof (induction io arbitrary: q)
  case Nil
  then show ?case
    by auto
next
  case (Cons xy io)

  obtain x y where xy = (x,y)
    using prod.exhaust by metis

  show ?case
    unfolding ‹xy = (x,y)›
    unfolding h-obs-language-iff[OF assms(1), of x y io q]
    unfolding is-in-language.simps
    apply (cases h-obs M q x y)
    apply auto[1]
    by (metis Cons.IH h-obs-state option.simps(5))
qed

lemma observable-paths-for-io :
  assumes observable M
  and     io ∈ LS M q
obtains p where paths-for-io M q io = {p}
proof −
  obtain p where path M q p and p-io p = io
    using assms(2) by auto
  then have p ∈ paths-for-io M q io
    unfolding paths-for-io-def
```

```
    by blast
  then show ?thesis
    using that[of p]
    using observable-path-unique[OF assms(1) ‹path M q p›] ‹p-io p = io›
    unfolding paths-for-io-def
    by force
qed

lemma io-targets-language :
  assumes q' ∈ io-targets M io q
  shows io ∈ LS M q
  using assms by auto


lemma observable-after-reachable-surj :
  assumes observable M
  shows (after-initial M) ‘ (L M) = reachable-states M
proof
  show after-initial M ‘ L M ⊆ reachable-states M
    using after-reachable[OF assms - reachable-states-initial]
    by blast
  show reachable-states M ⊆ after-initial M ‘ L M
    unfolding reachable-states-def
    using after-path[OF assms]
    using image-iff by fastforce
qed


lemma observable-minimal-size-r-language-distinct :
  assumes minimal M1
  and     minimal M2
  and     observable M1
  and     observable M2
  and     size-r M1 < size-r M2
shows L M1 ≠ L M2
proof
  assume L M1 = L M2

  define V where V = (λ q . SOME io . io ∈ L M1 ∧ after-initial M2 io = q)

  have ⋀ q . q ∈ reachable-states M2 ⟹ V q ∈ L M1 ∧ after-initial M2 (V q)
= q
  proof −
    fix q assume q ∈ reachable-states M2
    then have ∃ io . io ∈ L M1 ∧ after-initial M2 io = q
      unfolding ‹L M1 = L M2›
      by (metis assms(4) imageE observable-after-reachable-surj)
    then show V q ∈ L M1 ∧ after-initial M2 (V q) = q
      unfolding V-def
```

using *someI-ex*[*of* λ *io* . *io* $\in$ *L M1* $\wedge$ *after-initial M2 io* = *q*] **by** *blast*
qed
then have (*after-initial M1*) ‘ *V* ‘ *reachable-states M2* $\subseteq$ *reachable-states M1*
by (*metis assms*(*3*) *image-mono image-subsetI observable-after-reachable-surj*)
then have *card* (*after-initial M1* ‘ *V* ‘ *reachable-states M2*) $\leq$ *size-r M1*
using *reachable-states-finite*[*of M1*]
by (*meson card-mono*)

have (*after-initial M2*) ‘ *V* ‘ *reachable-states M2* = *reachable-states M2*
proof
show *after-initial M2* ‘ *V* ‘ *reachable-states M2* $\subseteq$ *reachable-states M2*
using ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M2* $\Longrightarrow$ *V q* $\in$ *L M1* $\wedge$ *after-initial M2* (*V q*) = *q*› **by** *auto*
show *reachable-states M2* $\subseteq$ *after-initial M2* ‘ *V* ‘ *reachable-states M2*
using ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M2* $\Longrightarrow$ *V q* $\in$ *L M1* $\wedge$ *after-initial M2* (*V q*) = *q*› *observable-after-reachable-surj*[*OF assms*(*4*)] **unfolding** ‹*L M1* = *L M2*›
using *image-iff* **by** *fastforce*
qed
then have *card* ((*after-initial M2*) ‘ *V* ‘ *reachable-states M2*) = *size-r M2*
by *auto*

have $*$: *finite* (*V* ‘ *reachable-states M2*)
by (*simp add*: *reachable-states-finite*)

have $**$: *card* ((*after-initial M1*) ‘ *V* ‘ *reachable-states M2*) < *card* ((*after-initial M2*) ‘ *V* ‘ *reachable-states M2*)
using *assms*(*5*) ‹*card* (*after-initial M1* ‘ *V* ‘ *reachable-states M2*) $\leq$ *size-r M1*›
unfolding ‹*card* ((*after-initial M2*) ‘ *V* ‘ *reachable-states M2*) = *size-r M2*›
by *linarith*

obtain *io1 io2* **where** *io1* $\in$ *V* ‘ *reachable-states M2*
io2 $\in$ *V* ‘ *reachable-states M2*
after-initial M2 io1 $\neq$ *after-initial M2 io2*
after-initial M1 io1 = *after-initial M1 io2*
using *finite-card-less-witnesses*[*OF* $* **$]
by *blast*
then have *io1* $\in$ *L M1* **and** *io2* $\in$ *L M1* **and** *io1* $\in$ *L M2* **and** *io2* $\in$ *L M2*
using ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M2* $\Longrightarrow$ *V q* $\in$ *L M1* $\wedge$ *after-initial M2* (*V q*) = *q*› **unfolding** ‹*L M1* = *L M2*›
by *auto*
then have *after-initial M1 io1* $\in$ *reachable-states M1*
after-initial M1 io2 $\in$ *reachable-states M1*
after-initial M2 io1 $\in$ *reachable-states M2*
after-initial M2 io2 $\in$ *reachable-states M2*
using *after-reachable*[*OF assms*(*3*) - *reachable-states-initial*] *after-reachable*[*OF assms*(*4*) - *reachable-states-initial*]
by *blast*+

obtain *io3* **where** *io3* $\in$ *LS M2* (*after-initial M2 io1*) = (*io3* $\notin$ *LS M2* (*after-initial*

M2 io2))
using *reachable-state-is-state*[*OF* ‹*after-initial M2 io1* ∈ *reachable-states M2*›]

reachable-state-is-state[*OF* ‹*after-initial M2 io2* ∈ *reachable-states M2*›]
‹*after-initial M2 io1* ≠ *after-initial M2 io2*› *assms*(*2*)
unfolding *minimal.simps* **by** *blast*
then have *io1@io3* ∈ *L M2* = (*io2@io3* ∉ *L M2*)
using *observable-after-language-append*[*OF assms*(*4*) ‹*io1* ∈ *L M2*›]
observable-after-language-append[*OF assms*(*4*) ‹*io2* ∈ *L M2*›]
observable-after-language-none[*OF assms*(*4*) ‹*io1* ∈ *L M2*›]
observable-after-language-none[*OF assms*(*4*) ‹*io2* ∈ *L M2*›]
by *blast*
moreover have *io1@io3* ∈ *L M1* = (*io2@io3* ∈ *L M1*)
by (*meson* ‹*after-initial M1 io1* = *after-initial M1 io2*› ‹*io1* ∈ *L M1*› ‹*io2* ∈ *L M1*› *assms*(*3*) *observable-after-eq*)
ultimately show *False*
using ‹*L M1* = *L M2*› **by** *blast*
qed

lemma *minimal-equivalence-size-r* :
assumes *minimal M1*
and *minimal M2*
and *observable M1*
and *observable M2*
and *L M1* = *L M2*
shows *size-r M1* = *size-r M2*
using *observable-minimal-size-r-language-distinct*[*OF assms*(*1*−*4*)]
observable-minimal-size-r-language-distinct[*OF assms*(*2*,*1*,*4*,*3*)]
assms(*5*)
using *nat-neq-iff* **by** *auto*

4.10 Conformity Relations

fun *is-io-reduction-state* :: (*'a*,*'b*,*'c*) *fsm* ⇒ *'a* ⇒ (*'d*,*'b*,*'c*) *fsm* ⇒ *'d* ⇒ *bool* **where**
is-io-reduction-state A a B b = (*LS A a* ⊆ *LS B b*)

abbreviation(*input*) *is-io-reduction A B* ≡ *is-io-reduction-state A* (*initial A*) *B* (*initial B*)
notation
is-io-reduction (‹- ⪯ -›)

fun *is-io-reduction-state-on-inputs* :: (*'a*,*'b*,*'c*) *fsm* ⇒ *'a* ⇒ *'b list set* ⇒ (*'d*,*'b*,*'c*) *fsm* ⇒ *'d* ⇒ *bool* **where**
is-io-reduction-state-on-inputs A a U B b = (LS_{in} *A a U* ⊆ LS_{in} *B b U*)

abbreviation(*input*) *is-io-reduction-on-inputs A U B* ≡ *is-io-reduction-state-on-inputs A* (*initial A*) *U B* (*initial B*)

notation
is-io-reduction-on-inputs (‹- ≼⟦-⟧ -›)

4.11 A Pass Relation for Reduction and Test Represented as Sets of Input-Output Sequences

definition *pass-io-set* :: $('a,'b,'c)$ *fsm* ⇒ $('b \times 'c)$ *list set* ⇒ *bool* **where**
pass-io-set M ios = (∀ *io x y* . *io*@[(*x*,*y*)] ∈ *ios* ⟶ (∀ *y'* . *io*@[(*x*,*y'*)] ∈ *L M* ⟶ *io*@[(*x*,*y'*)] ∈ *ios*))

definition *pass-io-set-maximal* :: $('a,'b,'c)$ *fsm* ⇒ $('b \times 'c)$ *list set* ⇒ *bool* **where**
pass-io-set-maximal M ios = (∀ *io x y io'* . *io*@[(*x*,*y*)]@*io'* ∈ *ios* ⟶ (∀ *y'* . *io*@[(*x*,*y'*)] ∈ *L M* ⟶ (∃ *io''*. *io*@[(*x*,*y'*)]@*io''* ∈ *ios*)))

lemma *pass-io-set-from-pass-io-set-maximal* :
pass-io-set-maximal M ios = *pass-io-set M* {*io'* . ∃ *io io''* . *io* = *io'*@*io''* ∧ *io* ∈ *ios*}
proof −
have ⋀ *io x y io'* . *io*@[(*x*,*y*)]@*io'* ∈ *ios* ⟹ *io*@[(*x*,*y*)] ∈ {*io'* . ∃ *io io''* . *io* = *io'*@*io''* ∧ *io* ∈ *ios*}
by *auto*
moreover have ⋀ *io x y* . *io*@[(*x*,*y*)] ∈ {*io'* . ∃ *io io''* . *io* = *io'*@*io''* ∧ *io* ∈ *ios*} ⟹ ∃ *io'* . *io*@[(*x*,*y*)]@*io'* ∈ *ios*
by *auto*
ultimately show *?thesis*
unfolding *pass-io-set-def pass-io-set-maximal-def*
by *meson*
qed

lemma *pass-io-set-maximal-from-pass-io-set* :
assumes *finite ios*
and ⋀ *io' io''* . *io'*@*io''* ∈ *ios* ⟹ *io'* ∈ *ios*
shows *pass-io-set M ios* = *pass-io-set-maximal M* {*io'* ∈ *ios* . ¬ (∃ *io''* . *io''* ≠ [] ∧ *io'*@*io''* ∈ *ios*)}
proof −
have ⋀ *io x y* . *io*@[(*x*,*y*)] ∈ *ios* ⟹ ∃ *io'* . *io*@[(*x*,*y*)]@*io'* ∈ {*io''* ∈ *ios* . ¬ (∃ *io'''* . *io'''* ≠ [] ∧ *io''*@*io'''* ∈ *ios*)}
proof −
fix *io x y* **assume** *io*@[(*x*,*y*)] ∈ *ios*
show ∃ *io'* . *io*@[(*x*,*y*)]@*io'* ∈ {*io''* ∈ *ios* . ¬ (∃ *io'''* . *io'''* ≠ [] ∧ *io''*@*io'''* ∈ *ios*)}
using *finite-set-elem-maximal-extension-ex*[*OF* ‹*io*@[(*x*,*y*)] ∈ *ios*› *assms*(*1*)]
by *force*
qed
moreover have ⋀ *io x y io'* . *io*@[(*x*,*y*)]@*io'* ∈ {*io''* ∈ *ios* . ¬ (∃ *io'''* . *io'''* ≠ [] ∧ *io''*@*io'''* ∈ *ios*)} ⟹ *io*@[(*x*,*y*)] ∈ *ios*
using ‹⋀ *io' io''* . *io'*@*io''* ∈ *ios* ⟹ *io'* ∈ *ios*› **by** *force*

ultimately show *?thesis*
unfolding *pass-io-set-def pass-io-set-maximal-def*
by *meson*
qed

4.12 Relaxation of IO based test suites to sets of input sequences

abbreviation(*input*) *input-portion xs* ≡ *map fst xs*

lemma *equivalence-io-relaxation* :
assumes $(L\ M1 = L\ M2) \longleftrightarrow (L\ M1 \cap T = L\ M2 \cap T)$
shows $(L\ M1 = L\ M2) \longleftrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\})$
proof
show $(L\ M1 = L\ M2) \Longrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\})$
by *blast*
show $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow L\ M1 = L\ M2$
proof −
have $*$:$\bigwedge M\ .\ \{io\ .\ io \in L\ M \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = L\ M \cap \{io\ .\ \exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io'\}$
by *blast*

have $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow (L\ M1 \cap T = L\ M2 \cap T)$
unfolding $*$ **by** *blast*
then show $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} = \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow L\ M1 = L\ M2$
using *assms* **by** *blast*
qed
qed

lemma *reduction-io-relaxation* :
assumes $(L\ M1 \subseteq L\ M2) \longleftrightarrow (L\ M1 \cap T \subseteq L\ M2 \cap T)$
shows $(L\ M1 \subseteq L\ M2) \longleftrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\})$
proof
show $(L\ M1 \subseteq L\ M2) \Longrightarrow (\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\})$
by *blast*

show $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow L\ M1 \subseteq L\ M2$
proof −
have $*$:$\bigwedge M\ .\ \{io\ .\ io \in L\ M \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq L\ M \cap \{io\ .\ \exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io'\}$
by *blast*

have $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow (L\ M1 \cap T \subseteq L\ M2 \cap T)$
unfolding $*$ **by** *blast*
then show $(\{io\ .\ io \in L\ M1 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\} \subseteq \{io\ .\ io \in L\ M2 \wedge (\exists\ io' \in T\ .\ input\text{-}portion\ io = input\text{-}portion\ io')\}) \Longrightarrow L\ M1 \subseteq L\ M2$
using *assms* **by** *blast*
qed
qed

4.13 Submachines

fun *is-submachine* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *fsm* $\Rightarrow$ *bool* **where**
is-submachine A B = (*initial A* = *initial B* $\wedge$ *transitions A* $\subseteq$ *transitions B* $\wedge$ *inputs A* = *inputs B* $\wedge$ *outputs A* = *outputs B* $\wedge$ *states A* $\subseteq$ *states B*)

lemma *submachine-path-initial* :
assumes *is-submachine A B*
and *path A* (*initial A*) *p*
shows *path B* (*initial B*) *p*
using *assms* **proof** (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a p*)
then show *?case*
by *fastforce*
qed

lemma *submachine-path* :
assumes *is-submachine A B*
and *path A q p*
shows *path B q p*
by (*meson assms*(*1*) *assms*(*2*) *is-submachine.elims*(*2*) *path-begin-state subsetD transition-subset-path*)

lemma *submachine-reduction* :

assumes *is-submachine A B*
shows *is-io-reduction A B*
using *submachine-path*[*OF assms*] *assms* **by** *auto*

lemma *complete-submachine-initial* :
assumes *is-submachine A B*
and *completely-specified A*
shows *completely-specified-state B* (*initial B*)
using *assms*(*1*) *assms*(*2*) *fsm-initial subset-iff* **by** *fastforce*

lemma *submachine-language* :
assumes *is-submachine S M*
shows $L\ S \subseteq L\ M$
by (*meson assms is-io-reduction-state.elims*(*2*) *submachine-reduction*)

lemma *submachine-observable* :
assumes *is-submachine S M*
and *observable M*
shows *observable S*
using *assms* **unfolding** *is-submachine.simps observable.simps* **by** *blast*

lemma *submachine-transitive* :
assumes *is-submachine S M*
and *is-submachine S′ S*
shows *is-submachine S′ M*
using *assms* **unfolding** *is-submachine.simps* **by** *force*

lemma *transitions-subset-path* :
assumes *set p* $\subseteq$ *transitions M*
and $p \neq []$
and *path S q p*
shows *path M q p*
using *assms* **by** (*induction p arbitrary*: *q*; *auto*)

lemma *transition-subset-paths* :
assumes *transitions S* $\subseteq$ *transitions M*
and *initial S* $\in$ *states M*
and *inputs S* = *inputs M*
and *outputs S* = *outputs M*
and *path S* (*initial S*) *p*
shows *path M* (*initial S*) *p*
using *assms*(*5*) **proof** (*induction p rule*: *rev-induct*)
case *Nil*

```
  then show ?case using assms(2) by auto
next
  case (snoc t p)
  then have path S (initial S) p
        and t ∈ transitions S
        and t-source t = target (initial S) p
        and path M (initial S) p
    by auto

  have t ∈ transitions M
    using assms(1) ‹t ∈ transitions S› by auto
  moreover have t-source t ∈ states M
    using ‹t-source t = target (initial S) p› ‹path M (initial S) p›
    using path-target-is-state by fastforce
  ultimately have t ∈ transitions M
    using ‹t ∈ transitions S› assms(3,4) by auto
  then show ?case
    using ‹path M (initial S) p›
    using snoc.prems by auto
qed


lemma submachine-reachable-subset :
  assumes is-submachine A B
shows reachable-states A ⊆ reachable-states B
  using assms submachine-path-initial[OF assms]
  unfolding is-submachine.simps reachable-states-def by force


lemma submachine-simps :
  assumes is-submachine A B
shows initial A = initial B
and   states A ⊆ states B
and   inputs A = inputs B
and   outputs A = outputs B
and   transitions A ⊆ transitions B
  using assms unfolding is-submachine.simps by blast+


lemma submachine-deadlock :
  assumes is-submachine A B
      and deadlock-state B q
    shows deadlock-state A q
  using assms(1) assms(2) in-mono by auto
```

4.14 Changing Initial States

```
lift-definition from-FSM :: ('a,'b,'c) fsm ⇒ 'a ⇒ ('a,'b,'c) fsm is FSM-Impl.from-FSMI
  by simp
```

lemma *from-FSM-simps*[*simp*]:
assumes $q \in$ *states M*
shows
initial (*from-FSM M q*) = *q*
inputs (*from-FSM M q*) = *inputs M*
outputs (*from-FSM M q*) = *outputs M*
transitions (*from-FSM M q*) = *transitions M*
states (*from-FSM M q*) = *states M* **using** *assms* **by** (*transfer*; *simp*)+

lemma *from-FSM-path-initial* :
assumes $q \in$ *states M*
shows *path M q p* = *path* (*from-FSM M q*) (*initial* (*from-FSM M q*)) *p*
by (*metis assms from-FSM-simps*(*1*) *from-FSM-simps*(*4*) *from-FSM-simps*(*5*) *order-refl*
transition-subset-path)

lemma *from-FSM-path* :
assumes $q \in$ *states M*
and *path* (*from-FSM M q*) *q′ p*
shows *path M q′ p*
using *assms*(*1*) *assms*(*2*) *path-transitions transitions-subset-path* **by** *fastforce*

lemma *from-FSM-reachable-states* :
assumes $q \in$ *reachable-states M*
shows *reachable-states* (*from-FSM M q*) $\subseteq$ *reachable-states M*
proof
from *assms* **obtain** *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *q*
unfolding *reachable-states-def* **by** *blast*
then have $q \in$ *states M*
by (*meson path-target-is-state*)

fix *q′* **assume** *q′* $\in$ *reachable-states* (*from-FSM M q*)
then obtain *p′* **where** *path* (*from-FSM M q*) *q p′* **and** *target q p′* = *q′*
unfolding *reachable-states-def from-FSM-simps*[*OF* ‹$q \in$ *states M*›] **by** *blast*
then have *path M* (*initial M*) (*p@p′*) **and** *target* (*initial M*) (*p@p′*) = *q′*
using *from-FSM-path*[*OF* ‹$q \in$ *states M*›] ‹*path M* (*initial M*) *p*›
using ‹*target* (*FSM.initial M*) *p* = *q*› **by** *auto*

then show *q′* $\in$ *reachable-states M*
unfolding *reachable-states-def* **by** *blast*
qed

lemma *submachine-from* :
assumes *is-submachine S M*

and $q \in$ *states S*
shows *is-submachine* (*from-FSM S q*) (*from-FSM M q*)
proof −
have *path S q* []
using *assms*(*2*) **by** *blast*
then have *path M q* []
by (*meson assms*(*1*) *submachine-path*)
then show *?thesis*
using *assms*(*1*) *assms*(*2*) **by** *force*
qed

lemma *from-FSM-path-rev-initial* :
assumes *path M q p*
shows *path* (*from-FSM M q*) *q p*
by (*metis* (*no-types*) *assms from-FSM-path-initial from-FSM-simps*(*1*) *path-begin-state*)

lemma *from-from*[*simp*] :
assumes *q1* $\in$ *states M*
and *q1'* $\in$ *states M*
shows *from-FSM* (*from-FSM M q1*) *q1'* = *from-FSM M q1'* (**is** *?M* = *?M'*)
proof −
have ∗: *q1'* $\in$ *states* (*from-FSM M q1*)
using *assms*(*2*) **unfolding** *from-FSM-simps*(*5*)[*OF assms*(*1*)] **by** *assumption*

have *initial ?M* = *initial ?M'*
and *states ?M* = *states ?M'*
and *inputs ?M* = *inputs ?M'*
and *outputs ?M* = *outputs ?M'*
and *transitions ?M* = *transitions ?M'*
unfolding *from-FSM-simps*[*OF* ∗] *from-FSM-simps*[*OF assms*(*1*)] *from-FSM-simps*[*OF assms*(*2*)] **by** *simp*+

then show *?thesis* **by** (*transfer*; *force*)
qed

lemma *from-FSM-completely-specified* :
assumes *completely-specified M*
shows *completely-specified* (*from-FSM M q*) **proof** (*cases q* $\in$ *states M*)
case *True*
then show *?thesis*
using *assms* **by** *auto*
next
case *False*
then have *from-FSM M q* = *M* **by** (*transfer*; *auto*)
then show *?thesis* **using** *assms* **by** *auto*
qed

lemma *from-FSM-single-input* :
  assumes *single-input M*
shows *single-input* (*from-FSM M q*) **proof** (*cases q* ∈ *states M*)
  case *True*
  then show *?thesis*
    using *assms*
    by (*metis from-FSM-simps*(*4*) *single-input.elims*(*1*))
next
  case *False*
  then have *from-FSM M q* = *M* **by** (*transfer*; *auto*)
  then show *?thesis* **using** *assms*
    by *presburger*
qed

lemma *from-FSM-acyclic* :
  assumes *q* ∈ *reachable-states M*
  and *acyclic M*
shows *acyclic* (*from-FSM M q*)
  using *assms*(*1*)
      acyclic-paths-from-reachable-states[*OF assms*(*2*), *of - q*]
      from-FSM-path[*of q M q*]
      path-target-is-state
      reachable-state-is-state[*OF assms*(*1*)]
      from-FSM-simps(*1*)
  unfolding *acyclic.simps*
        reachable-states-def
  by *force*

lemma *from-FSM-observable* :
  assumes *observable M*
shows *observable* (*from-FSM M q*)
proof (*cases q* ∈ *states M*)
  case *True*
  then show *?thesis*
    using *assms*
  proof −
   have *f1*: ∀ *f*. *observable f* = (∀ *a b c aa ab*. ((*a*::'*a*, *b*::'*b*, *c*::'*c*, *aa*) ∉ *FSM.transitions f* ∨ (*a*, *b*, *c*, *ab*) ∉ *FSM.transitions f*) ∨ *aa* = *ab*)
      by *force*
   have ∀ *a f*. *a* ∉ *FSM.states* (*f*::('*a*, '*b*, '*c*) *fsm*) ∨ *FSM.transitions* (*FSM.from-FSM f a*) = *FSM.transitions f*
      by (*meson from-FSM-simps*(*4*))
    then show *?thesis*
      using *f1 True assms* **by** *presburger*

```
  qed
next
  case False
  then have from-FSM M q = M by (transfer; auto)
  then show ?thesis using assms by presburger
qed


lemma observable-language-next :
  assumes io#ios ∈ LS M (t-source t)
  and     observable M
  and     t ∈ transitions M
  and     t-input t = fst io
  and     t-output t = snd io
shows ios ∈ L (from-FSM M (t-target t))
proof −
  obtain p where path M (t-source t) p and p-io p = io#ios
    using assms(1)
  proof −
    assume a1: ⋀p. ⟦path M (t-source t) p; p-io p = io # ios⟧ ⟹ thesis
    obtain pps :: ('a × 'b) list ⇒ 'c ⇒ ('c, 'a, 'b) fsm ⇒ ('c × 'a × 'b × 'c) list
where
      ∀ x0 x1 x2. (∃ v3. x0 = p-io v3 ∧ path x2 x1 v3) = (x0 = p-io (pps x0 x1 x2)
∧ path x2 x1 (pps x0 x1 x2))
      by moura
    then have ∃ ps. path M (t-source t) ps ∧ p-io ps = io # ios
      using assms(1) by auto
    then show ?thesis
      using a1 by meson
  qed
  then obtain t' p' where p = t' # p'
    by auto
  then have t' ∈ transitions M and t-source t' = t-source t and t-input t' = fst
io and t-output t' = snd io
    using ‹path M (t-source t) p› ‹p-io p = io#ios› by auto
  then have t = t'
    using assms(2,3,4,5) unfolding observable.simps
    by (metis (no-types, opaque-lifting) prod.expand)

  then have path M (t-target t) p' and p-io p' = ios
    using ‹p = t' # p'› ‹path M (t-source t) p› ‹p-io p = io#ios› by auto
  then have path (from-FSM M (t-target t)) (initial (from-FSM M (t-target t)))
p'
    by (meson assms(3) from-FSM-path-initial fsm-transition-target)

  then show ?thesis using ‹p-io p' = ios› by auto
qed
```

lemma *from-FSM-language* :
assumes $q \in$ *states M*
shows *L* (*from-FSM M q*) = *LS M q*
using *assms* **unfolding** *LS.simps* **by** (*meson from-FSM-path-initial*)

lemma *observable-transition-target-language-subset* :
assumes *LS M* (*t-source t1*) $\subseteq$ *LS M* (*t-source t2*)
and *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*
and *t-input t1* = *t-input t2*
and *t-output t1* = *t-output t2*
and *observable M*
shows *LS M* (*t-target t1*) $\subseteq$ *LS M* (*t-target t2*)
proof (*rule ccontr*)
assume $\neg$ *LS M* (*t-target t1*) $\subseteq$ *LS M* (*t-target t2*)
then obtain *ioF* **where** *ioF* $\in$ *LS M* (*t-target t1*) **and** *ioF* $\notin$ *LS M* (*t-target t2*)
by *blast*
then have (*t-input t1*, *t-output t1*)#*ioF* $\in$ *LS M* (*t-source t1*)
using *LS-prepend-transition assms*(*2*) **by** *blast*
then have $*$: (*t-input t1*, *t-output t1*)#*ioF* $\in$ *LS M* (*t-source t2*)
using *assms*(*1*) **by** *blast*

have *ioF* $\in$ *LS M* (*t-target t2*)
using *observable-language-next*[*OF* $*$ ‹*observable M*› ‹*t2* $\in$ *transitions M*›]
unfolding *assms*(*4*,*5*) *fst-conv snd-conv*
by (*metis assms*(*3*) *from-FSM-language fsm-transition-target*)
then show *False*
using ‹*ioF* $\notin$ *LS M* (*t-target t2*)› **by** *blast*
qed

lemma *observable-transition-target-language-eq* :
assumes *LS M* (*t-source t1*) = *LS M* (*t-source t2*)
and *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*
and *t-input t1* = *t-input t2*
and *t-output t1* = *t-output t2*
and *observable M*
shows *LS M* (*t-target t1*) = *LS M* (*t-target t2*)
using *observable-transition-target-language-subset*[*OF* - *assms*(*2*,*3*,*4*,*5*,*6*)]
observable-transition-target-language-subset[*OF* - *assms*(*3*,*2*) *assms*(*4*,*5*)[*symmetric*] *assms*(*6*)]
assms(*1*)
by *blast*

lemma *language-state-prepend-transition* :
assumes *io* $\in$ *LS* (*from-FSM A* (*t-target t*)) (*initial* (*from-FSM A* (*t-target t*)))

and $t \in$ *transitions A*
shows *p-io* [*t*] @ *io* $\in$ *LS A* (*t-source t*)
proof −
obtain *p* **where** *path* (*from-FSM A* (*t-target t*)) (*initial* (*from-FSM A* (*t-target t*))) *p*
and *p-io p* = *io*
using *assms*(*1*) **unfolding** *LS.simps* **by** *blast*
then have *path A* (*t-target t*) *p*
by (*meson assms*(*2*) *from-FSM-path-initial fsm-transition-target*)
then have *path A* (*t-source t*) (*t* # *p*)
using *assms*(*2*) **by** *auto*
then show *?thesis*
using ‹*p-io p* = *io*› **unfolding** *LS.simps*
by *force*
qed

lemma *observable-language-transition-target* :
assumes *observable M*
and $t \in$ *transitions M*
and (*t-input t*, *t-output t*) # *io* $\in$ *LS M* (*t-source t*)
shows *io* $\in$ *LS M* (*t-target t*)
by (*metis* (*no-types*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *from-FSM-language fsm-transition-target fst-conv observable-language-next snd-conv*)

lemma *LS-single-transition* :
[(*x*,*y*)] $\in$ *LS M q* $\longleftrightarrow$ ($\exists$ *t* $\in$ *transitions M* . *t-source t* = *q* $\wedge$ *t-input t* = *x* $\wedge$ *t-output t* = *y*)
proof
show [(*x*, *y*)] $\in$ *LS M q* $\Longrightarrow$ $\exists$ *t*$\in$*FSM.transitions M*. *t-source t* = *q* $\wedge$ *t-input t* = *x* $\wedge$ *t-output t* = *y*
by *auto*
show $\exists$ *t*$\in$*FSM.transitions M*. *t-source t* = *q* $\wedge$ *t-input t* = *x* $\wedge$ *t-output t* = *y* $\Longrightarrow$ [(*x*, *y*)] $\in$ *LS M q*
by (*metis LS-prepend-transition from-FSM-language fsm-transition-target language-contains-empty-sequence*)
qed

lemma *h-obs-language-append* :
assumes *observable M*
and *u* $\in$ *L M*
and *h-obs M* (*after-initial M u*) *x y* $\neq$ *None*
shows *u*@[(*x*,*y*)] $\in$ *L M*
using *after-language-iff*[*OF assms*(*1*,*2*), *of* [(*x*,*y*)]]
using *h-obs-None*[*OF assms*(*1*)] *assms*(*3*)
unfolding *LS-single-transition*
by (*metis old.prod.inject prod.collapse*)

lemma *h-obs-language-single-transition-iff* :

assumes *observable M*
shows $[(x,y)] \in LS\ M\ q \longleftrightarrow h\text{-}obs\ M\ q\ x\ y \neq None$
using *h-obs-None*[*OF assms*(*1*), *of q x y*]
unfolding *LS-single-transition*
by (*metis fst-conv prod.exhaust-sel snd-conv*)

lemma *minimal-failure-prefix-ob* :
assumes *observable M*
and *observable I*
and $qM \in states\ M$
and $qI \in states\ I$
and $io \in LS\ I\ qI - LS\ M\ qM$
obtains *io′ xy io″* **where** $io = io'@[xy]@io''$
and $io' \in LS\ I\ qI \cap LS\ M\ qM$
and $io'@[xy] \in LS\ I\ qI - LS\ M\ qM$
proof −
have $\exists\ io'\ xy\ io''\ .\ io = io'@[xy]@io'' \wedge io' \in LS\ I\ qI \cap LS\ M\ qM \wedge io'@[xy] \in LS\ I\ qI - LS\ M\ qM$
using *assms*(*3*,*4*,*5*) **proof** (*induction io arbitrary*: *qM qI*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons xy io*)
show *?case* **proof** (*cases* $[xy] \in LS\ I\ qI - LS\ M\ qM$)
case *True*

have $xy\ \#\ io = []@[xy]@io$
by *auto*
moreover have $[] \in LS\ I\ qI \cap LS\ M\ qM$
using ‹$qM \in states\ M$› ‹$qI \in states\ I$› **by** *auto*
moreover have $[]@[xy] \in LS\ I\ qI - LS\ M\ qM$
using *True* **by** *auto*
ultimately show *?thesis*
by *blast*
next
case *False*

obtain *x y* **where** $xy = (x,y)$
by (*meson surj-pair*)

have $[(x,y)] \in LS\ M\ qM$
using ‹$xy = (x,y)$› *False* ‹$xy\ \#\ io \in LS\ I\ qI - LS\ M\ qM$›
by (*metis DiffD1 DiffI append-Cons append-Nil language-prefix*)
then obtain qM' **where** $(qM,x,y,qM') \in transitions\ M$
by *auto*
then have $io \notin LS\ M\ qM'$
using *observable-language-transition-target*[*OF* ‹*observable M*›]
‹$xy = (x,y)$› ‹$xy\ \#\ io \in LS\ I\ qI - LS\ M\ qM$›

by (*metis DiffD2 LS-prepend-transition fst-conv snd-conv*)

have $[(x,y)] \in LS\ I\ qI$
using ‹$xy = (x,y)$› ‹$xy\ \#\ io \in LS\ I\ qI - LS\ M\ qM$›
by (*metis DiffD1 append-Cons append-Nil language-prefix*)
then obtain qI' **where** $(qI,x,y,qI') \in transitions\ I$
by *auto*
then have $io \in LS\ I\ qI'$
using *observable-language-next*[*of xy io I* (qI,x,y,qI'), *OF* - ‹*observable I*›]
‹$xy\ \#\ io \in LS\ I\ qI - LS\ M\ qM$› *fsm-transition-target*[*OF* ‹$(qI,x,y,qI') \in transitions\ I$›]
unfolding ‹$xy = (x,y)$› *fst-conv snd-conv*
by (*metis DiffD1 from-FSM-language*)

obtain $io'\ xy'\ io''$ **where** $io = io'@[xy']@io''$ **and** $io' \in LS\ I\ qI' \cap LS\ M\ qM'$ **and** $io'@[xy'] \in LS\ I\ qI' - LS\ M\ qM'$
using ‹$io \in LS\ I\ qI'$› ‹$io \notin LS\ M\ qM'$›
Cons.IH[*OF fsm-transition-target*[*OF* ‹$(qM,x,y,qM') \in transitions\ M$›]
fsm-transition-target[*OF* ‹$(qI,x,y,qI') \in transitions\ I$›]]
unfolding *fst-conv snd-conv*
by *blast*

have $xy\#io = (xy\#io')@[xy']@io''$
using ‹$io = io'@[xy']@io''$› ‹$xy = (x,y)$› **by** *auto*
moreover have $xy\#io' \in LS\ I\ qI \cap LS\ M\ qM$
using *LS-prepend-transition*[*OF* ‹$(qI,x,y,qI') \in transitions\ I$›, *of* io']
using *LS-prepend-transition*[*OF* ‹$(qM,x,y,qM') \in transitions\ M$›, *of* io']
using ‹$io' \in LS\ I\ qI' \cap LS\ M\ qM'$›
unfolding ‹$xy = (x,y)$› *fst-conv snd-conv*
by *auto*
moreover have $(xy\#io')@[xy'] \in LS\ I\ qI - LS\ M\ qM$
using *LS-prepend-transition*[*OF* ‹$(qI,x,y,qI') \in transitions\ I$›, *of* $io'@[xy']$]
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹$(qM,x,y,qM') \in transitions\ M$›, *of* $io'@[xy']$]
using ‹$io'@[xy'] \in LS\ I\ qI' - LS\ M\ qM'$›
unfolding ‹$xy = (x,y)$› *fst-conv snd-conv*
by *fastforce*
ultimately show *?thesis*
by *blast*
qed
qed
then show *?thesis*
using *that* **by** *blast*
qed

4.15 Language and Defined Inputs

lemma *defined-inputs-code* : *defined-inputs M q* = *t-input* ‘ *Set.filter* (λt . *t-source t* = *q*) (*transitions M*)

unfolding *defined-inputs-set* **by** *force*

lemma *defined-inputs-alt-def* : *defined-inputs M q* = {*t-input t* | *t* . *t* ∈ *transitions M* ∧ *t-source t* = *q*}
unfolding *defined-inputs-code* **by** *force*

lemma *defined-inputs-language-diff* :
assumes *x* ∈ *defined-inputs M1 q1*
and *x* ∉ *defined-inputs M2 q2*
obtains *y* **where** [(*x*,*y*)] ∈ *LS M1 q1* − *LS M2 q2*
using *assms* **unfolding** *defined-inputs-alt-def*
proof −
assume *a1*: *x* ∉ {*t-input t* |*t*. *t* ∈ *FSM.transitions M2* ∧ *t-source t* = *q2*}
assume *a2*: *x* ∈ {*t-input t* |*t*. *t* ∈ *FSM.transitions M1* ∧ *t-source t* = *q1*}
assume *a3*: ⋀*y*. [(*x*, *y*)] ∈ *LS M1 q1* − *LS M2 q2* ⟹ *thesis*
have *f4*: ∄*p*. *x* = *t-input p* ∧ *p* ∈ *FSM.transitions M2* ∧ *t-source p* = *q2*
using *a1* **by** *blast*
obtain *pp* :: *'a* ⇒ *'b* × *'a* × *'c* × *'b* **where**
∀*a*. ((∄*p*. *a* = *t-input p* ∧ *p* ∈ *FSM.transitions M1* ∧ *t-source p* = *q1*) ∨ *a* = *t-input* (*pp a*) ∧ *pp a* ∈ *FSM.transitions M1* ∧ *t-source* (*pp a*) = *q1*) ∧ ((∃*p*. *a* = *t-input p* ∧ *p* ∈ *FSM.transitions M1* ∧ *t-source p* = *q1*) ∨ (∀*p*. *a* ≠ *t-input p* ∨ *p* ∉ *FSM.transitions M1* ∨ *t-source p* ≠ *q1*))
by *moura*
then have *x* = *t-input* (*pp x*) ∧ *pp x* ∈ *FSM.transitions M1* ∧ *t-source* (*pp x*) = *q1*
using *a2* **by** *blast*
then show *?thesis*
using *f4 a3* **by** (*metis* (*no-types*) *DiffI LS-single-transition*)
qed

lemma *language-path-append* :
assumes *path M1 q1 p1*
and *io* ∈ *LS M1* (*target q1 p1*)
shows (*p-io p1* @ *io*) ∈ *LS M1 q1*
proof −
obtain *p2* **where** *path M1* (*target q1 p1*) *p2* **and** *p-io p2* = *io*
using *assms*(*2*) **by** *auto*
then have *path M1 q1* (*p1*@*p2*)
using *assms*(*1*) **by** *auto*
moreover have *p-io* (*p1*@*p2*) = (*p-io p1* @ *io*)
using ‹*p-io p2* = *io*› **by** *auto*
ultimately show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *language-intro*)
qed

lemma *observable-defined-inputs-diff-ob* :
assumes *observable M1*
and *observable M2*
and *path M1 q1 p1*

and *path M2 q2 p2*
and *p-io p1 = p-io p2*
and *x ∈ defined-inputs M1 (target q1 p1)*
and *x ∉ defined-inputs M2 (target q2 p2)*
obtains *y* **where** *(p-io p1)@[(x,y)] ∈ LS M1 q1 − LS M2 q2*
proof −
obtain *y* **where** *[(x,y)] ∈ LS M1 (target q1 p1) − LS M2 (target q2 p2)*
using *defined-inputs-language-diff[OF assms(6,7)]* **by** *blast*
then have *(p-io p1)@[(x,y)] ∈ LS M1 q1*
using *language-path-append[OF assms(3)]*
by *blast*
moreover have *(p-io p1)@[(x,y)] ∉ LS M2 q2*
by *(metis (mono-tags, lifting) DiffD2 ‹[(x, y)] ∈ LS M1 (target q1 p1) − LS M2 (target q2 p2)› assms(2) assms(4) assms(5) language-state-containment observable-path-suffix)*
ultimately show *?thesis*
using *that[of y]* **by** *blast*
qed

lemma *observable-defined-inputs-diff-language* :
assumes *observable M1*
and *observable M2*
and *path M1 q1 p1*
and *path M2 q2 p2*
and *p-io p1 = p-io p2*
and *defined-inputs M1 (target q1 p1) ≠ defined-inputs M2 (target q2 p2)*
shows *LS M1 q1 ≠ LS M2 q2*
proof −
obtain *x* **where** *(x ∈ defined-inputs M1 (target q1 p1) − defined-inputs M2 (target q2 p2))*
∨ (x ∈ defined-inputs M2 (target q2 p2) − defined-inputs M1 (target q1 p1))
using *assms* **by** *blast*
then consider *(x ∈ defined-inputs M1 (target q1 p1) − defined-inputs M2 (target q2 p2)) |*
(x ∈ defined-inputs M2 (target q2 p2) − defined-inputs M1 (target q1 p1))
by *blast*
then show *?thesis*
proof *cases*
case *1*
then show *?thesis*
using *observable-defined-inputs-diff-ob[OF assms(1−5), of x]* **by** *blast*
next
case *2*
then show *?thesis*
using *observable-defined-inputs-diff-ob[OF assms(2,1,4,3) assms(5)[symmetric], of x]* **by** *blast*

```
  qed
qed

fun maximal-prefix-in-language :: ('a,'b,'c) fsm ⇒ 'a ⇒ ('b ×'c) list ⇒ ('b ×'c)
list where
  maximal-prefix-in-language M q [] = [] |
  maximal-prefix-in-language M q ((x,y)#io) = (case h-obs M q x y of
    None ⇒ [] |
    Some q' ⇒ (x,y)#maximal-prefix-in-language M q' io)

lemma maximal-prefix-in-language-properties :
  assumes observable M
  and     q ∈ states M
shows maximal-prefix-in-language M q io ∈ LS M q
and   maximal-prefix-in-language M q io ∈ list.set (prefixes io)
proof −
  have maximal-prefix-in-language M q io ∈ LS M q ∧ maximal-prefix-in-language
M q io ∈ list.set (prefixes io)
    using assms(2) proof (induction io arbitrary: q)
    case Nil
    then show ?case by auto
  next
    case (Cons xy io)

    obtain x y where xy = (x,y)
      using prod.exhaust by metis

    show ?case proof (cases h-obs M q x y)
      case None
      then have maximal-prefix-in-language M q (xy#io) = []
        unfolding ‹xy = (x,y)›
        by auto
      then show ?thesis
      by (metis (mono-tags, lifting) Cons.prems append-self-conv2 from-FSM-language
language-contains-empty-sequence mem-Collect-eq prefixes-set)
    next
      case (Some q')
     then have ∗: maximal-prefix-in-language M q (xy#io) = (x,y)#maximal-prefix-in-language
M q' io
        unfolding ‹xy = (x,y)›
        by auto

      have q' ∈ states M
        using h-obs-state[OF Some] by auto
      then have maximal-prefix-in-language M q' io ∈ LS M q'
            and maximal-prefix-in-language M q' io ∈ list.set (prefixes io)
        using Cons.IH by auto

      have maximal-prefix-in-language M q (xy # io) ∈ LS M q
```

unfolding $*$
using *Some* ‹*maximal-prefix-in-language M q′ io* ∈ *LS M q′*›
by (*meson assms*(*1*) *h-obs-language-iff*)
moreover have *maximal-prefix-in-language M q* (*xy* # *io*) ∈ *list.set* (*prefixes* (*xy* # *io*))
unfolding $*$
unfolding ‹*xy* = (*x*,*y*)›
using ‹*maximal-prefix-in-language M q′ io* ∈ *list.set* (*prefixes io*)› *append-Cons*
unfolding *prefixes-set*
by *auto*
ultimately show *?thesis*
by *blast*
qed
qed
then show *maximal-prefix-in-language M q io* ∈ *LS M q*
and *maximal-prefix-in-language M q io* ∈ *list.set* (*prefixes io*)
by *auto*
qed

4.16 Further Reachability Formalisations

fun *reachable-k* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ *nat* ⇒ ′*a set* **where**
reachable-k M q n = {*target q p* | *p* . *path M q p* ∧ *length p* ≤ *n*}

lemma *reachable-k-0-initial* : *reachable-k M* (*initial M*) *0* = {*initial M*}
by *auto*

lemma *reachable-k-states* : *reachable-states M* = *reachable-k M* (*initial M*) (*size M* − *1*)
proof −
have ⋀*q*. *q* ∈ *reachable-states M* ⟹ *q* ∈ *reachable-k M* (*initial M*) (*size M* − *1*)
proof −
fix *q* **assume** *q* ∈ *reachable-states M*
then obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *q*
unfolding *reachable-states-def* **by** *blast*
then obtain *p′* **where** *path M* (*initial M*) *p′*
and *target* (*initial M*) *p′* = *target* (*initial M*) *p*
and *length p′* < *size M*
by (*metis acyclic-path-from-cyclic-path acyclic-path-length-limit*)
then show *q* ∈ *reachable-k M* (*initial M*) (*size M* − *1*)
using ‹*target* (*FSM.initial M*) *p* = *q*› *less-trans* **by** *auto*
qed

moreover have ⋀*x*. *x* ∈ *reachable-k M* (*initial M*) (*size M* − *1*) ⟹ *x* ∈ *reachable-states M*
unfolding *reachable-states-def reachable-k.simps* **by** *blast*

ultimately show *?thesis* **by** *blast*
qed

4.16.1 Induction Schemes

lemma *acyclic-induction* [*consumes 1*, *case-names reachable-state*]:
assumes *acyclic M*
and $\bigwedge$ *q* . *q* ∈ *reachable-states M* ⟹ ($\bigwedge$ *t* . *t* ∈ *transitions M* ⟹ ((*t-source t* = *q*) ⟹ *P* (*t-target t*))) ⟹ *P q*
shows ∀ *q* ∈ *reachable-states M* . *P q*
proof
fix *q* **assume** *q* ∈ *reachable-states M*

let *?k* = *Max* (*image length* {*p* . *path M q p*})
have *finite* {*p* . *path M q p*} **using** *acyclic-finite-paths-from-reachable-state*[*OF assms*(*1*)]
using ‹*q* ∈ *reachable-states M*› **unfolding** *reachable-states-def* **by** *force*
then have *k-prop*: (∀ *p* . *path M q p* ⟶ *length p* ≤ *?k*) **by** *auto*

moreover have $\bigwedge$ *q k* . *q* ∈ *reachable-states M* ⟹ (∀ *p* . *path M q p* ⟶ *length p* ≤ *k*) ⟹ *P q*
proof −
fix *q k* **assume** *q* ∈ *reachable-states M* **and** (∀ *p* . *path M q p* ⟶ *length p* ≤ *k*)
then show *P q*
proof (*induction k arbitrary*: *q*)
case *0*
then have {*p* . *path M q p*} = {[]} **using** *reachable-state-is-state*[*OF* ‹*q* ∈ *reachable-states M*›]
by *blast*
then have *LS M q* ⊆ {[]} **unfolding** *LS.simps* **by** *blast*
then have *deadlock-state M q* **using** *deadlock-state-alt-def* **by** *metis*
then show *?case* **using** *assms*(*2*)[*OF* ‹*q* ∈ *reachable-states M*›] **unfolding** *deadlock-state.simps* **by** *blast*
next
case (*Suc k*)
have $\bigwedge$ *t* . *t* ∈ *transitions M* ⟹ (*t-source t* = *q*) ⟹ *P* (*t-target t*)
proof −
fix *t* **assume** *t* ∈ *transitions M* **and** *t-source t* = *q*
then have *t-target t* ∈ *reachable-states M*
using ‹*q* ∈ *reachable-states M*› **using** *reachable-states-next* **by** *metis*
moreover have ∀*p*. *path M* (*t-target t*) *p* ⟶ *length p* ≤ *k*
using *Suc.prems*(*2*) ‹*t* ∈ *transitions M*› ‹*t-source t* = *q*› **by** *auto*
ultimately show *P* (*t-target t*)
using *Suc.IH* **unfolding** *reachable-states-def* **by** *blast*
qed
then show *?case* **using** *assms*(*2*)[*OF Suc.prems*(*1*)] **by** *blast*
qed

qed

ultimately show P q **using** ‹$q \in$ *reachable-states* M› **by** *blast*
qed

lemma *reachable-states-induct* [*consumes 1*, *case-names init transition*] :
assumes $q \in$ *reachable-states* M
and P (*initial* M)
and $\bigwedge$ t . $t \in$ *transitions* $M \Longrightarrow$ *t-source* $t \in$ *reachable-states* $M \Longrightarrow P$ (*t-source* t) $\Longrightarrow P$ (*t-target* t)
shows P q
proof −
from *assms*(*1*) **obtain** p **where** *path* M (*initial* M) p **and** *target* (*initial* M) p $= q$
unfolding *reachable-states-def* **by** *auto*
then show P q
proof (*induction* p *arbitrary*: q *rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *assms*(*2*) **by** *auto*
next
case (*snoc* t p)

then have *target* (*initial* M) p = *t-source* t
by *auto*
then have P (*t-source* t)
using *snoc.IH snoc.prems* **by** *auto*
moreover have $t \in$ *transitions* M
using *snoc.prems* **by** *auto*
moreover have *t-source* $t \in$ *reachable-states* M
by (*metis* ‹*target* (*FSM.initial* M) p = *t-source* t› *path-prefix reachable-states-intro snoc.prems*(*1*))
moreover have *t-target* $t = q$
using *snoc.prems* **by** *auto*
ultimately show *?case*
using *assms*(*3*) **by** *blast*
qed
qed

lemma *reachable-states-cases* [*consumes 1*, *case-names init transition*] :
assumes $q \in$ *reachable-states* M
and P (*initial* M)
and $\bigwedge$ t . $t \in$ *transitions* $M \Longrightarrow$ *t-source* $t \in$ *reachable-states* $M \Longrightarrow P$ (*t-target* t)
shows P q
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *reachable-states-induct*)

4.17 Further Path Enumeration Algorithms

fun *paths-for-input′* :: (′a ⇒ (′b × ′c × ′a) *set*) ⇒ ′b *list* ⇒ ′a ⇒ (′a,′b,′c) *path* ⇒ (′a,′b,′c) *path set* **where**
  paths-for-input′ f [] *q prev* = {*prev*} |
  paths-for-input′ f (*x*#*xs*) *q prev* = ⋃(*image* (λ(*x′*,*y′*,*q′*) . *paths-for-input′ f xs q′* (*prev*@[(*q*,*x*,*y′*,*q′*)])) (*Set.filter* (λ(*x′*,*y′*,*q′*) . *x′* = *x*) (*f q*)))

lemma *paths-for-input′-set* :
  assumes *q* ∈ *states M*
  shows *paths-for-input′* (*h-from M*) *xs q prev* = {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *xs*}
using *assms* **proof** (*induction xs arbitrary*: *q prev*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons x xs*)

  let *?UN* = ⋃(*image* (λ(*x′*,*y′*,*q′*) . *paths-for-input′* (*h-from M*) *xs q′* (*prev*@[(*q*,*x*,*y′*,*q′*)])) (*Set.filter* (λ(*x′*,*y′*,*q′*) . *x′* = *x*) (*h-from M q*)))

  have *?UN* = {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *x*#*xs*}
  proof
    have ⋀ *p* . *p* ∈ *?UN* ⟹ *p* ∈ {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *x*#*xs*}
    proof −
      fix *p* **assume** *p* ∈ *?UN*
      then obtain *y′ q′* **where** (*x*,*y′*,*q′*) ∈ (*Set.filter* (λ(*x′*,*y′*,*q′*) . *x′* = *x*) (*h-from M q*))
                and *p* ∈ *paths-for-input′* (*h-from M*) *xs q′* (*prev*@[(*q*,*x*,*y′*,*q′*)])
        by *auto*

      from ‹(*x*,*y′*,*q′*) ∈ (*Set.filter* (λ(*x′*,*y′*,*q′*) . *x′* = *x*) (*h-from M q*))› **have** *q′* ∈ *states M* **and** (*q*,*x*,*y′*,*q′*) ∈ *transitions M*
        using *fsm-transition-target* **unfolding** *h.simps* **by** *auto*

      have *p* ∈ {(*prev* @ [(*q*, *x*, *y′*, *q′*)]) @ *p* |*p*. *path M q′ p* ∧ *map fst* (*p-io p*) = *xs*}
        using ‹*p* ∈ *paths-for-input′* (*h-from M*) *xs q′* (*prev*@[(*q*,*x*,*y′*,*q′*)])›
        unfolding *Cons.IH*[*OF* ‹*q′* ∈ *states M*›] **by** *assumption*
      moreover have {(*prev* @ [(*q*, *x*, *y′*, *q′*)]) @ *p* |*p*. *path M q′ p* ∧ *map fst* (*p-io p*) = *xs*}
                ⊆ {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *x*#*xs*}
        using ‹(*q*,*x*,*y′*,*q′*) ∈ *transitions M*›
        using *cons* **by** *force*
      ultimately show *p* ∈ {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *x*#*xs*}
        by *blast*
    qed
    then show *?UN* ⊆ {*prev*@*p* | *p* . *path M q p* ∧ *map fst* (*p-io p*) = *x*#*xs*}
      by *blast*

have $\bigwedge p$. $p \in \{prev@p \mid p$. $path\ M\ q\ p \wedge map\ fst\ (p\text{-}io\ p) = x\#xs\} \Longrightarrow p \in$ *?UN*
proof −
fix *pp* **assume** $pp \in \{prev@p \mid p$. $path\ M\ q\ p \wedge map\ fst\ (p\text{-}io\ p) = x\#xs\}$
then obtain *p* **where** $pp = prev@p$ **and** $path\ M\ q\ p$ **and** $map\ fst\ (p\text{-}io\ p) = x\#xs$
by *fastforce*
then obtain $t\ p'$ **where** $p = t\#p'$ **and** $path\ M\ q\ (t\#p')$ **and** $map\ fst\ (p\text{-}io\ (t\#p')) = x\#xs$ **and** $map\ fst\ (p\text{-}io\ p') = xs$
by (*metis* (*no-types*, *lifting*) *map-eq-Cons-D*)
then have $path\ M\ (t\text{-}target\ t)\ p'$ **and** $t\text{-}source\ t = q$ **and** $t\text{-}input\ t = x$ **and** $t\text{-}target\ t \in states\ M$ **and** $t \in transitions\ M$
by *auto*

have $(x,t\text{-}output\ t,t\text{-}target\ t) \in (Set.filter\ (\lambda(x',y',q')\ .\ x' = x)\ (h\text{-}from\ M\ q))$
using ‹$t \in transitions\ M$› ‹$t\text{-}input\ t = x$› ‹$t\text{-}source\ t = q$›
unfolding *h.simps* **by** *auto*
moreover have $(prev@p) \in paths\text{-}for\text{-}input'\ (h\text{-}from\ M)\ xs\ (t\text{-}target\ t)\ (prev@[(q,x,t\text{-}output\ t,t\text{-}target\ t)])$
using *Cons.IH*[*OF* ‹$t\text{-}target\ t \in states\ M$›, *of* $prev@[(q,\ x,\ t\text{-}output\ t,\ t\text{-}target\ t)]$]
using ‹$\bigwedge thesis.\ (\bigwedge t\ p'.\ [\![p = t\ \#\ p';\ path\ M\ q\ (t\ \#\ p');\ map\ fst\ (p\text{-}io\ (t\ \#\ p')) = x\ \#\ xs;\ map\ fst\ (p\text{-}io\ p') = xs]\!] \Longrightarrow thesis) \Longrightarrow thesis$›
‹$p = t\ \#\ p'$›
‹$paths\text{-}for\text{-}input'\ (h\text{-}from\ M)\ xs\ (t\text{-}target\ t)\ (prev\ @\ [(q,\ x,\ t\text{-}output\ t,\ t\text{-}target\ t)])$
$= \{(prev\ @\ [(q,\ x,\ t\text{-}output\ t,\ t\text{-}target\ t)])\ @\ p \mid p.\ path\ M\ (t\text{-}target\ t)\ p \wedge map\ fst\ (p\text{-}io\ p) = xs\}$›
‹$t\text{-}input\ t = x$›
‹$t\text{-}source\ t = q$›
by *fastforce*

ultimately show $pp \in$ *?UN* **unfolding** ‹$pp = prev@p$›
by *blast*
qed
then show $\{prev@p \mid p$. $path\ M\ q\ p \wedge map\ fst\ (p\text{-}io\ p) = x\#xs\} \subseteq$ *?UN*
by (*meson subsetI*)
qed

then show *?case*
by (*metis paths-for-input'.simps*(*2*))

qed

definition $paths\text{-}for\text{-}input :: ('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow ('a,'b,'c)\ path\ set$
where
$paths\text{-}for\text{-}input\ M\ q\ xs = \{p\ .\ path\ M\ q\ p \wedge map\ fst\ (p\text{-}io\ p) = xs\}$

lemma *paths-for-input-set-code*[*code*] :
paths-for-input M q xs = (*if q* ∈ *states M then paths-for-input′* (*h-from M*) *xs q* [] *else* {})
using *paths-for-input′-set*[*of q M xs* []]
unfolding *paths-for-input-def*
by (*cases q* ∈ *states M*; *auto*; *simp add*: *path-begin-state*)

fun *paths-up-to-length-or-condition-with-witness′* ::
(′*a* ⇒ (′*b* × ′*c* × ′*a*) *set*) ⇒ ((′*a*,′*b*,′*c*) *path* ⇒ ′*d option*) ⇒ (′*a*,′*b*,′*c*) *path* ⇒ *nat* ⇒ ′*a* ⇒ ((′*a*,′*b*,′*c*) *path* × ′*d*) *set*
where
paths-up-to-length-or-condition-with-witness′ f P prev 0 q = (*case P prev of Some w* ⇒ {(*prev*,*w*)} | *None* ⇒ {}) |
paths-up-to-length-or-condition-with-witness′ f P prev (*Suc k*) *q* = (*case P prev of*
Some w ⇒ {(*prev*,*w*)} |
None ⇒ (⋃(*image* (λ(*x*,*y*,*q′*) . *paths-up-to-length-or-condition-with-witness′ f P* (*prev*@[(*q*,*x*,*y*,*q′*)]) *k q′*) (*f q*))))

lemma *paths-up-to-length-or-condition-with-witness′-set* :
assumes *q* ∈ *states M*
shows *paths-up-to-length-or-condition-with-witness′* (*h-from M*) *P prev k q*
= {(*prev*@*p*,*x*) | *p x* . *path M q p*
∧ *length p* ≤ *k*
∧ *P* (*prev*@*p*) = *Some x*
∧ (∀ *p′ p″* . (*p* = *p′*@*p″* ∧ *p″* ≠ []) ⟶ *P* (*prev*@*p′*) = *None*)}
using *assms* **proof** (*induction k arbitrary*: *q prev*)
case *0*
then show *?case* **proof** (*cases P prev*)
case *None* **then show** *?thesis* **by** *auto*
next
case (*Some w*)
then show *?thesis* **by** (*simp add*: *0.prems nil*)
qed
next
case (*Suc k*)
then show *?case* **proof** (*cases P prev*)
case (*Some w*)
then have (*prev*,*w*) ∈ {(*prev*@*p*,*x*) | *p x* . *path M q p*
∧ *length p* ≤ *Suc k*
∧ *P* (*prev*@*p*) = *Some x*
∧ (∀ *p′ p″* . (*p* = *p′*@*p″* ∧ *p″* ≠ []) ⟶ *P* (*prev*@*p′*) = *None*)}
by (*simp add*: *Suc.prems nil*)

then have $\{(prev@p,x) \mid p\ x\ .\ path\ M\ q\ p$
$\wedge\ length\ p \leq Suc\ k$
$\wedge\ P\ (prev@p) = Some\ x$
$\wedge\ (\forall\ p'\ p''\ .\ (p = p'@p'' \wedge p'' \neq []) \longrightarrow P\ (prev@p') = None)\}$
$= \{(prev,w)\}$
using *Some* **by** *fastforce*

then show *?thesis* **using** *Some* **by** *auto*
next
case *None*

have $(\bigcup(image\ (\lambda(x,y,q')\ .\ paths\text{-}up\text{-}to\text{-}length\text{-}or\text{-}condition\text{-}with\text{-}witness'\ (h\text{-}from\ M)\ P\ (prev@[(q,x,y,q')])\ k\ q')\ (h\text{-}from\ M\ q)))$
$= \{(prev@p,x) \mid p\ x\ .\ path\ M\ q\ p$
$\wedge\ length\ p \leq Suc\ k$
$\wedge\ P\ (prev@p) = Some\ x$
$\wedge\ (\forall\ p'\ p''\ .\ (p = p'@p'' \wedge p'' \neq []) \longrightarrow P\ (prev@p') = None)\}$
(**is** *?UN* = *?PX*)
proof −
have $*$: $\bigwedge\ pp\ .\ pp \in ?UN \Longrightarrow pp \in ?PX$
proof −
fix *pp* **assume** $pp \in ?UN$
then obtain $x\ y\ q'$ **where** $(x,y,q') \in h\text{-}from\ M\ q$
and $pp \in paths\text{-}up\text{-}to\text{-}length\text{-}or\text{-}condition\text{-}with\text{-}witness'\ (h\text{-}from\ M)\ P\ (prev@[(q,x,y,q')])\ k\ q'$
by *blast*
then have $(q,x,y,q') \in transitions\ M$ **by** *auto*
then have $q' \in states\ M$ **using** *fsm-transition-target* **by** *auto*

obtain $p\ w$ **where** $pp = ((prev@[(q,x,y,q')])@p,w)$
and $path\ M\ q'\ p$
and $length\ p \leq k$
and $P\ ((prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p) = Some\ w$
and $\bigwedge\ p'\ p''.\ p = p'\ @\ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ ((prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p') = None$
using ‹$pp \in paths\text{-}up\text{-}to\text{-}length\text{-}or\text{-}condition\text{-}with\text{-}witness'\ (h\text{-}from\ M)\ P\ (prev@[(q,x,y,q')])\ k\ q'$›
unfolding *Suc.IH*[*OF* ‹$q' \in states\ M$›, *of* $prev@[(q,x,y,q')]$]
by *blast*

have $path\ M\ q\ ((q,x,y,q')\#p)$
using ‹$path\ M\ q'\ p$› ‹$(q,x,y,q') \in transitions\ M$› **by** (*simp add*: *path-prepend-t*)

moreover have $length\ ((q,x,y,q')\#p) \leq Suc\ k$
using ‹$length\ p \leq k$› **by** *auto*
moreover have $P\ (prev\ @\ ([(q,\ x,\ y,\ q')]\ @\ p)) = Some\ w$
using ‹$P\ ((prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p) = Some\ w$› **by** *auto*

moreover have $\bigwedge p'\ p''.\ ((q,x,y,q')\#p) = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ (prev\ @\ p') = None$
using ‹$\bigwedge p'\ p''.\ p = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ ((prev\ @\ [(q,\ x,\ y,\ q')])\ @\ p') = None$›
using *None*
by (*metis* (*no-types*, *opaque-lifting*) *append.simps*(*1*) *append-Cons append-Nil2 append-assoc*
list.inject neq-Nil-conv)

ultimately show $pp \in ?PX$
unfolding ‹$pp = ((prev@[(q,x,y,q')])@p,w)$› **by** *auto*
qed

have $**$: $\bigwedge pp\ .\ pp \in ?PX \Longrightarrow pp \in ?UN$
proof −
fix *pp* **assume** $pp \in ?PX$
then obtain $p'\ w$ **where** $pp = (prev\ @\ p',\ w)$
and $path\ M\ q\ p'$
and $length\ p' \leq Suc\ k$
and $P\ (prev\ @\ p') = Some\ w$
and $\bigwedge p'\ p''.\ p' = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ (prev\ @\ p') = None$
by *blast*
moreover obtain $t\ p$ **where** $p' = t\#p$ **using** ‹$P\ (prev\ @\ p') = Some\ w$› **using** *None*
by (*metis append-Nil2 list.exhaust option.distinct*(*1*))

have $pp = ((prev\ @\ [t])@p,\ w)$
using ‹$pp = (prev\ @\ p',\ w)$› **unfolding** ‹$p' = t\#p$› **by** *auto*
have $path\ M\ q\ (t\#p)$
using ‹$path\ M\ q\ p'$› **unfolding** ‹$p' = t\#p$› **by** *auto*
have *p2*: $length\ (t\#p) \leq Suc\ k$
using ‹$length\ p' \leq Suc\ k$› **unfolding** ‹$p' = t\#p$› **by** *auto*
have *p3*: $P\ ((prev\ @\ [t])@p) = Some\ w$
using ‹$P\ (prev\ @\ p') = Some\ w$› **unfolding** ‹$p' = t\#p$› **by** *auto*
have *p4*: $\bigwedge p'\ p''.\ p = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ ((prev@[t])\ @\ p') = None$
using ‹$\bigwedge p'\ p''.\ p' = p' @ p'' \Longrightarrow p'' \neq [] \Longrightarrow P\ (prev\ @\ p') = None$› ‹$pp \in ?PX$›
unfolding ‹$pp = ((prev\ @\ [t])\ @\ p,\ w)$› ‹$p' = t\#p$›
by *auto*

have $t \in transitions\ M$ **and** *p1*: $path\ M\ (t\text{-}target\ t)\ p$ **and** $t\text{-}source\ t = q$
using ‹$path\ M\ q\ (t\#p)$› **by** *auto*
then have $t\text{-}target\ t \in states\ M$
and $(t\text{-}input\ t,\ t\text{-}output\ t,\ t\text{-}target\ t) \in h\text{-}from\ M\ q$
and $t\text{-}source\ t = q$
using *fsm-transition-target* **by** *auto*
then have $t = (q,t\text{-}input\ t,\ t\text{-}output\ t,\ t\text{-}target\ t)$

by *auto*

have $((prev\ @\ [t])@p,\ w) \in$ *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* (*prev*@[*t*]) *k* (*t-target t*)
unfolding *Suc.IH*[*OF* ‹*t-target t* ∈ *states M*›, *of prev*@[*t*]]
using *p1 p2 p3 p4* **by** *auto*

then show *pp* ∈ *?UN*
unfolding ‹*pp* = ((*prev* @ [*t*])@*p*, *w*)›
proof −
have *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* (*prev* @ [*t*]) *k* (*t-target t*)
= *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* (*prev* @ [(*q*, *t-input t*, *t-output t*, *t-target t*)]) *k* (*t-target t*)
using ‹*t* = (*q*, *t-input t*, *t-output t*, *t-target t*)› **by** *presburger*
then show ((*prev* @ [*t*]) @ *p*, *w*) ∈ (⋃(*b*, *c*, *a*)∈*h-from M q*. *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* (*prev* @ [(*q*, *b*, *c*, *a*)]) *k a*)
using ‹((*prev* @ [*t*]) @ *p*, *w*) ∈ *paths-up-to-length-or-condition-with-witness'* (*h-from M*) *P* (*prev* @ [*t*]) *k* (*t-target t*)›
‹(*t-input t*, *t-output t*, *t-target t*) ∈ *h-from M q*›
by *blast*
qed
qed

show *?thesis*
using *subsetI*[*of ?UN ?PX*, *OF* ∗] *subsetI*[*of ?PX ?UN*, *OF* ∗∗] *subset-antisym* **by** *blast*
qed

then show *?thesis*
using *None* **unfolding** *paths-up-to-length-or-condition-with-witness'.simps* **by** *simp*
qed
qed

definition *paths-up-to-length-or-condition-with-witness* ::
('*a*,'*b*,'*c*) *fsm* ⇒ (('*a*,'*b*,'*c*) *path* ⇒ '*d option*) ⇒ *nat* ⇒ '*a* ⇒ (('*a*,'*b*,'*c*) *path* × '*d*) *set*
where
paths-up-to-length-or-condition-with-witness M P k q
= {(*p*,*x*) | *p x* . *path M q p*
∧ *length p* ≤ *k*
∧ *P p* = *Some x*
∧ (∀ *p' p''* . (*p* = *p'*@*p''* ∧ *p''* ≠ []) ⟶ *P p'* = *None*)}

lemma *paths-up-to-length-or-condition-with-witness-code*[*code*] :

paths-up-to-length-or-condition-with-witness M P k q
= (*if q* ∈ *states M then paths-up-to-length-or-condition-with-witness′* (*h-from M*) *P* [] *k q*
else {})
proof (*cases q* ∈ *states M*)
case *True*
then show *?thesis*
unfolding *paths-up-to-length-or-condition-with-witness-def*
paths-up-to-length-or-condition-with-witness′-set[*OF True*]
by *auto*
next
case *False*
then show *?thesis*
unfolding *paths-up-to-length-or-condition-with-witness-def*
using *path-begin-state* **by** *fastforce*
qed

lemma *paths-up-to-length-or-condition-with-witness-finite* :
finite (*paths-up-to-length-or-condition-with-witness M P k q*)
proof −
have *paths-up-to-length-or-condition-with-witness M P k q*
⊆ {(*p*, *the* (*P p*)) | *p* . *path M q p* ∧ *length p* ≤ *k*}
unfolding *paths-up-to-length-or-condition-with-witness-def*
by *auto*
moreover have *finite* {(*p*, *the* (*P p*)) | *p* . *path M q p* ∧ *length p* ≤ *k*}
using *paths-finite*[*of M q k*]
by *simp*
ultimately show *?thesis*
using *rev-finite-subset* **by** *auto*
qed

4.18 More Acyclicity Properties

lemma *maximal-path-target-deadlock* :
assumes *path M* (*initial M*) *p*
and ¬(∃ *p′* . *path M* (*initial M*) *p′* ∧ *is-prefix p p′* ∧ *p* ≠ *p′*)
shows *deadlock-state M* (*target* (*initial M*) *p*)
proof −
have ¬(∃ *t* ∈ *transitions M* . *t-source t* = *target* (*initial M*) *p*)
using *assms*(*2*) **unfolding** *is-prefix-prefix*
by (*metis append-Nil2 assms*(*1*) *not-Cons-self2 path-append-transition same-append-eq*)
then show *?thesis* **by** *auto*
qed

lemma *path-to-deadlock-is-maximal* :
assumes *path M* (*initial M*) *p*
and *deadlock-state M* (*target* (*initial M*) *p*)
shows ¬(∃ *p′* . *path M* (*initial M*) *p′* ∧ *is-prefix p p′* ∧ *p* ≠ *p′*)

proof
assume $\exists p'.$ *path M* (*initial M*) $p' \wedge$ *is-prefix p* $p' \wedge p \neq p'$
then obtain p' **where** *path M* (*initial M*) p' **and** *is-prefix p* p' **and** $p \neq p'$ **by** *blast*
then have *length* $p' >$ *length p*
unfolding *is-prefix-prefix* **by** *auto*
then obtain *t p2* **where** $p' = p$ @ [*t*] @ *p2*
using ‹*is-prefix p* p'› **unfolding** *is-prefix-prefix*
by (*metis* ‹$p \neq p'$› *append.left-neutral append-Cons append-Nil2 non-sym-dist-pairs'.cases*)

then have *path M* (*initial M*) (*p*@[*t*])
using ‹*path M* (*initial M*) p'› **by** *auto*
then have $t \in$ *transitions M* **and** *t-source t* = *target* (*initial M*) *p*
by *auto*
then show *False*
using ‹*deadlock-state M* (*target* (*initial M*) *p*)› **unfolding** *deadlock-state.simps* **by** *blast*
qed

definition *maximal-acyclic-paths* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *path set* **where**
maximal-acyclic-paths M = {*p* . *path M* (*initial M*) *p*
$\wedge$ *distinct* (*visited-states* (*initial M*) *p*)
$\wedge \neg(\exists$ p' . $p' \neq []$ $\wedge$ *path M* (*initial M*) (*p*@p')
$\wedge$ *distinct* (*visited-states* (*initial M*) (*p*@p')))}

lemma *maximal-acyclic-paths-code*[*code*] :
maximal-acyclic-paths M = (*let ps* = *acyclic-paths-up-to-length M* (*initial M*) (*size M* − *1*)
in Set.filter (λp . $\neg$ ($\exists$ $p' \in ps$. $p' \neq p \wedge$ *is-prefix p* p')) *ps*)
proof −
have *scheme1*: $\bigwedge$ *P p* . ($\exists$ p' . $p' \neq []$ $\wedge$ *P* (*p*@p')) = ($\exists$ $p' \in \{p$. *P p*} . $p' \neq p \wedge$ *is-prefix p* p')
unfolding *is-prefix-prefix* **by** *blast*

have *scheme2*: $\bigwedge$ *p* . (*path M* (*FSM.initial M*) *p*
$\wedge$ *length p* $\leq$ *FSM.size M* − *1*
$\wedge$ *distinct* (*visited-states* (*FSM.initial M*) *p*))
= (*path M* (*FSM.initial M*) *p* $\wedge$ *distinct* (*visited-states* (*FSM.initial M*) *p*))
using *acyclic-path-length-limit* **by** *fastforce*

show *?thesis*
unfolding *maximal-acyclic-paths-def acyclic-paths-up-to-length.simps Let-def*
unfolding *scheme1*[*of* λp . *path M* (*initial M*) *p* $\wedge$ *distinct* (*visited-states*

(*initial M*) *p*)]
  unfolding *scheme2* **by** *fastforce*
qed

lemma *maximal-acyclic-path-deadlock* :
  assumes *acyclic M*
  and *path M* (*initial M*) *p*
shows ¬(∃ *p′* . *p′* ≠ [] ∧ *path M* (*initial M*) (*p*@*p′*) ∧ *distinct* (*visited-states* (*initial M*) (*p*@*p′*)))
    = *deadlock-state M* (*target* (*initial M*) *p*)
proof −
  have *deadlock-state M* (*target* (*initial M*) *p*) ⟹ ¬(∃ *p′* . *p′* ≠ [] ∧ *path M* (*initial M*) (*p*@*p′*)
      ∧ *distinct* (*visited-states* (*initial M*) (*p*@*p′*)))
    unfolding *deadlock-state.simps*
    using *assms*(*2*) **by** (*metis path.cases path-suffix*)
  then show *?thesis*
  by (*metis acyclic.elims*(*2*) *assms*(*1*) *assms*(*2*) *is-prefix-prefix maximal-path-target-deadlock*

      self-append-conv)
qed

lemma *maximal-acyclic-paths-deadlock-targets* :
  assumes *acyclic M*
  shows *maximal-acyclic-paths M*
    = { *p* . *path M* (*initial M*) *p* ∧ *deadlock-state M* (*target* (*initial M*) *p*)}
  using *maximal-acyclic-path-deadlock*[*OF assms*]
  unfolding *maximal-acyclic-paths-def*
  by (*metis* (*no-types*, *lifting*) *acyclic.elims*(*2*) *assms*)

lemma *cycle-from-cyclic-path* :
  assumes *path M q p*
  and ¬ *distinct* (*visited-states q p*)
obtains *i j* **where**
  take j (*drop i p*) ≠ []
  target (*target q* (*take i p*)) (*take j* (*drop i p*)) = (*target q* (*take i p*))
  path M (*target q* (*take i p*)) (*take j* (*drop i p*))
proof −
  obtain *i j* **where** *i* < *j* **and** *j* < *length* (*visited-states q p*)
      and (*visited-states q p*) ! *i* = (*visited-states q p*) ! *j*
    using *assms*(*2*) *non-distinct-repetition-indices* **by** *blast*

  have (*target q* (*take i p*)) = (*visited-states q p*) ! *i*
    using ‹*i* < *j*› ‹*j* < *length* (*visited-states q p*)›
    by (*metis less-trans take-last-index target.simps visited-states-take*)

then have (*target q* (*take i p*)) = (*visited-states q p*) ! *j*
using ‹(*visited-states q p*) ! *i* = (*visited-states q p*) ! *j*› **by** *auto*

have *p1*: *take* (*j*−*i*) (*drop i p*) ≠ []
using ‹*i* < *j*› ‹*j* < *length* (*visited-states q p*)› **by** *auto*

have *target* (*target q* (*take i p*)) (*take* (*j*−*i*) (*drop i p*)) = (*target q* (*take j p*))
using ‹*i* < *j*› **by** (*metis add-diff-inverse-nat less-asym′ path-append-target take-add*)
then have *p2*: *target* (*target q* (*take i p*)) (*take* (*j*−*i*) (*drop i p*)) = (*target q* (*take i p*))
using ‹(*target q* (*take i p*)) = (*visited-states q p*) ! *i*›
using ‹(*target q* (*take i p*)) = (*visited-states q p*) ! *j*›
by (*metis* ‹*j* < *length* (*visited-states q p*)› *take-last-index target.simps visited-states-take*)

have *p3*: *path M* (*target q* (*take i p*)) (*take* (*j*−*i*) (*drop i p*))
by (*metis append-take-drop-id assms*(*1*) *path-append-elim*)

show *?thesis* **using** *p1 p2 p3 that* **by** *blast*
qed

lemma *acyclic-single-deadlock-reachable* :
assumes *acyclic M*
and ⋀ *q′* . *q′* ∈ *reachable-states M* ⟹ *q′* = *qd* ∨ ¬ *deadlock-state M q′*
shows *qd* ∈ *reachable-states M*
using *acyclic-deadlock-reachable*[*OF assms*(*1*)]
using *assms*(*2*) **by** *auto*

lemma *acyclic-paths-to-single-deadlock* :
assumes *acyclic M*
and ⋀ *q′* . *q′* ∈ *reachable-states M* ⟹ *q′* = *qd* ∨ ¬ *deadlock-state M q′*
and *q* ∈ *reachable-states M*
obtains *p* **where** *path M q p* **and** *target q p* = *qd*
proof −
have *q* ∈ *states M* **using** *assms*(*3*) *reachable-state-is-state* **by** *metis*
have *acyclic* (*from-FSM M q*)
using *from-FSM-acyclic*[*OF assms*(*3*,*1*)] **by** *assumption*

have ∗: (⋀*q′*. *q′* ∈ *reachable-states* (*FSM.from-FSM M q*)
⟹ *q′* = *qd* ∨ ¬ *deadlock-state* (*FSM.from-FSM M q*) *q′*)
using *assms*(*2*) *from-FSM-reachable-states*[*OF assms*(*3*)]
unfolding *deadlock-state.simps from-FSM-simps*[*OF* ‹*q* ∈ *states M*›] **by** *blast*

obtain *p* **where** *path* (*from-FSM M q*) *q p* **and** *target q p* = *qd*

using *acyclic-single-deadlock-reachable*[*OF* ‹*acyclic* (*from-FSM M q*)› ∗]
unfolding *reachable-states-def from-FSM-simps*[*OF* ‹*q* ∈ *states M*›]
by *blast*

then show *?thesis*
using *that* **by** (*metis* ‹*q* ∈ *FSM.states M*› *from-FSM-path*)
qed

4.19 Elements as Lists

fun *states-as-list* :: (*'a* :: *linorder*, *'b*, *'c*) *fsm* ⇒ *'a list* **where**
states-as-list M = *sorted-list-of-set* (*states M*)

lemma *states-as-list-distinct* : *distinct* (*states-as-list M*) **by** *auto*

lemma *states-as-list-set* : *set* (*states-as-list M*) = *states M*
by (*simp add*: *fsm-states-finite*)

fun *reachable-states-as-list* :: (*'a* :: *linorder*, *'b*, *'c*) *fsm* ⇒ *'a list* **where**
reachable-states-as-list M = *sorted-list-of-set* (*reachable-states M*)

lemma *reachable-states-as-list-distinct* : *distinct* (*reachable-states-as-list M*) **by** *auto*

lemma *reachable-states-as-list-set* : *set* (*reachable-states-as-list M*) = *reachable-states M*
by (*metis fsm-states-finite infinite-super reachable-state-is-state reachable-states-as-list.simps*

set-sorted-list-of-set subsetI)

fun *inputs-as-list* :: (*'a*, *'b* :: *linorder*, *'c*) *fsm* ⇒ *'b list* **where**
inputs-as-list M = *sorted-list-of-set* (*inputs M*)

lemma *inputs-as-list-set* : *set* (*inputs-as-list M*) = *inputs M*
by (*simp add*: *fsm-inputs-finite*)

lemma *inputs-as-list-distinct* : *distinct* (*inputs-as-list M*) **by** *auto*

fun *transitions-as-list* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*'a*,*'b*,*'c*) *transition list* **where**
transitions-as-list M = *sorted-list-of-set* (*transitions M*)

lemma *transitions-as-list-set* : *set* (*transitions-as-list M*) = *transitions M*
by (*simp add*: *fsm-transitions-finite*)

fun *outputs-as-list* :: (*'a*,*'b*,*'c* :: *linorder*) *fsm* ⇒ *'c list* **where**
outputs-as-list M = *sorted-list-of-set* (*outputs M*)

lemma *outputs-as-list-set* : *set* (*outputs-as-list M*) = *outputs M*
by (*simp add*: *fsm-outputs-finite*)

fun *ftransitions* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a*,'*b*,'*c*) *transition fset* **where**
ftransitions M = *fset-of-list* (*transitions-as-list M*)

fun *fstates* :: ('*a* :: *linorder*,'*b*,'*c*) *fsm* ⇒ '*a fset* **where**
fstates M = *fset-of-list* (*states-as-list M*)

fun *finputs* :: ('*a*,'*b* :: *linorder*,'*c*) *fsm* ⇒ '*b fset* **where**
finputs M = *fset-of-list* (*inputs-as-list M*)

fun *foutputs* :: ('*a*,'*b*,'*c* :: *linorder*) *fsm* ⇒ '*c fset* **where**
foutputs M = *fset-of-list* (*outputs-as-list M*)

lemma *fstates-set* : *fset* (*fstates M*) = *states M*
using *fsm-states-finite*[*of M*] **by** (*simp add*: *fset-of-list.rep-eq*)

lemma *finputs-set* : *fset* (*finputs M*) = *inputs M*
using *fsm-inputs-finite*[*of M*] **by** (*simp add*: *fset-of-list.rep-eq*)

lemma *foutputs-set* : *fset* (*foutputs M*) = *outputs M*
using *fsm-outputs-finite*[*of M*] **by** (*simp add*: *fset-of-list.rep-eq*)

lemma *ftransitions-set*: *fset* (*ftransitions M*) = *transitions M*
by (*metis* (*no-types*) *fset-of-list.rep-eq ftransitions.simps transitions-as-list-set*)

lemma *ftransitions-source*:
q |∈| (*t-source* |`| *ftransitions M*) ⟹ *q* ∈ *states M*
using *ftransitions-set*[*of M*] *fsm-transition-source*[*of - M*]
by (*metis* (*no-types*, *opaque-lifting*) *fimageE*)

lemma *ftransitions-target*:
q |∈| (*t-target* |`| *ftransitions M*) ⟹ *q* ∈ *states M*
using *ftransitions-set*[*of M*] *fsm-transition-target*[*of - M*]
by (*metis* (*no-types*, *lifting*) *fimageE*)

4.20 Responses to Input Sequences

fun *language-for-input* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ '*b list* ⇒ ('*b*×'*c*) *list list* **where**
language-for-input M q [] = [[]] |
language-for-input M q (*x*#*xs*) =
(*let outs* = *outputs-as-list M*
in concat (*map* (λ*y* . *case h-obs M q x y of None* ⇒ [] | *Some q'* ⇒ *map* ((#) (*x*,*y*)) (*language-for-input M q' xs*)) *outs*))

lemma *language-for-input-set* :
  assumes *observable M*
  and $q \in$ *states M*
shows *list.set* (*language-for-input M q xs*) = {*io* . *io* ∈ *LS M q* ∧ *map fst io* = *xs*}
  using *assms*(*2*) **proof** (*induction xs arbitrary*: *q*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons x xs*)

  have *list.set* (*language-for-input M q* (*x*#*xs*)) ⊆ {*io* . *io* ∈ *LS M q* ∧ *map fst io* = (*x*#*xs*)}
  proof
    fix *io* **assume** *io* ∈ *list.set* (*language-for-input M q* (*x*#*xs*))
    then obtain *y* **where** *y* ∈ *outputs M*
                and *io* ∈ *list.set* (*case h-obs M q x y of None* ⇒ [] | *Some q′* ⇒ *map* ((#) (*x*,*y*)) (*language-for-input M q′ xs*))
      unfolding *outputs-as-list-set*[*symmetric*]
      by *auto*
    then obtain *q′* **where** *h-obs M q x y* = *Some q′* **and** *io* ∈ *list.set* (*map* ((#) (*x*,*y*)) (*language-for-input M q′ xs*))
      by (*cases h-obs M q x y*; *auto*)

    then obtain *io′* **where** *io* = (*x*,*y*)#*io′*
                and *io′* ∈ *list.set* (*language-for-input M q′ xs*)
      by *auto*

    then have *io′* ∈ *LS M q′* **and** *map fst io′* = *xs*
      using *Cons.IH*[*OF h-obs-state*[*OF* ‹*h-obs M q x y* = *Some q′*›]]
      by *blast*+
    then have (*x*,*y*)#*io′* ∈ *LS M q*
      using ‹*h-obs M q x y* = *Some q′*›
      unfolding *h-obs-language-iff*[*OF assms*(*1*), *of x y io′ q*]
      by *blast*
    then show *io* ∈ {*io* . *io* ∈ *LS M q* ∧ *map fst io* = (*x*#*xs*)}
      unfolding ‹*io* = (*x*,*y*)#*io′*›
      using ‹*map fst io′* = *xs*›
      by *auto*
  qed
  moreover have {*io* . *io* ∈ *LS M q* ∧ *map fst io* = (*x*#*xs*)} ⊆ *list.set* (*language-for-input M q* (*x*#*xs*))
  proof
    have *scheme* : ⋀ *x y f xs* . *y* ∈ *list.set* (*f x*) ⟹ *x* ∈ *list.set xs* ⟹ *y* ∈ *list.set* (*concat* (*map f xs*))
      by *auto*

    fix *io* **assume** *io* ∈ {*io* . *io* ∈ *LS M q* ∧ *map fst io* = (*x*#*xs*)}
    then have *io* ∈ *LS M q* **and** *map fst io* = (*x*#*xs*)
      by *auto*

then obtain *y io′* **where** *io* = (*x*,*y*)#*io′*
by *fastforce*
then have (*x*,*y*)#*io′* ∈ *LS M q*
using ‹*io* ∈ *LS M q*›
by *auto*
then obtain *q′* **where** *h-obs M q x y* = *Some q′* **and** *io′* ∈ *LS M q′*
unfolding *h-obs-language-iff*[*OF assms*(*1*), *of x y io′ q*]
by *blast*
moreover have *io′* ∈ *list.set* (*language-for-input M q′ xs*)
using *Cons.IH*[*OF h-obs-state*[*OF* ‹*h-obs M q x y* = *Some q′*›]] ‹*io′* ∈ *LS M q′*› ‹*map fst io* = (*x*#*xs*)›
unfolding ‹*io* = (*x*,*y*)#*io′*› **by** *auto*
ultimately have *io* ∈ *list.set* ((λ *y* .(*case h-obs M q x y of None* ⇒ [] | *Some q′* ⇒ *map* ((#) (*x*,*y*)) (*language-for-input M q′ xs*))) *y*)
unfolding ‹*io* = (*x*,*y*)#*io′*›
by *force*
moreover have *y* ∈ *list.set* (*outputs-as-list M*)
unfolding *outputs-as-list-set*
using *language-io*(*2*)[*OF* ‹(*x*,*y*)#*io′* ∈ *LS M q*›] **by** *auto*
ultimately show *io* ∈ *list.set* (*language-for-input M q* (*x*#*xs*))
unfolding *language-for-input.simps Let-def*
using *scheme*[*of io* (λ *y* .(*case h-obs M q x y of None* ⇒ [] | *Some q′* ⇒ *map* ((#) (*x*,*y*)) (*language-for-input M q′ xs*))) *y*]
by *blast*
qed
ultimately show *?case*
by *blast*
qed

4.21 Filtering Transitions

lift-definition *filter-transitions* ::
(′*a*,′*b*,′*c*) *fsm* ⇒ ((′*a*,′*b*,′*c*) *transition* ⇒ *bool*) ⇒ (′*a*,′*b*,′*c*) *fsm* **is** *FSM-Impl.filter-transitions*

proof −
fix *M* :: (′*a*,′*b*,′*c*) *fsm-impl*
fix *P* :: (′*a*,′*b*,′*c*) *transition* ⇒ *bool*
assume *well-formed-fsm M*
then show *well-formed-fsm* (*FSM-Impl.filter-transitions M P*)
unfolding *FSM-Impl.filter-transitions.simps* **by** *force*
qed

lemma *filter-transitions-simps*[*simp*] :
initial (*filter-transitions M P*) = *initial M*
states (*filter-transitions M P*) = *states M*
inputs (*filter-transitions M P*) = *inputs M*
outputs (*filter-transitions M P*) = *outputs M*
transitions (*filter-transitions M P*) = {*t* ∈ *transitions M* . *P t*}

by (*transfer*;*auto*)+

lemma *filter-transitions-submachine* :
is-submachine (*filter-transitions M P*) *M*
unfolding *filter-transitions-simps* **by** *fastforce*

lemma *filter-transitions-path* :
assumes *path* (*filter-transitions M P*) *q p*
shows *path M q p*
using *path-begin-state*[*OF assms*]
transition-subset-path[*of filter-transitions M P M*, *OF - assms*]
unfolding *filter-transitions-simps* **by** *blast*

lemma *filter-transitions-reachable-states* :
assumes $q \in$ *reachable-states* (*filter-transitions M P*)
shows $q \in$ *reachable-states M*
using *assms* **unfolding** *reachable-states-def filter-transitions-simps*
using *filter-transitions-path*[*of M P initial M*]
by *blast*

4.22 Filtering States

lift-definition *filter-states* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a* ⇒ *bool*) ⇒ ('*a*,'*b*,'*c*) *fsm*
is *FSM-Impl.filter-states*
proof −
fix *M* :: ('*a*,'*b*,'*c*) *fsm-impl*
fix *P* :: '*a* ⇒ *bool*
assume ∗: *well-formed-fsm M*

then show *well-formed-fsm* (*FSM-Impl.filter-states M P*)
by (*cases P* (*FSM-Impl.initial M*); *auto*)
qed

lemma *filter-states-simps*[*simp*] :
assumes *P* (*initial M*)
shows *initial* (*filter-states M P*) = *initial M*
states (*filter-states M P*) = *Set.filter P* (*states M*)
inputs (*filter-states M P*) = *inputs M*
outputs (*filter-states M P*) = *outputs M*
transitions (*filter-states M P*) = {*t* ∈ *transitions M* . *P* (*t-source t*) ∧ *P* (*t-target t*)}
using *assms* **by** (*transfer*;*auto*)+

lemma *filter-states-submachine* :
assumes *P* (*initial M*)
shows *is-submachine* (*filter-states M P*) *M*

using *filter-states-simps*[*of P M*, *OF assms*] **by** *fastforce*

fun *restrict-to-reachable-states* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *fsm* **where**
restrict-to-reachable-states M = *filter-states M* (λ *q* . *q* ∈ *reachable-states M*)

lemma *restrict-to-reachable-states-simps*[*simp*] :
shows *initial* (*restrict-to-reachable-states M*) = *initial M*
states (*restrict-to-reachable-states M*) = *reachable-states M*
inputs (*restrict-to-reachable-states M*) = *inputs M*
outputs (*restrict-to-reachable-states M*) = *outputs M*
transitions (*restrict-to-reachable-states M*)
= {*t* ∈ *transitions M* . (*t-source t*) ∈ *reachable-states M*}
proof −
show *initial* (*restrict-to-reachable-states M*) = *initial M*
states (*restrict-to-reachable-states M*) = *reachable-states M*
inputs (*restrict-to-reachable-states M*) = *inputs M*
outputs (*restrict-to-reachable-states M*) = *outputs M*

using *filter-states-simps*[*of* (λ *q* . *q* ∈ *reachable-states M*), *OF reachable-states-initial*]

using *reachable-state-is-state*[*of - M*] **by** *auto*

have *transitions* (*restrict-to-reachable-states M*)
= {*t* ∈ *transitions M*. (*t-source t*) ∈ *reachable-states M* ∧ (*t-target t*) ∈ *reachable-states M*}
using *filter-states-simps*[*of* (λ *q* . *q* ∈ *reachable-states M*), *OF reachable-states-initial*]

by *auto*
then show *transitions* (*restrict-to-reachable-states M*)
= {*t* ∈ *transitions M* . (*t-source t*) ∈ *reachable-states M*}
using *reachable-states-next*[*of - M*] **by** *auto*
qed

lemma *restrict-to-reachable-states-path* :
assumes *q* ∈ *reachable-states M*
shows *path M q p* = *path* (*restrict-to-reachable-states M*) *q p*
proof
show *path M q p* ⟹ *path* (*restrict-to-reachable-states M*) *q p*
proof −
assume *path M q p*
then show *path* (*restrict-to-reachable-states M*) *q p*
using *assms* **proof** (*induction p arbitrary*: *q rule*: *list.induct*)
case *Nil*
then show *?case*
using *restrict-to-reachable-states-simps*(*2*) **by** *fastforce*

```
    next
      case (Cons t' p')
      then have path M (t-target t') p' by auto
      moreover have t-target t' ∈ reachable-states M using Cons.prems
        by (metis path-cons-elim reachable-states-next)
      ultimately show ?case using Cons.IH
        by (metis (no-types, lifting) Cons.prems(1) Cons.prems(2) mem-Collect-eq
path.simps
              path-cons-elim restrict-to-reachable-states-simps(5))
    qed
  qed

  show path (restrict-to-reachable-states M) q p ⟹ path M q p
    by (metis (no-types, lifting) assms mem-Collect-eq reachable-state-is-state
          restrict-to-reachable-states-simps(5) subsetI transition-subset-path)
qed

lemma restrict-to-reachable-states-language :
  L (restrict-to-reachable-states M) = L M
  unfolding LS.simps
  unfolding restrict-to-reachable-states-simps
  unfolding restrict-to-reachable-states-path[OF reachable-states-initial, of M]
  by blast

lemma restrict-to-reachable-states-observable :
  assumes observable M
shows observable (restrict-to-reachable-states M)
  using assms unfolding observable.simps
  unfolding restrict-to-reachable-states-simps
  by blast

lemma restrict-to-reachable-states-minimal :
  assumes minimal M
  shows minimal (restrict-to-reachable-states M)
proof −
  have ⋀ q1 q2 . q1 ∈ reachable-states M ⟹
                q2 ∈ reachable-states M ⟹
                q1 ≠ q2 ⟹
            LS (restrict-to-reachable-states M) q1 ≠ LS (restrict-to-reachable-states
M) q2
  proof −
    fix q1 q2 assume q1 ∈ reachable-states M and q2 ∈ reachable-states M and
q1 ≠ q2
    then have q1 ∈ states M and q2 ∈ states M
      by (simp add: reachable-state-is-state)+
    then have LS M q1 ≠ LS M q2
      using ‹q1 ≠ q2› assms by auto
   then show LS (restrict-to-reachable-states M) q1 ≠ LS (restrict-to-reachable-states
M) q2
```

unfolding *LS.simps*
unfolding *restrict-to-reachable-states-path*[*OF* ‹*q1* ∈ *reachable-states M*›]
unfolding *restrict-to-reachable-states-path*[*OF* ‹*q2* ∈ *reachable-states M*›] **.**
qed
then show *?thesis*
unfolding *minimal.simps restrict-to-reachable-states-simps*
by *blast*
qed

lemma *restrict-to-reachable-states-reachable-states* :
reachable-states (*restrict-to-reachable-states M*) = *states* (*restrict-to-reachable-states M*)
proof
show *reachable-states* (*restrict-to-reachable-states M*) ⊆ *states* (*restrict-to-reachable-states M*)
by (*simp add*: *reachable-state-is-state subsetI*)
show *states* (*restrict-to-reachable-states M*) ⊆ *reachable-states* (*restrict-to-reachable-states M*)
proof
fix *q* **assume** *q* ∈ *states* (*restrict-to-reachable-states M*)
then have *q* ∈ *reachable-states M*
unfolding *restrict-to-reachable-states-simps* **.**
then show *q* ∈ *reachable-states* (*restrict-to-reachable-states M*)
unfolding *reachable-states-def*
unfolding *restrict-to-reachable-states-simps*
unfolding *restrict-to-reachable-states-path*[*OF reachable-states-initial*, *symmetric*] **.**
qed
qed

4.23 Adding Transitions

lift-definition *create-unconnected-fsm* :: *'a* ⇒ *'a set* ⇒ *'b set* ⇒ *'c set* ⇒ (*'a*,*'b*,*'c*) *fsm*
is *FSM-Impl.create-unconnected-FSMI* **by** (*transfer*; *simp*)

lemma *create-unconnected-fsm-simps* :
assumes *finite ns* **and** *finite ins* **and** *finite outs* **and** *q* ∈ *ns*
shows *initial* (*create-unconnected-fsm q ns ins outs*) = *q*
states (*create-unconnected-fsm q ns ins outs*) = *ns*
inputs (*create-unconnected-fsm q ns ins outs*) = *ins*
outputs (*create-unconnected-fsm q ns ins outs*) = *outs*
transitions (*create-unconnected-fsm q ns ins outs*) = {}
using *assms* **by** (*transfer*; *auto*)+

lift-definition *create-unconnected-fsm-from-lists* :: *'a* ⇒ *'a list* ⇒ *'b list* ⇒ *'c list* ⇒ (*'a*,*'b*,*'c*) *fsm*
is *FSM-Impl.create-unconnected-fsm-from-lists* **by** (*transfer*; *simp*)

lemma *create-unconnected-fsm-from-lists-simps* :
assumes $q \in$ *set ns*
shows *initial* (*create-unconnected-fsm-from-lists q ns ins outs*) = *q*
states (*create-unconnected-fsm-from-lists q ns ins outs*) = *set ns*
inputs (*create-unconnected-fsm-from-lists q ns ins outs*) = *set ins*
outputs (*create-unconnected-fsm-from-lists q ns ins outs*) = *set outs*
transitions (*create-unconnected-fsm-from-lists q ns ins outs*) = {}
using *assms* **by** (*transfer*; *auto*)+

lift-definition *create-unconnected-fsm-from-fsets* :: *'a* ⇒ *'a fset* ⇒ *'b fset* ⇒ *'c fset* ⇒ (*'a*,*'b*,*'c*) *fsm*
is *FSM-Impl.create-unconnected-fsm-from-fsets* **by** (*transfer*; *simp*)

lemma *create-unconnected-fsm-from-fsets-simps* :
assumes *q* |∈| *ns*
shows *initial* (*create-unconnected-fsm-from-fsets q ns ins outs*) = *q*
states (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ns*
inputs (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ins*
outputs (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset outs*
transitions (*create-unconnected-fsm-from-fsets q ns ins outs*) = {}
using *assms* **by** (*transfer*; *auto*)+

lift-definition *add-transitions* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'a*,*'b*,*'c*) *transition set* ⇒ (*'a*,*'b*,*'c*) *fsm*
is *FSM-Impl.add-transitions*
proof −
fix *M* :: (*'a*,*'b*,*'c*) *fsm-impl*
fix *ts* :: (*'a*,*'b*,*'c*) *transition set*
assume ∗: *well-formed-fsm M*

then show *well-formed-fsm* (*FSM-Impl.add-transitions M ts*)
proof (*cases* ∀ *t* ∈ *ts* . *t-source t* ∈ *FSM-Impl.states M* ∧ *t-input t* ∈ *FSM-Impl.inputs M*
∧ *t-output t* ∈ *FSM-Impl.outputs M* ∧ *t-target t* ∈ *FSM-Impl.states M*)
case *True*
then have *ts* ⊆ *FSM-Impl.states M* × *FSM-Impl.inputs M* × *FSM-Impl.outputs M* × *FSM-Impl.states M*
by *fastforce*
moreover have *finite* (*FSM-Impl.states M* × *FSM-Impl.inputs M* × *FSM-Impl.outputs M* × *FSM-Impl.states M*)
using ∗ **by** *blast*
ultimately have *finite ts*
using *rev-finite-subset* **by** *auto*
then show *?thesis* **using** ∗ **by** *auto*
next
case *False*
then show *?thesis* **using** ∗ **by** *auto*

qed
qed

lemma *add-transitions-simps* :
assumes $\bigwedge$ *t* . *t* ∈ *ts* ⟹ *t-source t* ∈ *states M* ∧ *t-input t* ∈ *inputs M* ∧ *t-output t* ∈ *outputs M* ∧ *t-target t* ∈ *states M*
shows *initial* (*add-transitions M ts*) = *initial M*
states (*add-transitions M ts*) = *states M*
inputs (*add-transitions M ts*) = *inputs M*
outputs (*add-transitions M ts*) = *outputs M*
transitions (*add-transitions M ts*) = *transitions M* ∪ *ts*
using *assms* **by** (*transfer*; *auto*)+

lift-definition *create-fsm-from-sets* :: *'a* ⇒ *'a set* ⇒ *'b set* ⇒ *'c set* ⇒ (*'a*,*'b*,*'c*) *transition set* ⇒ (*'a*,*'b*,*'c*) *fsm*
is *FSM-Impl.create-fsm-from-sets*
proof −
fix *q* :: *'a*
fix *qs* :: *'a set*
fix *ins* :: *'b set*
fix *outs* :: *'c set*
fix *ts* :: (*'a*,*'b*,*'c*) *transition set*

show *well-formed-fsm* (*FSM-Impl.create-fsm-from-sets q qs ins outs ts*)
proof (*cases q* ∈ *qs* ∧ *finite qs* ∧ *finite ins* ∧ *finite outs*)
case *True*

let *?M* = (*FSMI q qs ins outs* {})

show *?thesis* **proof** (*cases* ∀ *t* ∈ *ts* . *t-source t* ∈ *FSM-Impl.states ?M* ∧ *t-input t* ∈ *FSM-Impl.inputs ?M*
∧ *t-output t* ∈ *FSM-Impl.outputs ?M* ∧ *t-target t* ∈ *FSM-Impl.states ?M*)
case *True*
then have *ts* ⊆ *FSM-Impl.states ?M* × *FSM-Impl.inputs ?M* × *FSM-Impl.outputs ?M* × *FSM-Impl.states ?M*
by *fastforce*
moreover have *finite* (*FSM-Impl.states ?M* × *FSM-Impl.inputs ?M* × *FSM-Impl.outputs ?M* × *FSM-Impl.states ?M*)
using ‹*q* ∈ *qs* ∧ *finite qs* ∧ *finite ins* ∧ *finite outs*› **by** *force*
ultimately have *finite ts*
using *rev-finite-subset* **by** *auto*
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis* **by** *auto*

qed
next
case *False*
then show *?thesis* **by** *auto*
qed
qed

lemma *create-fsm-from-sets-simps* :
assumes $q \in qs$ **and** *finite qs* **and** *finite ins* **and** *finite outs*
assumes $\bigwedge t\ .\ t \in ts \Longrightarrow$ *t-source* $t \in qs \wedge$ *t-input* $t \in ins \wedge$ *t-output* $t \in outs \wedge$ *t-target* $t \in qs$
shows *initial* (*create-fsm-from-sets q qs ins outs ts*) = *q*
states (*create-fsm-from-sets q qs ins outs ts*) = *qs*
inputs (*create-fsm-from-sets q qs ins outs ts*) = *ins*
outputs (*create-fsm-from-sets q qs ins outs ts*) = *outs*
transitions (*create-fsm-from-sets q qs ins outs ts*) = *ts*
using *assms* **by** (*transfer*; *auto*)+

lemma *create-fsm-from-self* :
m = *create-fsm-from-sets* (*initial m*) (*states m*) (*inputs m*) (*outputs m*) (*transitions m*)
proof −
have $*$: $\bigwedge t\ .\ t \in$ *transitions m* $\Longrightarrow$ *t-source* $t \in$ *states m* $\wedge$ *t-input* $t \in$ *inputs m* $\wedge$ *t-output* $t \in$ *outputs m* $\wedge$ *t-target* $t \in$ *states m*
by *auto*
show *?thesis*
using *create-fsm-from-sets-simps*[*OF fsm-initial fsm-states-finite fsm-inputs-finite fsm-outputs-finite* $*$, *of transitions m*]
apply *transfer*
by *force*
qed

lemma *create-fsm-from-sets-surj* :
assumes *finite* (*UNIV* :: *'a set*)
and *finite* (*UNIV* :: *'b set*)
and *finite* (*UNIV* :: *'c set*)
shows *surj* (λ(*q*::*'a*,*Q*,*X*::*'b set*,*Y*::*'c set*,*T*) . *create-fsm-from-sets q Q X Y T*)
proof
show *range* (λ(*q*::*'a*,*Q*,*X*::*'b set*,*Y*::*'c set*,*T*) . *create-fsm-from-sets q Q X Y T*) $\subseteq$ *UNIV*
by *simp*
show *UNIV* $\subseteq$ *range* (λ(*q*::*'a*,*Q*,*X*::*'b set*,*Y*::*'c set*,*T*) . *create-fsm-from-sets q Q X Y T*)
proof
fix *m* **assume** $m \in$ (*UNIV* :: (*'a*,*'b*,*'c*) *fsm set*)
then have *m* = *create-fsm-from-sets* (*initial m*) (*states m*) (*inputs m*) (*outputs m*) (*transitions m*)
using *create-fsm-from-self* **by** *blast*
then have *m* = (λ(*q*::*'a*,*Q*,*X*::*'b set*,*Y*::*'c set*,*T*) . *create-fsm-from-sets q Q X*

Y T) (*initial m*,*states m*,*inputs m*,*outputs m*,*transitions m*)
by *auto*
then show $m \in$ *range* (λ(*q*::'*a*,*Q*,*X*::'*b set*,*Y*::'*c set*,*T*) . *create-fsm-from-sets q Q X Y T*)
by *blast*
qed
qed

4.24 Distinguishability

definition *distinguishes* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ ('*b* $\times$'*c*) *list* $\Rightarrow$ *bool* **where**
distinguishes M q1 q2 io = (*io* $\in$ *LS M q1* $\cup$ *LS M q2* $\wedge$ *io* $\notin$ *LS M q1* $\cap$ *LS M q2*)

definition *minimally-distinguishes* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ ('*b* $\times$'*c*) *list* $\Rightarrow$ *bool* **where**
minimally-distinguishes M q1 q2 io = (*distinguishes M q1 q2 io*
$\wedge$ ($\forall$ *io'* . *distinguishes M q1 q2 io'* $\longrightarrow$ *length io* $\leq$ *length io'*))

lemma *minimally-distinguishes-ex* :
assumes *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *LS M q1* $\neq$ *LS M q2*
obtains *v* **where** *minimally-distinguishes M q1 q2 v*
proof −
let *?vs* = {*v* . *distinguishes M q1 q2 v*}
define *vMin* **where** *vMin*: *vMin* = *arg-min length* (λ*v* . *v* $\in$ *?vs*)

obtain *v'* **where** *distinguishes M q1 q2 v'*
using *assms* **unfolding** *distinguishes-def* **by** *blast*
then have *vMin* $\in$ *?vs* $\wedge$ ($\forall$ *v''* . *distinguishes M q1 q2 v''* $\longrightarrow$ *length vMin* $\leq$ *length v''*)
unfolding *vMin* **using** *arg-min-nat-lemma*[*of* λ*v* . *distinguishes M q1 q2 v v'* *length*]
by *simp*
then show *?thesis*
using *that*[*of vMin*] **unfolding** *minimally-distinguishes-def* **by** *blast*
qed

lemma *distinguish-prepend* :
assumes *observable M*
and *distinguishes M* (*FSM.after M q1 io*) (*FSM.after M q2 io*) *w*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *io* $\in$ *LS M q1*
and *io* $\in$ *LS M q2*
shows *distinguishes M q1 q2* (*io*@*w*)
proof −

have $(io@w \in LS\ M\ q1) = (w \in LS\ M\ (after\ M\ q1\ io))$
using *assms*(*1*,*3*,*5*)
by (*metis after-language-iff*)
moreover have $(io@w \in LS\ M\ q2) = (w \in LS\ M\ (after\ M\ q2\ io))$
using *assms*(*1*,*4*,*6*)
by (*metis after-language-iff*)
ultimately show *?thesis*
using *assms*(*2*) **unfolding** *distinguishes-def* **by** *blast*
qed

lemma *distinguish-prepend-initial* :
assumes *observable M*
and *distinguishes M* (*after-initial M* (*io1@io*)) (*after-initial M* (*io2@io*)) *w*
and $io1@io \in L\ M$
and $io2@io \in L\ M$
shows *distinguishes M* (*after-initial M io1*) (*after-initial M io2*) (*io@w*)
proof −
have *f1*: $\forall\, ps\ psa\ f\ a.\ (ps::('b \times 'c)\ list)\ @\ psa \notin LS\ f\ (a::'a) \vee ps \in LS\ f\ a$
by (*meson language-prefix*)
then have *f2*: $io1 \in L\ M$
by (*meson assms*(*3*))
have *f3*: $io2 \in L\ M$
using *f1* **by** (*metis assms*(*4*))
have $io1 \in L\ M$
using *f1* **by** (*metis assms*(*3*))
then show *?thesis*
by (*metis after-is-state after-language-iff after-split assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *distinguish-prepend f3*)
qed

lemma *minimally-distinguishes-no-prefix* :
assumes *observable M*
and $u@w \in L\ M$
and $v@w \in L\ M$
and *minimally-distinguishes M* (*after-initial M u*) (*after-initial M v*) $(w@w'@w'')$
and $w' \neq []$
shows $\neg$*distinguishes M* (*after-initial M* (*u@w*)) (*after-initial M* (*v@w*)) w''
proof
assume *distinguishes M* (*after-initial M* (*u @ w*)) (*after-initial M* (*v @ w*)) w''
then have *distinguishes M* (*after-initial M u*) (*after-initial M v*) $(w@w'')$
using *assms*(*1*−*3*) *distinguish-prepend-initial* **by** *blast*
moreover have $length\ (w@w'') < length\ (w@w'@w'')$
using *assms*(*5*) **by** *auto*
ultimately show *False*
using *assms*(*4*) **unfolding** *minimally-distinguishes-def*
using *leD* **by** *blast*
qed

lemma *minimally-distinguishes-after-append* :
assumes *observable M*
and *minimal M*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *minimally-distinguishes M q1 q2* $(w@w')$
and $w' \neq []$
shows *minimally-distinguishes M* (*after M q1 w*) (*after M q2 w*) w'
proof −
have $\neg$ *distinguishes M q1 q2 w*
using *assms*(*5*,*6*)
by (*metis add.right-neutral add-le-cancel-left length-append length-greater-0-conv linorder-not-le minimally-distinguishes-def*)
then have $w \in$ *LS M q1* = ($w \in$ *LS M q2*)
unfolding *distinguishes-def*
by *blast*
moreover have $(w@w') \in$ *LS M q1* $\cup$ *LS M q2*
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
ultimately have $w \in$ *LS M q1* **and** $w \in$ *LS M q2*
by (*meson Un-iff language-prefix*)+

have $(w@w') \in$ *LS M q1* = ($w' \in$ *LS M* (*after M q1 w*))
by (*meson* ‹$w \in$ *LS M q1*› *after-language-iff assms*(*1*))
moreover have $(w@w') \in$ *LS M q2* = ($w' \in$ *LS M* (*after M q2 w*))
by (*meson* ‹$w \in$ *LS M q2*› *after-language-iff assms*(*1*))
ultimately have *distinguishes M* (*after M q1 w*) (*after M q2 w*) w'
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
moreover have $\bigwedge w''$. *distinguishes M* (*after M q1 w*) (*after M q2 w*) $w'' \Longrightarrow$ *length* $w' \leq$ *length* w''
proof −
fix w'' **assume** *distinguishes M* (*after M q1 w*) (*after M q2 w*) w''
then have *distinguishes M q1 q2* $(w@w'')$
by (*metis* ‹$w \in$ *LS M q1*› ‹$w \in$ *LS M q2*› *assms*(*1*) *assms*(*3*) *assms*(*4*) *distinguish-prepend*)
then have *length* $(w@w') \leq$ *length* $(w@w'')$
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
then show *length* $w' \leq$ *length* w''
by *auto*
qed
ultimately show *?thesis*
unfolding *minimally-distinguishes-def distinguishes-def*
by *blast*
qed

lemma *minimally-distinguishes-after-append-initial* :
assumes *observable M*
and *minimal M*
and $u \in L\ M$
and $v \in L\ M$
and *minimally-distinguishes M* (*after-initial M u*) (*after-initial M v*) ($w@w'$)
and $w' \neq []$
shows *minimally-distinguishes M* (*after-initial M* ($u@w$)) (*after-initial M* ($v@w$)) w'
proof −

have ¬ *distinguishes M* (*after-initial M u*) (*after-initial M v*) *w*
using *assms*(*5*,*6*)
by (*metis add.right-neutral add-le-cancel-left length-append length-greater-0-conv linorder-not-le minimally-distinguishes-def*)
then have $w \in LS\ M$ (*after-initial M u*) = ($w \in LS\ M$ (*after-initial M v*))
unfolding *distinguishes-def*
by *blast*
moreover have ($w@w'$) $\in LS\ M$ (*after-initial M u*) $\cup$ $LS\ M$ (*after-initial M v*)
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
ultimately have $w \in LS\ M$ (*after-initial M u*) **and** $w \in LS\ M$ (*after-initial M v*)
by (*meson Un-iff language-prefix*)+

have ($w@w'$) $\in LS\ M$ (*after-initial M u*) = ($w' \in LS\ M$ (*after-initial M* ($u@w$)))
by (*meson* ‹$w \in LS\ M$ (*after-initial M u*)› *after-language-append-iff after-language-iff assms*(*1*) *assms*(*3*))
moreover have ($w@w'$) $\in LS\ M$ (*after-initial M v*) = ($w' \in LS\ M$ (*after-initial M* ($v@w$)))
by (*meson* ‹$w \in LS\ M$ (*after-initial M v*)› *after-language-append-iff after-language-iff assms*(*1*) *assms*(*4*))
ultimately have *distinguishes M* (*after-initial M* ($u@w$)) (*after-initial M* ($v@w$)) w'
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
moreover have $\bigwedge w''$. *distinguishes M* (*after-initial M* ($u@w$)) (*after-initial M* ($v@w$)) $w'' \Longrightarrow$ *length* $w' \leq$ *length* w''
proof −
fix w'' **assume** *distinguishes M* (*after-initial M* ($u@w$)) (*after-initial M* ($v@w$)) w''
then have *distinguishes M* (*after-initial M u*) (*after-initial M v*) ($w@w''$)
by (*meson* ‹$w \in LS\ M$ (*after-initial M u*)› ‹$w \in LS\ M$ (*after-initial M v*)› *after-language-iff assms*(*1*) *assms*(*3*) *assms*(*4*) *distinguish-prepend-initial*)
then have *length* ($w@w'$) $\leq$ *length* ($w@w''$)
using *assms*(*5*) **unfolding** *minimally-distinguishes-def distinguishes-def*
by *blast*
then show *length* $w' \leq$ *length* w''
by *auto*

```
  qed
  ultimately show ?thesis
    unfolding minimally-distinguishes-def distinguishes-def
    by blast
qed


lemma minimally-distinguishes-proper-prefixes-card :
  assumes observable M
  and     minimal M
  and     q1 ∈ states M
  and     q2 ∈ states M
  and     minimally-distinguishes M q1 q2 w
  and     S ⊆ states M
shows card {w' . w' ∈ set (prefixes w) ∧ w' ≠ w ∧ after M q1 w' ∈ S ∧ after M
q2 w' ∈ S} ≤ card S − 1
(is ?P S)
proof −

  define k where k = card S
  then show ?thesis
    using assms(6)
  proof (induction k arbitrary: S rule: less-induct)
    case (less k)

    then have finite S
      by (metis fsm-states-finite rev-finite-subset)

    show ?case proof (cases k)
      case 0
      then have S = {}
        using less.prems ‹finite S› by auto
      then show ?thesis
        by fastforce
    next
      case (Suc k')

      show ?thesis proof (cases {w' . w' ∈ set (prefixes w) ∧ w' ≠ w ∧ after M
q1 w' ∈ S ∧ after M q2 w' ∈ S} = {})
        case True
        then show ?thesis
          by (metis bot.extremum dual-order.eq-iff obtain-subset-with-card-n)
      next
        case False

        define wk where wk: wk = arg-max length (λwk . wk ∈ {w' . w' ∈ set
(prefixes w) ∧ w' ≠ w ∧ after M q1 w' ∈ S ∧ after M q2 w' ∈ S})
```

obtain wk' **where** $*$:$wk' \in \{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\}$
using *False* **by** *blast*
have *finite* $\{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\}$
by (*metis* (*no-types*) *Collect-mem-eq List.finite-set finite-Collect-conjI*)
then have $wk \in \{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\}$
and $\bigwedge wk' . wk' \in \{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\} \Longrightarrow length\ wk' \leq length\ wk$
using *False* **unfolding** *wk*
using *arg-max-nat-lemma*[*of* $(\lambda wk . wk \in \{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\})$, *OF* $*$]
by (*meson finite-maxlen*)+

then have $wk \in set\ (prefixes\ w)$ **and** $wk \neq w$ **and** $after\ M\ q1\ wk \in S$ **and** $after\ M\ q2\ wk \in S$
by *blast*+

obtain *wk-suffix* **where** $w = wk@wk\text{-}suffix$ **and** $wk\text{-}suffix \neq []$
using ‹$wk \in set\ (prefixes\ w)$›
using *prefixes-set-ob* ‹$wk \neq w$›
by *blast*

have $distinguishes\ M\ (after\ M\ q1\ []) (after\ M\ q2\ [])\ w$
using ‹*minimally-distinguishes M q1 q2 w*›
by (*metis after.simps*(*1*) *minimally-distinguishes-def*)

have $minimally\text{-}distinguishes\ M\ (after\ M\ q1\ wk)\ (after\ M\ q2\ wk)\ wk\text{-}suffix$
using ‹$minimally\text{-}distinguishes\ M\ q1\ q2\ w$› ‹$wk\text{-}suffix \neq []$›
unfolding ‹$w = wk@wk\text{-}suffix$›
using *minimally-distinguishes-after-append*[*OF assms*(*1*,*2*,*3*,*4*), *of wk wk-suffix*]
by *blast*
then have $distinguishes\ M\ (after\ M\ q1\ wk)\ (after\ M\ q2\ wk)\ wk\text{-}suffix$
unfolding *minimally-distinguishes-def*
by *auto*
then have $wk\text{-}suffix \in LS\ M\ (after\ M\ q1\ wk) = (wk\text{-}suffix \notin LS\ M\ (after\ M\ q2\ wk))$
unfolding *distinguishes-def* **by** *blast*

define *S1* **where** *S1*: $S1 = Set.filter\ (\lambda q . wk\text{-}suffix \in LS\ M\ q)\ S$
define *S2* **where** *S2*: $S2 = Set.filter\ (\lambda q . wk\text{-}suffix \notin LS\ M\ q)\ S$

have $S = S1 \cup S2$
unfolding *S1 S2* **by** *auto*
moreover have $S1 \cap S2 = \{\}$

unfolding *S1 S2* **by** *auto*
ultimately have *card S* = *card S1* + *card S2*
using ‹*finite S*› *card-Un-disjoint* **by** *blast*

have *S1* ⊆ *states M* **and** *S2* ⊆ *states M*
using ‹*S* = *S1* ∪ *S2*› *less.prems*(*2*) **by** *blast*+

have *S1* ≠ {} **and** *S2* ≠ {}
using ‹*wk-suffix* ∈ *LS M* (*after M q1 wk*) = (*wk-suffix* ∉ *LS M* (*after M q2 wk*))› ‹*after M q1 wk* ∈ *S*› ‹*after M q2 wk* ∈ *S*›
unfolding *S1 S2* **by** *auto*
then have *card S1* > *0* **and** *card S2* > *0*
using ‹*S* = *S1* ∪ *S2*› ‹*finite S*›
by (*meson card-0-eq finite-Un neq0-conv*)+
then have *card S1* < *k* **and** *card S2* < *k*
using ‹*card S* = *card S1* + *card S2*› **unfolding** *less.prems*
by *auto*

define *W* **where** *W*: *W* = (λ *S1 S2* . {*w'* . *w'* ∈ *set* (*prefixes w*) ∧ *w'* ≠ *w* ∧ *after M q1 w'* ∈ *S1* ∧ *after M q2 w'* ∈ *S2*})
then have ⋀ *S' S''* . *W S' S''* ⊆ *set* (*prefixes w*)
by *auto*
then have *W-finite*: ⋀ *S' S''* . *finite* (*W S' S''*)
using *List.finite-set*[*of prefixes w*]
by (*meson finite-subset*)

have ⋀ *w'* . *w'* ∈ *W S S* ⟹ *w'* ≠ *wk* ⟹ *after M q1 w'* ∈ *S1* = (*after M q2 w'* ∈ *S1*)
proof −
fix *w'* **assume** ∗:*w'* ∈ *W S S* **and** *w'* ≠ *wk*
then have *w'* ∈ *set* (*prefixes w*) **and** *w'* ≠ *w* **and** *after M q1 w'* ∈ *S* **and** *after M q2 w'* ∈ *S*
unfolding *W*
by *blast*+

then have *w'* ∈ *LS M q1*
by (*metis IntE UnCI UnE append-self-conv assms*(*5*) *distinguishes-def language-prefix leD length-append length-greater-0-conv less-add-same-cancel1 minimally-distinguishes-def prefixes-set-ob*)
have *w'* ∈ *LS M q2*
by (*metis IntE UnCI* ‹*w'* ∈ *LS M q1*› ‹*w'* ∈ *set* (*prefixes w*)› ‹*w'* ≠ *w*› *append-Nil2 assms*(*5*) *distinguishes-def leD length-append length-greater-0-conv less-add-same-cancel1 minimally-distinguishes-def prefixes-set-ob*)

have *length w'* < *length wk*
using ‹*w'* ≠ *wk*› ∗
‹⋀ *wk'* . *wk'* ∈ {*w'* . *w'* ∈ *set* (*prefixes w*) ∧ *w'* ≠ *w* ∧ *after M q1 w'* ∈ *S* ∧ *after M q2 w'* ∈ *S*} ⟹ *length wk'* ≤ *length wk*›

unfolding *W*
by (*metis* (*no-types*, *lifting*) ‹*w* = *wk* @ *wk-suffix*› ‹*w′* ∈ *set* (*prefixes w*)› *append-eq-append-conv le-neq-implies-less prefixes-set-ob*)

show *after M q1 w′* ∈ *S1* = (*after M q2 w′* ∈ *S1*)
proof (*rule ccontr*)
assume (*after M q1 w′* ∈ *S1*) ≠ (*after M q2 w′* ∈ *S1*)
then have (*after M q1 w′* ∈ *S1* ∧ (*after M q2 w′* ∈ *S2*)) ∨ (*after M q1 w′* ∈ *S2* ∧ (*after M q2 w′* ∈ *S1*))
using ‹*after M q1 w′* ∈ *S*› ‹*after M q2 w′* ∈ *S*›
unfolding ‹*S* = *S1* ∪ *S2*›
by *blast*
then have *wk-suffix* ∈ *LS M* (*after M q1 w′*) = (*wk-suffix* ∉ *LS M* (*after M q2 w′*))
using *S1 S2* **by** *auto*
then have *distinguishes M* (*after M q1 w′*) (*after M q2 w′*) *wk-suffix*
unfolding *distinguishes-def* **by** *blast*
then have *distinguishes M q1 q2* (*w′*@*wk-suffix*)
using *distinguish-prepend*[*OF assms*(*1*) - ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*w′* ∈ *LS M q1*› ‹*w′* ∈ *LS M q2*›]
by *blast*
moreover have *length* (*w′*@*wk-suffix*) < *length* (*wk*@*wk-suffix*)
using ‹*length w′* < *length wk*›
by *auto*
ultimately show *False*
using ‹*minimally-distinguishes M q1 q2 w*›
unfolding ‹*w* = *wk*@*wk-suffix*› *minimally-distinguishes-def*
by *auto*
qed
qed

have ⋀ *x* . *x* ∈ *W S1 S2* ∪ *W S2 S1* ⟹ *x* = *wk*
proof −
fix *x* **assume** *x* ∈ *W S1 S2* ∪ *W S2 S1*
then have *x* ∈ *W S S*
unfolding *W* ‹*S* = *S1* ∪ *S2*› **by** *blast*
show *x* = *wk*
using ‹*x* ∈ *W S1 S2* ∪ *W S2 S1*›
using ‹⋀ *w′* . *w′* ∈ *W S S* ⟹ *w′* ≠ *wk* ⟹ *after M q1 w′* ∈ *S1* = (*after M q2 w′* ∈ *S1*)›[*OF* ‹*x* ∈ *W S S*›]
unfolding *W*
using ‹*S1* ∩ *S2* = {}›
by *blast*
qed
moreover have *wk* ∈ *W S1 S2* ∪ *W S2 S1*
unfolding *W*

using ‹$wk \in \{w' . w' \in set\ (prefixes\ w) \wedge w' \neq w \wedge after\ M\ q1\ w' \in S \wedge after\ M\ q2\ w' \in S\}$›
‹$wk\text{-}suffix \in LS\ M\ (after\ M\ q1\ wk) = (wk\text{-}suffix \notin LS\ M\ (after\ M\ q2\ wk))$›
using $S1$ **by** *clarsimp* (*auto simp add*: ‹$S = S1 \cup S2$›)
ultimately have $W\ S1\ S2 \cup W\ S2\ S1 = \{wk\}$
by *blast*

have $W\ S\ S = (W\ S1\ S1 \cup W\ S2\ S2 \cup (W\ S1\ S2 \cup W\ S2\ S1))$
unfolding W ‹$S = S1 \cup S2$› **by** *blast*
moreover have $W\ S1\ S1 \cap W\ S2\ S2 = \{\}$
using ‹$S1 \cap S2 = \{\}$› **unfolding** W
by *blast*
moreover have $W\ S1\ S1 \cap (W\ S1\ S2 \cup W\ S2\ S1) = \{\}$
unfolding W
using ‹$S1 \cap S2 = \{\}$›
by *blast*
moreover have $W\ S2\ S2 \cap (W\ S1\ S2 \cup W\ S2\ S1) = \{\}$
unfolding W
using ‹$S1 \cap S2 = \{\}$›
by *blast*
moreover have *finite* $(W\ S1\ S1)$ **and** *finite* $(W\ S2\ S2)$ **and** *finite* $\{wk\}$
using *W-finite* **by** *auto*
ultimately have $card\ (W\ S\ S) = card\ (W\ S1\ S1) + card\ (W\ S2\ S2) + 1$
unfolding ‹$W\ S1\ S2 \cup W\ S2\ S1 = \{wk\}$›
by (*metis card-Un-disjoint finite-UnI inf-sup-distrib2 is-singletonI is-singleton-altdef sup-idem*)
moreover have $card\ (W\ S1\ S1) \leq card\ S1 - 1$
using *less.IH*[*OF* ‹$card\ S1 < k$› - ‹$S1 \subseteq states\ M$›]
unfolding W **by** *blast*
moreover have $card\ (W\ S2\ S2) \leq card\ S2 - 1$
using *less.IH*[*OF* ‹$card\ S2 < k$› - ‹$S2 \subseteq states\ M$›]
unfolding W **by** *blast*
ultimately have $card\ (W\ S\ S) \leq card\ S - 1$
using ‹$card\ S = card\ S1 + card\ S2$›
using ‹$card\ S1 < k$› ‹$card\ S2 < k$› *less.prems*(*1*) **by** *linarith*
then show *?thesis*
unfolding W **.**
qed
qed
qed
qed

lemma *minimally-distinguishes-proper-prefix-in-language* :
assumes *minimally-distinguishes* $M\ q1\ q2\ io$
and $io' \in set\ (prefixes\ io)$
and $io' \neq io$
shows $io' \in LS\ M\ q1 \cap LS\ M\ q2$

proof −
  have *io* ∈ *LS M q1* ∨ *io* ∈ *LS M q2*
    using *assms*(*1*) **unfolding** *minimally-distinguishes-def distinguishes-def* **by** *blast*
  then have *io′* ∈ *LS M q1* ∨ *io′* ∈ *LS M q2*
    by (*metis assms*(*2*) *prefixes-set-ob language-prefix*)

  have *length io′* < *length io*
    using *assms*(*2*,*3*) **unfolding** *prefixes-set* **by** *auto*
  then have *io′* ∈ *LS M q1* ⟷ *io′* ∈ *LS M q2*
    using *assms*(*1*) **unfolding** *minimally-distinguishes-def distinguishes-def*
    by (*metis Int-iff Un-Int-eq*(*1*) *Un-Int-eq*(*2*) *leD*)
  then show *?thesis*
    using ‹*io′* ∈ *LS M q1* ∨ *io′* ∈ *LS M q2*›
    by *blast*
qed

lemma *distinguishes-not-Nil*:
  assumes *distinguishes M q1 q2 io*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
shows *io* ≠ []
  using *assms* **unfolding** *distinguishes-def* **by** *auto*

fun *does-distinguish* :: (*′a*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ (*′b* × *′c*) *list* ⇒ *bool* **where**
  does-distinguish M q1 q2 io = (*is-in-language M q1 io* ≠ *is-in-language M q2 io*)

lemma *does-distinguish-correctness* :
  assumes *observable M*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
shows *does-distinguish M q1 q2 io* = *distinguishes M q1 q2 io*
  unfolding *does-distinguish.simps*
    is-in-language-iff[*OF assms*(*1*,*2*)]
    is-in-language-iff[*OF assms*(*1*,*3*)]
    distinguishes-def
  by *blast*

lemma *h-obs-distinguishes* :
  assumes *observable M*
  and *h-obs M q1 x y* = *Some q1′*
  and *h-obs M q2 x y* = *None*
shows *distinguishes M q1 q2* [(*x*,*y*)]
  using *assms*(*2*,*3*) *LS-single-transition*[*of x y M*] **unfolding** *distinguishes-def h-obs-Some*[*OF assms*(*1*)] *h-obs-None*[*OF assms*(*1*)]
  by (*metis Int-iff UnI1* ‹⋀*y x q*. (*h-obs M q x y* = *None*) = (∄ *q′*. (*q*, *x*, *y*, *q′*) ∈ *FSM.transitions M*)› *assms*(*1*) *assms*(*2*) *fst-conv h-obs-language-iff option.distinct*(*1*) *snd-conv*)

lemma *distinguishes-sym* :
assumes *distinguishes M q1 q2 io*
shows *distinguishes M q2 q1 io*
using *assms* **unfolding** *distinguishes-def* **by** *blast*

lemma *distinguishes-after-prepend* :
assumes *observable M*
and *h-obs M q1 x y* $\neq$ *None*
and *h-obs M q2 x y* $\neq$ *None*
and *distinguishes M* (*FSM.after M q1* [(*x*,*y*)]) (*FSM.after M q2* [(*x*,*y*)]) γ
shows *distinguishes M q1 q2* ((*x*,*y*)#γ)
proof −
have [(*x*,*y*)] $\in$ *LS M q1*
using *assms*(*2*) *h-obs-language-single-transition-iff*[*OF assms*(*1*)] **by** *auto*

have [(*x*,*y*)] $\in$ *LS M q2*
using *assms*(*3*) *h-obs-language-single-transition-iff*[*OF assms*(*1*)] **by** *auto*

show *?thesis*
using *after-language-iff*[*OF assms*(*1*) ‹[(*x*,*y*)] $\in$ *LS M q1*›, *of* γ]
using *after-language-iff*[*OF assms*(*1*) ‹[(*x*,*y*)] $\in$ *LS M q2*›, *of* γ]
using *assms*(*4*)
unfolding *distinguishes-def*
by *simp*
qed

lemma *distinguishes-after-initial-prepend* :
assumes *observable M*
and *io1* $\in$ *L M*
and *io2* $\in$ *L M*
and *h-obs M* (*after-initial M io1*) *x y* $\neq$ *None*
and *h-obs M* (*after-initial M io2*) *x y* $\neq$ *None*
and *distinguishes M* (*after-initial M* (*io1*@[(*x*,*y*)])) (*after-initial M* (*io2*@[(*x*,*y*)]))
γ
shows *distinguishes M* (*after-initial M io1*) (*after-initial M io2*) ((*x*,*y*)#γ)
by (*metis after-split assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) *distinguishes-after-prepend h-obs-language-append*)

4.25 Extending FSMs by single elements

lemma *fsm-from-list-simps*[*simp*] :
initial (*fsm-from-list q ts*) = (*case ts of* [] $\Rightarrow$ *q* | (*t*#*ts*) $\Rightarrow$ *t-source t*)
states (*fsm-from-list q ts*) = (*case ts of* [] $\Rightarrow$ {*q*} | (*t*#*ts*′) $\Rightarrow$ ((*image t-source* (*set ts*)) $\cup$ (*image t-target* (*set ts*))))
inputs (*fsm-from-list q ts*) = *image t-input* (*set ts*)
outputs (*fsm-from-list q ts*) = *image t-output* (*set ts*)
transitions (*fsm-from-list q ts*) = *set ts*
by (*cases ts*; *transfer*; *simp*)+

lift-definition *add-transition* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *transition* $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.add-transition*
by *simp*

lemma *add-transition-simps*[*simp*]:
assumes *t-source t* $\in$ *states M* **and** *t-input t* $\in$ *inputs M* **and** *t-output t* $\in$ *outputs M* **and** *t-target t* $\in$ *states M*
shows
initial (*add-transition M t*) = *initial M*
inputs (*add-transition M t*) = *inputs M*
outputs (*add-transition M t*) = *outputs M*
transitions (*add-transition M t*) = *insert t* (*transitions M*)
states (*add-transition M t*) = *states M* **using** *assms* **by** (*transfer*; *simp*)+

lift-definition *add-state* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.add-state*
by *simp*

lemma *add-state-simps*[*simp*]:
initial (*add-state M q*) = *initial M*
inputs (*add-state M q*) = *inputs M*
outputs (*add-state M q*) = *outputs M*
transitions (*add-state M q*) = *transitions M*
states (*add-state M q*) = *insert q* (*states M*) **by** (*transfer*; *simp*)+

lift-definition *add-input* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'b$ $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.add-input*
by *simp*

lemma *add-input-simps*[*simp*]:
initial (*add-input M x*) = *initial M*
inputs (*add-input M x*) = *insert x* (*inputs M*)
outputs (*add-input M x*) = *outputs M*
transitions (*add-input M x*) = *transitions M*
states (*add-input M x*) = *states M* **by** (*transfer*; *simp*)+

lift-definition *add-output* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'c$ $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.add-output*
by *simp*

lemma *add-output-simps*[*simp*]:
initial (*add-output M y*) = *initial M*
inputs (*add-output M y*) = *inputs M*
outputs (*add-output M y*) = *insert y* (*outputs M*)
transitions (*add-output M y*) = *transitions M*
states (*add-output M y*) = *states M* **by** (*transfer*; *simp*)+

lift-definition *add-transition-with-components* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *transition* $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.add-transition-with-components*
by *simp*

lemma *add-transition-with-components-simps*[*simp*]:
initial (*add-transition-with-components M t*) = *initial M*
inputs (*add-transition-with-components M t*) = *insert* (*t-input t*) (*inputs M*)
outputs (*add-transition-with-components M t*) = *insert* (*t-output t*) (*outputs M*)
transitions (*add-transition-with-components M t*) = *insert t* (*transitions M*)
states (*add-transition-with-components M t*) = *insert* (*t-target t*) (*insert* (*t-source t*) (*states M*))
by (*transfer*; *simp*)+

4.26 Renaming Elements

lift-definition *rename-states* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a* ⇒ '*d*) ⇒ ('*d*,'*b*,'*c*) *fsm* **is** *FSM-Impl.rename-states*
by *simp*

lemma *rename-states-simps*[*simp*]:
initial (*rename-states M f*) = *f* (*initial M*)
states (*rename-states M f*) = *f* ‘ (*states M*)
inputs (*rename-states M f*) = *inputs M*
outputs (*rename-states M f*) = *outputs M*
transitions (*rename-states M f*) = (λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions M*
by (*transfer*; *simp*)+

lemma *rename-states-isomorphism-language-state* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
and *q* ∈ *states M*
shows *LS* (*rename-states M f*) (*f q*) = *LS M q*
proof −

have ∗: *bij-betw f* (*FSM.states M*) (*FSM.states* (*FSM.rename-states M f*))
using *assms rename-states-simps* **by** *auto*

have ∗∗: *f* (*initial M*) = *initial* (*rename-states M f*)
using *rename-states-simps* **by** *auto*

have ∗∗∗: (⋀*q x y q*′.
q ∈ *states M* ⟹
q′ ∈ *states M* ⟹ ((*q*, *x*, *y*, *q*′) ∈ *transitions M*) = ((*f q*, *x*, *y*, *f q*′) ∈ *transitions* (*rename-states M f*)))
proof
fix *q x y q*′ **assume** *q* ∈ *states M* **and** *q*′ ∈ *states M*

show (*q*, *x*, *y*, *q*′) ∈ *transitions M* ⟹ (*f q*, *x*, *y*, *f q*′) ∈ *transitions* (*rename-states M f*)
unfolding *assms rename-states-simps* **by** *force*

show (*f q*, *x*, *y*, *f q*′) ∈ *transitions* (*rename-states M f*) ⟹ (*q*, *x*, *y*, *q*′) ∈

transitions M
proof −
assume (*f q*, *x*, *y*, *f q′*) ∈ *transitions* (*rename-states M f*)
then obtain *t* **where** (*f q*, *x*, *y*, *f q′*) = (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))
and *t* ∈ *transitions M*
unfolding *assms rename-states-simps*
by *blast*
then have *t-source t* ∈ *states M* **and** *t-target t* ∈ *states M* **and** *f* (*t-source t*) = *f q* **and** *f* (*t-target t*) = *f q′* **and** *t-input t* = *x* **and** *t-output t* = *y*
by *auto*

have *f q* ∈ *states* (*rename-states M f*) **and** *f q′* ∈ *states* (*rename-states M f*)
using ‹(*f q*, *x*, *y*, *f q′*) ∈ *transitions* (*rename-states M f*)›
by *auto*

have *t-source t* = *q*
using ‹*f* (*t-source t*) = *f q*› ‹*q* ∈ *states M*› ‹*t-source t* ∈ *states M*›
using *assms* **unfolding** *bij-betw-def inj-on-def*
by *blast*
moreover have *t-target t* = *q′*
using ‹*f* (*t-target t*) = *f q′*› ‹*q′* ∈ *states M*› ‹*t-target t* ∈ *states M*›
using *assms* **unfolding** *bij-betw-def inj-on-def*
by *blast*
ultimately show (*q*, *x*, *y*, *q′*) ∈ *transitions M*
using ‹*t-input t* = *x*› ‹*t-output t* = *y*› ‹*t* ∈ *transitions M*›
by *auto*
qed
qed

show *?thesis*
using *language-equivalence-from-isomorphism*[*OF* ∗ ∗∗ ∗∗∗ *assms*(*2*)]
by *blast*
qed

lemma *rename-states-isomorphism-language* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
shows *L* (*rename-states M f*) = *L M*
using *rename-states-isomorphism-language-state*[*OF assms fsm-initial*]
unfolding *rename-states-simps* **.**

lemma *rename-states-observable* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
and *observable M*
shows *observable* (*rename-states M f*)
proof −
have ⋀ *q1 x y q1′ q1′′* . (*q1*,*x*,*y*,*q1′*) ∈ *transitions* (*rename-states M f*) ⟹ (*q1*,*x*,*y*,*q1′′*) ∈ *transitions* (*rename-states M f*) ⟹ *q1′* = *q1′′*

proof −
fix *q1 x y q1′ q1″*
assume (*q1*,*x*,*y*,*q1′*) ∈ *transitions* (*rename-states M f*) **and** (*q1*,*x*,*y*,*q1″*) ∈ *transitions* (*rename-states M f*)
then obtain *t′ t″* **where** *t′* ∈ *transitions M*
and *t″* ∈ *transitions M*
and (*f* (*t-source t′*), *t-input t′*, *t-output t′*, *f* (*t-target t′*)) = (*q1*,*x*,*y*,*q1′*)
and (*f* (*t-source t″*), *t-input t″*, *t-output t″*, *f* (*t-target t″*)) = (*q1*,*x*,*y*,*q1″*)
unfolding *rename-states-simps*
by *force*

then have *f* (*t-source t′*) = *f* (*t-source t″*)
by *auto*
moreover have *t-source t′* ∈ *states M* **and** *t-source t″* ∈ *states M*
using ‹*t′* ∈ *transitions M*› ‹*t″* ∈ *transitions M*›
by *auto*
ultimately have *t-source t′* = *t-source t″*
using *assms*(*1*)
unfolding *bij-betw-def inj-on-def* **by** *blast*
then have *t-target t′* = *t-target t″*
using *assms*(*2*) **unfolding** *observable.simps*
by (*metis Pair-inject* ‹(*f* (*t-source t″*), *t-input t″*, *t-output t″*, *f* (*t-target t″*)) = (*q1*, *x*, *y*, *q1″*)› ‹(*f* (*t-source t′*), *t-input t′*, *t-output t′*, *f* (*t-target t′*)) = (*q1*, *x*, *y*, *q1′*)› ‹*t′* ∈ *FSM.transitions M*› ‹*t″* ∈ *FSM.transitions M*›)
then show *q1′* = *q1″*
using ‹(*f* (*t-source t″*), *t-input t″*, *t-output t″*, *f* (*t-target t″*)) = (*q1*, *x*, *y*, *q1″*)› ‹(*f* (*t-source t′*), *t-input t′*, *t-output t′*, *f* (*t-target t′*)) = (*q1*, *x*, *y*, *q1′*)› **by** *auto*
qed
then show *?thesis*
unfolding *observable-alt-def* **by** *blast*
qed

lemma *rename-states-minimal* :
assumes *bij-betw f* (*states M*) (*f* ‘ *states M*)
and *minimal M*
shows *minimal* (*rename-states M f*)
proof −
have ⋀ *q q′* . *q* ∈ *f* ‘ *FSM.states M* ⟹ *q′* ∈ *f* ‘ *FSM.states M* ⟹ *q* ≠ *q′* ⟹ *LS* (*rename-states M f*) *q* ≠ *LS* (*rename-states M f*) *q′*
proof −
fix *q q′* **assume** *q* ∈ *f* ‘ *FSM.states M* **and** *q′* ∈ *f* ‘ *FSM.states M* **and** *q* ≠ *q′*

then obtain *fq fq′* **where** *fq* ∈ *states M* **and** *fq′* ∈ *states M* **and** *q* = *f fq* **and** *q′* = *f fq′*
by *auto*

then have $fq \neq fq'$
using ‹$q \neq q'$› **by** *auto*
then have $LS\ M\ fq \neq LS\ M\ fq'$
by (*meson* ‹$fq \in FSM.states\ M$› ‹$fq' \in FSM.states\ M$› *assms*(*2*) *minimal.elims*(*2*))
then show LS (*rename-states* $M\ f$) $q \neq LS$ (*rename-states* $M\ f$) q'
using *rename-states-isomorphism-language-state*[*OF assms*(*1*)]
by (*simp add*: ‹$fq \in FSM.states\ M$› ‹$fq' \in FSM.states\ M$› ‹$q = f\ fq$› ‹$q' = f\ fq'$›)
qed
then show *?thesis*
by *auto*
qed

fun *index-states* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ (*nat*,'*b*,'*c*) *fsm* **where**
index-states M = *rename-states M* (*assign-indices* (*states M*))

lemma *assign-indices-bij-betw*: *bij-betw* (*assign-indices* (*FSM.states M*)) (*FSM.states M*) (*assign-indices* (*FSM.states M*) ‘ *FSM.states M*)
using *assign-indices-bij*[*OF fsm-states-finite*[*of M*]]
by (*simp add*: *bij-betw-def*)

lemma *index-states-language* :
L (*index-states M*) = *L M*
using *rename-states-isomorphism-language*[*of assign-indices* (*states M*) *M*, *OF assign-indices-bij-betw*]
by *auto*

lemma *index-states-observable* :
assumes *observable M*
shows *observable* (*index-states M*)
using *rename-states-observable*[*of assign-indices* (*states M*), *OF assign-indices-bij-betw assms*]
unfolding *index-states.simps* **.**

lemma *index-states-minimal* :
assumes *minimal M*
shows *minimal* (*index-states M*)
using *rename-states-minimal*[*of assign-indices* (*states M*), *OF assign-indices-bij-betw assms*]
unfolding *index-states.simps* **.**

fun *index-states-integer* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ (*integer*,'*b*,'*c*) *fsm* **where**
index-states-integer M = *rename-states M* (*integer-of-nat* ∘ *assign-indices* (*states M*))

lemma *assign-indices-integer-bij-betw*: *bij-betw* (*integer-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*) ((*integer-of-nat* ∘ *assign-indices* (*states M*)) ‘ *FSM.states M*)
proof −
  have ∗: *inj-on* (*assign-indices* (*FSM.states M*)) (*FSM.states M*)
    using *assign-indices-bij*[*OF fsm-states-finite*[*of M*]]
    unfolding *bij-betw-def*
    by *auto*
  then have *inj-on* (*integer-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*)
    unfolding *inj-on-def*
    by (*metis comp-apply nat-of-integer-integer-of-nat*)
  then show *?thesis*
    unfolding *bij-betw-def*
    by *auto*
qed

lemma *index-states-integer-language* :
  L (*index-states-integer M*) = *L M*
  using *rename-states-isomorphism-language*[*of integer-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-integer-bij-betw*]
  by *auto*

lemma *index-states-integer-observable* :
  assumes *observable M*
  shows *observable* (*index-states-integer M*)
  using *rename-states-observable*[*of integer-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-integer-bij-betw assms*]
  unfolding *index-states-integer.simps* **.**

lemma *index-states-integer-minimal* :
  assumes *minimal M*
  shows *minimal* (*index-states-integer M*)
  using *rename-states-minimal*[*of integer-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-integer-bij-betw assms*]
  unfolding *index-states-integer.simps* **.**

4.27 Canonical Separators

lift-definition *canonical-separator′* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ((′*a* × ′*a*),′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) + ′*a*,′*b*,′*c*) *fsm* **is** *FSM-Impl.canonical-separator′*
proof −
  fix *A* :: (′*a*,′*b*,′*c*) *fsm-impl*
  fix *B* :: (′*a* × ′*a*,′*b*,′*c*) *fsm-impl*
  fix *q1* :: ′*a*
  fix *q2* :: ′*a*
  assume *well-formed-fsm A* **and** *well-formed-fsm B*

  then have *p1a*: *fsm-impl.initial A* ∈ *fsm-impl.states A*

and *p2a*: *finite* (*fsm-impl.states A*)
and *p3a*: *finite* (*fsm-impl.inputs A*)
and *p4a*: *finite* (*fsm-impl.outputs A*)
and *p5a*: *finite* (*fsm-impl.transitions A*)
and *p6a*: (∀ *t*∈*fsm-impl.transitions A*.
t-source t ∈ *fsm-impl.states A* ∧
t-input t ∈ *fsm-impl.inputs A* ∧ *t-target t* ∈ *fsm-impl.states A* ∧
t-output t ∈ *fsm-impl.outputs A*)
and *p1b*: *fsm-impl.initial B* ∈ *fsm-impl.states B*
and *p2b*: *finite* (*fsm-impl.states B*)
and *p3b*: *finite* (*fsm-impl.inputs B*)
and *p4b*: *finite* (*fsm-impl.outputs B*)
and *p5b*: *finite* (*fsm-impl.transitions B*)
and *p6b*: (∀ *t*∈*fsm-impl.transitions B*.
t-source t ∈ *fsm-impl.states B* ∧
t-input t ∈ *fsm-impl.inputs B* ∧ *t-target t* ∈ *fsm-impl.states B* ∧
t-output t ∈ *fsm-impl.outputs B*)
by *simp+*

let *?P* = *FSM-Impl.canonical-separator′ A B q1 q2*

show *well-formed-fsm ?P* **proof** (*cases fsm-impl.initial B* = (*q1*,*q2*))
case *False*
then show *?thesis* **by** *auto*
next
case *True*

let *?f* = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*fsm-impl.transitions A*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}))

have ⋀ *qx* . (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*fsm-impl.transitions A*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {})) *qx* = (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ *fsm-impl.transitions A*}) *qx*
proof −
fix *qx*
show ⋀ *qx* . (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*fsm-impl.transitions A*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {})) *qx* = (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ *fsm-impl.transitions A*}) *qx*
unfolding *set-as-map-def* **by** (*cases* ∃ *z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ *fsm-impl.transitions A*; *auto*)
qed
moreover have ⋀ *qx* . (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ *fsm-impl.transitions A*}) *qx* = (λ *qx* . {*y* | *y* . ∃ *q′* . (*fst qx*, *snd qx*, *y*, *q′*) ∈ *fsm-impl.transitions A*}) *qx*
proof −
fix *qx*
show (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q′*). ((*q*, *x*), *y*)) ‘ *fsm-impl.transitions A*}) *qx* = (λ *qx* . {*y* | *y* . ∃ *q′* . (*fst qx*, *snd qx*, *y*, *q′*) ∈ *fsm-impl.transitions A*}) *qx*

by *force*
qed
ultimately have $*$: *?f* = (λ *qx* . {*y* | *y* . $\exists$ *q′* . (*fst qx*, *snd qx*, *y*, *q′*) $\in$ *fsm-impl.transitions A*})
by *blast*

let *?shifted-transitions′* = *shifted-transitions* (*fsm-impl.transitions B*)
let *?distinguishing-transitions-lr* = *distinguishing-transitions ?f q1 q2* (*fsm-impl.states B*) (*fsm-impl.inputs B*)
let *?ts* = *?shifted-transitions′* $\cup$ *?distinguishing-transitions-lr*

have *FSM-Impl.states ?P* = (*image Inl* (*FSM-Impl.states B*)) $\cup$ {*Inr q1*, *Inr q2*}
and *FSM-Impl.transitions ?P* = *?ts*
unfolding *FSM-Impl.canonical-separator′.simps Let-def True* **by** *simp+*

have *p2*: *finite* (*fsm-impl.states ?P*)
unfolding ‹*FSM-Impl.states ?P* = (*image Inl* (*FSM-Impl.states B*)) $\cup$ {*Inr q1*, *Inr q2*}› **using** *p2b* **by** *blast*

have *fsm-impl.initial ?P* = *Inl* (*q1*,*q2*) **by** *auto*
then have *p1*: *fsm-impl.initial ?P* $\in$ *fsm-impl.states ?P*
using *p1a p1b* **unfolding** *canonical-separator′.simps True* **by** *auto*
have *p3*: *finite* (*fsm-impl.inputs ?P*)
using *p3a p3b* **by** *auto*
have *p4*: *finite* (*fsm-impl.outputs ?P*)
using *p4a p4b* **by** *auto*

have *finite* (*fsm-impl.states B* $\times$ *fsm-impl.inputs B*)
using *p2b p3b* **by** *blast*
moreover have $**$: $\bigwedge$ *x q1* . *finite* ({*y* |*y*. $\exists$ *q′*. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q′*) $\in$ *fsm-impl.transitions A*})
proof −
fix *x q1*
have {*y* |*y*. $\exists$ *q′*. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q′*) $\in$ *fsm-impl.transitions A*} = {*t-output t* | *t* . *t* $\in$ *fsm-impl.transitions A* $\wedge$ *t-source t* = *q1* $\wedge$ *t-input t* = *x*}
by *auto*
then have {*y* |*y*. $\exists$ *q′*. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q′*) $\in$ *fsm-impl.transitions A*} $\subseteq$ *image t-output* (*fsm-impl.transitions A*)
unfolding *fst-conv snd-conv* **by** *blast*
moreover have *finite* (*image t-output* (*fsm-impl.transitions A*))
using *p5a* **by** *auto*
ultimately show *finite* ({*y* |*y*. $\exists$ *q′*. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q′*) $\in$ *fsm-impl.transitions A*})
by (*simp add*: *finite-subset*)
qed
ultimately have *finite ?distinguishing-transitions-lr*
unfolding $*$ *distinguishing-transitions-def* **by** *force*
moreover have *finite ?shifted-transitions′*

unfolding *shifted-transitions-def* **using** *p5b* **by** *auto*
ultimately have *finite ?ts* **by** *blast*
then have *p5*: *finite* (*fsm-impl.transitions ?P*)
by *simp*

have *fsm-impl.inputs ?P* = *fsm-impl.inputs A* ∪ *fsm-impl.inputs B*
using *True* **by** *auto*
have *fsm-impl.outputs ?P* = *fsm-impl.outputs A* ∪ *fsm-impl.outputs B*
using *True* **by** *auto*

have ⋀ *t* . *t* ∈ *?shifted-transitions′* ⟹ *t-source t* ∈ *fsm-impl.states ?P* ∧ *t-target t* ∈ *fsm-impl.states ?P*
unfolding ‹*FSM-Impl.states ?P* = (*image Inl* (*FSM-Impl.states B*)) ∪ {*Inr q1*, *Inr q2*}› *shifted-transitions-def*
using *p6b* **by** *force*
moreover have ⋀ *t* . *t* ∈ *?distinguishing-transitions-lr* ⟹ *t-source t* ∈ *fsm-impl.states ?P* ∧ *t-target t* ∈ *fsm-impl.states ?P*
unfolding ‹*FSM-Impl.states ?P* = (*image Inl* (*FSM-Impl.states B*)) ∪ {*Inr q1*, *Inr q2*}› *distinguishing-transitions-def* ∗ **by** *force*
ultimately have ⋀ *t* . *t* ∈ *?ts* ⟹ *t-source t* ∈ *fsm-impl.states ?P* ∧ *t-target t* ∈ *fsm-impl.states ?P*
by *blast*
moreover have ⋀ *t* . *t* ∈ *?shifted-transitions′* ⟹ *t-input t* ∈ *fsm-impl.inputs ?P* ∧ *t-output t* ∈ *fsm-impl.outputs ?P*
proof −
have ⋀ *t* . *t* ∈ *?shifted-transitions′* ⟹ *t-input t* ∈ *fsm-impl.inputs B* ∧ *t-output t* ∈ *fsm-impl.outputs B*
unfolding *shifted-transitions-def* **using** *p6b* **by** *auto*
then show ⋀ *t* . *t* ∈ *?shifted-transitions′* ⟹ *t-input t* ∈ *fsm-impl.inputs ?P* ∧ *t-output t* ∈ *fsm-impl.outputs ?P*
unfolding ‹*fsm-impl.inputs ?P* = *fsm-impl.inputs A* ∪ *fsm-impl.inputs B*›
‹*fsm-impl.outputs ?P* = *fsm-impl.outputs A* ∪ *fsm-impl.outputs B*›
by *blast*
qed
moreover have ⋀ *t* . *t* ∈ *?distinguishing-transitions-lr* ⟹ *t-input t* ∈ *fsm-impl.inputs ?P* ∧ *t-output t* ∈ *fsm-impl.outputs ?P*
unfolding ∗ *distinguishing-transitions-def* **using** *p6a p6b True* **by** *auto*
ultimately have *p6*: (∀ *t*∈*fsm-impl.transitions ?P*.
t-source t ∈ *fsm-impl.states ?P* ∧
t-input t ∈ *fsm-impl.inputs ?P* ∧ *t-target t* ∈ *fsm-impl.states ?P* ∧
t-output t ∈ *fsm-impl.outputs ?P*)
unfolding ‹*FSM-Impl.transitions ?P* = *?ts*› **by** *blast*

show *well-formed-fsm ?P*
using *p1 p2 p3 p4 p5 p6* **by** *linarith*
qed
qed

lemma *canonical-separator′-simps* :
assumes *initial P* = (*q1*,*q2*)
shows *initial* (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator′ M P q1 q2*) = (*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*}
inputs (*canonical-separator′ M P q1 q2*) = *inputs M* ∪ *inputs P*
outputs (*canonical-separator′ M P q1 q2*) = *outputs M* ∪ *outputs P*
transitions (*canonical-separator′ M P q1 q2*)
= *shifted-transitions* (*transitions P*)
∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states P*) (*inputs P*)
using *assms* **unfolding** *h-out-code* **by** (*transfer*; *auto*)+

lemma *canonical-separator′-simps-without-assm* :
initial (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then* (*image Inl* (*states P*)) ∪ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
inputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then inputs M* ∪ *inputs P else* {})
outputs (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then outputs M* ∪ *outputs P else* {})
transitions (*canonical-separator′ M P q1 q2*) = (*if initial P* = (*q1*,*q2*) *then shifted-transitions* (*transitions P*) ∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states P*) (*inputs P*) *else* {})
unfolding *h-out-code* **by** (*transfer*; *simp add*: *Let-def*)+

end

5 Product Machines

This theory defines the construction of product machines. A product machine of two finite state machines essentially represents all possible parallel executions of those two machines.

theory *Product-FSM*
imports *FSM*
begin

lift-definition *product* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*d*,′*b*,′*c*) *fsm* ⇒ (′*a* × ′*d*,′*b*,′*c*) *fsm* **is** *FSM-Impl.product*
proof −
fix *A* :: (′*a*,′*b*,′*c*) *fsm-impl*
fix *B* :: (′*d*,′*b*,′*c*) *fsm-impl*
assume *well-formed-fsm A* **and** *well-formed-fsm B*

then have *p1a*: *fsm-impl.initial A* ∈ *fsm-impl.states A*
and *p2a*: *finite* (*fsm-impl.states A*)
and *p3a*: *finite* (*fsm-impl.inputs A*)
and *p4a*: *finite* (*fsm-impl.outputs A*)

and *p5a*: *finite* (*fsm-impl.transitions A*)
and *p6a*: (∀ *t*∈*fsm-impl.transitions A*.
t-source t ∈ *fsm-impl.states A* ∧
t-input t ∈ *fsm-impl.inputs A* ∧ *t-target t* ∈ *fsm-impl.states A* ∧
t-output t ∈ *fsm-impl.outputs A*)
and *p1b*: *fsm-impl.initial B* ∈ *fsm-impl.states B*
and *p2b*: *finite* (*fsm-impl.states B*)
and *p3b*: *finite* (*fsm-impl.inputs B*)
and *p4b*: *finite* (*fsm-impl.outputs B*)
and *p5b*: *finite* (*fsm-impl.transitions B*)
and *p6b*: (∀ *t*∈*fsm-impl.transitions B*.
t-source t ∈ *fsm-impl.states B* ∧
t-input t ∈ *fsm-impl.inputs B* ∧ *t-target t* ∈ *fsm-impl.states B* ∧
t-output t ∈ *fsm-impl.outputs B*)
by *simp+*

let *?P* = *FSM-Impl.product A B*

have *fsm-impl.initial ?P* ∈ *fsm-impl.states ?P*
using *p1a p1b* **by** *auto*
moreover have *finite* (*fsm-impl.states ?P*)
using *p2a p2b* **by** *auto*
moreover have *finite* (*fsm-impl.inputs ?P*)
using *p3a p3b* **by** *auto*
moreover have *finite* (*fsm-impl.outputs ?P*)
using *p4a p4b* **by** *auto*
moreover have *finite* (*fsm-impl.transitions ?P*)
using *p5a p5b* **unfolding** *product-code-naive* **by** *auto*
moreover have (∀ *t*∈*fsm-impl.transitions ?P*.
t-source t ∈ *fsm-impl.states ?P* ∧
t-input t ∈ *fsm-impl.inputs ?P* ∧ *t-target t* ∈ *fsm-impl.states ?P* ∧
t-output t ∈ *fsm-impl.outputs ?P*)
using *p6a p6b* **by** *auto*

ultimately show *well-formed-fsm* (*FSM-Impl.product A B*)
by *blast*
qed

abbreviation *left-path p* ≡ *map* (λ*t*. (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) *p*
abbreviation *right-path p* ≡ *map* (λ*t*. (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) *p*
abbreviation *zip-path p1 p2* ≡ (*map* (λ *t* . ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), (*t-target* (*fst t*), *t-target* (*snd t*))))
(*zip p1 p2*))

lemma *product-simps*[*simp*]:
initial (*product A B*) = (*initial A*, *initial B*)
states (*product A B*) = (*states A*) × (*states B*)
inputs (*product A B*) = *inputs A* ∪ *inputs B*
outputs (*product A B*) = *outputs A* ∪ *outputs B*
by (*transfer*; *simp*)+

lemma *product-transitions-def* :
transitions (*product A B*) = {((*qA*,*qB*),*x*,*y*,(*qA*′,*qB*′)) | *qA qB x y qA*′ *qB*′ . (*qA*,*x*,*y*,*qA*′) ∈ *transitions A* ∧ (*qB*,*x*,*y*,*qB*′) ∈ *transitions B*}
by (*transfer*; *simp*)+

lemma *product-transitions-alt-def* :
transitions (*product A B*) = {((*t-source tA*, *t-source tB*),*t-input tA*, *t-output tA*, (*t-target tA*, *t-target tB*)) | *tA tB* . *tA* ∈ *transitions A* ∧ *tB* ∈ *transitions B* ∧ *t-input tA* = *t-input tB* ∧ *t-output tA* = *t-output tB*}
(**is** *?T1* = *?T2*)
proof −
have ⋀ *t* . *t* ∈ *?T1* ⟹ *t* ∈ *?T2*
proof −
fix *tt* **assume** *tt* ∈ *?T1*
then obtain *qA qB x y qA*′ *qB*′ **where** *tt* = ((*qA*,*qB*),*x*,*y*,(*qA*′,*qB*′)) **and** (*qA*,*x*,*y*,*qA*′) ∈ *transitions A* **and** (*qB*,*x*,*y*,*qB*′) ∈ *transitions B*
unfolding *product-transitions-def* **by** *blast*
then have ((*t-source* (*qA*,*x*,*y*,*qA*′), *t-source* (*qB*,*x*,*y*,*qB*′)),*t-input* (*qA*,*x*,*y*,*qA*′), *t-output* (*qA*,*x*,*y*,*qA*′), (*t-target* (*qA*,*x*,*y*,*qA*′), *t-target* (*qB*,*x*,*y*,*qB*′))) ∈ *?T2*
by *auto*
then show *tt* ∈ *?T2*
unfolding ‹*tt* = ((*qA*,*qB*),*x*,*y*,(*qA*′,*qB*′))› *fst-conv snd-conv* **by** *assumption*
qed
moreover have ⋀ *t* . *t* ∈ *?T2* ⟹ *t* ∈ *?T1*
proof −
fix *tt* **assume** *tt* ∈ *?T2*
then obtain *tA tB* **where** *tt* = ((*t-source tA*, *t-source tB*),*t-input tA*, *t-output tA*, (*t-target tA*, *t-target tB*))
and *tA* ∈ *transitions A* **and** *tB* ∈ *transitions B* **and** *t-input tA* = *t-input tB* **and** *t-output tA* = *t-output tB*
by *blast*
then have (*t-source tA*, *t-input tA*, *t-output tA*, *t-target tA*) ∈ *transitions A*
and (*t-source tB*, *t-input tA*, *t-output tA*, *t-target tB*) ∈ *transitions B*
by (*metis prod.collapse*)+
then show *tt* ∈ *?T1*
unfolding *product-transitions-def* ‹*tt* = ((*t-source tA*, *t-source tB*),*t-input tA*, *t-output tA*, (*t-target tA*, *t-target tB*))› **by** *blast*
qed
ultimately show *?thesis* **by** *blast*

qed

lemma *zip-path-last* : *length xs* = *length ys* $\Longrightarrow$ (*zip-path* (*xs* @ [*x*]) (*ys* @ [*y*])) = (*zip-path xs ys*)@(*zip-path* [*x*] [*y*])
  by (*induction xs ys rule*: *list-induct2*; *simp*)

lemma *product-path-from-paths* :
  assumes *path A* (*initial A*) *p1*
    and *path B* (*initial B*) *p2*
    and *p-io p1* = *p-io p2*
  shows *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
    and *target* (*initial* (*product A B*)) (*zip-path p1 p2*) = (*target* (*initial A*) *p1*, *target* (*initial B*) *p2*)
proof −
  have *initial* (*product A B*) = (*initial A*, *initial B*) **by** *auto*
  then have (*initial A*, *initial B*) ∈ *states* (*product A B*)
    by (*metis fsm-initial*)

  have *length p1* = *length p2* **using** *assms*(*3*)
    using *map-eq-imp-length-eq* **by** *blast*
  then have *c*: *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
      ∧ *target* (*initial* (*product A B*)) (*zip-path p1 p2*) = (*target* (*initial A*) *p1*, *target* (*initial B*) *p2*)
    using *assms* **proof** (*induction p1 p2 rule*: *rev-induct2*)
    case *Nil*

    then have *path* (*product A B*) (*initial* (*product A B*)) (*zip-path* [] [])
      using ‹*initial* (*product A B*) = (*initial A*, *initial B*)› ‹(*initial A*, *initial B*) ∈ *states* (*product A B*)›
      by (*metis Nil-is-map-conv path.nil zip-Nil*)
    moreover have *target* (*initial* (*product A B*)) (*zip-path* [] []) = (*target* (*initial A*) [], *target* (*initial B*) [])
      using ‹*initial* (*product A B*) = (*initial A*, *initial B*)› **by** *auto*
    ultimately show *?case* **by** *fast*
  next
    case (*snoc x xs y ys*)

    have *path A* (*initial A*) *xs* **using** *snoc.prems*(*1*) **by** *auto*
    moreover have *path B* (*initial B*) *ys* **using** *snoc.prems*(*2*) **by** *auto*
    moreover have *p-io xs* = *p-io ys* **using** *snoc.prems*(*3*) **by** *auto*
    ultimately have ∗:*path* (*product A B*) (*initial* (*product A B*)) (*zip-path xs ys*)
      and ∗∗:*target* (*initial* (*product A B*)) (*zip-path xs ys*) = (*target* (*initial A*) *xs*, *target* (*initial B*) *ys*)
      using *snoc.IH* **by** *blast*+
    then have (*target* (*initial A*) *xs*, *target* (*initial B*) *ys*) ∈ *states* (*product A B*)
      by (*metis* (*no-types*, *lifting*) *path-target-is-state*)
    then have (*t-source x*, *t-source y*) ∈ *states* (*product A B*)

using *snoc.prems*(*1*−*2*) **by** (*metis path-cons-elim path-suffix*)

have $x \in$ *transitions A* **using** *snoc.prems*(*1*) **by** *auto*
moreover have $y \in$ *transitions B* **using** *snoc.prems*(*2*) **by** *auto*
moreover have *t-input x* = *t-input y* **using** *snoc.prems*(*3*) **by** *auto*
moreover have *t-output x* = *t-output y* **using** *snoc.prems*(*3*) **by** *auto*
ultimately have ((*t-source x*, *t-source y*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*)) ∈ *transitions* (*product A B*)
unfolding *product-transitions-alt-def* **by** *blast*

moreover have *t-source x* = *target* (*initial A*) *xs* **using** *snoc.prems*(*1*) **by** *auto*
moreover have *t-source y* = *target* (*initial B*) *ys* **using** *snoc.prems*(*2*) **by** *auto*
ultimately have ((*target* (*initial A*) *xs*, *target* (*initial B*) *ys*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*)) ∈ *transitions* (*product A B*)
using ‹(*t-source x*, *t-source y*) ∈ *states* (*product A B*)›
by *simp*
then have ∗∗∗: *path* (*product A B*) (*initial* (*product A B*)) ((*zip-path xs ys*)@[((*target* (*initial A*) *xs*, *target* (*initial B*) *ys*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*))])
using ∗ ∗∗
by (*metis* (*no-types*, *lifting*) *fst-conv path-append-transition*)

have *t-target x* = *target* (*initial A*) (*xs*@[*x*]) **by** *auto*
moreover have *t-target y* = *target* (*initial B*) (*ys*@[*y*]) **by** *auto*
ultimately have ∗∗∗∗: *target* (*initial* (*product A B*)) ((*zip-path xs ys*)@[((*target* (*initial A*) *xs*, *target* (*initial B*) *ys*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*))])
= (*target* (*initial A*) (*xs*@[*x*]), *target* (*initial B*) (*ys*@[*y*]))
by *fastforce*

have (*zip-path* [*x*] [*y*]) = [((*target* (*initial A*) *xs*, *target* (*initial B*) *ys*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*))]
using ‹*t-source x* = *target* (*initial A*) *xs*› ‹*t-source y* = *target* (*initial B*) *ys*› **by** *auto*
moreover have (*zip-path* (*xs* @ [*x*]) (*ys* @ [*y*])) = (*zip-path xs ys*)@(*zip-path* [*x*] [*y*])
using *zip-path-last*[*of xs ys x y*, *OF snoc.hyps*] **by** *assumption*
ultimately have ∗∗∗∗∗:(*zip-path* (*xs*@[*x*]) (*ys*@[*y*]))
= (*zip-path xs ys*)@[((*target* (*initial A*) *xs*, *target* (*initial B*) *ys*), *t-input x*, *t-output x*, (*t-target x*, *t-target y*))]
by *auto*
then have *path* (*product A B*) (*initial* (*product A B*)) (*zip-path* (*xs*@[*x*]) (*ys*@[*y*]))
using ∗∗∗ **by** *presburger*
moreover have *target* (*initial* (*product A B*)) (*zip-path* (*xs*@[*x*]) (*ys*@[*y*]))
= (*target* (*initial A*) (*xs*@[*x*]), *target* (*initial B*) (*ys*@[*y*]))
using ∗∗∗∗ ∗∗∗∗∗ **by** *auto*
ultimately show *?case* **by** *linarith*

qed

from *c* **show** *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
by *auto*
from *c* **show** *target* (*initial* (*product A B*)) (*zip-path p1 p2*)
= (*target* (*initial A*) *p1*, *target* (*initial B*) *p2*)
by *auto*
qed

lemma *paths-from-product-path* :
assumes *path* (*product A B*) (*initial* (*product A B*)) *p*
shows *path A* (*initial A*) (*left-path p*)
and *path B* (*initial B*) (*right-path p*)
and *target* (*initial A*) (*left-path p*) = *fst* (*target* (*initial* (*product A B*)) *p*)
and *target* (*initial B*) (*right-path p*) = *snd* (*target* (*initial* (*product A B*)) *p*)
proof −
have *path A* (*initial A*) (*left-path p*)
∧ *path B* (*initial B*) (*right-path p*)
∧ *target* (*initial A*) (*left-path p*) = *fst* (*target* (*initial* (*product A B*)) *p*)
∧ *target* (*initial B*) (*right-path p*) = *snd* (*target* (*initial* (*product A B*)) *p*)
using *assms* **proof** (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc t p*)
then have *path* (*product A B*) (*initial* (*product A B*)) *p* **by** *fast*
then have *path A* (*initial A*) (*left-path p*)
and *path B* (*initial B*) (*right-path p*)
and *target* (*initial A*) (*left-path p*) = *fst* (*target* (*initial* (*product A B*)) *p*)
and *target* (*initial B*) (*right-path p*) = *snd* (*target* (*initial* (*product A B*)) *p*)
using *snoc.IH* **by** *fastforce+*

then have *t-source t* = (*target* (*initial A*) (*left-path p*), *target* (*initial B*) (*right-path p*))
using *snoc.prems* **by** (*metis* (*no-types*, *lifting*) *path-cons-elim path-suffix prod.collapse*)

have ∗∗∗: *target* (*initial A*) (*left-path* (*p*@[*t*])= *fst* (*target* (*initial* (*product A B*)) (*p*@[*t*]))
by *fastforce*
have ∗∗∗∗: *target* (*initial B*) (*right-path* (*p*@[*t*])= *snd* (*target* (*initial* (*product A B*)) (*p*@[*t*]))
by *fastforce*

have *t* ∈ *transitions* (*product A B*) **using** *snoc.prems* **by** *auto*

then have (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*)) ∈ *transitions*

A
unfolding *product-transitions-alt-def* **by** *force*
moreover have *target* (*initial A*) (*left-path p*) = *fst* (*t-source t*)
using ‹*t-source t* = (*target* (*initial A*) (*left-path p*), *target* (*initial B*) (*right-path p*))› **by** *auto*
ultimately have *path A* (*initial A*) ((*left-path p*)@[(*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))])
by (*simp add*: ‹*path A* (*initial A*) (*map* (λ*t*. (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) *p*)› *path-append-transition*)
then have ∗: *path A* (*initial A*) (*left-path* (*p*@[*t*])) **by** *auto*

have (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*)) ∈ *transitions B*
using ‹*t* ∈ *transitions* (*product A B*)› **unfolding** *product-transitions-alt-def* **by** *auto*
moreover have *target* (*initial B*) (*right-path p*) = *snd* (*t-source t*)
using ‹*t-source t* = (*target* (*initial A*) (*left-path p*), *target* (*initial B*) (*right-path p*))› **by** *auto*
ultimately have *path B* (*initial B*) ((*right-path p*)@[(*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))])
by (*simp add*: ‹*path B* (*initial B*) (*map* (λ*t*. (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) *p*)› *path-append-transition*)
then have ∗∗: *path B* (*initial B*) (*right-path* (*p*@[*t*])) **by** *auto*

show *?case* **using** ∗ ∗∗ ∗∗∗ ∗∗∗∗ **by** *blast*
qed

then show *path A* (*initial A*) (*left-path p*)
and *path B* (*initial B*) (*right-path p*)
and *target* (*initial A*) (*left-path p*) = *fst* (*target* (*initial* (*product A B*)) *p*)
and *target* (*initial B*) (*right-path p*) = *snd* (*target* (*initial* (*product A B*)) *p*)
by *linarith*+
qed

lemma *zip-path-left-right*[*simp*] :
(*zip-path* (*left-path p*) (*right-path p*)) = *p* **by** (*induction p*; *auto*)

lemma *product-reachable-state-paths* :
assumes (*q1*,*q2*) ∈ *reachable-states* (*product A B*)
obtains *p1 p2*
where *path A* (*initial A*) *p1*
and *path B* (*initial B*) *p2*
and *target* (*initial A*) *p1* = *q1*
and *target* (*initial B*) *p2* = *q2*
and *p-io p1* = *p-io p2*
and *path* (*product A B*) (*initial* (*product A B*)) (*zip-path p1 p2*)
and *target* (*initial* (*product A B*)) (*zip-path p1 p2*) = (*q1*,*q2*)

proof −
  let *?P* = *product A B*
  from *assms* **obtain** *p* **where** *path ?P* (*initial ?P*) *p* **and** *target* (*initial ?P*) *p* = (*q1*,*q2*)
    unfolding *reachable-states-def* **by** *auto*

  have *path A* (*initial A*) (*left-path p*)
   and *path B* (*initial B*) (*right-path p*)
   and *target* (*initial A*) (*left-path p*) = *q1*
   and *target* (*initial B*) (*right-path p*) = *q2*
    using *paths-from-product-path*[*OF* ‹*path ?P* (*initial ?P*) *p*›] ‹*target* (*initial ?P*) *p* = (*q1*,*q2*)› **by** *auto*

  moreover have *p-io* (*left-path p*) = *p-io* (*right-path p*) **by** *auto*
  moreover have *path* (*product A B*) (*initial* (*product A B*)) (*zip-path* (*left-path p*) (*right-path p*))
    using ‹*path ?P* (*initial ?P*) *p*› **by** *auto*
  moreover have *target* (*initial* (*product A B*)) (*zip-path* (*left-path p*) (*right-path p*)) = (*q1*,*q2*)
    using ‹*target* (*initial ?P*) *p* = (*q1*,*q2*)› **by** *auto*
  ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *product-reachable-states*[*iff*] :
  (*q1*,*q2*) ∈ *reachable-states* (*product A B*) ⟷ (∃ *p1 p2* . *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
proof
  show (*q1*,*q2*) ∈ *reachable-states* (*product A B*) ⟹ (∃ *p1 p2* . *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
    using *product-reachable-state-paths*[*of q1 q2 A B*] **by** *blast*
  show (∃ *p1 p2* . *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*) ⟹ (*q1*,*q2*) ∈ *reachable-states* (*product A B*)
  proof −
    assume (∃ *p1 p2* . *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
    then obtain *p1 p2* **where** *path A* (*initial A*) *p1* ∧ *path B* (*initial B*) *p2* ∧ *target* (*initial A*) *p1* = *q1* ∧ *target* (*initial B*) *p2* = *q2* ∧ *p-io p1* = *p-io p2*
      by *blast*
    then show *?thesis*
      using *product-path-from-paths*[*of A p1 B p2*] **unfolding** *reachable-states-def*
      by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
  qed
qed

lemma *left-path-zip* : *length p1* = *length p2* $\Longrightarrow$ *left-path* (*zip-path p1 p2*) = *p1*
  by (*induction p1 p2 rule*: *list-induct2*; *simp*)

lemma *right-path-zip* : *length p1* = *length p2* $\Longrightarrow$ *p-io p1* = *p-io p2* $\Longrightarrow$ *right-path* (*zip-path p1 p2*) = *p2*
  by (*induction p1 p2 rule*: *list-induct2*; *simp*)

lemma *zip-path-append-left-right* : *length p1* = *length p2* $\Longrightarrow$ *zip-path* (*p1*@(*left-path p*)) (*p2*@(*right-path p*)) = (*zip-path p1 p2*)@*p*
proof (*induction p1 p2 rule*: *list-induct2*)
  case *Nil*
  then show *?case* **by** (*induction p*; *simp*)
next
  case (*Cons x xs y ys*)
  then show *?case* **by** *simp*
qed

lemma *product-path*:
  path (*product A B*) (*q1*,*q2*) *p* $\longleftrightarrow$ (*path A q1* (*left-path p*) $\wedge$ *path B q2* (*right-path p*))
proof (*induction p arbitrary*: *q1 q2*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons t p*)

  have *path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*) $\Longrightarrow$ (*path A q1* (*left-path* (*t* # *p*)) $\wedge$ *path B q2* (*right-path* (*t* # *p*)))
  proof −
    assume *path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*)
   then obtain *x y qA′ qB′* **where** *t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*)) **using** *prod.collapse*
      by (*metis path-cons-elim*)
    then have ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*)) $\in$ *transitions* (*product A B*)
      using ‹*path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*)› **by** *auto*
      then have (*q1*, *x*, *y*, *qA′*) $\in$ *FSM.transitions A* **and** (*q2*, *x*, *y*, *qB′*) $\in$ *FSM.transitions B*
      unfolding *product-transitions-def* **by** *blast+*
    moreover have (*path A qA′* (*left-path p*) $\wedge$ *path B qB′* (*right-path p*))
      using *Cons.IH*[*of qA′ qB′*] ‹*path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*)› **unfolding** ‹*t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*))› **by** *auto*
    ultimately show *?thesis*
      unfolding ‹*t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*))›
      by (*simp add*: *path-prepend-t*)
  qed

  moreover have *path A q1* (*left-path* (*t* # *p*)) $\Longrightarrow$ *path B q2* (*right-path* (*t* #

p)) $\Longrightarrow$ *path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*)
proof −
assume *path A q1* (*left-path* (*t* # *p*)) **and** *path B q2* (*right-path* (*t* # *p*))
then obtain *x y qA′ qB′* **where** *t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*)) **using** *prod.collapse*
by (*metis* (*no-types*, *lifting*) *fst-conv list.simps*(*9*) *path-cons-elim*)
then have (*q1*, *x*, *y*, *qA′*) ∈ *FSM.transitions A* **and** (*q2*, *x*, *y*, *qB′*) ∈ *FSM.transitions B*
using ‹*path A q1* (*left-path* (*t* # *p*))› ‹*path B q2* (*right-path* (*t* # *p*))› **by** *auto*
then have ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*)) ∈ *transitions* (*product A B*)
unfolding *product-transitions-def* **by** *blast*
moreover have *path* (*Product-FSM.product A B*) (*qA′*, *qB′*) *p*
using *Cons.IH*[*of qA′ qB′*] ‹*path A q1* (*left-path* (*t* # *p*))› ‹*path B q2* (*right-path* (*t* # *p*))› **unfolding** ‹*t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*))› **by** *auto*
ultimately show *path* (*Product-FSM.product A B*) (*q1*, *q2*) (*t* # *p*)
unfolding ‹*t* = ((*q1*,*q2*),*x*,*y*,(*qA′*,*qB′*))›
by (*simp add*: *path-prepend-t*)
qed

ultimately show *?case* **by** *force*
qed

lemma *product-path-rev*:
assumes *p-io p1* = *p-io p2*
shows *path* (*product A B*) (*q1*,*q2*) (*zip-path p1 p2*) $\longleftrightarrow$ (*path A q1 p1* ∧ *path B q2 p2*)
proof −
have *length p1* = *length p2* **using** *assms*
using *map-eq-imp-length-eq* **by** *blast*
then have (*map* (λ *t* . (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) (*map* (λ *t* . ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), (*t-target* (*fst t*), *t-target* (*snd t*)))) (*zip p1 p2*))) = *p1*
by (*induction p1 p2 arbitrary*: *q1 q2 rule*: *list-induct2*; *auto*)

moreover have (*map* (λ *t* . (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) (*map* (λ *t* . ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), (*t-target* (*fst t*), *t-target* (*snd t*)))) (*zip p1 p2*))) = *p2*
using ‹*length p1* = *length p2*› *assms* **by** (*induction p1 p2 arbitrary*: *q1 q2 rule*: *list-induct2*; *auto*)

ultimately show *?thesis* **using** *product-path*[*of A B q1 q2* (*map* (λ *t* . ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), (*t-target* (*fst t*), *t-target* (*snd t*)))) (*zip p1 p2*))]
by *auto*
qed

lemma *product-language-state* :
shows *LS* (*product A B*) (*q1*,*q2*) = *LS A q1* ∩ *LS B q2*

```
proof
  show LS (product A B) (q1, q2) ⊆ LS A q1 ∩ LS B q2
  proof
    fix io assume io ∈ LS (product A B) (q1, q2)
    then obtain p where io = p-io p
                  and path (product A B) (q1,q2) p
      by auto
    then obtain p1 p2 where path A q1 p1
                      and path B q2 p2
                      and io = p-io p1
                      and io = p-io p2
      using product-path[of A B q1 q2 p] by fastforce
    then show io ∈ LS A q1 ∩ LS B q2
      unfolding LS.simps by blast
  qed

  show LS A q1 ∩ LS B q2 ⊆ LS (product A B) (q1, q2)
  proof
    fix io assume io ∈ LS A q1 ∩ LS B q2
    then obtain p1 p2 where path A q1 p1
                      and path B q2 p2
                      and io = p-io p1
                      and io = p-io p2
                      and p-io p1 = p-io p2
      by auto

    let ?p = zip-path p1 p2


    have length p1 = length p2
      using ‹p-io p1 = p-io p2› map-eq-imp-length-eq by blast
    moreover have p-io ?p = p-io (map fst (zip p1 p2)) by auto
    ultimately have p-io ?p = p-io p1 by auto

    then have p-io ?p = io
      using ‹io = p-io p1› by auto
    moreover have path (product A B) (q1, q2) ?p
     using product-path-rev[OF ‹p-io p1 = p-io p2›, of A B q1 q2] ‹path A q1 p1›
‹path B q2 p2› by auto
    ultimately show io ∈ LS (product A B) (q1, q2)
      unfolding LS.simps by blast
  qed
qed


lemma product-language : L (product A B) = L A ∩ L B
  unfolding product-simps product-language-state by blast
```

lemma *product-transition-split-ob* :
assumes $t \in$ *transitions* (*product A B*)
obtains *t1 t2*
where *t1* $\in$ *transitions A* $\wedge$ *t-source t1* = *fst* (*t-source t*) $\wedge$ *t-input t1* = *t-input t* $\wedge$ *t-output t1* = *t-output t* $\wedge$ *t-target t1* = *fst* (*t-target t*)
and *t2* $\in$ *transitions B* $\wedge$ *t-source t2* = *snd* (*t-source t*) $\wedge$ *t-input t2* = *t-input t* $\wedge$ *t-output t2* = *t-output t* $\wedge$ *t-target t2* = *snd* (*t-target t*)
using *assms* **unfolding** *product-transitions-alt-def*
by *auto*

lemma *product-transition-split* :
assumes $t \in$ *transitions* (*product A B*)
shows (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*)) $\in$ *transitions A*
and (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*)) $\in$ *transitions B*
using *product-transition-split-ob*[*OF assms*] *prod.collapse* **by** *fastforce*+

lemma *product-target-split*:
assumes *target* (*q1*,*q2*) *p* = (*q1'*,*q2'*)
shows *target q1* (*left-path p*) = *q1'*
and *target q2* (*right-path p*) = *q2'*
using *assms* **by** (*induction p arbitrary*: *q1 q2*; *force*)+

lemma *target-single-transition*[*simp*] : *target q1* [(*q1*, *x*, *y*, *q1'*)] = *q1'*
by *auto*

lemma *product-undefined-input* :
assumes $\neg$ ($\exists$ *t* $\in$ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)).
t-source t = *qq* $\wedge$ *t-input t* = *x*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
shows $\neg$ ($\exists$ *t1* $\in$ *transitions M*. $\exists$ *t2* $\in$ *transitions M*.
t-source t1 = *fst qq* $\wedge$
t-source t2 = *snd qq* $\wedge$
t-input t1 = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*)
proof
assume $\exists$ *t1* $\in$ *transitions M*. $\exists$ *t2* $\in$ *transitions M*.
t-source t1 = *fst qq* $\wedge$
t-source t2 = *snd qq* $\wedge$
t-input t1 = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*
then obtain *t1 t2* **where** *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*
and *t-source t1* = *fst qq*
and *t-source t2* = *snd qq*
and *t-input t1* = *x*
and *t-input t1* = *t-input t2*

and *t-output t1* = *t-output t2*
by *force*

have ((*t-source t1*, *t-source t2*), *t-input t1*, *t-output t1*, *t-target t1*, *t-target t2*) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
unfolding *product-transitions-alt-def*
unfolding *from-FSM-simps*[*OF assms*(*2*)]
unfolding *from-FSM-simps*[*OF assms*(*3*)]
using ‹*t1* ∈ *transitions M*› ‹*t2* ∈ *transitions M*› ‹*t-input t1* = *t-input t2*› ‹*t-output t1* = *t-output t2*› **by** *blast*
then show *False*
unfolding ‹*t-source t1* = *fst qq*› ‹*t-source t2* = *snd qq*› ‹*t-input t1* = *x*› *prod.collapse*
using *assms*(*1*) **by** *auto*
qed

5.1 Product Machines and Changing Initial States

lemma *product-from-reachable-next* :
assumes ((*q1*,*q2*),*x*,*y*,(*q1'*,*q2'*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*from-FSM* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*, *q2'*)) = (*product* (*from-FSM M q1'*) (*from-FSM M q2'*))
(**is** *?P1* = *?P2*)
proof −
have (*q1*,*x*,*y*,*q1'*) ∈ *transitions* (*from-FSM M q1*)
and (*q2*,*x*,*y*,*q2'*) ∈ *transitions* (*from-FSM M q2*)
using *assms*(*1*) **unfolding** *product-transitions-def* **by** *blast+*
then have *q1'* ∈ *states* (*from-FSM M q1*) **and** *q2'* ∈ *states* (*from-FSM M q2*)
using *fsm-transition-target* **by** *auto*

have *q1'* ∈ *states* (*from-FSM M q1'*) **and** *q1'* ∈ *states M* **and** *q1* ∈ *states M*
using ‹*q1'* ∈ *FSM.states* (*FSM.from-FSM M q1*)› *assms*(*2*) *reachable-state-is-state* **by** *fastforce+*
have *q2'* ∈ *states* (*from-FSM M q2'*) **and** *q2'* ∈ *states M* **and** *q2* ∈ *states M*
using ‹*q2'* ∈ *FSM.states* (*FSM.from-FSM M q2*)› *assms*(*3*) *reachable-state-is-state* **by** *fastforce+*

have *initial ?P1* = *initial ?P2*
and *states ?P1* = *states ?P2*
and *inputs ?P1* = *inputs ?P2*
and *outputs ?P1* = *outputs ?P2*
and *transitions ?P1* = *transitions ?P2*
using *from-FSM-simps*[*OF fsm-transition-target*[*OF assms*(*1*)]]
unfolding *snd-conv*
unfolding *product-simps*
unfolding *product-transitions-def*

unfolding *from-FSM-simps*[*OF* ‹*q1'* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q2'* ∈ *states M*›]
unfolding *from-FSM-simps*[*OF* ‹*q1* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q2* ∈ *states M*›]
by *auto*

then show *?thesis* **by** (*transfer*; *auto*)
qed

lemma *from-FSM-product-inputs* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (*inputs* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = (*inputs M*)
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *from-FSM-product-outputs* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (*outputs* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = (*outputs M*)
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *from-FSM-product-initial* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) = (*q1*,*q2*)
by (*simp add*: *assms*(*1*) *assms*(*2*))

lemma *product-from-reachable-next'* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*))))
and *fst* (*t-source t*) ∈ *states M*
and *snd* (*t-source t*) ∈ *states M*
shows (*from-FSM* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*)))) (*fst* (*t-target t*),*snd* (*t-target t*))) = (*product* (*from-FSM M* (*fst* (*t-target t*))) (*from-FSM M* (*snd* (*t-target t*))))
proof −
have ((*fst* (*t-source t*), *snd* (*t-source t*)), *t-input t*, *t-output t*, *fst* (*t-target t*), *snd* (*t-target t*)) = *t*
by *simp*
then show *?thesis*
by (*metis* (*no-types*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *product-from-reachable-next*)
qed

lemma *product-from-reachable-next'-path* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*))))
and *fst* (*t-source t*) ∈ *states M*

and *snd* (*t-source t*) ∈ *states M*
shows *path* (*from-FSM* (*product* (*from-FSM M* (*fst* (*t-source t*))) (*from-FSM M* (*snd* (*t-source t*)))) (*fst* (*t-target t*),*snd* (*t-target t*))) (*fst* (*t-target t*),*snd* (*t-target t*)) *p* = *path* (*product* (*from-FSM M* (*fst* (*t-target t*))) (*from-FSM M* (*snd* (*t-target t*)))) (*fst* (*t-target t*),*snd* (*t-target t*)) *p*
(**is** *path ?P1 ?q p* = *path ?P2 ?q p*)
proof −
have *i1*: *initial ?P1* = *?q*
using *assms*(*1*) *fsm-transition-target* **by** *fastforce*
have *i2*: *initial ?P2* = *?q*
proof −
have ((*fst* (*t-source t*), *snd* (*t-source t*)), *t-input t*, *t-output t*, *fst* (*t-target t*), *snd* (*t-target t*)) = *t*
by *auto*
then show *?thesis*
by (*metis* (*no-types*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *i1 product-from-reachable-next*)
qed

have *h12*: *transitions ?P1* = *transitions ?P2* **using** *product-from-reachable-next′*[*OF assms*] **by** *simp*

show *?thesis* **proof** (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*metis* (*full-types*) *i1 i2 fsm-initial path.nil*)
next
case (*snoc t p*)
show *?case*
by (*metis h12 path-append-transition path-append-transition-elim*(*1*) *path-append-transition-elim*(*2*) *path-append-transition-elim*(*3*) *snoc.IH*)
qed
qed

lemma *product-from-transition*:
assumes (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *transitions* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*)) = *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
proof −
have *q1′* ∈ *states M* **and** *q2′* ∈ *states M*
using *assms*(*1*) **unfolding** *product-simps from-FSM-simps*[*OF assms*(*2*)] *from-FSM-simps*[*OF assms*(*3*)] **by** *auto*
show *?thesis*
unfolding *product-transitions-def from-FSM-simps*[*OF* ‹*q1* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q1′* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q2* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q2′* ∈ *states M*›] **by** *blast*
qed

lemma *product-from-path*:
assumes (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *path* (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) (*q1'*,*q2'*) *p*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*,*q2'*) *p*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *from-FSM-path-initial from-FSM-simps*(*5*) *from-from mem-Sigma-iff product-path product-simps*(*2*))

lemma *product-from-path-previous* :
assumes *path* (*product* (*from-FSM M* (*fst* (*t-target t*)))
(*from-FSM M* (*snd* (*t-target t*))))
(*t-target t*) *p* (**is** *path ?Pt* (*t-target t*) *p*)
and *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*t-target t*) *p* (**is** *path ?P* (*t-target t*) *p*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *fsm-transition-target prod.collapse product-from-path*)

lemma *product-from-transition-shared-state* :
assumes *t* ∈ *transitions* (*product* (*from-FSM M q1'*) (*from-FSM M q2'*))
and (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
by (*metis assms product-from-transition*)

lemma *product-from-not-completely-specified* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*,*q2'*)
and (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ *completely-specified-state* (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) (*q1'*,*q2'*)
proof −
have *q1'* ∈ *states M* **and** *q2'* ∈ *states M*
using *assms*(*2*) **unfolding** *product-simps from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)] **by** *auto*
show *?thesis*

using *from-FSM-product-inputs*[*OF assms*(*3*) *assms*(*4*)]

using *from-FSM-product-inputs*[*OF* ‹*q1'* ∈ *states M*› ‹*q2'* ∈ *states M*›]
proof −
have *FSM.transitions* (*Product-FSM.product* (*FSM.from-FSM M q1'*) (*FSM.from-FSM M q2'*)) = *FSM.transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
by (*metis* (*no-types*) ‹(*q1'*, *q2'*) ∈ *FSM.states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))› *assms*(*3*) *assms*(*4*) *product-from-transition*)
then show *?thesis*
using ‹*FSM.inputs* (*Product-FSM.product* (*FSM.from-FSM M q1'*) (*FSM.from-FSM M q2'*)) = *FSM.inputs M*› ‹*FSM.inputs* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) = *FSM.inputs M*› ‹¬ *completely-specified-state* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) (*q1'*, *q2'*)›
by *fastforce*
qed
qed

lemma *from-product-initial-paths-ex* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (∃ *p1 p2*.
path (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1* ∧
path (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2* ∧
target (*initial* (*from-FSM M q1*)) *p1* = *q1* ∧
target (*initial* (*from-FSM M q2*)) *p2* = *q2* ∧ *p-io p1* = *p-io p2*)
proof −
have *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) [] **by** *blast*
moreover have *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) [] **by** *blast*
moreover have
target (*initial* (*from-FSM M q1*)) [] = *q1* ∧
target (*initial* (*from-FSM M q2*)) [] = *q2* ∧ *p-io* [] = *p-io* []
unfolding *from-FSM-simps*[*OF assms*(*1*)] *from-FSM-simps*[*OF assms*(*2*)] **by** *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *product-observable* :
assumes *observable M1*
and *observable M2*
shows *observable* (*product M1 M2*) (**is** *observable ?P*)
proof −
have ⋀ *t1 t2* . *t1* ∈ *transitions ?P* ⟹ *t2* ∈ *transitions ?P* ⟹ *t-source t1* = *t-source t2* ⟹ *t-input t1* = *t-input t2* ⟹ *t-output t1* = *t-output t2* ⟹ *t-target t1* = *t-target t2*
proof −
fix *t1 t2* **assume** *t1* ∈ *transitions ?P* **and** *t2* ∈ *transitions ?P* **and** *t-source t1* = *t-source t2* **and** *t-input t1* = *t-input t2* **and** *t-output t1* = *t-output t2*

```
    let ?t1L = (fst (t-source t1), t-input t1, t-output t1, fst (t-target t1))
    let ?t1R = (snd (t-source t1), t-input t1, t-output t1, snd (t-target t1))
    let ?t2L = (fst (t-source t2), t-input t2, t-output t2, fst (t-target t2))
    let ?t2R = (snd (t-source t2), t-input t2, t-output t2, snd (t-target t2))

    have t-target ?t1L = t-target ?t2L
      using product-transition-split(1)[OF ‹t1 ∈ transitions ?P›]
            product-transition-split(1)[OF ‹t2 ∈ transitions ?P›]
            ‹observable M1›
            ‹t-source t1 = t-source t2›
            ‹t-input t1 = t-input t2›
            ‹t-output t1 = t-output t2› by auto
    moreover have t-target ?t1R = t-target ?t2R
      using product-transition-split(2)[OF ‹t1 ∈ transitions ?P›]
            product-transition-split(2)[OF ‹t2 ∈ transitions ?P›]
            ‹observable M2›
            ‹t-source t1 = t-source t2›
            ‹t-input t1 = t-input t2›
            ‹t-output t1 = t-output t2› by auto
    ultimately show t-target t1 = t-target t2
      by (metis prod.exhaust-sel snd-conv)
  qed
  then show ?thesis unfolding observable.simps by blast
qed


lemma product-observable-self-transitions :
  assumes q ∈ reachable-states (product M M)
  and     observable M
shows fst q = snd q
proof −
  let ?P = product M M

  have ⋀ p . path ?P (initial ?P) p ⟹ fst (target (initial ?P) p) = snd (target
(initial ?P) p)
  proof −
    fix p assume path ?P (initial ?P) p
    then show fst (target (initial ?P) p) = snd (target (initial ?P) p)
    proof (induction p rule: rev-induct)
      case Nil
      then show ?case by simp
    next
      case (snoc t p)

      have path ?P (initial ?P) p and path ?P (target (initial ?P) p) [t]
          using path-append-elim[of ?P initial ?P p [t], OF ‹path (product M M)
(initial (product M M)) (p @ [t])›] by blast+
      then have t ∈ transitions ?P
        by blast
```

```
      have t-source t = target (initial ?P) p
        using snoc.prems by fastforce


      let ?t1 = (fst (t-source t), t-input t, t-output t, fst (t-target t))
      let ?t2 = (snd (t-source t), t-input t, t-output t, snd (t-target t))
      have ?t1 ∈ transitions M and ?t2 ∈ transitions M
        using product-transition-split[OF ‹t ∈ transitions ?P›] by auto
      moreover have t-source ?t1 = t-source ?t2
        using ‹t-source t = target (initial ?P) p› snoc.IH[OF ‹path ?P (initial ?P)
p›]
        by (metis fst-conv)
      moreover have t-input ?t1 = t-input ?t2
        by auto
      moreover have t-output ?t1 = t-output ?t2
        by auto
      ultimately have t-target ?t1 = t-target ?t2
        using ‹observable M› unfolding observable.simps by blast
      then have fst (t-target t) = snd (t-target t)
        by auto
      then show ?case unfolding target.simps visited-states.simps
      proof −
       show fst (last (initial (product M M) # map t-target (p @ [t]))) = snd (last
(initial (product M M) # map t-target (p @ [t])))
        using ‹fst (t-target t) = snd (t-target t)› last-map last-snoc length-append-singleton
length-map by force
      qed
    qed
  qed

  then show ?thesis
    using assms(1) unfolding reachable-states-def
    by blast
qed


lemma zip-path-eq-left :
  assumes length xs1 = length xs2
  and     length xs2 = length ys1
  and     length ys1 = length ys2
  and     zip-path xs1 xs2 = zip-path ys1 ys2
shows xs1 = ys1
  using assms by (induction xs1 xs2 ys1 ys2 rule: list-induct4; auto)


lemma zip-path-eq-right :
  assumes length xs1 = length xs2
  and     length xs2 = length ys1
  and     length ys1 = length ys2
```

and *p-io xs2* = *p-io ys2*
and *zip-path xs1 xs2* = *zip-path ys1 ys2*
shows *xs2* = *ys2*
using *assms* **by** (*induction xs1 xs2 ys1 ys2 rule*: *list-induct4*; *auto*)

lemma *zip-path-merge* :
(*zip-path* (*left-path p*) (*right-path p*)) = *p*
by (*induction p*; *auto*)

lemma *product-from-reachable-path'* :
assumes *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1'*, *q2'*) *p*
and *q1* ∈ *reachable-states M*
and *q2* ∈ *reachable-states M*
shows *path* (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) (*q1'*, *q2'*) *p*
by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *from-FSM-path from-FSM-path-rev-initial product-path reachable-state-is-state*)

lemma *product-from* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *product* (*from-FSM M q1*) (*from-FSM M q2*) = *from-FSM* (*product M M*) (*q1*,*q2*) (**is** *?PF* = *?FP*)
proof −
have (*q1*,*q2*) ∈ *states* (*product M M*)
using *assms* **unfolding** *product-simps* **by** *auto*

have *initial ?FP* = *initial ?PF*
and *inputs ?FP* = *inputs ?PF*
and *outputs ?FP* = *outputs ?PF*
and *states ?FP* = *states ?PF*
and *transitions ?FP* = *transitions ?PF*
unfolding *product-simps*
from-FSM-simps[*OF assms*(*1*)]
from-FSM-simps[*OF assms*(*2*)]
from-FSM-simps[*OF* ‹(*q1*,*q2*) ∈ *states* (*product M M*)›]
product-transitions-def
by *auto*
then show *?thesis* **by** (*transfer*; *auto*)
qed

lemma *product-from-from* :
assumes (*q1'*,*q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*product* (*from-FSM M q1'*) (*from-FSM M q2'*)) = (*from-FSM* (*product*

(*from-FSM M q1*) (*from-FSM M q2*)) (*q1′,q2′*))
using *product-from*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *from-FSM-simps*(*5*) *from-from mem-Sigma-iff product-simps*(*2*))

lemma *submachine-transition-product-from* :
assumes *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) ∈ *transitions S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *is-submachine* (*from-FSM S* (*q1′*,*q2′*)) (*product* (*from-FSM M q1′*) (*from-FSM M q2′*))
proof −
have ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *assms*(*1*) *assms*(*2*) **by** *auto*
have (*q1′*,*q2′*) ∈ *states S* **using** *fsm-transition-target assms*(*2*) **by** *auto*
show *?thesis*
using *product-from-reachable-next*[*OF* ‹((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))› *assms*(*3*,*4*)]
submachine-from[*OF assms*(*1*) ‹(*q1′*,*q2′*) ∈ *states S*›]
by *simp*
qed

lemma *submachine-transition-complete-product-from* :
assumes *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *completely-specified S*
and ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) ∈ *transitions S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *completely-specified* (*from-FSM S* (*q1′*,*q2′*))
proof −
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))
let *?P′* = (*product* (*from-FSM M q1′*) (*from-FSM M q2′*))
let *?F* = (*from-FSM S* (*q1′*,*q2′*))

have *initial ?P* = (*q1*,*q2*)
by (*simp add*: *assms*(*4*) *assms*(*5*) *reachable-state-is-state*)

then have *initial S* = (*q1*,*q2*)
using *assms*(*1*) **by** (*metis is-submachine.simps*)
then have (*q1′*,*q2′*) ∈ *states S*
using *assms*(*3*)
using *fsm-transition-target* **by** *fastforce*
then have *states ?F* = *states S*
using *from-FSM-simps*(*5*) **by** *simp*
moreover have *inputs ?F* = *inputs S*

using *from-FSM-simps*(*2*) ‹(*q1'*,*q2'*) ∈ *states S*› **by** *simp*
ultimately show *completely-specified ?F*
using *assms*(*2*) **unfolding** *completely-specified.simps*
by (*meson assms*(*2*) *completely-specified.elims*(*2*) *from-FSM-completely-specified*)
qed

5.2 Calculating Acyclic Intersection Languages

lemma *acyclic-product* :
assumes *acyclic B*
shows *acyclic* (*product A B*)
proof −
show *acyclic* (*product A B*)
proof (*rule ccontr*)
assume ¬ *FSM.acyclic* (*Product-FSM.product A B*)
then obtain *p* **where** *path* (*product A B*) (*initial* (*product A B*)) *p* **and** ¬ *distinct* (*visited-states* (*initial* (*product A B*)) *p*)
by *auto*

have *path B* (*initial B*) (*right-path p*)
using *product-path*[*of A B*] ‹*path* (*product A B*) (*initial* (*product A B*)) *p*›
unfolding *product-simps*
by *auto*

moreover have ¬ *distinct* (*visited-states* (*initial B*) (*right-path p*))
proof −
obtain *i j* **where** *i* < *j* **and** *j* < *length* ((*initial A*, *initial B*) # *map t-target p*) **and** ((*initial A*, *initial B*) # *map t-target p*) ! *i* = ((*initial A*, *initial B*) # *map t-target p*) ! *j*
using ‹¬ *distinct* (*visited-states* (*initial* (*product A B*)) *p*)›
unfolding *visited-states.simps product-simps*
using *non-distinct-repetition-indices* **by** *blast*

then have *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *i*) = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *j*)
by *simp*

have ∗ :*i* < *length* ((*initial B*) # *map t-target* (*right-path p*))
and ∗∗:*j* < *length* ((*initial B*) # *map t-target* (*right-path p*))
using ‹*i* < *j*› ‹*j* < *length* ((*initial A*, *initial B*) # *map t-target p*)› **by** *auto*

have *right-nth*: ⋀ *i* . *i* < *length* ((*initial B*) # *map t-target* (*right-path p*)) ⟹ ((*initial B*) # *map t-target* (*right-path p*)) ! *i* = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *i*)
proof −
have ((*initial B*) # *map t-target* (*right-path p*)) ! *0* = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *0*)
by *simp*

moreover have $\bigwedge$ *i* . *Suc i* < *length* ((*initial B*) # *map t-target* (*right-path p*)) $\Longrightarrow$ ((*initial B*) # *map t-target* (*right-path p*)) ! *Suc i* = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *Suc i*)
by *auto*
ultimately show $\bigwedge$ *i* . *i* < *length* ((*initial B*) # *map t-target* (*right-path p*)) $\Longrightarrow$ ((*initial B*) # *map t-target* (*right-path p*)) ! *i* = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *i*)
using *less-Suc-eq-0-disj* **by** *auto*
qed

have ((*initial B*) # *map t-target* (*right-path p*)) ! *i* = ((*initial B*) # *map t-target* (*right-path p*)) ! *j*
using ‹*snd* (((*initial A*, *initial B*) # *map t-target p*) ! *i*) = *snd* (((*initial A*, *initial B*) # *map t-target p*) ! *j*)›
unfolding *right-nth*[*OF* ∗] *right-nth*[*OF* ∗∗]
by *assumption*
then show *?thesis*
unfolding *visited-states.simps product-simps*
using *non-distinct-repetition-indices-rev*[*OF* ‹*i* < *j*› ∗∗] **by** *blast*
qed
ultimately show *False*
using ‹*acyclic B*› **unfolding** *acyclic.simps* **by** *blast*
qed
qed

lemma *acyclic-product-path-length* :
assumes *acyclic B*
and *path* (*product A B*) (*initial* (*product A B*)) *p*
shows *length p* < *size B*
proof −
have ∗:*path B* (*initial B*) (*right-path p*)
using *product-path*[*of A B*] ‹*path* (*product A B*) (*initial* (*product A B*)) *p*›
unfolding *product-simps*
by *auto*
then have ∗∗: *distinct* (*visited-states* (*initial B*) (*right-path p*))
using *assms* **unfolding** *acyclic.simps* **by** *blast*

have *length* (*right-path p*) < *size B*
using *acyclic-path-length-limit*[*OF* ∗ ∗∗] **by** *assumption*
then show *length p* < *size B*
by *auto*
qed

lemma *acyclic-language-alt-def* :
assumes *acyclic A*
shows *image p-io* (*acyclic-paths-up-to-length A* (*initial A*) (*size A* − *1*)) = *L A*
proof −

let *?ps* = *acyclic-paths-up-to-length A* (*initial A*) (*size A* − *1*)
have $\bigwedge$ *p* . *path A* (*initial A*) *p* $\Longrightarrow$ *length p* $\leq$ *FSM.size A* − *1*
using *acyclic-path-length-limit assms* **unfolding** *acyclic.simps*
by *fastforce*
then have *?ps* = {*p*. *path A* (*initial A*) *p*}
using *assms* **unfolding** *acyclic-paths-up-to-length.simps acyclic.simps* **by** *blast*
then show *?thesis* **unfolding** *LS.simps* **by** *blast*
qed

definition *acyclic-language-intersection* :: (*'a*,*'b*,*'c*) *fsm* $\Rightarrow$ (*'d*,*'b*,*'c*) *fsm* $\Rightarrow$ (*'b* $\times$ *'c*) *list set* **where**
acyclic-language-intersection M A = (*let P* = *product M A in image p-io* (*acyclic-paths-up-to-length P* (*initial P*) (*size A* − *1*)))

lemma *acyclic-language-intersection-completeness* :
assumes *acyclic A*
shows *acyclic-language-intersection M A* = *L M* $\cap$ *L A*
proof −
let *?P* = *product M A*
let *?ps* = *acyclic-paths-up-to-length ?P* (*initial ?P*) (*size A* − *1*)

have *L ?P* = *L M* $\cap$ *L A*
using *product-language* **by** *blast*

have $\bigwedge$ *p* . *path ?P* (*initial ?P*) *p* $\Longrightarrow$ *length p* $\leq$ *FSM.size A* − *1*
using *acyclic-product-path-length*[*OF assms*]
by *fastforce*
then have *?ps* = {*p*. *path ?P* (*initial ?P*) *p*}
using *acyclic-product*[*OF assms*] **unfolding** *acyclic-paths-up-to-length.simps acyclic.simps* **by** *blast*
then have *image p-io ?ps* = *L ?P*
unfolding *LS.simps* **by** *blast*
then show *?thesis*
using *product-language* **unfolding** *acyclic-language-intersection-def Let-def* **by** *blast*
qed

end

6 Minimisation by OFSM Tables

This theory presents the classical algorithm for transforming observable FSMs into language-equivalent observable and minimal FSMs in analogy to the minimisation of finite automata.

theory *Minimisation*

imports *FSM*
begin

6.1 OFSM Tables

OFSM tables partition the states of an FSM based on an initial partition and an iteration counter. States are in the same element of the 0th table iff they are in the same element of the initial partition. States q1, q2 are in the same element of the (k+1)-th table if they are in the same element of the k-th table and furthermore for each IO pair (x,y) either (x,y) is not in the language of both q1 and q2 or it is in the language of both states and the states q1', q2' reached via (x,y) from q1 and q2, respectively, are in the same element of the k-th table.

fun *ofsm-table* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a \Rightarrow 'a$ *set*$)$ $\Rightarrow$ *nat* $\Rightarrow$ $'a \Rightarrow 'a$ *set* **where**
ofsm-table M f 0 q = (*if q* $\in$ *states M then f q else* {}) |
ofsm-table M f (*Suc k*) *q* = (*let*
prev-table = *ofsm-table M f k*
in $\{q' \in$ *prev-table q* . $\forall$ *x* $\in$ *inputs M* . $\forall$ *y* $\in$ *outputs M* . (*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q' x y of Some qT'* $\Rightarrow$ *prev-table qT* = *prev-table qT'* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q' x y* = *None*) $\}$)

lemma *ofsm-table-non-state* :
assumes $q \notin$ *states M*
shows *ofsm-table M f k q* = {}
using *assms* **by** (*induction k*; *auto*)

lemma *ofsm-table-subset*:
assumes $i \leq j$
shows *ofsm-table M f j q* $\subseteq$ *ofsm-table M f i q*
proof −
have $*$: $\bigwedge$ *k* . *ofsm-table M f* (*Suc k*) *q* $\subseteq$ *ofsm-table M f k q*
proof −
fix *k* **show** *ofsm-table M f* (*Suc k*) *q* $\subseteq$ *ofsm-table M f k q*
proof (*cases k*)
case *0*
show *?thesis* **unfolding** *0 ofsm-table.simps Let-def* **by** *blast*
next
case (*Suc k'*)

show *?thesis*
unfolding *Suc ofsm-table.simps Let-def* **by** *force*
qed
qed

show *?thesis*
using *assms*
proof (*induction j*)

case *0*
then show *?case* **by** *auto*
next
case (*Suc x*)
then show *?case* **using** $*$[*of x*]
using *le-SucE* **by** *blast*
qed
qed

lemma *ofsm-table-case-helper* :
(*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q′ x y of Some qT′* $\Rightarrow$ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q′ x y* = *None*)
= ((∃ *qT qT′* . *h-obs M q x y* = *Some qT* ∧ *h-obs M q′ x y* = *Some qT′* ∧ *ofsm-table M f k qT* = *ofsm-table M f k qT′*) ∨ (*h-obs M q x y* = *None* ∧ *h-obs M q′ x y* = *None*))
proof −
have $*$: ⋀ *a b P* . (*case a of Some a′* $\Rightarrow$ (*case b of Some b′* $\Rightarrow$ *P a′ b′* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *b* = *None*)
= ((∃ *a′ b′* . *a* = *Some a′* ∧ *b* = *Some b′* ∧ *P a′ b′*) ∨ (*a* = *None* ∧ *b* = *None*))
(**is** ⋀ *a b P* . *?P1 a b P* = *?P2 a b P*)
proof
fix *a b P*
show *?P1 a b P* ⟹ *?P2 a b P* **using** *case-optionE*[*of b* = *None* λ*a′* . (*case b of Some b′* $\Rightarrow$ *P a′ b′* | *None* $\Rightarrow$ *False*) *a*]
by (*metis case-optionE*)
show *?P2 a b P* ⟹ *?P1 a b P* **by** *auto*
qed

show *?thesis*
using $*$[*of h-obs M q′ x y* λ*qT qT′* . *ofsm-table M f k qT* = *ofsm-table M f k qT′ h-obs M q x y*] **.**
qed

lemma *ofsm-table-case-helper-neg* :
(¬ (*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q′ x y of Some qT′* $\Rightarrow$ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q′ x y* = *None*))
= ((∃ *qT qT′* . *h-obs M q x y* = *Some qT* ∧ *h-obs M q′ x y* = *Some qT′* ∧ *ofsm-table M f k qT* ≠ *ofsm-table M f k qT′*) ∨ (*h-obs M q x y* = *None* ⟷ *h-obs M q′ x y* ≠ *None*))
unfolding *ofsm-table-case-helper* **by** *force*

lemma *ofsm-table-fixpoint* :
assumes $i \leq j$

and $\bigwedge q\ .\ q \in states\ M \Longrightarrow ofsm\text{-}table\ M\ f\ (Suc\ i)\ q = ofsm\text{-}table\ M\ f\ i\ q$
and $q \in states\ M$
shows $ofsm\text{-}table\ M\ f\ j\ q = ofsm\text{-}table\ M\ f\ i\ q$
proof −

have $*$: $\bigwedge k\ .\ k \geq i \Longrightarrow (\bigwedge q\ .\ q \in states\ M \Longrightarrow ofsm\text{-}table\ M\ f\ (Suc\ k)\ q = ofsm\text{-}table\ M\ f\ k\ q)$
proof −

fix k :: *nat* **assume** $k \geq i$
then show $\bigwedge q\ .\ q \in states\ M \Longrightarrow ofsm\text{-}table\ M\ f\ (Suc\ k)\ q = ofsm\text{-}table\ M\ f\ k\ q$
proof (*induction k*)
case *0*
then show *?case* **using** *assms*(*2*) **by** *auto*
next
case (*Suc k*)

show $ofsm\text{-}table\ M\ f\ (Suc\ (Suc\ k))\ q = ofsm\text{-}table\ M\ f\ (Suc\ k)\ q$
proof (*cases* $i = Suc\ k$)
case *True*
then show *?thesis* **using** *assms*(*2*)[*OF* ‹$q \in states\ M$›] **by** *simp*
next
case *False*
then have $i \leq k$
using ‹$i \leq Suc\ k$› **by** *auto*

have *h-obs-state*: $\bigwedge q\ x\ y\ qT\ .\ h\text{-}obs\ M\ q\ x\ y = Some\ qT \Longrightarrow qT \in states\ M$
using *h-obs-state* **by** *fastforce*

show *?thesis*
proof (*rule ccontr*)
assume $ofsm\text{-}table\ M\ f\ (Suc\ (Suc\ k))\ q \neq ofsm\text{-}table\ M\ f\ (Suc\ k)\ q$
moreover have $ofsm\text{-}table\ M\ f\ (Suc\ (Suc\ k))\ q \subseteq ofsm\text{-}table\ M\ f\ (Suc\ k)\ q$
using *ofsm-table-subset*
by (*metis* (*full-types*) *Suc-n-not-le-n nat-le-linear*)
ultimately obtain q' **where** $q' \notin \{q' \in ofsm\text{-}table\ M\ f\ (Suc\ k)\ q\ .\ \forall\ x \in inputs\ M\ .\ \forall\ y \in outputs\ M\ .\ (case\ h\text{-}obs\ M\ q\ x\ y\ of\ Some\ qT \Rightarrow (case\ h\text{-}obs\ M\ q'\ x\ y\ of\ Some\ qT' \Rightarrow ofsm\text{-}table\ M\ f\ (Suc\ k)\ qT = ofsm\text{-}table\ M\ f\ (Suc\ k)\ qT' \mid None \Rightarrow False) \mid None \Rightarrow h\text{-}obs\ M\ q'\ x\ y = None)\ \}$
and $q' \in ofsm\text{-}table\ M\ f\ (Suc\ k)\ q$
using *ofsm-table.simps*(*2*)[*of M f Suc k q*] **unfolding** *Let-def* **by** *blast*
then have $\neg(\forall\ x \in inputs\ M\ .\ \forall\ y \in outputs\ M\ .\ (case\ h\text{-}obs\ M\ q\ x\ y\ of\ Some\ qT \Rightarrow (case\ h\text{-}obs\ M\ q'\ x\ y\ of\ Some\ qT' \Rightarrow ofsm\text{-}table\ M\ f\ (Suc\ k)\ qT = ofsm\text{-}table\ M\ f\ (Suc\ k)\ qT' \mid None \Rightarrow False) \mid None \Rightarrow h\text{-}obs\ M\ q'\ x\ y = None))$
by *blast*
then obtain $x\ y$ **where** $x \in inputs\ M$ **and** $y \in outputs\ M$ **and** $\neg$ (*case*

h-obs M q x y of Some qT ⇒ (*case h-obs M q′ x y of Some qT′* ⇒ *ofsm-table M f* (*Suc k*) *qT* = *ofsm-table M f* (*Suc k*) *qT′* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q′ x y* = *None*)
by *blast*
then consider ∃ *qT qT′* . *h-obs M q x y* = *Some qT* ∧ *h-obs M q′ x y* = *Some qT′* ∧ *ofsm-table M f* (*Suc k*) *qT* ≠ *ofsm-table M f* (*Suc k*) *qT′* |
(*h-obs M q x y* = *None* ⟷ *h-obs M q′ x y* ≠ *None*)
unfolding *ofsm-table-case-helper-neg* **by** *blast*
then show *False* **proof** *cases*
case *1*
then obtain *qT qT′* **where** *h-obs M q x y* = *Some qT* **and** *h-obs M q′ x y* = *Some qT′* **and** *ofsm-table M f* (*Suc k*) *qT* ≠ *ofsm-table M f* (*Suc k*) *qT′*
by *blast*
then have *ofsm-table M f k qT* ≠ *ofsm-table M f k qT′*
using *Suc.IH*[*OF h-obs-state*[*OF* ‹*h-obs M q x y* = *Some qT*›] ‹*i* ≤ *k*›]
Suc.IH[*OF h-obs-state*[*OF* ‹*h-obs M q′ x y* = *Some qT′*›] ‹*i* ≤ *k*›]
by *fast*
moreover have *q′* ∈ *ofsm-table M f k q*
using *ofsm-table-subset*[*of k Suc k*] ‹*q′* ∈ *ofsm-table M f* (*Suc k*) *q*› **by** *force*
ultimately have *ofsm-table M f* (*Suc k*) *q* ≠ *ofsm-table M f k q*
using ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*› ‹*h-obs M q x y* = *Some qT*› ‹*h-obs M q′ x y* = *Some qT′*›
unfolding *ofsm-table.simps*(*2*) *Let-def* **by** *force*
then show *?thesis*
using *Suc.IH*[*OF Suc.prems*(*1*) ‹*i* ≤ *k*›] **by** *simp*
next
case *2*
then have ¬ (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q′ x y of Some qT′* ⇒ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q′ x y* = *None*)
unfolding *ofsm-table-case-helper-neg* **by** *blast*
moreover have *q′* ∈ *ofsm-table M f k q*
using *ofsm-table-subset*[*of k Suc k*] ‹*q′* ∈ *ofsm-table M f* (*Suc k*) *q*› **by** *force*
ultimately have *ofsm-table M f* (*Suc k*) *q* ≠ *ofsm-table M f k q*
using ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*›
unfolding *ofsm-table.simps*(*2*) *Let-def* **by** *force*
then show *?thesis*
using *Suc.IH*[*OF Suc.prems*(*1*) ‹*i* ≤ *k*›] **by** *simp*
qed
qed
qed
qed
qed

show *?thesis*
using *assms*(*1*) **proof** (*induction j*)
case *0*

```
    then show ?case by auto
  next
    case (Suc j)

    show ?case proof (cases i = Suc j)
      case True
      then show ?thesis by simp
    next
      case False
      then have i ≤ j
        using Suc.prems(1) by auto
      then have ofsm-table M f j q = ofsm-table M f i q
        using Suc.IH by auto
      moreover have ofsm-table M f (Suc j) q = ofsm-table M f j q
        using *[OF ‹i≤j› ‹q∈states M›] by assumption
      ultimately show ?thesis
        by blast
    qed
  qed
qed
```

```
function ofsm-table-fix :: ('a,'b,'c) fsm ⇒ ('a ⇒ 'a set) ⇒ nat ⇒ 'a ⇒ 'a set
where
  ofsm-table-fix M f k = (let
    cur-table = ofsm-table M (λq. f q ∩ states M) k;
    next-table = ofsm-table M (λq. f q ∩ states M) (Suc k)
  in if (∀ q ∈ states M . cur-table q = next-table q)
    then cur-table
    else ofsm-table-fix M f (Suc k))
  by pat-completeness auto
termination
proof −
  {
    fix M :: ('a,'b,'c) fsm
    and f :: ('a ⇒ 'a set)
    and k :: nat

    define f' where f': f' = (λq. f q ∩ states M)

    assume ∃ q∈FSM.states M. ofsm-table M (λq. f q ∩ states M) k q ≠ ofsm-table
M (λq. f q ∩ states M) (Suc k) q
    then obtain q where q ∈ states M
                  and ofsm-table M f' k q ≠ ofsm-table M f' (Suc k) q
      unfolding f' by blast

      have *: ⋀ k . (∑ q∈FSM.states M. card (ofsm-table M f' k q)) = card
(ofsm-table M f' k q) + (∑ q∈FSM.states M − {q}. card (ofsm-table M f' k q))
```

using ‹$q \in$ *states M*› **by** (*meson fsm-states-finite sum.remove*)

have $\bigwedge$ *q* . *ofsm-table M f′* (*Suc k*) *q* $\subseteq$ *ofsm-table M f′ k q*
using *ofsm-table-subset*[*of k Suc k M*] **by** *auto*
moreover have $\bigwedge$ *q* . *finite* (*ofsm-table M f′ k q*)
proof −
fix *q*
have *ofsm-table M* (λ*q*. *f q* $\cap$ *states M*) *k q* $\subseteq$ *ofsm-table M* (λ*q*. *f q* $\cap$ *states M*) *0 q*
using *ofsm-table-subset*[*of 0 k M* (λ*q*. *f q* $\cap$ *FSM.states M*) *q*] **by** *auto*
then have *ofsm-table M f′ k q* $\subseteq$ *states M*
unfolding *f′*
using *ofsm-table-non-state*[*of q M* (λ*q*. *f q* $\cap$ *FSM.states M*) *k*]
by *force*
then show *finite* (*ofsm-table M f′ k q*)
using *fsm-states-finite finite-subset* **by** *auto*
qed
ultimately have $\bigwedge$ *q* . *card* (*ofsm-table M f′* (*Suc k*) *q*) $\leq$ *card* (*ofsm-table M f′ k q*)
by (*simp add*: *card-mono*)
then have ($\sum$ *q*$\in$*FSM.states M* − {*q*}. *card* (*ofsm-table M f′* (*Suc k*) *q*)) $\leq$ ($\sum$ *q*$\in$*FSM.states M* − {*q*}. *card* (*ofsm-table M f′ k q*))
by (*simp add*: *sum-mono*)
moreover have *card* (*ofsm-table M f′* (*Suc k*) *q*) < *card* (*ofsm-table M f′ k q*)
using ‹*ofsm-table M f′ k q* $\neq$ *ofsm-table M f′* (*Suc k*) *q*› ‹*ofsm-table M f′* (*Suc k*) *q* $\subseteq$ *ofsm-table M f′ k q*› ‹*finite* (*ofsm-table M f′ k q*)›
by (*metis psubsetI psubset-card-mono*)
ultimately have ($\sum$ *q*$\in$*FSM.states M*. *card* (*ofsm-table M* (λ*q*. *f q* $\cap$ *states M*) (*Suc k*) *q*)) < ($\sum$ *q*$\in$*FSM.states M*. *card* (*ofsm-table M* (λ*q*. *f q* $\cap$ *states M*) *k q*))
unfolding *f′*[*symmetric*] $*$
by *linarith*
} **note** *t* = *this*

show *?thesis*
apply (*relation measure* (λ (*M*, *f*, *k*) . $\sum$ *q* $\in$ *states M* . *card* (*ofsm-table M* (λ*q*. *f q* $\cap$ *states M*) *k q*)))
apply (*simp del*: *h-obs.simps ofsm-table.simps*)+
by (*erule t*)
qed

lemma *ofsm-table-restriction-to-states* :
assumes $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *f q* $\subseteq$ *states M*
and *q* $\in$ *states M*
shows *ofsm-table M f k q* = *ofsm-table M* (λ*q* . *f q* $\cap$ *states M*) *k q*
using *assms*(*2*) **proof** (*induction k arbitrary*: *q*)
case *0*
then show *?case* **using** *assms*(*1*) **by** *auto*

next
  case (*Suc k*)

  have $\bigwedge$ *x y q q′* . (*case h-obs M q x y of None* ⇒ *h-obs M q′ x y* = *None* | *Some qT* ⇒ (*case h-obs M q′ x y of None* ⇒ *False* | *Some qT′* ⇒ *ofsm-table M f k qT* = *ofsm-table M f k qT′*))
                = (*case h-obs M q x y of None* ⇒ *h-obs M q′ x y* = *None* | *Some qT* ⇒ (*case h-obs M q′ x y of None* ⇒ *False* | *Some qT′* ⇒ *ofsm-table M* (λ*q* . *f q* ∩ *states M*) *k qT* = *ofsm-table M* (λ*q* . *f q* ∩ *states M*) *k qT′*))
      (**is** $\bigwedge$ *x y q q′* . *?C1 x y q q′* = *?C2 x y q q′*)
  proof −
    fix *x y q q′*
    show *?C1 x y q q′* = *?C2 x y q q′*
      using *Suc.IH*[*OF h-obs-state*, *of q x y*]
      using *Suc.IH*[*OF h-obs-state*, *of q′ x y*]
      by (*cases h-obs M q x y*; *cases h-obs M q′ x y*; *auto*)
  qed
  then show *?case*
    unfolding *ofsm-table.simps Let-def Suc.IH*[*OF Suc.prems*]
    by *blast*
qed

lemma *ofsm-table-fix-length* :
  assumes $\bigwedge$ *q* . *q* ∈ *states M* ⟹ *f q* ⊆ *states M*
  obtains *k* **where** $\bigwedge$ *q* . *q* ∈ *states M* ⟹ *ofsm-table-fix M f 0 q* = *ofsm-table M f k q* **and** $\bigwedge$ *q k′* . *q* ∈ *states M* ⟹ *k′* ≥ *k* ⟹ *ofsm-table M f k′ q* = *ofsm-table M f k q*
proof −

  have ∃ *k* . ∀ *q* ∈ *states M* . ∀ *k′* ≥ *k* . *ofsm-table M f k′ q* = *ofsm-table M f k q*
  proof −

    have ∃ *fp* . ∀ *q k′* . *q* ∈ *states M* ⟶ *k′* ≥ (*fp q*) ⟶ *ofsm-table M f k′ q* = *ofsm-table M f* (*fp q*) *q*
    proof
      fix *q*
      let *?assignK* = λ *q* . *SOME k* . ∀ *k′* ≥ *k* . *ofsm-table M f k′ q* = *ofsm-table M f k q*

      have $\bigwedge$ *q k′* . *q* ∈ *states M* ⟹ *k′* ≥ *?assignK q* ⟹ *ofsm-table M f k′ q* = *ofsm-table M f* (*?assignK q*) *q*
      proof −
        fix *q k′* **assume** *q* ∈ *states M* **and** *k′* ≥ *?assignK q*
        then have *p1*: *finite* (*ofsm-table M f 0 q*)
          using *fsm-states-finite assms*(*1*)
          using *infinite-super* **by** *fastforce*

        have ∃ *k* . ∀ *k′* ≥ *k* . *ofsm-table M f k′ q* = *ofsm-table M f k q*

using *finite-subset-mapping-limit*[*of* λ *k* . *ofsm-table M f k q*, *OF p1 ofsm-table-subset*] **by** *metis*
have $\forall$ $k' \geq$ (*?assignK q*) . *ofsm-table M f k' q* = *ofsm-table M f* (*?assignK q*) *q*
using *someI-ex*[*of* λ *k* . $\forall$ $k' \geq k$. *ofsm-table M f k' q* = *ofsm-table M f k q*, *OF* ‹$\exists$ *k* . $\forall$ $k' \geq k$. *ofsm-table M f k' q* = *ofsm-table M f k q*›] **by** *assumption*
then show *ofsm-table M f k' q* = *ofsm-table M f* (*?assignK q*) *q*
using ‹$k' \geq$ *?assignK q*› **by** *blast*
qed
then show $\forall$ *q k'*. *q* $\in$ *states M* $\longrightarrow$ *?assignK q* $\leq k' \longrightarrow$ *ofsm-table M f k' q* = *ofsm-table M f* (*?assignK q*) *q*
by *blast*
qed
then obtain *assignK* **where** *assignK-prop*: $\bigwedge$ *q k'* . *q* $\in$ *states M* $\Longrightarrow k' \geq$ *assignK q* $\Longrightarrow$ *ofsm-table M f k' q* = *ofsm-table M f* (*assignK q*) *q*
by *blast*

have *finite* (*assignK* ‘ *states M*)
by (*simp add*: *fsm-states-finite*)
moreover have *assignK* ‘ *FSM.states M* $\neq$ {}
using *fsm-initial* **by** *auto*
ultimately obtain *k* **where** *k* $\in$ (*assignK* ‘ *states M*) **and** $\bigwedge$ *k'* . *k'* $\in$ (*assignK* ‘ *states M*) $\Longrightarrow k' \leq k$
using *Max-elem*[*OF* ‹*finite* (*assignK* ‘ *states M*)› ‹*assignK* ‘ *FSM.states M* $\neq$ {}›] **by** (*meson eq-Max-iff*)

have $\bigwedge$ *q k'* . *q* $\in$ *states M* $\Longrightarrow k' \geq k \Longrightarrow$ *ofsm-table M f k' q* = *ofsm-table M f k q*
proof −
fix *q k'* **assume** $k' \geq k$ **and** *q* $\in$ *states M*
then have $k' \geq$ *assignK q*
using ‹$\bigwedge$ *k'* . *k'* $\in$ (*assignK* ‘ *states M*) $\Longrightarrow k' \leq k$›
using *dual-order.trans* **by** *auto*
then show *ofsm-table M f k' q* = *ofsm-table M f k q*
using *assignK-prop* ‹$\bigwedge$*k'*. *k'* $\in$ *assignK* ‘ *FSM.states M* $\Longrightarrow k' \leq k$› ‹*q* $\in$ *FSM.states M*› **by** *blast*
qed
then show *?thesis*
by *blast*
qed
then obtain *k* **where** *k-prop*: $\bigwedge$ *q k'* . *q* $\in$ *states M* $\Longrightarrow k' \geq k \Longrightarrow$ *ofsm-table M f k' q* = *ofsm-table M f k q*
by *blast*
then have $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table M f k q* = *ofsm-table M f* (*Suc k*) *q*
by (*metis* (*full-types*) *le-SucI order-refl*)

let *?ks* = (*Set.filter* (λ *k* . $\forall$ *q* $\in$ *states M* . *ofsm-table M f k q* = *ofsm-table M f*

(*Suc k*) *q*) {..*k*})
have *f1*: *finite ?ks*
by *simp*
moreover have *f2*: *?ks* ≠ {}
apply (*clarsimp simp only*: *Set.filter-eq simp flip*: *ex-in-conv*)
using ‹⋀*q*. *q* ∈ *states M* ⟹ *ofsm-table M f k q* = *ofsm-table M f* (*Suc k*) *q*›
apply *blast*
done
ultimately obtain *kMin* **where** *kMin* ∈ *?ks* **and** ⋀ *k′* . *k′* ∈ *?ks* ⟹ *k′* ≥ *kMin*
using *Min-elem*[*OF f1 f2*] **by** (*meson eq-Min-iff*)

have *k1*: ⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table M f* (*Suc kMin*) *q* = *ofsm-table M f kMin q*
using ‹*kMin* ∈ *?ks*›
by (*metis* (*mono-tags*, *lifting*) *Set.filter-eq mem-Collect-eq*)

have *k2*: ⋀ *k′* . (⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table M f k′ q* = *ofsm-table M f* (*Suc k′*) *q*) ⟹ *k′* ≥ *kMin*
proof −
fix *k′* **assume** ⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table M f k′ q* = *ofsm-table M f* (*Suc k′*) *q*
show *k′* ≥ *kMin* **proof** (*cases k′* ∈ *?ks*)
case *True*
then show *?thesis* **using** ‹⋀ *k′* . *k′* ∈ *?ks* ⟹ *k′* ≥ *kMin*› **by** *blast*
next
case *False*
then have *k′* > *k*
using ‹⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table M f k′ q* = *ofsm-table M f* (*Suc k′*) *q*›
by (*smt* (*verit*, *best*) *Set.filter-eq atMost-iff dual-order.refl mem-Collect-eq nat-less-le*
nat-neq-iff)
moreover have *kMin* ≤ *k*
using ‹*kMin* ∈ *?ks*› **by** *auto*
ultimately show *?thesis*
by *auto*
qed
qed

have ⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table-fix M f 0 q* = *ofsm-table M* (λ *q* . *f q* ∩ *states M*) *kMin q*
proof −
fix *q* **assume** *q* ∈ *states M*
show *ofsm-table-fix M f 0 q* = *ofsm-table M* (λ *q* . *f q* ∩ *states M*) *kMin q*
proof (*cases kMin*)
case *0*

have ∀ *q*∈*FSM.states M*. *ofsm-table M* (λ*q*. *f q* ∩ *FSM.states M*) *0 q* =

```
ofsm-table M (λq. f q ∩ FSM.states M) (Suc 0) q
      using k1
      using ofsm-table-restriction-to-states[of M f -, OF assms(1) - ]
      using 0 by blast
    then show ?thesis
      apply (subst ofsm-table-fix.simps)
      unfolding  0 Let-def by force
  next
    case (Suc kMin')

    have ∗: ⋀ i . i < kMin ⟹ ¬(∀  q ∈ states M . ofsm-table M f i q = ofsm-table
M f (Suc i) q)
      using k2
      by (meson leD)
    have ⋀ i . i < kMin ⟹ ofsm-table-fix M f 0 = ofsm-table-fix M f (Suc i)
    proof −
      fix i assume i < kMin
      then show ofsm-table-fix M f 0 = ofsm-table-fix M f (Suc i)
      proof (induction i)
        case 0
        show ?case
          using ∗[OF 0]  ofsm-table-restriction-to-states[of - f, OF assms(1) - ]
unfolding ofsm-table-fix.simps[of M f 0] Let-def
          by (metis (no-types, lifting))
      next
        case (Suc i)
        then have i < kMin by auto

        have ofsm-table-fix M f (Suc i) = ofsm-table-fix M f (Suc (Suc i))
          using ∗[OF ‹Suc i < kMin›] ofsm-table-restriction-to-states[of - f, OF
assms(1) - ] unfolding ofsm-table-fix.simps[of M f Suc i] Let-def by metis
        then show ?case using Suc.IH[OF ‹i < kMin›]
          by presburger
      qed
    qed
    then have ofsm-table-fix M f 0 = ofsm-table-fix M f kMin
      using Suc by blast
    moreover have ofsm-table-fix M f kMin q = ofsm-table M f kMin q
    proof −
      have ∀ q∈FSM.states M. ofsm-table M (λq. f q ∩ FSM.states M) kMin q
= ofsm-table M (λq. f q ∩ FSM.states M) (Suc kMin) q
        using ofsm-table-restriction-to-states[of - f, OF assms(1) - ]
        using k1 by blast
      then show ?thesis
        using ofsm-table-restriction-to-states[of - f, OF assms(1) - ] ‹q ∈ states
M›
        unfolding ofsm-table-fix.simps[of M f kMin] Let-def
        by presburger
    qed
```

```
      ultimately show ?thesis
        using ofsm-table-restriction-to-states[of - f, OF assms(1) ‹q ∈ states M›]
        by presburger
    qed
  qed
  moreover have ⋀ q k′ . q ∈ states M ⟹ k′ ≥ kMin ⟹ ofsm-table M f k′ q
= ofsm-table M f kMin q
    using ofsm-table-fixpoint[OF - k1 ] by blast
  ultimately show ?thesis
    using that[of kMin]
    using ofsm-table-restriction-to-states[of M f, OF assms(1) - ]
    by blast
qed

lemma ofsm-table-containment :
  assumes q ∈ states M
  and     ⋀ q . q ∈ states M ⟹ q ∈ f q
  shows q ∈ ofsm-table M f k q
proof (induction k)
  case 0
  then show ?case using assms by auto
next
  case (Suc k)
  then show ?case
    unfolding ofsm-table.simps Let-def option.case-eq-if
    by auto
qed

lemma ofsm-table-states :
  assumes ⋀ q . q ∈ states M ⟹ f q ⊆ states M
  and     q ∈ states M
shows  ofsm-table M f k q ⊆ states M
proof −
  have ofsm-table M f k q ⊆ ofsm-table M f 0 q
    using ofsm-table-subset[OF le0] by metis
  moreover have ofsm-table M f 0 q ⊆ states M
    using assms
    unfolding ofsm-table.simps(1) by (metis (full-types))
  ultimately show ?thesis
    by blast
qed
```

6.1.1 Properties of Initial Partitions

```
definition equivalence-relation-on-states :: ('a,'b,'c) fsm ⇒ ('a ⇒ 'a set) ⇒ bool
where
  equivalence-relation-on-states M f =
      (equiv (states M) {(q1,q2) | q1 q2 . q1 ∈ states M ∧ q2 ∈ f q1}
       ∧ (∀  q ∈ states M . f q ⊆ states M))
```

lemma *equivalence-relation-on-states-refl* :
assumes *equivalence-relation-on-states M f*
and $q \in$ *states M*
shows $q \in f\ q$
using *assms* **unfolding** *equivalence-relation-on-states-def equiv-def refl-on-def* **by** *blast*

lemma *equivalence-relation-on-states-sym* :
assumes *equivalence-relation-on-states M f*
and $q1 \in$ *states M*
and $q2 \in f\ q1$
shows $q1 \in f\ q2$
using *assms* **unfolding** *equivalence-relation-on-states-def equiv-def sym-def* **by** *blast*

lemma *equivalence-relation-on-states-trans* :
assumes *equivalence-relation-on-states M f*
and $q1 \in$ *states M*
and $q2 \in f\ q1$
and $q3 \in f\ q2$
shows $q3 \in f\ q1$
proof −
have $(q1,q2) \in \{(q1,q2) \mid q1\ q2\ .\ q1 \in$ *states M* $\wedge\ q2 \in f\ q1\}$
using *assms*(*2*,*3*) **by** *blast*
then have $q2 \in$ *states M*
using *assms*(*1*) **unfolding** *equivalence-relation-on-states-def*
by *auto*
then have $(q2,q3) \in \{(q1,q2) \mid q1\ q2\ .\ q1 \in$ *states M* $\wedge\ q2 \in f\ q1\}$
using *assms*(*4*) **by** *blast*
moreover have *trans* $\{(q1,q2) \mid q1\ q2\ .\ q1 \in$ *states M* $\wedge\ q2 \in f\ q1\}$
using *assms*(*1*) **unfolding** *equivalence-relation-on-states-def equiv-def* **by** *auto*
ultimately show *?thesis*
using ‹$(q1,q2) \in \{(q1,q2) \mid q1\ q2\ .\ q1 \in$ *states M* $\wedge\ q2 \in f\ q1\}$›
unfolding *trans-def* **by** *blast*
qed

lemma *equivalence-relation-on-states-ran* :
assumes *equivalence-relation-on-states M f*
and $q \in$ *states M*
shows $f\ q \subseteq$ *states M*
using *assms* **unfolding** *equivalence-relation-on-states-def* **by** *blast*

6.1.2 Properties of OFSM tables for initial partitions based on equivalence relations

lemma *h-obs-io* :
assumes *h-obs M q x y* = *Some q′*
shows $x \in$ *inputs M* **and** $y \in$ *outputs M*

```
proof −
  have snd ‘ Set.filter (λ (y′,q′) . y′ = y) (h M (q,x)) ≠ {}
    using assms by (auto simp add: Let-def card-1-singleton-iff split: if-splits)
  then show x ∈ inputs M and y ∈ outputs M
    unfolding h-simps
    using fsm-transition-input fsm-transition-output
    by fastforce+
qed


lemma ofsm-table-language :
  assumes q′ ∈ ofsm-table M f k q
  and     length io ≤ k
  and     q ∈ states M
  and     equivalence-relation-on-states M f
shows is-in-language M q io ⟷ is-in-language M q′ io
and   is-in-language M q io ⟹ (after M q′ io) ∈ f (after M q io)
proof −
  have (is-in-language M q io ⟷ is-in-language M q′ io) ∧ (is-in-language M q
io ⟶ (after M q′ io) ∈ f (after M q io))
    using assms(1,2,3)
  proof (induction k arbitrary: q q′ io)
    case 0
    then have io = [] by auto
    then show ?case
      using 0.prems(1,3) by auto
  next
    case (Suc k)

    show ?case proof (cases length io ≤ k)
      case True
      have ∗: q′ ∈ ofsm-table M f k q
        using ‹q′ ∈ ofsm-table M f (Suc k) q› ofsm-table-subset
        by (metis (full-types) le-SucI order-refl subsetD)
      show ?thesis using Suc.IH[OF ∗ True ‹q ∈ states M›] by assumption
    next
      case False
      then have length io = Suc k
        using ‹length io ≤ Suc k› by auto
      then obtain ioT ioP where io = ioT#ioP
        by (meson length-Suc-conv)
      then have length ioP ≤ k
        using ‹length io ≤ Suc k› by auto

      obtain x y where io = (x,y)#ioP
        using ‹io = ioT#ioP› prod.exhaust-sel
        by fastforce

      have ofsm-table M f (Suc k) q = {q′ ∈ ofsm-table M f k q . ∀ x ∈ inputs M
```

. ∀ y ∈ outputs M . (case h-obs M q x y of Some qT ⇒ (case h-obs M q′ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q′ x y = None) }
unfolding *ofsm-table.simps Let-def* **by** *blast*
then have q′ ∈ ofsm-table M f k q
and ∗: ⋀ x y . x ∈ inputs M ⟹ y ∈ outputs M ⟹ (case h-obs M q x y of Some qT ⇒ (case h-obs M q′ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q′ x y = None)
using ‹q′ ∈ ofsm-table M f (Suc k) q› **by** *blast+*

show *?thesis*
unfolding ‹io = (x,y)#ioP›
proof −
have *is-in-language* M q ((x,y)#ioP) ⟹ *is-in-language* M q′ ((x,y)#ioP) ∧ *after* M q′ ((x,y)#ioP) ∈ f (*after* M q ((x,y)#ioP))
proof −
assume *is-in-language* M q ((x,y)#ioP)

then obtain qT **where** *h-obs* M q x y = Some qT **and** *is-in-language* M qT ioP
by (*metis case-optionE is-in-language.simps*(*2*))
moreover have (*case h-obs* M q x y *of* Some qT ⇒ (*case h-obs* M q′ x y *of* Some qT′ ⇒ *ofsm-table* M f k qT = *ofsm-table* M f k qT′ | None ⇒ False) | None ⇒ *h-obs* M q′ x y = None)
using ∗[*of* x y, *OF h-obs-io*[*OF* ‹*h-obs* M q x y = Some qT›]] **.**
ultimately obtain qT′ **where** *h-obs* M q′ x y = Some qT′ **and** *ofsm-table* M f k qT = *ofsm-table* M f k qT′
using *ofsm-table-case-helper*[*of* M q′ x y f k q]
unfolding *ofsm-table.simps* **by** *force*
then have qT′ ∈ *ofsm-table* M f k qT
using *ofsm-table-containment*[*OF h-obs-state equivalence-relation-on-states-refl*[*OF* ‹*equivalence-relation-on-states* M f›]]
by *metis*

have (*is-in-language* M qT ioP) = (*is-in-language* M qT′ ioP)
(*is-in-language* M qT ioP ⟶ *after* M qT′ ioP ∈ f (*after* M qT ioP))
using *Suc.IH*[*OF* ‹qT′ ∈ *ofsm-table* M f k qT› ‹*length* ioP ≤ k› *h-obs-state*[*OF* ‹*h-obs* M q x y = Some qT›]]
by *blast+*

have (*is-in-language* M qT′ ioP)
using ‹(*is-in-language* M qT ioP) = (*is-in-language* M qT′ ioP)› ‹*is-in-language* M qT ioP›
by *auto*
then have *is-in-language* M q′ ((x,y)#ioP)
unfolding *is-in-language.simps* ‹*h-obs* M q′ x y = Some qT′› **by** *auto*
moreover have *after* M q′ ((x,y)#ioP) ∈ f (*after* M q ((x,y)#ioP))
unfolding *after.simps* ‹*h-obs* M q′ x y = Some qT′› ‹*h-obs* M q x y = Some qT›

using ‹(*is-in-language M qT ioP* ⟶ *after M qT′ ioP* ∈ *f* (*after M qT ioP*))› ‹*is-in-language M qT ioP*›
by *auto*
ultimately show *is-in-language M q′* ((*x*,*y*)#*ioP*) ∧ *after M q′* ((*x*,*y*)#*ioP*) ∈ *f* (*after M q* ((*x*,*y*)#*ioP*))
by *blast*
qed
moreover have *is-in-language M q′* ((*x*,*y*)#*ioP*) ⟹ *is-in-language M q* ((*x*,*y*)#*ioP*)
proof −
assume *is-in-language M q′* ((*x*,*y*)#*ioP*)

then obtain *qT′* **where** *h-obs M q′ x y* = *Some qT′* **and** *is-in-language M qT′ ioP*
by (*metis case-optionE is-in-language.simps*(*2*))
moreover have (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q′ x y of Some qT′* ⇒ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q′ x y* = *None*)
using ∗[*of x y*, *OF h-obs-io*[*OF* ‹*h-obs M q′ x y* = *Some qT′*›]] **.**
ultimately obtain *qT* **where** *h-obs M q x y* = *Some qT* **and** *ofsm-table M f k qT* = *ofsm-table M f k qT′*
using *ofsm-table-case-helper*[*of M q′ x y f k q*]
unfolding *ofsm-table.simps* **by** *force*
then have *qT* ∈ *ofsm-table M f k qT′*
using *ofsm-table-containment*[*OF h-obs-state equivalence-relation-on-states-refl*[*OF* ‹*equivalence-relation-on-states M f*›]]
by *metis*

have (*is-in-language M qT ioP*) = (*is-in-language M qT′ ioP*)
using *Suc.IH*[*OF* ‹*qT* ∈ *ofsm-table M f k qT′*› ‹*length ioP* ≤ *k*› *h-obs-state*[*OF* ‹*h-obs M q′ x y* = *Some qT′*›]]
by *blast*
then have *is-in-language M qT ioP*
using ‹*is-in-language M qT′ ioP*›
by *auto*
then show *is-in-language M q* ((*x*,*y*)#*ioP*)
unfolding *is-in-language.simps* ‹*h-obs M q x y* = *Some qT*› **by** *auto*
qed
ultimately show *is-in-language M q* ((*x*, *y*) # *ioP*) = *is-in-language M q′* ((*x*, *y*) # *ioP*) ∧ (*is-in-language M q* ((*x*, *y*) # *ioP*) ⟶ *after M q′* ((*x*, *y*) # *ioP*) ∈ *f* (*after M q* ((*x*, *y*) # *ioP*)))
by *blast*
qed
qed
qed
then show *is-in-language M q io* = *is-in-language M q′ io* **and** (*is-in-language M q io* ⟹ *after M q′ io* ∈ *f* (*after M q io*))
by *blast*+
qed

lemma *after-is-state-is-in-language* :
  assumes $q \in$ *states M*
  and *is-in-language M q io*
  shows *FSM.after M q io* $\in$ *states M*
  using *assms*
proof (*induction io arbitrary*: *q*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons a io*)
  then obtain *x y* **where** *a* = (*x*,*y*) **using** *prod.exhaust* **by** *metis*
  show *?case*
    using ‹*is-in-language M q* (*a* # *io*)› *Cons.IH*[*OF h-obs-state*[*of M q x y*]]
    unfolding ‹*a* = (*x*,*y*)›
    unfolding *after.simps is-in-language.simps*
    by (*metis option.case-eq-if option.exhaust-sel*)
qed

lemma *ofsm-table-elem* :
  assumes $q \in$ *states M*
  and $q' \in$ *states M*
  and *equivalence-relation-on-states M f*
  and $\bigwedge$ *io* . *length io* $\leq k \Longrightarrow$ *is-in-language M q io* $\longleftrightarrow$ *is-in-language M q′ io*
  and $\bigwedge$ *io* . *length io* $\leq k \Longrightarrow$ *is-in-language M q io* $\Longrightarrow$ (*after M q′ io*) $\in f$ (*after M q io*)
shows $q' \in$ *ofsm-table M f k q*
  using *assms*(*1*,*2*,*4*,*5*) **proof** (*induction k arbitrary*: *q q′*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc k*)

  have $q' \in$ *ofsm-table M f k q*
    using *Suc.IH*[*OF Suc.prems*(*1*,*2*)] *Suc.prems*(*3*,*4*) **by** *auto*

  moreover have $\bigwedge$ *x y* . *x* $\in$ *inputs M* $\Longrightarrow$ *y* $\in$ *outputs M* $\Longrightarrow$ (*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q′ x y of Some qT′* $\Rightarrow$ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q′ x y* = *None*)
  proof −
    fix *x y* **assume** *x* $\in$ *inputs M* **and** *y* $\in$ *outputs M*
    show (*case h-obs M q x y of Some qT* $\Rightarrow$ (*case h-obs M q′ x y of Some qT′* $\Rightarrow$ *ofsm-table M f k qT* = *ofsm-table M f k qT′* | *None* $\Rightarrow$ *False*) | *None* $\Rightarrow$ *h-obs M q′ x y* = *None*)
    proof (*cases* $\exists$ *qT qT′* . *h-obs M q x y* = *Some qT* $\wedge$ *h-obs M q′ x y* = *Some*

qT′)
case *True*
then obtain *qT qT′* **where** *h-obs M q x y = Some qT* **and** *h-obs M q′ x y = Some qT′*
by *blast*

have ∗: ⋀ *io* . *length io* ≤ *k* ⟹ *is-in-language M qT io = is-in-language M qT′ io*
proof −
fix *io* :: (′*b* × ′*c*) *list*
assume *length io* ≤ *k*

have *is-in-language M qT io = is-in-language M q* ([(*x*,*y*)]@*io*)
using ‹*h-obs M q x y = Some qT*› **by** *auto*
moreover have *is-in-language M qT′ io = is-in-language M q′* ([(*x*,*y*)]@*io*)
using ‹*h-obs M q′ x y = Some qT′*› **by** *auto*
ultimately show *is-in-language M qT io = is-in-language M qT′ io*
using *Suc.prems*(*3*) ‹*length io* ≤ *k*›
by (*metis append.left-neutral append-Cons length-Cons not-less-eq-eq*)
qed

have *ofsm-table M f k qT = ofsm-table M f k qT′*
proof

have *qT* ∈ *states M*
using *h-obs-state*[*OF* ‹*h-obs M q x y = Some qT*›] **.**
have *qT′* ∈ *states M*
using *h-obs-state*[*OF* ‹*h-obs M q′ x y = Some qT′*›] **.**

show *ofsm-table M f k qT* ⊆ *ofsm-table M f k qT′*
proof
fix *s* **assume** *s* ∈ *ofsm-table M f k qT*
then have *s* ∈ *states M*
using *ofsm-table-subset*[*of 0 k M f qT*] *equivalence-relation-on-states-ran*[*OF assms*(*3*) ‹*qT* ∈ *states M*›] ‹*qT* ∈ *states M*› **by** *auto*
have ∗∗: (⋀*io*. *length io* ≤ *k* ⟹ *is-in-language M qT′ io = is-in-language M s io*)
using *ofsm-table-language*(*1*)[*OF* ‹*s* ∈ *ofsm-table M f k qT*› - ‹*qT*∈ *states M*› *assms*(*3*)] ∗ **by** *blast*
have ∗∗∗: (⋀*io*. *length io* ≤ *k* ⟹ *is-in-language M qT′ io* ⟹ *after M s io* ∈ *f* (*after M qT′ io*))
proof −
fix *io* **assume** *length io* ≤ *k* **and** *is-in-language M qT′ io*
then have *is-in-language M qT io*
using ∗ **by** *blast*
then have *after M s io* ∈ *f* (*after M qT io*)
using *ofsm-table-language*(*2*)[*OF* ‹*s* ∈ *ofsm-table M f k qT*› ‹*length io* ≤ *k*› ‹*qT*∈ *states M*› *assms*(*3*)]
by *blast*

have *after M qT io = after M q ((x,y)#io)*
using ‹*h-obs M q x y = Some qT*› **by** *auto*
moreover have *after M qT′ io = after M q′ ((x,y)#io)*
using ‹*h-obs M q′ x y = Some qT′*› **by** *auto*
moreover have *is-in-language M q ((x,y)#io)*
using ‹*h-obs M q x y = Some qT*› ‹*is-in-language M qT io*› **by** *auto*
ultimately have *after M qT′ io ∈ f (after M qT io)*
using *Suc.prems(4)* ‹*length io ≤ k*›
by (*metis Suc-le-mono length-Cons*)

show *after M s io ∈ f (after M qT′ io)*
using *equivalence-relation-on-states-trans*[*OF* ‹*equivalence-relation-on-states M f*› *after-is-state-is-in-language*[*OF* ‹*qT′ ∈ states M*› ‹*is-in-language M qT′ io*›] *equivalence-relation-on-states-sym*[*OF* ‹*equivalence-relation-on-states M f*› *after-is-state-is-in-language*[*OF* ‹*qT ∈ states M*› ‹*is-in-language M qT io*›] ‹*after M qT′ io ∈ f (after M qT io)*›] ‹*after M s io ∈ f (after M qT io)*›] **.**
qed
show *s ∈ ofsm-table M f k qT′*
using *Suc.IH*[*OF* ‹*qT′ ∈ states M*› ‹*s ∈ states M*› ∗∗ ∗∗∗] **by** *blast*

qed

show *ofsm-table M f k qT′ ⊆ ofsm-table M f k qT*
proof
fix *s* **assume** *s ∈ ofsm-table M f k qT′*
then have *s ∈ states M*
using *ofsm-table-subset*[*of 0 k M f qT′*] *equivalence-relation-on-states-ran*[*OF assms(3)* ‹*qT′ ∈ states M*›] ‹*qT′ ∈ states M*› **by** *auto*
have ∗∗: (⋀*io. length io ≤ k* ⟹ *is-in-language M qT io = is-in-language M s io*)
using *ofsm-table-language(1)*[*OF* ‹*s ∈ ofsm-table M f k qT′*› - ‹*qT′∈ states M*› *assms(3)*] ∗ **by** *blast*
have ∗∗∗: (⋀*io. length io ≤ k* ⟹ *is-in-language M qT io* ⟹ *after M s io ∈ f (after M qT io)*)
proof −
fix *io* **assume** *length io ≤ k* **and** *is-in-language M qT io*
then have *is-in-language M qT′ io*
using ∗ **by** *blast*
then have *after M s io ∈ f (after M qT′ io)*
using *ofsm-table-language(2)*[*OF* ‹*s ∈ ofsm-table M f k qT′*› ‹*length io ≤ k*› ‹*qT′∈ states M*› *assms(3)*]
by *blast*

have *after M qT′ io = after M q′ ((x,y)#io)*
using ‹*h-obs M q′ x y = Some qT′*› **by** *auto*
moreover have *after M qT io = after M q ((x,y)#io)*

```
            using ‹h-obs M q x y = Some qT› by auto
          moreover have is-in-language M q' ((x,y)#io)
            using ‹h-obs M q' x y = Some qT'› ‹is-in-language M qT' io› by auto
          ultimately have after M qT io ∈ f (after M qT' io)
            using Suc.prems(4) ‹length io ≤ k›
             by (metis Suc.prems(3) Suc-le-mono ‹is-in-language M qT io› ‹qT ∈
FSM.states M› after-is-state-is-in-language assms(3) equivalence-relation-on-states-sym
length-Cons)

          show after M s io ∈ f (after M qT io)
         using equivalence-relation-on-states-trans[OF ‹equivalence-relation-on-states
M f› after-is-state-is-in-language[OF ‹qT ∈ states M› ‹is-in-language M qT io›]
                                                    equivalence-relation-on-states-sym[OF
‹equivalence-relation-on-states M f› after-is-state-is-in-language[OF ‹qT' ∈ states
M› ‹is-in-language M qT' io›]
                                                    ‹after M qT io ∈ f (after M qT'
io)›] ‹after M s io ∈ f (after M qT' io)›] .
        qed
        show s ∈ ofsm-table M f k qT
            using Suc.IH[OF ‹qT ∈ states M› ‹s ∈ states M› ** ***] by blast

      qed
    qed
    then show ?thesis
      unfolding ‹h-obs M q x y = Some qT› ‹h-obs M q' x y = Some qT'›
      by auto
  next
    case False
    have h-obs M q x y = None ∧ h-obs M q' x y = None
    proof (rule ccontr)
      assume ¬ (h-obs M q x y = None ∧ h-obs M q' x y = None)
      then have is-in-language M q [(x,y)] ∨ is-in-language M q' [(x,y)]
        unfolding is-in-language.simps
        using option.disc-eq-case(2) by blast
      moreover have is-in-language M q [(x,y)] ≠ is-in-language M q' [(x,y)]
        using False
        by (metis calculation case-optionE is-in-language.simps(2))
      moreover have length [(x,y)] ≤ Suc k
        by auto
      ultimately show False
        using Suc.prems(3) by blast
    qed
    then show ?thesis
      unfolding ofsm-table-case-helper
      by blast
  qed
qed

ultimately show ?case
```

unfolding *Suc ofsm-table.simps Let-def* **by** *force*
qed

lemma *ofsm-table-set* :
assumes $q \in$ *states M*
and *equivalence-relation-on-states M f*
shows *ofsm-table M f k q* $= \{q' \,.\, q' \in$ *states M* $\wedge$ ($\forall$ *io . length io* $\leq k \longrightarrow$ (*is-in-language M q io* $\longleftrightarrow$ *is-in-language M q′ io*) $\wedge$ (*is-in-language M q io* $\longrightarrow$ *after M q′ io* $\in f$ (*after M q io*)))$\}$
using *ofsm-table-language*[*OF - - assms*(*1*,*2*)]
ofsm-table-states[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*2*)] *assms*(*1*)]
ofsm-table-elem[*OF assms*(*1*) - *assms*(*2*)]
by *blast*

lemma *ofsm-table-set-observable* :
assumes *observable M* **and** $q \in$ *states M*
and *equivalence-relation-on-states M f*
shows *ofsm-table M f k q* $= \{q' \,.\, q' \in$ *states M* $\wedge$ ($\forall$ *io . length io* $\leq k \longrightarrow$ (*io* $\in$ *LS M q* $\longleftrightarrow$ *io* $\in$ *LS M q′*) $\wedge$ (*io* $\in$ *LS M q* $\longrightarrow$ *after M q′ io* $\in f$ (*after M q io*)))$\}$
unfolding *ofsm-table-set*[*OF assms*(*2*,*3*)]
unfolding *is-in-language-iff*[*OF assms*(*1*,*2*)]
using *is-in-language-iff*[*OF assms*(*1*)]
by *blast*

lemma *ofsm-table-eq-if-elem* :
assumes *q1* $\in$ *states M* **and** *q2* $\in$ *states M* **and** *equivalence-relation-on-states M f*
shows (*ofsm-table M f k q1* = *ofsm-table M f k q2*) = (*q2* $\in$ *ofsm-table M f k q1*)
proof
show *ofsm-table M f k q1* = *ofsm-table M f k q2* $\Longrightarrow$ *q2* $\in$ *ofsm-table M f k q1*
using *ofsm-table-containment*[*OF assms*(*2*) *equivalence-relation-on-states-refl*[*OF* ‹*equivalence-relation-on-states M f*›]]
by *blast*

show *q2* $\in$ *ofsm-table M f k q1* $\Longrightarrow$ *ofsm-table M f k q1* = *ofsm-table M f k q2*
proof −
assume $*$: *q2* $\in$ *ofsm-table M f k q1*

have *ofsm-table M f k q1* $= \{q' \in$ *FSM.states M*. $\forall$ *io. length io* $\leq k \longrightarrow$ (*is-in-language M q1 io*) = (*is-in-language M q′ io*) $\wedge$ (*is-in-language M q1 io* $\longrightarrow$ *after M q′ io* $\in f$ (*after M q1 io*))$\}$
using *ofsm-table-set*[*OF assms*(*1*,*3*)] **by** *auto*

moreover have *ofsm-table M f k q2* $= \{q' \in$ *FSM.states M*. $\forall$ *io. length io* $\leq$

$k \longrightarrow$ (*is-in-language M q1 io*) = (*is-in-language M q′ io*) $\wedge$ (*is-in-language M q1 io* $\longrightarrow$ *after M q′ io* $\in$ *f* (*after M q1 io*))}

proof −

have *ofsm-table M f k q2* = {*q′* $\in$ *FSM.states M*. $\forall$ *io*. *length io* $\leq$ *k* $\longrightarrow$ (*is-in-language M q2 io*) = (*is-in-language M q′ io*) $\wedge$ (*is-in-language M q2 io* $\longrightarrow$ *after M q′ io* $\in$ *f* (*after M q2 io*))}

using *ofsm-table-set*[*OF assms*(*2*,*3*)] **by** *auto*

moreover have $\bigwedge$ *io* . *length io* $\leq$ *k* $\Longrightarrow$ (*is-in-language M q1 io*) = (*is-in-language M q2 io*)

using *ofsm-table-language*(*1*)[*OF* $*$ - *assms*(*1*,*3*)] **by** *blast*

moreover have $\bigwedge$ *io q′* . *q′* $\in$ *states M* $\Longrightarrow$ *length io* $\leq$ *k* $\Longrightarrow$ (*is-in-language M q2 io* $\longrightarrow$ *after M q′ io* $\in$ *f* (*after M q2 io*)) = (*is-in-language M q1 io* $\longrightarrow$ *after M q′ io* $\in$ *f* (*after M q1 io*))

using *ofsm-table-language*(*2*)[*OF* $*$ - *assms*(*1*,*3*)]

by (*meson after-is-state-is-in-language assms*(*1*) *assms*(*2*) *assms*(*3*) *calculation*(*2*) *equivalence-relation-on-states-sym equivalence-relation-on-states-trans*)

ultimately show *?thesis*

by *blast*

qed

ultimately show *?thesis*

by *blast*

qed

qed

lemma *ofsm-table-fix-language* :

fixes *M* :: (*′a*,*′b*,*′c*) *fsm*

assumes *q′* $\in$ *ofsm-table-fix M f 0 q*

and *q* $\in$ *states M*

and *observable M*

and *equivalence-relation-on-states M f*

shows *LS M q* = *LS M q′*

and *io* $\in$ *LS M q* $\Longrightarrow$ *after M q′ io* $\in$ *f* (*after M q io*)

proof −

obtain *k* **where** $*$:$\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table-fix M f 0 q* = *ofsm-table M f k q*

and $**$: $\bigwedge$ *q k′* . *q* $\in$ *states M* $\Longrightarrow$ *k′* $\geq$ *k* $\Longrightarrow$ *ofsm-table M f k′ q* = *ofsm-table M f k q*

using *ofsm-table-fix-length*[*of M f*,*OF equivalence-relation-on-states-ran*[*OF assms*(*4*)]]

by *blast*

have *q′* $\in$ *ofsm-table M f k q*

using $*$ *assms*(*1*,*2*) **by** *blast*

then have *q′* $\in$ *states M*

by (*metis assms*(*2*) *assms*(*4*) *equivalence-relation-on-states-ran le0 ofsm-table.simps*(*1*) *ofsm-table-subset subset-iff*)

have $\bigwedge k'$. $q' \in$ *ofsm-table M f k′ q*
proof −
fix k' **show** $q' \in$ *ofsm-table M f k′ q*
proof (*cases* $k' \leq k$)
case *True*
show *?thesis* **using** *ofsm-table-subset*[*OF True*, *of M f q*] ‹$q' \in$ *ofsm-table M f k q*› **by** *blast*
next
case *False*
then have $k \leq k'$
by *auto*
show *?thesis*
unfolding ∗∗[*OF assms*(*2*) ‹$k \leq k'$›]
using ‹$q' \in$ *ofsm-table M f k q*› **by** *assumption*
qed
qed

have $\bigwedge io$. $io \in LS\ M\ q \longleftrightarrow io \in LS\ M\ q'$
proof −
fix *io* :: (′*b* × ′*c*) *list*
show $io \in LS\ M\ q \longleftrightarrow io \in LS\ M\ q'$
using *ofsm-table-language*(*1*)[*OF* ‹$q' \in$ *ofsm-table M f* (*length io*) *q*› - *assms*(*2*,*4*), *of io*]
using *is-in-language-iff*[*OF assms*(*3*,*2*)] *is-in-language-iff*[*OF assms*(*3*) ‹$q' \in$ *states M*›]
by *blast*
qed
then show $LS\ M\ q = LS\ M\ q'$
by *blast*

show $io \in LS\ M\ q \Longrightarrow$ *after M q′ io* $\in f$ (*after M q io*)
using *ofsm-table-language*(*2*)[*OF* ‹$q' \in$ *ofsm-table M f* (*length io*) *q*› - *assms*(*2*,*4*), *of io*]
using *is-in-language-iff*[*OF assms*(*3*,*2*)] *is-in-language-iff*[*OF assms*(*3*) ‹$q' \in$ *states M*›]
by *blast*
qed

lemma *ofsm-table-same-language* :
assumes $LS\ M\ q = LS\ M\ q'$
and $\bigwedge io$. $io \in LS\ M\ q \Longrightarrow$ *after M q′ io* $\in f$ (*after M q io*)
and *observable M*
and $q' \in$ *states M*
and $q \in$ *states M*
and *equivalence-relation-on-states M f*

shows *ofsm-table M f k q = ofsm-table M f k q′*
using *assms(1,2,4,5)*
proof (*induction k arbitrary*: *q q′*)
case *0*
then show *?case*
by (*metis after.simps(1) assms(6) from-FSM-language language-contains-empty-sequence ofsm-table.simps(1) ofsm-table-eq-if-elem*)
next
case (*Suc k*)

have *ofsm-table M f (Suc k) q = {q″ ∈ ofsm-table M f k q′ . ∀ x ∈ inputs M . ∀ y ∈ outputs M . (case h-obs M q x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None) }*
using *Suc.IH*[*OF Suc.prems*] **unfolding** *ofsm-table.simps Suc Let-def Suc* **by** *simp*

moreover have *ofsm-table M f (Suc k) q′ = {q″ ∈ ofsm-table M f k q′ . ∀ x ∈ inputs M . ∀ y ∈ outputs M . (case h-obs M q′ x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None) }*
unfolding *ofsm-table.simps Suc Let-def*
by *auto*

moreover have *{q″ ∈ ofsm-table M f k q′ . ∀ x ∈ inputs M . ∀ y ∈ outputs M . (case h-obs M q x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None) }*
= {q″ ∈ ofsm-table M f k q′ . ∀ x ∈ inputs M . ∀ y ∈ outputs M . (case h-obs M q′ x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None) }
proof −
have ⋀ *q″ x y . q″ ∈ ofsm-table M f k q′ ⟹ x ∈ inputs M ⟹ y ∈ outputs M ⟹*
(case h-obs M q x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None)
= (case h-obs M q′ x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None)
proof −

fix *q″ x y* **assume** *q″ ∈ ofsm-table M f k q′* **and** *x ∈ inputs M* **and** *y ∈ outputs M*

have ∗:*(∃ qT . h-obs M q x y = Some qT) = (∃ qT′ . h-obs M q′ x y = Some qT′)*
proof −

have $([(x,y)] \in LS\ M\ q) = ([(x,y)] \in LS\ M\ q')$
using ‹$LS\ M\ q = LS\ M\ q'$› **by** *auto*
then have $(\exists\ qT\ .\ (q,\ x,\ y,\ qT) \in FSM.transitions\ M) = (\exists\ qT'\ .\ (q',\ x,\ y,\ qT') \in FSM.transitions\ M)$
unfolding *LS-single-transition* **by** *force*
then show $(\exists\ qT\ .\ h\text{-}obs\ M\ q\ x\ y = Some\ qT) = (\exists\ qT'\ .\ h\text{-}obs\ M\ q'\ x\ y = Some\ qT')$
unfolding *h-obs-Some*[*OF* ‹*observable M*›] **using** ‹*observable M*› **unfolding** *observable-alt-def* **by** *blast*
qed

have $**$: $(case\ h\text{-}obs\ M\ q\ x\ y\ of\ Some\ qT \Rightarrow (case\ h\text{-}obs\ M\ q'\ x\ y\ of\ Some\ qT' \Rightarrow ofsm\text{-}table\ M\ f\ k\ qT = ofsm\text{-}table\ M\ f\ k\ qT' \mid None \Rightarrow False) \mid None \Rightarrow h\text{-}obs\ M\ q'\ x\ y = None)$
proof (*cases h-obs M q x y*)
case *None*
then show *?thesis* **using** $*$ **by** *auto*
next
case (*Some qT*)
show *?thesis* **proof** (*cases h-obs M q' x y*)
case *None*
then show *?thesis* **using** $*$ **by** *auto*
next
case (*Some qT'*)

have $(q,x,y,qT) \in transitions\ M$
using ‹$h\text{-}obs\ M\ q\ x\ y = Some\ qT$› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] **by** *blast*
have $(q',x,y,qT') \in transitions\ M$
using ‹$h\text{-}obs\ M\ q'\ x\ y = Some\ qT'$› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] **by** *blast*

have $LS\ M\ qT = LS\ M\ qT'$
using *observable-transition-target-language-eq*[*OF* - ‹$(q,x,y,qT) \in transitions\ M$› ‹$(q',x,y,qT') \in transitions\ M$› - - ‹*observable M*›]
‹$LS\ M\ q = LS\ M\ q'$›
by *auto*
moreover have $(\bigwedge io.\ io \in LS\ M\ qT \Longrightarrow after\ M\ qT'\ io \in f\ (after\ M\ qT\ io))$
proof −
fix *io* **assume** $io \in LS\ M\ qT$

have $io \in LS\ M\ qT'$
using ‹$io \in LS\ M\ qT$› *calculation* **by** *auto*

have $after\ M\ qT\ io = after\ M\ q\ ((x,y)\#io)$
using *after-h-obs-prepend*[*OF* ‹*observable M*› ‹$h\text{-}obs\ M\ q\ x\ y = Some\ qT$› ‹$io \in LS\ M\ qT$›]

by *simp*
moreover have *after M qT′ io = after M q′ ((x,y)#io)*
using *after-h-obs-prepend*[*OF* ‹*observable M*› ‹*h-obs M q′ x y = Some qT′*› ‹*io ∈ LS M qT′*›]
by *simp*
moreover have *(x,y)#io ∈ LS M q*
using ‹*h-obs M q x y = Some qT*› ‹*io ∈ LS M qT*› **unfolding** *h-obs-language-iff*[*OF* ‹*observable M*›]
by *blast*
ultimately show *after M qT′ io ∈ f (after M qT io)*
using *Suc.prems*(*2*) **by** *presburger*
qed

ultimately have *ofsm-table M f k qT = ofsm-table M f k qT′*
using *Suc.IH*[*OF - - fsm-transition-target*[*OF* ‹*(q′,x,y,qT′) ∈ transitions M*›] *fsm-transition-target*[*OF* ‹*(q,x,y,qT) ∈ transitions M*›]]
unfolding *snd-conv*
by *blast*
then show *?thesis*
using ‹*h-obs M q x y = Some qT*› ‹*h-obs M q′ x y = Some qT′*› **by** *auto*
qed
qed

show (*case h-obs M q x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None*)
= (*case h-obs M q′ x y of Some qT ⇒ (case h-obs M q″ x y of Some qT′ ⇒ ofsm-table M f k qT = ofsm-table M f k qT′ | None ⇒ False) | None ⇒ h-obs M q″ x y = None*) (**is** *?P*)

proof (*cases h-obs M q x y*)
case *None*
then have *h-obs M q′ x y = None*
using ∗ **by** *auto*
show *?thesis* **unfolding** *None* ‹*h-obs M q′ x y = None*› **by** *auto*
next
case (*Some qT*)
then obtain *qT′* **where** *h-obs M q′ x y = Some qT′*
using ‹*(∃ qT . h-obs M q x y = Some qT) = (∃ qT′ . h-obs M q′ x y = Some qT′)*› **by** *auto*

show *?thesis*
proof (*cases h-obs M q″ x y*)
case *None*
then show *?thesis* **using** ∗
by (*metis Some option.case-eq-if option.simps*(*5*))
next

case (*Some qT''*)
show *?thesis*
using $**$
unfolding *Some* ‹*h-obs M q x y = Some qT*› ‹*h-obs M q' x y = Some qT'*› **by** *auto*
qed
qed
qed

then show *?thesis*
by *blast*
qed

ultimately show *?case* **by** *blast*
qed

lemma *ofsm-table-fix-set* :
assumes $q \in$ *states M*
and *observable M*
and *equivalence-relation-on-states M f*
shows *ofsm-table-fix M f 0 q* = $\{q' \in$ *states M . LS M q'* = *LS M q* $\wedge$ ($\forall$ *io* $\in$ *LS M q . after M q' io* $\in$ *f* (*after M q io*))$\}$
proof

have *ofsm-table-fix M f 0 q* $\subseteq$ *ofsm-table M f 0 q*
using *ofsm-table-fix-length*[*of M f*]
ofsm-table-subset[*OF zero-le*, *of M f - q*]
by (*metis assms*(*1*) *assms*(*3*) *equivalence-relation-on-states-ran*)
then have *ofsm-table-fix M f 0 q* $\subseteq$ *states M*
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*3*)] *assms*(*1*)] **by** *blast*
then show *ofsm-table-fix M f 0 q* $\subseteq$ $\{q' \in$ *states M . LS M q'* = *LS M q* $\wedge$ ($\forall$ *io* $\in$ *LS M q . after M q' io* $\in$ *f* (*after M q io*))$\}$
using *ofsm-table-fix-language*[*OF - assms*] **by** *blast*

show $\{q' \in$ *states M . LS M q'* = *LS M q* $\wedge$ ($\forall$ *io* $\in$ *LS M q . after M q' io* $\in$ *f* (*after M q io*))$\}$ $\subseteq$ *ofsm-table-fix M f 0 q*
proof
fix *q'* **assume** $q' \in \{q' \in$ *states M . LS M q'* = *LS M q* $\wedge$ ($\forall$ *io* $\in$ *LS M q . after M q' io* $\in$ *f* (*after M q io*))$\}$
then have $q' \in$ *states M* **and** *LS M q'* = *LS M q* **and** $\bigwedge$ *io . io* $\in$ *LS M q* $\Longrightarrow$ *after M q' io* $\in$ *f* (*after M q io*)
by *blast+*

then have $\bigwedge$ *io . io* $\in$ *LS M q'* $\Longrightarrow$ *after M q io* $\in$ *f* (*after M q' io*)
by (*metis after-is-state assms*(*2*) *assms*(*3*) *equivalence-relation-on-states-sym*)

obtain k **where** $\bigwedge q\ .\ q \in states\ M \Longrightarrow ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q = ofsm\text{-}table\ M\ f\ k\ q$
and $\bigwedge q\ k'\ .\ q \in states\ M \Longrightarrow k' \geq k \Longrightarrow ofsm\text{-}table\ M\ f\ k'\ q = ofsm\text{-}table\ M\ f\ k\ q$
using *ofsm-table-fix-length*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*3*)]] **by** *blast*

have $ofsm\text{-}table\ M\ f\ k\ q' = ofsm\text{-}table\ M\ f\ k\ q$
using *ofsm-table-same-language*[*OF* ‹$LS\ M\ q' = LS\ M\ q$› ‹$\bigwedge io\ .\ io \in LS\ M\ q' \Longrightarrow after\ M\ q\ io \in f\ (after\ M\ q'\ io)$› *assms*(*2*,*1*) ‹$q' \in states\ M$› *assms*(*3*)]
by *blast*
then show $q' \in ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q$
using *ofsm-table-containment*[*OF* ‹$q' \in states\ M$›, *of f k*]
using ‹$\bigwedge q\ .\ q \in states\ M \Longrightarrow ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q = ofsm\text{-}table\ M\ f\ k\ q$›
by (*metis assms*(*1*) *assms*(*3*) *equivalence-relation-on-states-refl*)
qed
qed

lemma *ofsm-table-fix-eq-if-elem* :
assumes $q1 \in states\ M$ **and** $q2 \in states\ M$
and $equivalence\text{-}relation\text{-}on\text{-}states\ M\ f$
shows $(ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1 = ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q2) = (q2 \in ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1)$
proof
have $(\bigwedge q.\ q \in FSM.states\ M \Longrightarrow q \in f\ q)$
using *assms*(*3*)
by (*meson equivalence-relation-on-states-refl*)

show $ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1 = ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q2 \Longrightarrow q2 \in ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1$
using *ofsm-table-containment*[*of - M f*, *OF assms*(*2*) ‹$(\bigwedge q.\ q \in FSM.states\ M \Longrightarrow q \in f\ q)$›]
using *ofsm-table-fix-length*[*of M f*]
by (*metis assms*(*2*) *assms*(*3*) *equivalence-relation-on-states-ran*)

show $q2 \in ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1 \Longrightarrow ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q1 = ofsm\text{-}table\text{-}fix\ M\ f\ 0\ q2$
using *ofsm-table-eq-if-elem*[*OF assms*(*1*,*2*,*3*)]
using *ofsm-table-fix-length*[*of M f*]
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *equivalence-relation-on-states-ran*)
qed

lemma *ofsm-table-refinement-disjoint* :
assumes $q1 \in states\ M$ **and** $q2 \in states\ M$
and $equivalence\text{-}relation\text{-}on\text{-}states\ M\ f$
and $ofsm\text{-}table\ M\ f\ k\ q1 \neq ofsm\text{-}table\ M\ f\ k\ q2$

shows *ofsm-table M f (Suc k) q1* $\neq$ *ofsm-table M f (Suc k) q2*
proof −
  have *ofsm-table M f (Suc k) q1* $\subseteq$ *ofsm-table M f k q1*
  and *ofsm-table M f (Suc k) q2* $\subseteq$ *ofsm-table M f k q2*
    using *ofsm-table-subset*[*of k Suc k M f*]
    by *fastforce+*
  moreover have *ofsm-table M f k q1* $\cap$ *ofsm-table M f k q2* = {}
  proof (*rule ccontr*)
    assume *ofsm-table M f k q1* $\cap$ *ofsm-table M f k q2* $\neq$ {}
    then obtain *q* **where** *q* $\in$ *ofsm-table M f k q1*
                and *q* $\in$ *ofsm-text M f k q2*
      by *blast*
    then have *q* $\in$ *states M*
        using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*3*)] *assms*(*1*)]
      by *blast*

    have *ofsm-table M f k q1* = *ofsm-table M f k q2*
      using ‹*q* $\in$ *ofsm-table M f k q1*› ‹*q* $\in$ *ofsm-table M f k q2*›
      unfolding *ofsm-table-eq-if-elem*[*OF assms*(*1*) ‹*q* $\in$ *states M*› *assms*(*3*), *symmetric*]
      unfolding *ofsm-table-eq-if-elem*[*OF assms*(*2*) ‹*q* $\in$ *states M*› *assms*(*3*), *symmetric*]
      by *blast*
    then show *False*
      using *assms*(*4*) **by** *simp*
  qed
  ultimately show *?thesis*
    by (*metis Int-subset-iff all-not-in-conv assms*(*2*) *assms*(*3*) *ofsm-table-eq-if-elem subset-empty*)
qed

lemma *ofsm-table-partition-finite* :
  assumes *equivalence-relation-on-states M f*
shows *finite* (*ofsm-table M f k* ‘ *states M*)
  using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*]]

    fsm-states-finite[*of M*]
  unfolding *finite-Pow-iff*[*of states M*, *symmetric*]
  by *simp*

lemma *ofsm-table-refinement-card* :
  assumes *equivalence-relation-on-states M f*
  and *A* $\subseteq$ *states M*
  and *i* $\leq$ *j*
shows *card* (*ofsm-table M f j* ‘ *A*) $\geq$ *card* (*ofsm-table M f i* ‘ *A*)
proof −
  have $\bigwedge$ *k* . *card* (*ofsm-table M f* (*Suc k*) ‘ *A*) $\geq$ *card* (*ofsm-table M f k* ‘ *A*)

```
  proof −
    fix k show card (ofsm-table M f (Suc k) ‘ A) ≥ card (ofsm-table M f k ‘ A)
    proof (rule ccontr)

      have finite A
        using fsm-states-finite[of M] assms(2)
        using finite-subset by blast

      assume ¬ card (ofsm-table M f k ‘ A) ≤ card (ofsm-table M f (Suc k) ‘ A)
      then have card (ofsm-table M f (Suc k) ‘ A) < card (ofsm-table M f k ‘ A)
        by simp
      then obtain q1 q2 where q1 ∈ A
                        and q2 ∈ A
                        and ofsm-table M f k q1 ≠ ofsm-table M f k q2
                        and ofsm-table M f (Suc k) q1 = ofsm-table M f (Suc k) q2
        using finite-card-less-witnesses[OF ‹finite A›] by blast
      then show False
        using ofsm-table-refinement-disjoint[OF - - assms(1), of q1 q2 k]
        using assms(2)
        by blast
    qed
  qed
  then show ?thesis
    using lift-Suc-mono-le[OF - assms(3), where f=λ k . card (ofsm-table M f k
‘ A)]
    by blast
qed



lemma ofsm-table-refinement-card-fix-Suc :
  assumes equivalence-relation-on-states M f
  and     card (ofsm-table M f (Suc k) ‘ states M) = card (ofsm-table M f k ‘ states
M)
  and     q ∈ states M
shows ofsm-table M f (Suc k) q = ofsm-table M f k q
proof (rule ccontr)
  assume ofsm-table M f (Suc k) q ≠ ofsm-table M f k q
  then have ofsm-table M f (Suc k) q ⊂ ofsm-table M f k q
    using ofsm-table-subset
    by (metis Suc-leD order-refl psubsetI)
  then obtain q′ where q′ ∈ ofsm-table M f k q
                 and q′ ∉ ofsm-table M f (Suc k) q
    by blast

  then have q′ ∈ states M
   using ofsm-table-states[of M f, OF equivalence-relation-on-states-ran[OF assms(1)]
assms(3)]  by blast
```

have *card-qq*: $\bigwedge$ *k* . *card* (*ofsm-table M f k* ‘ *states M*)
= *card* (*ofsm-table M f k* ‘ (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})) + *card* (*ofsm-table M f k* ‘ ($\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′}))
proof −
fix *k*
have *states M* = (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})) ∪ $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)] ‹*q* ∈ *states M*›]
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)] ‹*q*′ ∈ *states M*›]
by *blast*
then have *finite* (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′}))
and *finite* ($\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′}))
using *fsm-states-finite*[*of M*] *finite-Un*[*of* (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})) $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})]
by *force*+
then have ∗:*finite* (*ofsm-table M f k* ‘ (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})))

and ∗∗:*finite* (*ofsm-table M f k* ‘ $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′}))
by *blast*+
have ∗∗∗:(*ofsm-table M f k* ‘ (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′}))) ∩ (*ofsm-table M f k* ‘ $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})) = {}
proof (*rule ccontr*)
assume *ofsm-table M f k* ‘ (*FSM.states M* − $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})) ∩ *ofsm-table M f k* ‘ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′}) ≠ {}
then obtain *Q* **where** *Q* ∈ *ofsm-table M f k* ‘ (*FSM.states M* − $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′}))
and *Q* ∈ *ofsm-table M f k* ‘ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})
by *blast*

obtain *q1* **where** *q1* ∈ (*FSM.states M* − $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′}))
and *Q* = *ofsm-table M f k q1*
using ‹*Q* ∈ *ofsm-table M f k* ‘ (*FSM.states M* − $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′}))› **by** *blast*
moreover obtain *q2* **where** *q2* ∈ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})
and *Q* = *ofsm-table M f k q2*
using ‹*Q* ∈ *ofsm-table M f k* ‘ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})› **by** *blast*
ultimately have *ofsm-table M f k q1* = *ofsm-table M f k q2*
by *auto*

have *q1* ∈ *states M* **and** *q1* ∉ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})
using ‹*q1* ∈ (*FSM.states M* − $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′}))›
by *blast*+
have *q2* ∈ *states M*
using ‹*q2* ∈ $\bigcup$ (*ofsm-table M f k* ‘ {*q*, *q*′})› ‹*states M* = (*states M* − $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})) ∪ $\bigcup$(*ofsm-table M f k* ‘ {*q*,*q*′})›
by *blast*

have *q1* ∈ *ofsm-table M f k q2*
using ‹*ofsm-table M f k q1* = *ofsm-table M f k q2*›
using *ofsm-table-eq-if-elem*[*OF* ‹*q2* ∈ *states M*› ‹*q1* ∈ *states M*› *assms*(*1*)]
by *blast*
moreover have *q2* ∈ *ofsm-table M f k q* ∨ *q2* ∈ *ofsm-table M f k q′*
using ‹*q2* ∈ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})›
by *blast*
ultimately have *q1* ∈ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})
unfolding *ofsm-table-eq-if-elem*[*OF* ‹*q* ∈ *states M*› ‹*q2* ∈ *states M*› *assms*(*1*), *symmetric*]
unfolding *ofsm-table-eq-if-elem*[*OF* ‹*q′* ∈ *states M*› ‹*q2* ∈ *states M*› *assms*(*1*), *symmetric*]
by *blast*
then show *False*
using ‹*q1* ∉ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})›
by *blast*
qed

show *card* (*ofsm-table M f k* ‘ *states M*)
= *card* (*ofsm-table M f k* ‘ (*states M* − ⋃(*ofsm-table M f k* ‘ {*q*,*q′*}))) + *card* (*ofsm-table M f k* ‘ (⋃(*ofsm-table M f k* ‘ {*q*,*q′*})))
using *card-Un-disjoint*[*OF* ∗ ∗∗ ∗∗∗]
using ‹*states M* = (*states M* − ⋃(*ofsm-table M f k* ‘ {*q*,*q′*})) ∪ ⋃(*ofsm-table M f k* ‘ {*q*,*q′*})›
by (*metis image-Un*)
qed

have *s1*: ⋀ *k* . (*states M* − ⋃(*ofsm-table M f k* ‘ {*q*,*q′*})) ⊆ *states M*
and *s2*: ⋀ *k* . (⋃(*ofsm-table M f k* ‘ {*q*,*q′*})) ⊆ *states M*
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)] ‹*q* ∈ *states M*›]
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)] ‹*q′* ∈ *states M*›]
by *blast*+

have *card* (*ofsm-table M f* (*Suc k*) ‘ *states M*) > *card* (*ofsm-table M f k* ‘ *states M*)
proof −
have ∗: ⋃ (*ofsm-table M f* (*Suc k*) ‘ {*q*, *q′*}) ⊆ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})
using *ofsm-table-subset*
by (*metis SUP-mono′ lessI less-imp-le-nat*)

have *card* (*ofsm-table M f k* ‘ (*FSM.states M* − ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}))) ≤ *card* (*ofsm-table M f* (*Suc k*) ‘ (*FSM.states M* − ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})))
using *ofsm-table-refinement-card*[*OF assms*(*1*), **where** *i*=*k* **and** *j*=*Suc k*, *OF s1*]
using *le-SucI* **by** *blast*

moreover have *card* (*ofsm-table M f* (*Suc k*) ‘ (*FSM.states M* − ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}))) ≤ *card* (*ofsm-table M f* (*Suc k*) ‘ (*FSM.states M* − ⋃ (*ofsm-table M f* (*Suc k*) ‘ {*q*, *q′*})))
using ∗
using *fsm-states-finite*[*of M*]
by (*meson Diff-mono card-mono finite-Diff finite-imageI image-mono subset-refl*)
ultimately have *card* (*ofsm-table M f k* ‘ (*FSM.states M* − ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}))) ≤ *card* (*ofsm-table M f* (*Suc k*) ‘ (*FSM.states M* − ⋃ (*ofsm-table M f* (*Suc k*) ‘ {*q*, *q′*})))
by *presburger*
moreover have *card* (*ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})) < *card* (*ofsm-table M f* (*Suc k*) ‘ ⋃ (*ofsm-table M f* (*Suc k*) ‘ {*q*, *q′*}))
proof −
have ∗: ⋀ *k* . *ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}) = {*ofsm-table M f k q*, *ofsm-table M f k q′*}
proof −
fix *k* **show** *ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}) = {*ofsm-table M f k q*, *ofsm-table M f k q′*}
proof
show *ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*}) ⊆ {*ofsm-table M f k q*, *ofsm-table M f k q′*}
proof
fix *Q* **assume** *Q* ∈ *ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})
then obtain *qq* **where** *Q* = *ofsm-table M f k qq*
and *qq* ∈ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})
by *blast*

then have *qq* ∈ *ofsm-table M f k q* ∨ *qq* ∈ *ofsm-table M f k q′*
by *blast*
then have *qq* ∈ *states M*
using *ofsm-table-states*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)]] ‹*q* ∈ *states M*› ‹*q′* ∈ *states M*›
by *blast*

have *ofsm-table M f k qq* = *ofsm-table M f k q* ∨ *ofsm-table M f k qq* = *ofsm-table M f k q′*
using ‹*qq* ∈ *ofsm-table M f k q* ∨ *qq* ∈ *ofsm-table M f k q′*›
using *ofsm-table-eq-if-elem*[*OF* - ‹*qq* ∈ *states M*› *assms*(*1*)] ‹*q* ∈ *states M*› ‹*q′* ∈ *states M*›
by *blast*
then show *Q* ∈ {*ofsm-table M f k q*, *ofsm-table M f k q′*}
using ‹*Q* = *ofsm-table M f k qq*›
by *blast*
qed
show {*ofsm-table M f k q*, *ofsm-table M f k q′*} ⊆ *ofsm-table M f k* ‘ ⋃ (*ofsm-table M f k* ‘ {*q*, *q′*})
using *ofsm-table-containment*[*of* - *M f*, *OF* - *equivalence-relation-on-states-refl*[*OF assms*(*1*)]] ‹*q* ∈ *states M*› ‹*q′* ∈ *states M*›

```
        by blast
      qed
    qed

    have ofsm-table M f k q = ofsm-table M f k q'
      using ‹q' ∈ ofsm-table M f k q›
      using ofsm-table-eq-if-elem[OF ‹q ∈ states M› ‹q' ∈ states M› assms(1)]
      by blast
    moreover have ofsm-table M f (Suc k) q ≠ ofsm-table M f (Suc k) q'
      using ‹q' ∉ ofsm-table M f (Suc k) q›
      using ofsm-table-eq-if-elem[OF ‹q ∈ states M› ‹q' ∈ states M› assms(1)]
      by blast
    ultimately show ?thesis
      unfolding ∗
    by (metis card-insert-if finite.emptyI finite.insertI insert-absorb insert-absorb2
insert-not-empty lessI singleton-insert-inj-eq)
  qed
  ultimately show ?thesis
    unfolding card-qq by presburger
 qed
 then show False
  using assms(2) by linarith
qed


lemma ofsm-table-refinement-card-fix :
 assumes equivalence-relation-on-states M f
 and     card (ofsm-table M f j ‘ states M) = card (ofsm-table M f i ‘ states M)
 and     q ∈ states M
 and     i ≤ j
shows ofsm-table M f j q = ofsm-table M f i q
 using assms (2,4) proof (induction j−i arbitrary: i j)
 case 0
 then have i = j by auto
 then show ?case by auto
next
 case (Suc k)
 then have j ≥ Suc i and k = j − Suc i
  by auto

 have ∗:card (ofsm-table M f j ‘ FSM.states M) = card (ofsm-table M f (Suc i) ‘
FSM.states M)
 and  ∗∗:card (ofsm-table M f (Suc i) ‘ FSM.states M) = card (ofsm-table M f i
‘ FSM.states M)
  using ofsm-table-refinement-card[OF assms(1), where A=states M]
  by (metis Suc.prems(1) ‹Suc i ≤ j› eq-iff le-SucI)+


 show ?case
```

```
    using Suc.hyps(1)[OF ‹k = j − Suc i› ∗ ‹Suc i ≤ j›]
    using ofsm-table-refinement-card-fix-Suc[OF assms(1) ∗∗ assms(3)]
    by blast
qed


lemma ofsm-table-partition-fixpoint-Suc :
  assumes equivalence-relation-on-states M f
  and     q ∈ states M
shows ofsm-table M f (size M − card (f ‘ states M)) q = ofsm-table M f (Suc
(size M − card (f ‘ states M))) q
proof −

  have ⋀ q . q ∈ states M ⟹ f q = ofsm-table M f 0 q
    unfolding ofsm-table.simps by auto

  define n where n: n = (λ i . card (ofsm-table M f i ‘ states M))

  have ⋀ i j . i ≤ j ⟹ n i ≤ n j
    unfolding n
    using ofsm-table-refinement-card[OF assms(1), where A=states M]
    by blast
  moreover have ⋀ i j m . i < j ⟹ n i = n j ⟹ j ≤ m ⟹ n i = n m
  proof −
    fix i j m assume i < j and n i = n j and j ≤ m
    then have Suc i ≤ j and i ≤ Suc i and i ≤ m
      by auto

    have ⋀ q . q ∈ states M ⟹ ofsm-table M f j q = ofsm-table M f i q
      using ‹i < j› ‹n i = n j› ofsm-table-refinement-card-fix[OF assms(1) -]
      unfolding n
      using less-imp-le-nat by presburger
    then have ⋀ q . q ∈ states M ⟹ ofsm-table M f (Suc i) q = ofsm-table M f
i q
      using ofsm-table-subset[OF ‹Suc i ≤ j›, of M f]
      using ofsm-table-subset[OF ‹i ≤ Suc i›, of M f]
      by blast
    then have ⋀ q . q ∈ states M ⟹ ofsm-table M f m q = ofsm-table M f i q
      using ofsm-table-fixpoint[OF ‹i ≤ m›]
      by metis
    then show n i = n m
      unfolding n
      by auto
  qed
  moreover have ⋀ i . n i ≤ size M
    unfolding n
   using ofsm-table-states[of M f, OF equivalence-relation-on-states-ran[OF assms(1)]]
    using fsm-states-finite[of M]
    by (simp add: card-image-le)
```

ultimately obtain k **where** $n\ (Suc\ k) = n\ k$
and $k \leq size\ M - n\ 0$
using *monotone-function-with-limit-witness-helper*[**where** $f{=}n$ **and** $k{=}size\ M$]
by *blast*

then show *?thesis*
unfolding n
using ‹$\bigwedge q\ .\ q \in states\ M \Longrightarrow f\ q = ofsm\text{-}table\ M\ f\ 0\ q$›[*symmetric*]

using *ofsm-table-refinement-card-fix-Suc*[*OF assms*(*1*) -]
using *ofsm-table-fixpoint*[*OF* - - *assms*(*2*)]
by (*metis* (*mono-tags*, *lifting*) *image-cong nat-le-linear not-less-eq-eq*)
qed

lemma *ofsm-table-partition-fixpoint* :
assumes *equivalence-relation-on-states M f*
and $size\ M \leq m$
and $q \in states\ M$
shows $ofsm\text{-}table\ M\ f\ (m - card\ (f\ `\ states\ M))\ q = ofsm\text{-}table\ M\ f\ (Suc\ (m - card\ (f\ `\ states\ M)))\ q$
proof −
have $*$: $size\ M - card\ (f\ `\ states\ M) \leq m - card\ (f\ `\ states\ M)$
using *assms*(*2*) **by** *simp*
have $**$: $(size\ M - card\ (f\ `\ states\ M)) \leq Suc\ (m - card\ (f\ `\ states\ M))$
using *assms*(*2*) **by** *simp*

have $***$: $\bigwedge q\ .\ q \in FSM.states\ M \Longrightarrow ofsm\text{-}table\ M\ f\ (FSM.size\ M - card\ (f\ `\ FSM.states\ M))\ q = ofsm\text{-}table\ M\ f\ (Suc\ (FSM.size\ M - card\ (f\ `\ FSM.states\ M)))\ q$
using *ofsm-table-partition-fixpoint-Suc*[*OF assms*(*1*)] **.**

have $ofsm\text{-}table\ M\ f\ (m - card\ (f\ `\ states\ M))\ q = ofsm\text{-}table\ M\ f\ (FSM.size\ M - card\ (f\ `\ FSM.states\ M))\ q$
using *ofsm-table-fixpoint*[*OF* $*$ - *assms*(*3*)] $***$
by *blast*
moreover have $ofsm\text{-}table\ M\ f\ (Suc\ (m - card\ (f\ `\ states\ M)))\ q = ofsm\text{-}table\ M\ f\ (FSM.size\ M - card\ (f\ `\ FSM.states\ M))\ q$
using *ofsm-table-fixpoint*[*OF* $**$ - *assms*(*3*), *of f*] $***$
by *blast*
ultimately show *?thesis*
by *simp*
qed

lemma *ofsm-table-fix-partition-fixpoint* :
assumes *equivalence-relation-on-states M f*

and *size M* $\leq$ *m*
and *q* $\in$ *states M*
shows *ofsm-table M f* (*m* − *card* (*f* ‘ *states M*)) *q* = *ofsm-table-fix M f 0 q*
proof −

obtain *k* **where** *k1*: *ofsm-table-fix M f 0 q* = *ofsm-table M f k q*
and *k2*: $\bigwedge$ *k′* . *k′* $\geq$ *k* $\Longrightarrow$ *ofsm-table M f k′ q* = *ofsm-table M f k q*
using *ofsm-table-fix-length*[*of M f*, *OF equivalence-relation-on-states-ran*[*OF assms*(*1*)]]
assms(*3*)
by *metis*

have *m1*: $\bigwedge$ *k′* . *k′* $\geq$ *m* − *card* (*f* ‘ *states M*) $\Longrightarrow$ *ofsm-table M f k′ q* = *ofsm-table M f* (*m* − *card* (*f* ‘ *states M*)) *q*
using *ofsm-table-partition-fixpoint*[*OF assms*(*1*,*2*)]
using *ofsm-table-fixpoint*[*OF - - assms*(*3*)]
by *presburger*

show *?thesis* **proof** (*cases k* $\leq$ *m* − *card* (*f* ‘ *states M*))
case *True*
show *?thesis*
using *k1 k2*[*OF True*] **by** *simp*
next
case *False*
then have *k* $\geq$ *m* − *card* (*f* ‘ *states M*)
by *auto*
then have *ofsm-table M f k q* = *ofsm-table M f* (*m* − *card* (*f* ‘ *states M*)) *q*
using *ofsm-table-partition-fixpoint*[*OF assms*(*1*,*2*)]
using *ofsm-table-fixpoint*[*OF - - assms*(*3*)]
by *presburger*
then show *?thesis*
using *k1* **by** *simp*
qed
qed

6.2 A minimisation function based on OFSM-tables

lemma *language-equivalence-classes-preserve-observability*:
assumes *transitions M′* = (λ *t* . ({*q* $\in$ *states M* . *LS M q* = *LS M* (*t-source t*)}, *t-input t*, *t-output t*, {*q* $\in$ *states M* . *LS M q* = *LS M* (*t-target t*)})) ‘ *transitions M*
and *observable M*
shows *observable M′*
proof −
have $\bigwedge$ *t1 t2* . *t1* $\in$ *transitions M′* $\Longrightarrow$
t2 $\in$ *transitions M′* $\Longrightarrow$
t-source t1 = *t-source t2* $\Longrightarrow$
t-input t1 = *t-input t2* $\Longrightarrow$
t-output t1 = *t-output t2* $\Longrightarrow$

t-target t1 = *t-target t2*
proof −
fix *t1 t2* **assume** *t1* ∈ *transitions M′* **and** *t2* ∈ *transitions M′* **and** *t-source t1* = *t-source t2* **and** *t-input t1* = *t-input t2* **and** *t-output t1* = *t-output t2*

obtain *t1′* **where** *t1′-def*: *t1* = ({*q* ∈ *states M* . *LS M q* = *LS M* (*t-source t1′*)} , *t-input t1′*, *t-output t1′*, {*q* ∈ *states M* . *LS M q* = *LS M* (*t-target t1′*)})
and *t1′* ∈ *transitions M*
using ‹*t1* ∈ *transitions M′*› *assms*(*1*) **by** *auto*
obtain *t2′* **where** *t2′-def*: *t2* = ({*q* ∈ *states M* . *LS M q* = *LS M* (*t-source t2′*)} , *t-input t2′*, *t-output t2′*, {*q* ∈ *states M* . *LS M q* = *LS M* (*t-target t2′*)})
and *t2′* ∈ *transitions M*
using ‹*t2* ∈ *transitions M′*› *assms*(*1*) ‹*t-input t1* = *t-input t2*› ‹*t-output t1* = *t-output t2*› **by** *auto*

have {*q* ∈ *FSM.states M*. *LS M q* = *LS M* (*t-source t1′*)} = {*q* ∈ *FSM.states M*. *LS M q* = *LS M* (*t-source t2′*)}
using *t1′-def t2′-def* ‹*t-source t1* = *t-source t2*›
by (*metis* (*no-types*, *lifting*) *fst-eqD*)
then have *LS M* (*t-source t1′*) = *LS M* (*t-source t2′*)
using *fsm-transition-source*[*OF* ‹*t1′* ∈ *transitions M*›] *fsm-transition-source*[*OF* ‹*t2′* ∈ *transitions M*›] **by** *blast*
then have *LS M* (*t-target t1′*) = *LS M* (*t-target t2′*)
using *observable-transition-target-language-eq*[*OF* - ‹*t1′* ∈ *transitions M*› ‹*t2′* ∈ *transitions M*› - - ‹*observable M*›]
using ‹*t-input t1* = *t-input t2*› ‹*t-output t1* = *t-output t2*›
unfolding *t1′-def t2′-def fst-conv snd-conv* **by** *blast*
then show *t-target t1* = *t-target t2*
unfolding *t1′-def t2′-def snd-conv* **by** *blast*
qed
then show *?thesis*
unfolding *observable.simps* **by** *blast*
qed

lemma *language-equivalence-classes-retain-language-and-induce-minimality* :
assumes *transitions M′* = (λ *t* . ({*q* ∈ *states M* . *LS M q* = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {*q* ∈ *states M* . *LS M q* = *LS M* (*t-target t*)})) ‘ *transitions M*
and *states M′* = (λ*q* . {*q′* ∈ *states M* . *LS M q* = *LS M q′*}) ‘ *states M*
and *initial M′* = {*q′* ∈ *states M* . *LS M q′* = *LS M* (*initial M*)}
and *observable M*
shows *L M* = *L M′*
and *minimal M′*
proof −
have *observable M′*

using *assms*(*1*,*4*) *language-equivalence-classes-preserve-observability* **by** *blast*

have *ls-prop*: $\bigwedge$ *io q* . *q* $\in$ *states M* $\Longrightarrow$ (*io* $\in$ *LS M q*) $\longleftrightarrow$ (*io* $\in$ *LS M'* {*q'* $\in$ *states M* . *LS M q* = *LS M q'*})
proof −
fix *io q* **assume** *q* $\in$ *states M*
then show (*io* $\in$ *LS M q*) $\longleftrightarrow$ (*io* $\in$ *LS M'* {*q'* $\in$ *states M* . *LS M q* = *LS M q'*})
proof (*induction io arbitrary*: *q*)
case *Nil*
then show *?case* **using** *assms*(*2*) **by** *auto*
next
case (*Cons xy io*)

obtain *x y* **where** *xy* = (*x*,*y*)
using *surjective-pairing* **by** *blast*

have *xy#io* $\in$ *LS M q* $\Longrightarrow$ *xy#io* $\in$ *LS M'* {*q'* $\in$ *states M* . *LS M q* = *LS M q'*}
proof −
assume *xy#io* $\in$ *LS M q*
then obtain *p* **where** *path M q p* **and** *p-io p* = *xy#io*
unfolding *LS.simps mem-Collect-eq* **by** (*metis* (*no-types*, *lifting*))

let *?t* = *hd p*
let *?p* = *tl p*
let *?q'* = {*q'* $\in$ *states M* . *LS M* (*t-target ?t*) = *LS M q'*}

have *p* = *?t* # *?p* **and** *p-io ?p* = *io* **and** *t-input ?t* = *x* **and** *t-output ?t* = *y*
using ‹*p-io p* = *xy#io*› **unfolding** ‹*xy* = (*x*,*y*)› **by** *auto*
moreover have *?t* $\in$ *transitions M* **and** *path M* (*t-target ?t*) *?p* **and** *t-source ?t* = *q*
using ‹*path M q p*› *path-cons-elim*[*of M q ?t ?p*] *calculation* **by** *auto*
ultimately have [(*x*,*y*)] $\in$ *LS M q*
unfolding *LS-single-transition*[*of x y M q*] **by** *auto*
then have *io* $\in$ *LS M* (*t-target ?t*)
using *observable-language-next*[*OF* - ‹*observable M*›, *of* (*x*,*y*) *io*, *OF* - ‹*?t* $\in$ *transitions M*›]
‹*xy#io* $\in$ *LS M q*›
unfolding ‹*xy* = (*x*,*y*)› ‹*t-source ?t* = *q*› ‹*t-input ?t* = *x*› ‹*t-output ?t* = *y*›
by (*metis* ‹*?t* $\in$ *FSM.transitions M*› *from-FSM-language fsm-transition-target fst-conv snd-conv*)
then have *io* $\in$ *LS M'* *?q'*
using *Cons.IH*[*OF fsm-transition-target*[*OF* ‹*?t* $\in$ *transitions M*›]] **by** *blast*
then obtain *p'* **where** *path M'* *?q'* *p'* **and** *p-io p'* = *io*
by *auto*

have $*$: $(\{q' \in states\ M\ .\ LS\ M\ q = LS\ M\ q'\},x,y,\{q' \in states\ M\ .\ LS\ M\ (t\text{-}target\ ?t) = LS\ M\ q'\}) \in transitions\ M'$
using ‹$?t \in transitions\ M$› ‹$t\text{-}source\ ?t = q$› ‹$t\text{-}input\ ?t = x$› ‹$t\text{-}output\ ?t = y$›
unfolding *assms*(*1*) **by** *auto*

show $xy\#io \in LS\ M'\ \{q' \in states\ M\ .\ LS\ M\ q = LS\ M\ q'\}$
using *LS-prepend-transition*[*OF* $*$] **unfolding** *snd-conv fst-conv* ‹$xy = (x,y)$›
using ‹$io \in LS\ M'\ ?q'$› **by** *blast*
qed
moreover have $xy\#io \in LS\ M'\ \{q' \in states\ M\ .\ LS\ M\ q = LS\ M\ q'\} \Longrightarrow xy\#io \in LS\ M\ q$
proof −
let $?q = \{q' \in states\ M\ .\ LS\ M\ q = LS\ M\ q'\}$
assume $xy\#io \in LS\ M'\ ?q$
then obtain p **where** $path\ M'\ ?q\ p$ **and** $p\text{-}io\ p = xy\#io$
unfolding *LS.simps mem-Collect-eq* **by** (*metis* (*no-types*, *lifting*))

let $?t = hd\ p$
let $?p = tl\ p$

have $p = ?t\ \#\ ?p$ **and** $p\text{-}io\ ?p = io$ **and** $t\text{-}input\ ?t = x$ **and** $t\text{-}output\ ?t = y$
using ‹$p\text{-}io\ p = xy\#io$› **unfolding** ‹$xy = (x,y)$› **by** *auto*
then have $path\ M'\ ?q\ (?t\#?p)$
using ‹$path\ M'\ ?q\ p$› **by** *auto*
then have $?t \in transitions\ M'$ **and** $path\ M'\ (t\text{-}target\ ?t)\ ?p$ **and** $t\text{-}source\ ?t = ?q$
by *force*+
then have $io \in LS\ M'\ (t\text{-}target\ ?t)$
using ‹$p\text{-}io\ ?p = io$› **by** *auto*

obtain *t0* **where** *t0-def*: $?t = (\lambda\ t\ .\ (\{q \in states\ M\ .\ LS\ M\ q = LS\ M\ (t\text{-}source\ t)\}\ ,\ t\text{-}input\ t,\ t\text{-}output\ t,\ \{q \in states\ M\ .\ LS\ M\ q = LS\ M\ (t\text{-}target\ t)\}))\ t0$
and $t0 \in transitions\ M$
using ‹$?t \in transitions\ M'$›
unfolding *assms*(*1*)
by *auto*
then have $t\text{-}source\ t0 \in ?q$
using ‹$t\text{-}source\ ?t = ?q$›
by (*metis* (*mono-tags*, *lifting*) *fsm-transition-source fst-eqD mem-Collect-eq*)

then have $LS\ M\ q = LS\ M\ (t\text{-}source\ t0)$
by *auto*

moreover have $[(x,y)] \in LS\ M\ (t\text{-}source\ t0)$
using *t0-def* ‹*t-input ?t = x*› ‹*t0 ∈ transitions M*› ‹*t-output ?t = y*› ‹*t-source t0 ∈ ?q*› **unfolding** *LS-single-transition* **by** *auto*
ultimately obtain t **where** $t \in transitions\ M$ **and** $t\text{-}source\ t = q$ **and** $t\text{-}input\ t = x$ **and** $t\text{-}output\ t = y$
by (*metis LS-single-transition*)

have $LS\ M\ (t\text{-}target\ t) = LS\ M\ (t\text{-}target\ t0)$
using *observable-transition-target-language-eq*[*OF* -‹*t ∈ transitions M*› ‹*t0 ∈ transitions M*› - - ‹*observable M*›]
using ‹$LS\ M\ q = LS\ M\ (t\text{-}source\ t0)$›
unfolding ‹$t\text{-}source\ t = q$› ‹$t\text{-}input\ t = x$› ‹$t\text{-}output\ t = y$›
using *t0-def* ‹*t-input ?t = x*› ‹*t-output ?t = y*›
by *auto*
moreover have $t\text{-}target\ ?t = \{q' \in FSM.states\ M.\ LS\ M\ (t\text{-}target\ t) = LS\ M\ q'\}$
using *calculation t0-def* **by** *fastforce*
ultimately have $io \in LS\ M\ (t\text{-}target\ t)$
using *Cons.IH*[*OF fsm-transition-target*[*OF* ‹*t ∈ transitions M*›]]
‹$io \in LS\ M'\ (t\text{-}target\ ?t)$›
by *auto*
then show $xy\#io \in LS\ M\ q$
unfolding ‹$t\text{-}source\ t = q$›[*symmetric*] ‹$xy = (x,y)$›
using ‹$t\text{-}input\ t = x$› ‹$t\text{-}output\ t = y$›
using *LS-prepend-transition* ‹*t ∈ FSM.transitions M*›
by *blast*
qed

ultimately show *?case*
by *blast*
qed
qed

have $L\ M' = LS\ M'\ \{q' \in states\ M\ .\ LS\ M\ (initial\ M) = LS\ M\ q'\}$
using *assms*(*3*)
by (*metis* (*mono-tags*, *lifting*) *Collect-cong*)
then show $L\ M = L\ M'$
using *ls-prop*[*OF fsm-initial*] **by** *blast*

show *minimal M′*
proof −
have $\bigwedge\ q\ q'\ .\ q \in states\ M' \Longrightarrow q' \in states\ M' \Longrightarrow LS\ M'\ q = LS\ M'\ q' \Longrightarrow q = q'$
proof −

fix $q\ q'$ **assume** $q \in states\ M'$ **and** $q' \in states\ M'$ **and** $LS\ M'\ q = LS\ M'\ q'$

obtain qM **where** $q = \{q \in states\ M\ .\ LS\ M\ qM = LS\ M\ q\}$ **and** $qM \in states\ M$

using ‹$q \in$ *states* M'› *assms*(*2*) **by** *auto*
obtain qM' **where** $q' = \{q \in$ *states* M . *LS* M $qM' =$ *LS* M $q\}$ **and** $qM' \in$ *states* M
using ‹$q' \in$ *states* M'› *assms*(*2*) **by** *auto*

have *LS* M $qM =$ *LS* M' q
using *ls-prop*[*OF* ‹$qM \in$ *states* M›] **unfolding** ‹$q = \{q \in$ *states* M . *LS* M $qM =$ *LS* M $q\}$› **by** *blast*
moreover have *LS* M $qM' =$ *LS* M' q'
using *ls-prop*[*OF* ‹$qM' \in$ *states* M›] **unfolding** ‹$q' = \{q \in$ *states* M . *LS* M $qM' =$ *LS* M $q\}$› **by** *blast*
ultimately have *LS* M $qM =$ *LS* M qM'
using ‹*LS* M' $q =$ *LS* M' q'› **by** *blast*
then show $q = q'$
unfolding ‹$q = \{q \in$ *states* M . *LS* M $qM =$ *LS* M $q\}$› ‹$q' = \{q \in$ *states* M . *LS* M $qM' =$ *LS* M $q\}$› **by** *blast*
qed
then show *?thesis*
unfolding *minimal-alt-def* **by** *blast*
qed
qed

fun *minimise* :: ($'a$ :: *linorder*,$'b$ :: *linorder*,$'c$ :: *linorder*) *fsm* ⇒ ($'a$ *set*,$'b$,$'c$) *fsm*
where
minimise M = (*let*
eq-class = *ofsm-table-fix* M (λq . *states* M) *0*;
ts = (λ t . (*eq-class* (*t-source* t), *t-input* t, *t-output* t, *eq-class* (*t-target* t))) ‘ (*transitions* M);
q0 = *eq-class* (*initial* M);
eq-states = *eq-class* |‘| *fstates* M;
M' = *create-unconnected-fsm-from-fsets q0 eq-states* (*finputs* M) (*foutputs* M)
in add-transitions M' *ts*)

lemma *minimise-initial-partition* :
equivalence-relation-on-states M (λq . *states* M)
proof −
let *?r* = $\{(q1,q2) \mid q1\ q2$. $q1 \in$ *states* $M \wedge q2 \in (\lambda q$. *states* $M)$ $q1\}$

have *refl-on* (*FSM.states* M) *?r*
unfolding *refl-on-def* **by** *blast*
moreover have *sym ?r*
unfolding *sym-def* **by** *blast*
moreover have *trans ?r*
unfolding *trans-def* **by** *blast*
ultimately show *?thesis*
unfolding *equivalence-relation-on-states-def equiv-def* **by** *auto*

qed

lemma *minimise-props*:
assumes *observable M*
shows *initial* (*minimise M*) = {q' ∈ *states M* . *LS M* q' = *LS M* (*initial M*)}
and *states* (*minimise M*) = (λq . {q' ∈ *states M* . *LS M* q = *LS M* q'}) ‘ *states M*
and *inputs* (*minimise M*) = *inputs M*
and *outputs* (*minimise M*) = *outputs M*
and *transitions* (*minimise M*) = (λ *t* . ({q ∈ *states M* . *LS M* q = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {q ∈ *states M* . *LS M* q = *LS M* (*t-target t*)})) ‘ *transitions M*
proof −

let *?f* = λq . *states M*

define *eq-class* **where** *eq-class* = *ofsm-table-fix M* (λq . *states M*) *0*
moreover define M' **where** M'*-def*: M' = *create-unconnected-fsm-from-fsets* (*eq-class* (*initial M*)) (*eq-class* |‘| *fstates M*) (*finputs M*) (*foutputs M*)
ultimately have ∗: *minimise M* = *add-transitions* M' ((λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*))
by *auto*

have ∗∗: ⋀ q . q ∈ *states M* ⟹ *eq-class* q = {q' ∈ *FSM.states M*. *LS M* q = *LS M* q'}
using *ofsm-table-fix-set*[*OF - assms minimise-initial-partition*] ‹*eq-class* = *ofsm-table-fix M ?f 0*› *after-is-state*[*OF* ‹*observable M*›] **by** *blast*
then have ∗∗∗∗: ⋀ q . q ∈ *states M* ⟹ *eq-class* q = {q' ∈ *FSM.states M*. *LS M* q' = *LS M* q}
using *ofsm-table-fix-set*[*OF - assms*] ‹*eq-class* = *ofsm-table-fix M ?f 0*› **by** *blast*

have ∗∗∗: (*eq-class* (*initial M*)) |∈| (*eq-class* |‘| *fstates M*)
using *fsm-initial*[*of M*] *fstates-set* **by** *fastforce*

have *m1*:*initial* M' = {q' ∈ *states M* . *LS M* q' = *LS M* (*initial M*)}
by (*metis* (*mono-tags*) ∗∗∗ ∗∗∗∗ M'*-def create-unconnected-fsm-from-fsets-simps*(*1*) *fsm-initial*)

have *m2*: *states* M' = (λq . {q' ∈ *states M* . *LS M* q = *LS M* q'}) ‘ *states M*
unfolding M'*-def*
proof −
have *FSM.states* (*FSM.create-unconnected-fsm-from-fsets* (*eq-class* (*FSM.initial M*)) (*eq-class* |‘| *fstates M*) (*finputs M*) (*foutputs M*)) = *eq-class* ‘ *FSM.states M*
by (*metis* (*no-types*) ∗∗∗ *create-unconnected-fsm-from-fsets-simps*(*2*) *fset.set-map fstates-set*)
then show *FSM.states* (*FSM.create-unconnected-fsm-from-fsets* (*eq-class* (*FSM.initial M*)) (*eq-class* |‘| *fstates M*) (*finputs M*) (*foutputs M*)) = (λ*a*. {*aa* ∈ *FSM.states*

M. *LS M a* = *LS M aa*}) ‘ *FSM.states M*
using ∗∗ **by** *force*
qed

have *m3*: *inputs M′* = *inputs M*
using *create-unconnected-fsm-from-fsets-simps*(*3*)[*OF* ∗∗∗] *finputs-set* **unfolding** *M′-def* **by** *metis*

have *m4*: *outputs M′* = *outputs M*
using *create-unconnected-fsm-from-fsets-simps*(*4*)[*OF* ∗∗∗] *foutputs-set* **unfolding** *M′-def* **by** *metis*

have *m5*: *transitions M′* = {}
using *create-unconnected-fsm-from-fsets-simps*(*5*)[*OF* ∗∗∗] **unfolding** *M′-def* **by** *force*

let *?ts* = ((λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*))
have *wf*: ⋀ *t* . *t* ∈*?ts* ⟹ *t-source t* ∈ *states M′* ∧ *t-input t* ∈ *inputs M′* ∧ *t-output t* ∈ *outputs M′* ∧ *t-target t* ∈ *states M′*
proof −
fix *t* **assume** *t* ∈ *?ts*
then obtain *tM* **where** *tM* ∈ *transitions M*
and ∗: *t* = (λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) *tM*
by *blast*

have *t-source t* ∈ *states M′*
using *fsm-transition-source*[*OF* ‹*tM* ∈ *transitions M*›]
unfolding *m2* ∗ ∗∗[*OF fsm-transition-source*[*OF* ‹*tM* ∈ *transitions M*›]] **by** *auto*
moreover have *t-input t* ∈ *inputs M′*
unfolding *m3* ∗ **using** *fsm-transition-input*[*OF* ‹*tM* ∈ *transitions M*›] **by** *auto*
moreover have *t-output t* ∈ *outputs M′*
unfolding *m4* ∗ **using** *fsm-transition-output*[*OF* ‹*tM* ∈ *transitions M*›] **by** *auto*
moreover have *t-target t* ∈ *states M′*
using *fsm-transition-target*[*OF* ‹*tM* ∈ *transitions M*›]
unfolding *m2* ∗ ∗∗[*OF fsm-transition-target*[*OF* ‹*tM* ∈ *transitions M*›]] **by** *auto*
ultimately show *t-source t* ∈ *states M′* ∧ *t-input t* ∈ *inputs M′* ∧ *t-output t* ∈ *outputs M′* ∧ *t-target t* ∈ *states M′*
by *simp*
qed

show *initial* (*minimise M*) = {*q′* ∈ *states M* . *LS M q′* = *LS M* (*initial M*)}
using *add-transitions-simps*(*1*)[*OF wf*] **unfolding** ∗ *m1* **.**

show *states* (*minimise M*) = (λq . {$q' \in$ *states M* . *LS M q* = *LS M q'*}) ‘ *states M*
using *add-transitions-simps*(*2*)[*OF wf*] **unfolding** ∗ *m2* **.**

show *inputs* (*minimise M*) = *inputs M*
using *add-transitions-simps*(*3*)[*OF wf*] **unfolding** ∗ *m3* **.**

show *outputs* (*minimise M*) = *outputs M*
using *add-transitions-simps*(*4*)[*OF wf*] **unfolding** ∗ *m4* **.**

show *transitions* (*minimise M*) = (λ *t* . ({$q \in$ *states M* . *LS M q* = *LS M* (*t-source t*)} , *t-input t*, *t-output t*, {$q \in$ *states M* . *LS M q* = *LS M* (*t-target t*)})) ‘ *transitions M*
using *add-transitions-simps*(*5*)[*OF wf*] ∗∗∗∗[*OF fsm-transition-source*] ∗∗∗∗[*OF fsm-transition-target*] **unfolding** ∗ *m5* **by** *auto*
qed

lemma *minimise-observable*:
assumes *observable M*
shows *observable* (*minimise M*)
using *language-equivalence-classes-preserve-observability*[*OF minimise-props*(*5*)[*OF assms*] *assms*]
by *assumption*

lemma *minimise-minimal*:
assumes *observable M*
shows *minimal* (*minimise M*)
using *language-equivalence-classes-retain-language-and-induce-minimality*(*2*)[*OF minimise-props*(*5*,*2*,*1*)[*OF assms*] *assms*]
by *assumption*

lemma *minimise-language*:
assumes *observable M*
shows *L* (*minimise M*) = *L M*
using *language-equivalence-classes-retain-language-and-induce-minimality*(*1*)[*OF minimise-props*(*5*,*2*,*1*)[*OF assms*] *assms*]
by *blast*

lemma *minimal-observable-code* :
assumes *observable M*
shows *minimal M* = ($\forall$ $q \in$ *states M* . *ofsm-table-fix M* (λq . *states M*) *0 q* = {*q*})
proof
show *minimal M* $\Longrightarrow$ ($\forall$ $q \in$ *states M* . *ofsm-table-fix M* (λq . *states M*) *0 q* = {*q*})
proof
fix *q* **assume** *minimal M* **and** $q \in$ *states M*

then show *ofsm-table-fix M* (λq . *states M*) *0 q* = {*q*}
unfolding *ofsm-table-fix-set*[*OF* ‹*q* ∈ *states M*› ‹*observable M*› *minimise-initial-partition*]
minimal-alt-def
using *after-is-state*[*OF* ‹*observable M*›]
by *blast*
qed

show ∀ *q*∈*FSM.states M*. *ofsm-table-fix M* (λq . *states M*) *0 q* = {*q*} ⟹ *minimal M*
using *ofsm-table-fix-set*[*OF* - ‹*observable M*› *minimise-initial-partition*] *after-is-state*[*OF* ‹*observable M*›]
unfolding *minimal-alt-def*
by *blast*
qed

lemma *minimise-states-subset* :
assumes *observable M*
and *q* ∈ *states* (*minimise M*)
shows *q* ⊆ *states M*
using *assms*(*2*)
unfolding *minimise-props*[*OF assms*(*1*)]
by *auto*

lemma *minimise-states-finite* :
assumes *observable M*
and *q* ∈ *states* (*minimise M*)
shows *finite q*
using *minimise-states-subset*[*OF assms*] *fsm-states-finite*[*of M*]
using *finite-subset* **by** *auto*

end

7 Computation of distinguishing traces based on OFSM tables

This theory implements an algorithm for finding minimal length distinguishing traces for observable minimal FSMs based on OFSM tables.

theory *Distinguishability*
imports *Minimisation HOL.List*
begin

7.1 Finding Diverging OFSM Tables

definition *ofsm-table-fixpoint-value* :: ('*a*,'*b*,'*c*) *fsm* ⇒ *nat* **where**
ofsm-table-fixpoint-value M = (*SOME k* . (∀ *q* . *q* ∈ *states M* ⟶ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) *k q*) ∧ (∀ *q k'* . *q* ∈ *states M* ⟶ *k'* ≥ *k* ⟶ *ofsm-table M* (λq . *states M*) *k' q* = *ofsm-table M* (λq . *states M*) *k q*))

function *find-first-distinct-ofsm-table-gt* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
find-first-distinct-ofsm-table-gt M q1 q2 k =
(*if q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M* $\wedge$ ((*ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*))
then (*if ofsm-table M* (λq . *states M*) *k q1* $\neq$ *ofsm-table M* (λq . *states M*) *k q2*
then k
else find-first-distinct-ofsm-table-gt M q1 q2 (*Suc k*))
else 0)
using *prod-cases4* **by** *blast+*
termination
proof −
{
fix *M* :: $('a,'b,'c)$ *fsm*
fix *q1 q2 k*
assume *q1* $\in$ *FSM.states M* $\wedge$ *q2* $\in$ *FSM.states M* $\wedge$ *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
ofsm-table M (λq . *states M*) *k q1* = *ofsm-table M* (λq . *states M*) *k q2*
then have *q1* $\in$ *FSM.states M* **and** *q2* $\in$ *FSM.states M*
and *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
by *force+*

let *?k* = *ofsm-table-fixpoint-value M*
obtain k' **where** $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) k' *q* **and** $\bigwedge$ *q* k'' . *q* $\in$ *states M* $\Longrightarrow$ $k'' \geq k'$ $\Longrightarrow$ *ofsm-table M* (λq . *states M*) k'' *q* = *ofsm-table M* (λq . *states M*) k' *q*
using *ofsm-table-fix-length*[*of M* (λq . *states M*)]
by *blast*
then have ($\forall$ *q* . *q* $\in$ *states M* $\longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) k' *q*) $\wedge$ ($\forall$ *q* k'' . *q* $\in$ *states M* $\longrightarrow$ $k'' \geq k'$ $\longrightarrow$ *ofsm-table M* (λq . *states M*) k'' *q* = *ofsm-table M* (λq . *states M*) k' *q*)
by *blast*
then have $*$:$\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) *?k q*
and $**$: $\bigwedge$ *q* k'' . *q* $\in$ *states M* $\Longrightarrow$ $k'' \geq$ *?k* $\Longrightarrow$ *ofsm-table M* (λq . *states M*) k'' *q* = *ofsm-table M* (λq . *states M*) *?k q*
using *some-eq-imp*[*of* λ *k* . ($\forall$ *q* . *q* $\in$ *states M* $\longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) *k q*) $\wedge$ ($\forall$ *q* k' . *q* $\in$ *states M* $\longrightarrow$ $k' \geq k$ $\longrightarrow$ *ofsm-table M* (λq . *states M*) k' *q* = *ofsm-table M* (λq . *states M*) *k q*) *?k* k']
unfolding *ofsm-table-fixpoint-value-def*
by *blast+*

have *?k* > *k*

using $*$
‹*ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*›
‹*ofsm-table M* (λq . *states M*) *k q1* = *ofsm-table M* (λq . *states M*) *k q2*›
$**$[*OF* ‹*q1* $\in$ *states M*›]
$**$[*OF* ‹*q2* $\in$ *states M*›]
by (*metis* ‹*q1* $\in$ *FSM.states M* $\wedge$ *q2* $\in$ *FSM.states M* $\wedge$ *ofsm-table-fix M* (λq. *FSM.states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq. *FSM.states M*) *0 q2*› *leI*)
then have *?k* − *Suc k* < *?k* − *k*
by *simp*
} **note** *t* = *this*

show *?thesis*
apply (*relation measure* (λ (*M*, *q1*, *q2*, *k*) . *ofsm-table-fixpoint-value M* − *k*))
apply *auto*[*1*]
apply (*simp del*: *observable.simps ofsm-table-fix.simps*)
by (*erule t*)
qed

partial-function (*tailrec*) *find-first-distinct-ofsm-table-no-check* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
find-first-distinct-ofsm-table-no-check-def[*code*]:
find-first-distinct-ofsm-table-no-check M q1 q2 k =
(*if ofsm-table M* (λq . *states M*) *k q1* $\neq$ *ofsm-table M* (λq . *states M*) *k q2*
then k
else find-first-distinct-ofsm-table-no-check M q1 q2 (*Suc k*))

fun *find-first-distinct-ofsm-table-gt'* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
find-first-distinct-ofsm-table-gt' M q1 q2 k =
(*if q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M* $\wedge$ ((*q2* $\notin$ *ofsm-table-fix M* (λq . *states M*) *0 q1*))
then find-first-distinct-ofsm-table-no-check M q1 q2 k
else 0)

lemma *find-first-distinct-ofsm-table-gt-code*[*code*] :
find-first-distinct-ofsm-table-gt M q1 q2 k = *find-first-distinct-ofsm-table-gt' M q1 q2 k*
proof (*cases q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M* $\wedge$ ((*ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*)))
case *False*
have *find-first-distinct-ofsm-table-gt M q1 q2 k* = *0*
using *False*
by (*metis find-first-distinct-ofsm-table-gt.simps*)
moreover have *find-first-distinct-ofsm-table-gt' M q1 q2 k* = *0*
proof (*cases q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M*)
case *True*

then have *q1* ∈ *FSM.states M* **and** *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* = *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
using *False* **by** *force+*
then have *q2* ∈ *ofsm-table-fix M* (λ*q* . *states M*) *0 q1*
using *ofsm-table-fix-eq-if-elem*[*of q1 M q2*]
using *minimise-initial-partition*
by *blast*
then show *?thesis*
by (*metis find-first-distinct-ofsm-table-gt′.simps*)
next
case *False*
then show *?thesis* **by** (*meson find-first-distinct-ofsm-table-gt′.simps*)
qed
ultimately show *?thesis*
by *simp*
next
case *True*
then have *q1* ∈ *FSM.states M* **and** *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
by *force+*
then have *q2* ∉ *ofsm-table-fix M* (λ*q* . *states M*) *0 q1*
using *ofsm-table-fix-eq-if-elem*[*of q1 M q2*]
using *minimise-initial-partition*
by *blast*

obtain *k′* **where** ⋀ *q* . *q* ∈ *states M* ⟹ *ofsm-table-fix M* (λ*q* . *states M*) *0 q* = *ofsm-table M* (λ*q* . *states M*) *k′ q* **and** ⋀ *q k″* . *q* ∈ *states M* ⟹ *k″* ≥ *k′* ⟹ *ofsm-table M* (λ*q* . *states M*) *k″ q* = *ofsm-table M* (λ*q* . *states M*) *k′ q*
using *ofsm-table-fix-length*[*of M* (λ*q* . *states M*)]
by *blast*

have *f1*: *find-first-distinct-ofsm-table-gt M q1 q2* =
(λ*x*. *if ofsm-table M* (λ*q* . *states M*) *x q1* ≠ *ofsm-table M* (λ*q* . *states M*) *x q2*
then x
else find-first-distinct-ofsm-table-gt M q1 q2 (*Suc x*))
using *find-first-distinct-ofsm-table-gt.simps*[*of M q1 q2*]
using *True*
by *meson*

have *f2*: *find-first-distinct-ofsm-table-no-check M q1 q2* =
(λ*x*. *if ofsm-table M* (λ*q* . *states M*) *x q1* ≠ *ofsm-table M* (λ*q* . *states M*) *x q2*
then x
else find-first-distinct-ofsm-table-no-check M q1 q2 (*Suc x*))
using *True find-first-distinct-ofsm-table-no-check.simps*[*of M q1 q2*]
by *meson*

have ($\bigwedge x.\ k' \leq x \Longrightarrow$ *ofsm-table M* (λq . *states M*) *x q1* $\neq$ *ofsm-table M* (λq . *states M*) *x q2*)
using ‹$\bigwedge$ *q k''* . *q* $\in$ *states M* $\Longrightarrow$ *k''* $\geq$ *k'* $\Longrightarrow$ *ofsm-table M* (λq . *states M*) *k''* *q* = *ofsm-table M* (λq . *states M*) *k'* *q*› ‹*q1* $\in$ *FSM.states M*› ‹*q2* $\in$ *FSM.states M*›
by (*metis True* ‹$\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) *k'* *q*›)

have *find-first-distinct-ofsm-table-gt' M q1 q2 k* = *find-first-distinct-ofsm-table-no-check M q1 q2 k*
using *True* ‹*q2* $\notin$ *ofsm-table-fix M* (λq . *states M*) *0 q1*› *find-first-distinct-ofsm-table-gt'.simps*[*of M*]
by *meson*
then show *?thesis*
using *recursion-renaming-helper*[*OF f1 f2* ‹($\bigwedge x.\ k' \leq x \Longrightarrow$ *ofsm-table M* (λq . *states M*) *x q1* $\neq$ *ofsm-table M* (λq . *states M*) *x q2*)›, *of k'*]
by *simp*
qed

lemma *find-first-distinct-ofsm-table-gt-is-first-gt* :
assumes *q1* $\in$ *FSM.states M*
and *q2* $\in$ *FSM.states M*
and *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
shows *ofsm-table M* (λq . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q1* $\neq$ *ofsm-table M* (λq . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q2*
and *k* $\leq$ *k'* $\Longrightarrow$ *k'* < (*find-first-distinct-ofsm-table-gt M q1 q2 k*) $\Longrightarrow$ *ofsm-table M* (λq . *states M*) *k' q1* = *ofsm-table M* (λq . *states M*) *k' q2*
proof −

have *f*: *find-first-distinct-ofsm-table-gt M q1 q2* =
(λx. *if ofsm-table M* (λq . *states M*) *x q1* $\neq$ *ofsm-table M* (λq . *states M*) *x q2*
then x
else find-first-distinct-ofsm-table-gt M q1 q2 (*Suc x*))
using *assms find-first-distinct-ofsm-table-gt.simps*[*of M*]
by *meson*

obtain *kx* **where** $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ *ofsm-table-fix M* (λq . *states M*) *0 q* = *ofsm-table M* (λq . *states M*) *kx q* **and** $\bigwedge$ *q k''* . *q* $\in$ *states M* $\Longrightarrow$ *k''* $\geq$ *kx* $\Longrightarrow$ *ofsm-table M* (λq . *states M*) *k''* *q* = *ofsm-table M* (λq . *states M*) *kx q*
using *ofsm-table-fix-length*[*of M* (λq . *states M*)]
by *blast*
have *P*: ($\bigwedge x.\ kx \leq x \Longrightarrow$ *ofsm-table M* (λq . *states M*) *x q1* $\neq$ *ofsm-table M* (λq . *states M*) *x q2*)
using ‹$\bigwedge$ *q k''* . *q* $\in$ *states M* $\Longrightarrow$ *k''* $\geq$ *kx* $\Longrightarrow$ *ofsm-table M* (λq . *states M*) *k''* *q* = *ofsm-table M* (λq . *states M*) *kx q*› ‹*q1* $\in$ *FSM.states M*› ‹*q2* $\in$ *FSM.states*

M›
by (*metis assms* ‹$\bigwedge$ *q* . *q* ∈ *states M* ⟹ *ofsm-table-fix M* (λ*q* . *states M*) *0 q* = *ofsm-table M* (λ*q* . *states M*) *kx q*›)

show *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table-gt M q1 q2 k*) *q2*
using *minimal-fixpoint-helper*(*1*)[*OF f P*, *of kx k*] **.**

show *k* ≤ *k′* ⟹ *k′* < (*find-first-distinct-ofsm-table-gt M q1 q2 k*) ⟹ *ofsm-table M* (λ*q* . *states M*) *k′ q1* = *ofsm-table M* (λ*q* . *states M*) *k′ q2*
using *minimal-fixpoint-helper*(*2*)[*OF f P*, *of kx k k′*]
by *auto*
qed

abbreviation(*input*) *find-first-distinct-ofsm-table M q1 q2* ≡ *find-first-distinct-ofsm-table-gt M q1 q2 0*

lemma *find-first-distinct-ofsm-table-is-first* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
shows *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table M q1 q2*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*find-first-distinct-ofsm-table M q1 q2*) *q2*
and *k′* < (*find-first-distinct-ofsm-table M q1 q2*) ⟹ *ofsm-table M* (λ*q* . *states M*) *k′ q1* = *ofsm-table M* (λ*q* . *states M*) *k′ q2*
using *find-first-distinct-ofsm-table-gt-is-first-gt*[*OF assms*, *of 0*] **by** *blast+*

fun *select-diverging-ofsm-table-io* :: (*′a*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *nat* ⇒ (*′b* × *′c*) × (*′a option* × *′a option*) **where**
select-diverging-ofsm-table-io M q1 q2 k = (*let*
ins = *inputs-as-list M*;
outs = *outputs-as-list M*;
table = *ofsm-table M* (λ*q* . *states M*) (*k*−*1*);
f = (λ (*x*,*y*) . *case* (*h-obs M q1 x y*, *h-obs M q2 x y*)
of
(*Some q1′*, *Some q2′*) ⇒ *if table q1′* ≠ *table q2′*
then Some ((*x*,*y*),(*Some q1′*, *Some q2′*))
else None |
(*None*,*None*) ⇒ *None* |
(*Some q1′*, *None*) ⇒ *Some* ((*x*,*y*),(*Some q1′*, *None*)) |
(*None*, *Some q2′*) ⇒ *Some* ((*x*,*y*),(*None*, *Some q2′*)))
in
hd (*List.map-filter f* (*List.product ins outs*)))

lemma *select-diverging-ofsm-table-io-Some* :

assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q2*
obtains *x y*
where *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
and ⋀ *q1′ q2′* . *h-obs M q1 x y* = *Some q1′* ⟹ *h-obs M q2 x y* = *Some q2′* ⟹ *ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
and *h-obs M q1 x y* ≠ *None* ∨ *h-obs M q2 x y* ≠ *None*
proof −

let *?res* = *select-diverging-ofsm-table-io M q1 q2* (*Suc k*)

define *f* **where** *f*: *f* = (λ (*x*,*y*) . *case* (*h-obs M q1 x y*, *h-obs M q2 x y*)
of
(*Some q1′*, *Some q2′*) ⇒ *if ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
then Some ((*x*,*y*),(*Some q1′*, *Some q2′*))
else None |
(*None*,*None*) ⇒ *None* |
(*Some q1′*, *None*) ⇒ *Some* ((*x*,*y*),(*Some q1′*, *None*)) |
(*None*, *Some q2′*) ⇒ *Some* ((*x*,*y*),(*None*, *Some q2′*)))

have *f1*: ⋀ *x y* . *f* (*x*,*y*) ≠ *None* ⟹ *f* (*x*,*y*) = *Some* ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
proof −
fix *x y* **assume** *f* (*x*,*y*) ≠ *None*
then show *f* (*x*,*y*) = *Some* ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
unfolding *f* **by** (*cases h-obs M q1 x y*; *cases h-obs M q2 x y*; *auto*)
qed

have *f2* : ⋀ *q1′ q2′ x y* . *f* (*x*,*y*) = *Some* ((*x*,*y*),(*Some q1′*, *Some q2′*)) ⟹ *ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
proof −
fix *q1′ q2′ x y* **assume** ∗: *f* (*x*,*y*) = *Some* ((*x*,*y*),(*Some q1′*, *Some q2′*))
then have ∗∗: *f* (*x*,*y*) = *Some* ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
using *f1* **by** *auto*
show *ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
using ∗ ∗∗ **unfolding** *f* **by** (*cases h-obs M q1 x y*; *cases h-obs M q2 x y*; *auto*)
qed

have *f3*: ⋀ *x y* . *f* (*x*,*y*) ≠ *None* ⟹ *h-obs M q1 x y* ≠ *None* ∨ *h-obs M q2 x y* ≠ *None*
proof −
fix *x y* **assume** *f* (*x*,*y*) ≠ *None*
then show *h-obs M q1 x y* ≠ *None* ∨ *h-obs M q2 x y* ≠ *None*

unfolding *f* **by** (*cases h-obs M q1 x y*; *cases h-obs M q2 x y*; *auto*)
qed

have ∗: *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = *hd* (*List.map-filter f* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*)))
unfolding *f select-diverging-ofsm-table-io.simps Let-def*
using *diff-Suc-1* **by** *presburger*

let *?P* = ∀ *x y* . *x* ∈ *inputs M* ⟶ *y* ∈ *outputs M* ⟶ (*h-obs M q1 x y* = *None* ⟷ *h-obs M q2 x y* = *None*)
show *?thesis* **proof** (*cases ?P*)
case *False*
then obtain *x y* **where** *x* ∈ *inputs M* **and** *y* ∈ *outputs M* **and** ¬ (*h-obs M q1 x y* = *None* ⟷ *h-obs M q2 x y* = *None*)
by *blast*
then consider *h-obs M q1 x y* = *None* ∧ (∃ *q2′* . *h-obs M q2 x y* = *Some q2′*)
|
h-obs M q2 x y = *None* ∧ (∃ *q1′* . *h-obs M q1 x y* = *Some q1′*)
by *fastforce*
then show *?thesis* **proof** *cases*
case *1*
then obtain *q2′* **where** *h-obs M q1 x y* = *None* **and** *h-obs M q2 x y* = *Some q2′* **by** *blast*
then have *f* (*x*,*y*) = *Some* ((*x*,*y*),(*None*, *Some q2′*))
unfolding *f* **by** *force*
moreover have (*x*,*y*) ∈ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
using ‹*y* ∈ *outputs M*› *outputs-as-list-set*[*of M*]
using ‹*x* ∈ *inputs M*› *inputs-as-list-set*[*of M*]
using *image-iff* **by** *fastforce*
ultimately have (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))) ≠ []
unfolding *List.map-filter-def*
by (*metis* (*mono-tags*, *lifting*) *Nil-is-map-conv filter-empty-conv option.discI*)
then have ∗∗: *?res* ∈ *set* (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*)))
unfolding ∗ **using** *hd-in-set* **by** *simp*

obtain *xR yR* **where** (*xR*,*yR*) ∈ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
and *res*: *f* (*xR*,*yR*) = *Some ?res*
using *map-filter-elem*[*OF* ∗∗]
by (*metis prod.exhaust-sel*)

have *p1*: *?res* = ((*xR*,*yR*),(*h-obs M q1 xR yR*, *h-obs M q2 xR yR*))
using *res f1*
by (*metis option.distinct*(*1*) *option.sel*)

then have *p2*: $\bigwedge$ *q1′ q2′* . *h-obs M q1 xR yR* = *Some q1′* $\Longrightarrow$ *h-obs M q2 xR yR* = *Some q2′* $\Longrightarrow$ *ofsm-table M* (λ*q* . *states M*) *k q1′* $\neq$ *ofsm-table M* (λ*q* . *states M*) *k q2′*
using *res f1 f2* **by** *auto*
have *p3*: *h-obs M q1 xR yR* $\neq$ *None* $\vee$ *h-obs M q2 xR yR* $\neq$ *None*
using *res f3* **by** *blast*

show *?thesis* **using** *that p1 p2 p3* **by** *blast*
next
case *2*
then obtain *q1′* **where** *h-obs M q2 x y* = *None* **and** *h-obs M q1 x y* = *Some q1′* **by** *blast*
then have *f* (*x*,*y*) = *Some* ((*x*,*y*),(*Some q1′*, *None*))
unfolding *f* **by** *force*
moreover have (*x*,*y*) $\in$ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
using ‹*y* $\in$ *outputs M*› *outputs-as-list-set*[*of M*]
using ‹*x* $\in$ *inputs M*› *inputs-as-list-set*[*of M*]
using *image-iff* **by** *fastforce*
ultimately have (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))) $\neq$ []
unfolding *List.map-filter-def*
by (*metis* (*mono-tags*, *lifting*) *Nil-is-map-conv filter-empty-conv option.discI*)
then have $**$: *?res* $\in$ *set* (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*)))
unfolding $*$ **using** *hd-in-set* **by** *simp*

obtain *xR yR* **where** (*xR*,*yR*) $\in$ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
and *res*: *f* (*xR*,*yR*) = *Some ?res*
using *map-filter-elem*[*OF* $**$]
by (*metis prod.exhaust-sel*)

have *p1*: *?res* = ((*xR*,*yR*),(*h-obs M q1 xR yR*, *h-obs M q2 xR yR*))
using *res f1*
by (*metis option.distinct*(*1*) *option.sel*)
then have *p2*: $\bigwedge$ *q1′ q2′* . *h-obs M q1 xR yR* = *Some q1′* $\Longrightarrow$ *h-obs M q2 xR yR* = *Some q2′* $\Longrightarrow$ *ofsm-table M* (λ*q* . *states M*) *k q1′* $\neq$ *ofsm-table M* (λ*q* . *states M*) *k q2′*
using *res f1 f2* **by** *auto*
have *p3*: *h-obs M q1 xR yR* $\neq$ *None* $\vee$ *h-obs M q2 xR yR* $\neq$ *None*
using *res f3* **by** *blast*

show *?thesis* **using** *that p1 p2 p3* **by** *blast*
qed
next
case *True*

obtain *io* **where** *length io* $\leq$ *Suc k* **and** *io* $\in$ *LS M q1* $\cup$ *LS M q2* **and** *io* $\notin$

LS M q1 ∩ *LS M q2*
using ‹*ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q2*›
unfolding *ofsm-table-set*[*OF assms*(*2*) *minimise-initial-partition*] *ofsm-table-set*[*OF assms*(*3*) *minimise-initial-partition*]
unfolding *is-in-language-iff*[*OF assms*(*1*,*2*)] *is-in-language-iff*[*OF assms*(*1*,*3*)]
by *blast*
then have *io* ≠ []
using *assms*(*2*) *assms*(*3*) **by** *auto*
then have *io* = [*hd io*] @ *tl io*
by (*metis append.left-neutral append-Cons list.exhaust-sel*)
then obtain *x y* **where** *hd io* = (*x*,*y*)
by (*meson prod.exhaust-sel*)

have [(*x*,*y*)] ∈ *LS M q1* ∩ *LS M q2*
proof −
have [(*x*,*y*)] ∈ *LS M q1* ∪ *LS M q2*
using ‹*io* ∈ *LS M q1* ∪ *LS M q2*› *language-prefix* ‹*hd io* = (*x*,*y*)› ‹*io* = [*hd io*] @ *tl io*›
by (*metis Un-iff*)
then have *x* ∈ *inputs M* **and** *y* ∈ *outputs M*
by *auto*

consider [(*x*,*y*)] ∈ *LS M q1* | [(*x*,*y*)] ∈ *LS M q2*
using ‹[(*x*,*y*)] ∈ *LS M q1* ∪ *LS M q2*› **by** *blast*
then show *?thesis*
proof *cases*
case *1*
then have *h-obs M q1 x y* ≠ *None*
using *h-obs-None*[*OF* ‹*observable M*›] **unfolding** *LS-single-transition* **by** *auto*
then have *h-obs M q2 x y* ≠ *None*
using *True* ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*› **by** *meson*
then show *?thesis*
using *1 h-obs-None*[*OF* ‹*observable M*›]
by (*metis IntI LS-single-transition fst-conv snd-conv*)
next
case *2*
then have *h-obs M q2 x y* ≠ *None*
using *h-obs-None*[*OF* ‹*observable M*›] **unfolding** *LS-single-transition* **by** *auto*
then have *h-obs M q1 x y* ≠ *None*
using *True* ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*› **by** *meson*
then show *?thesis*
using *2 h-obs-None*[*OF* ‹*observable M*›]
by (*metis IntI LS-single-transition fst-conv snd-conv*)
qed
qed
then obtain *q1′ q2′* **where** (*q1*,*x*,*y*,*q1′*) ∈ *transitions M*

and $(q2,x,y,q2') \in$ *transitions M*
using *LS-single-transition* **by** *force*
then have $q1' \in$ *states M* **and** $q2' \in$ *states M* **using** *fsm-transition-target* **by** *auto*

have *tl io* $\in$ *LS M q1'* $\cup$ *LS M q2'*
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹$(q1,x,y,q1') \in$ *transitions M*›]
observable-language-transition-target[*OF* ‹*observable M*› ‹$(q2,x,y,q2') \in$ *transitions M*›]
‹*io* $\in$ *LS M q1* $\cup$ *LS M q2*›
unfolding *fst-conv snd-conv*
by (*metis Un-iff* ‹*hd io* = (*x*, *y*)› ‹*io* = [*hd io*] @ *tl io*› *append-Cons append-Nil*)

moreover have *tl io* $\notin$ *LS M q1'* $\cap$ *LS M q2'*
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹$(q1,x,y,q1') \in$ *transitions M*›]
observable-language-transition-target[*OF* ‹*observable M*› ‹$(q2,x,y,q2') \in$ *transitions M*›]
‹*io* $\in$ *LS M q1* $\cup$ *LS M q2*›
unfolding *fst-conv snd-conv*
by (*metis Int-iff LS-prepend-transition* ‹(*q1*, *x*, *y*, *q1'*) $\in$ *FSM.transitions M*› ‹(*q2*, *x*, *y*, *q2'*) $\in$ *FSM.transitions M*› ‹*hd io* = (*x*, *y*)› ‹*io* $\neq$ []› ‹*io* $\notin$ *LS M q1* $\cap$ *LS M q2*› *fst-conv list.collapse snd-conv*)
moreover have *length* (*tl io*) $\leq$ *k*
using ‹*length io* $\leq$ *Suc k*› **by** *auto*
ultimately have *ofsm-table M* (λq . *states M*) *k q1'* $\neq$ *ofsm-table M* (λq . *states M*) *k q2'*
unfolding *ofsm-table-set-observable*[*OF assms*(*1*) ‹*q1'* $\in$ *states M*› *minimise-initial-partition*] *ofsm-table-set-observable*[*OF assms*(*1*) ‹*q2'* $\in$ *states M*› *minimise-initial-partition*]
using ‹*q1'* $\in$ *states M*› ‹*q2'* $\in$ *states M*› *after-is-state*[*OF assms*(*1*)]
by *blast*
moreover have *h-obs M q1 x y* = *Some q1'*
using ‹$(q1,x,y,q1') \in$ *transitions M*› ‹*observable M*› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] *observable-alt-def* **by** *auto*
moreover have *h-obs M q2 x y* = *Some q2'*
using ‹$(q2,x,y,q2') \in$ *transitions M*› ‹*observable M*› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] *observable-alt-def* **by** *auto*
ultimately have *f* (x,y) = *Some* $((x,y)$,(*Some q1'*, *Some q2'*))
unfolding *f* **by** *force*

moreover have $(x,y) \in$ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
using *fsm-transition-output*[*OF* ‹$(q1,x,y,q1') \in$ *transitions M*›] *outputs-as-list-set*[*of M*]
using *fsm-transition-input*[*OF* ‹$(q1,x,y,q1') \in$ *transitions M*›] *inputs-as-list-set*[*of M*]
using *image-iff* **by** *fastforce*
ultimately have (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list*

M))) ≠ []
unfolding *List.map-filter-def*
by (*metis* (*mono-tags*, *lifting*) *Nil-is-map-conv filter-empty-conv option.discI*)
then have ∗∗: *?res* ∈ *set* (*List.map-filter f* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*)))
unfolding ∗ **using** *hd-in-set* **by** *simp*

obtain *xR yR* **where** (*xR*,*yR*) ∈ *set* (*List.product*(*inputs-as-list M*) (*outputs-as-list M*))
and *res*: *f* (*xR*,*yR*) = *Some ?res*
using *map-filter-elem*[*OF* ∗∗]
by (*metis prod.exhaust-sel*)

have *p1*: *?res* = ((*xR*,*yR*),(*h-obs M q1 xR yR*, *h-obs M q2 xR yR*))
using *res f1*
by (*metis option.distinct*(*1*) *option.sel*)
then have *p2*: ⋀ *q1′ q2′* . *h-obs M q1 xR yR* = *Some q1′* ⟹ *h-obs M q2 xR yR* = *Some q2′* ⟹ *ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
using *res f1 f2* **by** *auto*
have *p3*: *h-obs M q1 xR yR* ≠ *None* ∨ *h-obs M q2 xR yR* ≠ *None*
using *res f3* **by** *blast*
show *?thesis* **using** *that p1 p2 p3* **by** *blast*
qed
qed

7.2 Assembling Distinguishing Traces

fun *assemble-distinguishing-sequence-from-ofsm-table* :: (′*a*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ (′*b* × ′*c*) *list* **where**
assemble-distinguishing-sequence-from-ofsm-table M q1 q2 0 = [] |
assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (*Suc k*) = (*case*
select-diverging-ofsm-table-io M q1 q2 (*Suc k*)
of
((*x*,*y*),(*Some q1′*,*Some q2′*)) ⇒ (*x*,*y*) # (*assemble-distinguishing-sequence-from-ofsm-table M q1′ q2′ k*) |
((*x*,*y*),-) ⇒ [(*x*,*y*)])

lemma *assemble-distinguishing-sequence-from-ofsm-table-distinguishes* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∈ *LS M q1* ∪ *LS M q2*
and *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∉ *LS M q1* ∩ *LS M q2*
and *butlast* (*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k*) ∈ *LS*

M q1 ∩ *LS M q2*
proof −
have *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∈ *LS M q1* ∪ *LS M q2*
∧ *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∉ *LS M q1* ∩ *LS M q2*
∧ *butlast* (*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k*) ∈ *LS M q1* ∩ *LS M q2*
using *assms*(*2*,*3*,*4*)
proof (*induction k arbitrary*: *q1 q2*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)

obtain *x y* **where** *s1*: *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
and *s2*: ⋀ *q1′ q2′* . *h-obs M q1 x y* = *Some q1′* ⟹ *h-obs M q2 x y* = *Some q2′* ⟹ *ofsm-table M* (λ*q* . *states M*) *k q1′* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2′*
and *s3*: *h-obs M q1 x y* ≠ *None* ∨ *h-obs M q2 x y* ≠ *None*
using *select-diverging-ofsm-table-io-Some*[*OF assms*(*1*) *Suc.prems*]
by *blast*

consider (*a*) *h-obs M q1 x y* = *None* ∧ *h-obs M q2 x y* ≠ *None* |
(*b*) *h-obs M q1 x y* ≠ *None* ∧ *h-obs M q2 x y* = *None* |
(*c*) *h-obs M q1 x y* ≠ *None* ∧ *h-obs M q2 x y* ≠ *None*
using *s3* **by** *blast*
then show *?case* **proof** *cases*
case *a*
then obtain *q2′* **where** *h-obs M q1 x y* = *None* **and** *h-obs M q2 x y* = *Some q2′*
by *blast*
then have *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),(*None*, *Some q2′*))
using *s1* **by** *auto*
then have ∗:*assemble-distinguishing-sequence-from-ofsm-table M q1 q2* (*Suc k*) = [(*x*,*y*)]
by *auto*

have [(*x*,*y*)] ∈ *LS M q1* ∪ *LS M q2*
using ‹*h-obs M q2 x y* = *Some q2′*› *LS-single-transition*[*of x y M*]
by (*metis UnI2 h-obs-None*[*OF* ‹*observable M*›] *a fst-conv snd-conv*)
moreover have [(*x*,*y*)] ∉ *LS M q1* ∩ *LS M q2*
using ‹*h-obs M q1 x y* = *None*› *LS-single-transition*[*of x y M*]
unfolding *h-obs-None*[*OF* ‹*observable M*›] **by** *force*
moreover have *butlast* [(*x*,*y*)] ∈ *LS M q1* ∩ *LS M q2*
using *Suc.prems*(*1*,*2*) **by** *auto*
ultimately show *?thesis*

unfolding ∗ **by** *simp*
next
case *b*
then obtain *q1′* **where** *h-obs M q2 x y = None* **and** *h-obs M q1 x y = Some q1′*
by *blast*
then have *select-diverging-ofsm-table-io M q1 q2 (Suc k) = ((x,y),(Some q1′,None))*
using *s1* **by** *auto*
then have ∗:*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k) = [(x,y)]*
by *auto*

have *[(x,y)] ∈ LS M q1 ∪ LS M q2*
using ‹*h-obs M q1 x y = Some q1′*› *LS-single-transition*[*of x y M*]
by (*metis UnI1 assms*(*1*) *b fst-conv h-obs-None snd-conv*)
moreover have *[(x,y)] ∉ LS M q1 ∩ LS M q2*
using ‹*h-obs M q2 x y = None*› *LS-single-transition*[*of x y M*]
unfolding *h-obs-None*[*OF* ‹*observable M*›] **by** *force*
moreover have *butlast [(x,y)] ∈ LS M q1 ∩ LS M q2*
using *Suc.prems*(*1*,*2*) **by** *auto*
ultimately show *?thesis*
unfolding ∗ **by** *simp*
next
case *c*
then obtain *q1′ q2′* **where** *h-obs M q1 x y = Some q1′* **and** *h-obs M q2 x y = Some q2′*
by *blast*
then have *select-diverging-ofsm-table-io M q1 q2 (Suc k) = ((x,y),(Some q1′, Some q2′))*
using *s1* **by** *auto*
then have *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k) = (x,y) # (assemble-distinguishing-sequence-from-ofsm-table M q1′ q2′ k)*
by *auto*
moreover define *subseq* **where** *subseq*: *subseq = (assemble-distinguishing-sequence-from-ofsm-table M q1′ q2′ k)*
ultimately have ∗:*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k) = (x,y) # subseq*
by *auto*

have *(q1,x,y,q1′) ∈ transitions M*
using ‹*h-obs M q1 x y = Some q1′*› *h-obs-Some*[*OF* ‹*observable M*›] **by** *blast*
then have *q1′ ∈ states M*
using *fsm-transition-target* **by** *auto*
have *(q2,x,y,q2′) ∈ transitions M*
using ‹*h-obs M q2 x y = Some q2′*› *h-obs-Some*[*OF* ‹*observable M*›] **by** *blast*
then have *q2′ ∈ states M*

using *fsm-transition-target* **by** *auto*

have *i1*: *subseq* ∈ *LS M q1'* ∪ *LS M q2'*
and *i2*: *subseq* ∉ *LS M q1'* ∩ *LS M q2'*
and *i3*: *butlast subseq* ∈ *LS M q1'* ∩ *LS M q2'*
using *Suc.IH*[*OF* ‹*q1'* ∈ *states M*› ‹*q2'* ∈ *states M*› *s2*[*OF* ‹*h-obs M q1 x y = Some q1'*› ‹*h-obs M q2 x y = Some q2'*›]]
unfolding *subseq* **by** *blast+*

have (*x*,*y*) # *subseq* ∈ *LS M q1* ∪ *LS M q2*
using *i1* ‹(*q1*,*x*,*y*,*q1'*) ∈ *transitions M*› ‹(*q2*,*x*,*y*,*q2'*) ∈ *transitions M*›
by (*metis LS-prepend-transition Un-iff fst-conv snd-conv*)
moreover have (*x*,*y*) # *subseq* ∉ *LS M q1* ∩ *LS M q2*
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹(*q1*,*x*,*y*,*q1'*) ∈ *transitions M*›, *of subseq*]
observable-language-transition-target[*OF* ‹*observable M*› ‹(*q2*,*x*,*y*,*q2'*) ∈ *transitions M*›, *of subseq*]
i2
unfolding *fst-conv snd-conv*
by *blast*
moreover have *butlast* ((*x*,*y*) # *subseq*) ∈ *LS M q1* ∩ *LS M q2*
using *i3* ‹(*q1*,*x*,*y*,*q1'*) ∈ *transitions M*› ‹(*q2*,*x*,*y*,*q2'*) ∈ *transitions M*›
by (*metis Int-iff LS-prepend-transition LS-single-transition append-butlast-last-id butlast.simps*(*2*) *fst-conv language-prefix snd-conv*)
ultimately show *?thesis*
unfolding ∗ **by** *simp*
qed
qed

then show *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∈ *LS M q1* ∪ *LS M q2*
and *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* ∉ *LS M q1* ∩ *LS M q2*
and *butlast* (*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k*) ∈ *LS M q1* ∩ *LS M q2*
by *blast+*
qed

lemma *assemble-distinguishing-sequence-from-ofsm-table-length* :
length (*assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k*) ≤ *k*
proof (*induction k arbitrary*: *q1 q2*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
obtain *x y A B* **where** ∗:*select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),*A*,*B*)
using *prod.exhaust* **by** *metis*

```
  show ?case proof (cases A)
    case None
    then have assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k)
= [(x,y)]
    unfolding assemble-distinguishing-sequence-from-ofsm-table.simps ∗ case-prod-conv
by auto
    then show ?thesis
      by (metis Suc-le-length-iff length-Cons list.distinct(1) not-less-eq-eq)
  next
    case (Some q1′)
    show ?thesis proof (cases B)
      case None
      then have assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k)
= [(x,y)]
      unfolding assemble-distinguishing-sequence-from-ofsm-table.simps ∗ case-prod-conv
Some by auto
      then show ?thesis
        by (metis Suc-le-length-iff length-Cons list.distinct(1) not-less-eq-eq)
    next
      case (Some q2′)
      show ?thesis
        unfolding assemble-distinguishing-sequence-from-ofsm-table.simps ∗ ‹A =
Some q1′› Some case-prod-conv
        using Suc.IH[of q1′ q2′]
        by simp
    qed
  qed
qed

lemma ofsm-table-fix-partition-fixpoint-trivial-partition :
  assumes q ∈ states M
shows ofsm-table-fix M (λq. FSM.states M) 0 q = ofsm-table M (λq. FSM.states
M) (size M − 1) q
proof −
  have ((λq. FSM.states M) ‘ FSM.states M) = {states M}
    using fsm-initial[of M]
    by auto
  then have ∗:card ((λq. FSM.states M) ‘ FSM.states M) = 1
    by auto

  show ?thesis
    using ofsm-table-fix-partition-fixpoint[OF minimise-initial-partition - assms, of
size M]
    unfolding ∗
    by blast
qed
```

fun *get-distinguishing-sequence-from-ofsm-tables* :: ('a,'b::*linorder*,'c::*linorder*) *fsm* ⇒ 'a ⇒ 'a ⇒ ('b × 'c) *list* **where**
get-distinguishing-sequence-from-ofsm-tables M q1 q2 = (*let*
k = *find-first-distinct-ofsm-table M q1 q2*
in assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k)

lemma *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *LS M q1* ∪ *LS M q2*
and *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∉ *LS M q1* ∩ *LS M q2*
and *butlast* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*) ∈ *LS M q1* ∩ *LS M q2*
proof −
have *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
using ‹*minimal M*› **unfolding** *minimal-observable-code*[*OF assms*(*1*)]
using *assms*(*3*,*4*,*5*) **by** *blast*

let *?k* = *find-first-distinct-ofsm-table-gt M q1 q2 0*
have *ofsm-table M* (λ*q* . *states M*) *?k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *?k q2*
using *find-first-distinct-ofsm-table-is-first*(*1*)[*OF assms*(*3*,*4*) ‹*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*›] **.**

have ∗:*get-distinguishing-sequence-from-ofsm-tables M q1 q2* = *assemble-distinguishing-sequence-from-ofsm-ta*
M q1 q2 ?k
by *auto*

show *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *LS M q1* ∪ *LS M q2*
and *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∉ *LS M q1* ∩ *LS M q2*
and *butlast* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*) ∈ *LS M q1* ∩ *LS M q2*
using *assemble-distinguishing-sequence-from-ofsm-table-distinguishes*[*OF assms*(*1*,*3*,*4*) ‹*ofsm-table M* (λ*q* . *states M*) *?k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *?k q2*›]
unfolding ∗
by *blast*+
qed

lemma *get-distinguishing-sequence-from-ofsm-tables-distinguishes* :
assumes *observable M*

and *minimal M*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *q1* $\neq$ *q2*
shows *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*)
using *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace*(*1*,*2*)[*OF assms*]
unfolding *distinguishes-def*
by *blast*

7.3 Minimal Distinguishing Traces

lemma *get-distinguishing-sequence-from-ofsm-tables-is-minimally-distinguishing* :
fixes *M* :: ('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
assumes *observable M*
and *minimal M*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *q1* $\neq$ *q2*
shows *minimally-distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*)
proof −

have $*$:*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
using ‹*minimal M*› **unfolding** *minimal-observable-code*[*OF assms*(*1*)]
using *assms*(*3*,*4*,*5*) **by** *blast*

obtain *k* **where** *k* = *find-first-distinct-ofsm-table M q1 q2*
and *get-distinguishing-sequence-from-ofsm-tables M q1 q2* = *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k*
by *auto*
then have *length* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*) $\leq$ *k*
using *assemble-distinguishing-sequence-from-ofsm-table-length*
by *metis*
moreover have $\bigwedge$ *io* . *length io* < *k* $\Longrightarrow$ $\neg$*distinguishes M q1 q2 io*
proof −
fix *io* :: ('*b* $\times$ '*c*) *list*
assume *length io* < *k*
then have *ofsm-table M* (λ*q*. *FSM.states M*) (*length io*) *q1* = *ofsm-table M* (λ*q*. *FSM.states M*) (*length io*) *q2*
using *find-first-distinct-ofsm-table-is-first*[*OF assms*(*3*,*4*) $*$]
unfolding ‹*k* = *find-first-distinct-ofsm-table M q1 q2*›
by *blast*
then show $\neg$*distinguishes M q1 q2 io*
using *ofsm-table-set-observable*[*OF assms*(*1*,*3*) *minimise-initial-partition*]
using *ofsm-table-set-observable*[*OF assms*(*1*,*4*) *minimise-initial-partition*]
unfolding *distinguishes-def*

by (*metis* (*mono-tags*, *lifting*) *Int-iff Un-iff assms*(*3*) *le-refl mem-Collect-eq ofsm-table-containment*)
qed
ultimately show *?thesis*
using *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace*(*1*,*2*)[*OF assms*]
unfolding *minimally-distinguishes-def distinguishes-def*
using *le-neq-implies-less not-le-imp-less*
by *blast*
qed

lemma *minimally-distinguishes-length* :
assumes *observable M*
and *minimal M*
and $q1 \in$ *states M*
and $q2 \in$ *states M*
and $q1 \neq q2$
and *minimally-distinguishes M q1 q2 io*
shows *length io* $\leq$ *size M* $-$ *1*
proof $-$

have *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
using ‹*minimal M*› **unfolding** *minimal-observable-code*[*OF assms*(*1*)]
using *assms*(*3*,*4*,*5*) **by** *blast*
then have *ofsm-table M* (λq. *FSM.states M*) (*FSM.size M* $-$ *1*) *q1* $\neq$ *ofsm-table M* (λq. *FSM.states M*) (*FSM.size M* $-$ *1*) *q2*
using *ofsm-table-fix-partition-fixpoint-trivial-partition assms*(*3*,*4*)
by *metis*
then obtain *io'* **where** *distinguishes M q1 q2 io'* **and** *length io'* $\leq$ *size M* $-$ *1*
unfolding *ofsm-table-set-observable*[*OF assms*(*1*,*3*) *minimise-initial-partition*]
unfolding *ofsm-table-set-observable*[*OF assms*(*1*,*4*) *minimise-initial-partition*]
unfolding *distinguishes-def*
by *blast*
then show *?thesis*
using *assms*(*6*) **unfolding** *minimally-distinguishes-def*
using *dual-order.trans* **by** *blast*
qed

end

8 Properties of Sets of IO Sequences

This theory contains various definitions for properties of sets of IO-traces.

theory *IO-Sequence-Set*
imports *FSM*
begin

fun *output-completion* :: $('a \times 'b)$ *list set* $\Rightarrow$ $'b$ *set* $\Rightarrow$ $('a \times 'b)$ *list set* **where**
output-completion P Out = $P \cup \{io@[(fst\ xy,\ y)] \mid io\ xy\ y\ .\ y \in Out \wedge io@[xy] \in P \wedge io@[(fst\ xy,\ y)] \notin P\}$

fun *output-complete-sequences* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
output-complete-sequences M P = $(\forall\ io \in P\ .\ io = [] \vee (\forall\ y \in (outputs\ M)\ .\ (butlast\ io)@[(fst\ (last\ io),\ y)] \in P))$

fun *acyclic-sequences* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
acyclic-sequences M q P = $(\forall\ p\ .\ (path\ M\ q\ p \wedge p\text{-}io\ p \in P) \longrightarrow distinct\ (visited\text{-}states\ q\ p))$

fun *acyclic-sequences'* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
acyclic-sequences' M q P = $(\forall\ io \in P\ .\ \forall\ p \in (paths\text{-}for\text{-}io\ M\ q\ io)\ .\ distinct\ (visited\text{-}states\ q\ p))$

lemma *acyclic-sequences-alt-def*[*code*] : *acyclic-sequences M P* = *acyclic-sequences' M P*
unfolding *acyclic-sequences'.simps acyclic-sequences.simps paths-for-io-def*
by *blast*

fun *single-input-sequences* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
single-input-sequences M P = $(\forall\ xys1\ xys2\ xy1\ xy2\ .\ (xys1@[xy1] \in P \wedge xys2@[xy2] \in P \wedge io\text{-}targets\ M\ xys1\ (initial\ M) = io\text{-}targets\ M\ xys2\ (initial\ M)) \longrightarrow fst\ xy1 = fst\ xy2)$

fun *single-input-sequences'* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
single-input-sequences' M P = $(\forall\ io1 \in P\ .\ \forall\ io2 \in P\ .\ io1 = [] \vee io2 = [] \vee ((io\text{-}targets\ M\ (butlast\ io1)\ (initial\ M) = io\text{-}targets\ M\ (butlast\ io2)\ (initial\ M)) \longrightarrow fst\ (last\ io1) = fst\ (last\ io2)))$

lemma *single-input-sequences-alt-def*[*code*] : *single-input-sequences M P* = *single-input-sequences' M P*
unfolding *single-input-sequences.simps single-input-sequences'.simps*
by (*metis append-butlast-last-id append-is-Nil-conv butlast-snoc last-snoc not-Cons-self*)

fun *output-complete-for-FSM-sequences-from-state* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
output-complete-for-FSM-sequences-from-state M q P = $(\forall\ io\ xy\ t\ .\ io@[xy] \in P \wedge t \in transitions\ M \wedge t\text{-}source\ t \in io\text{-}targets\ M\ io\ q \wedge t\text{-}input\ t = fst\ xy \longrightarrow io@[(fst\ xy,\ t\text{-}output\ t)] \in P)$

lemma *output-complete-for-FSM-sequences-from-state-alt-def* :
shows *output-complete-for-FSM-sequences-from-state M q P* = $(\forall\ xys\ xy\ y\ .\ (xys@[xy] \in P \wedge (\exists\ q' \in (io\text{-}targets\ M\ xys\ q)\ .\ [(fst\ xy,y)] \in LS\ M\ q')) \longrightarrow$

xys@[(*fst xy*,*y*)] ∈ *P*)
proof −
have ⋀ *xys xy y q′* . *q′* ∈ (*io-targets M xys q*) ⟹ [(*fst xy*,*y*)] ∈ *LS M q′* ⟹ ∃ *t* . *t* ∈ *transitions M* ∧ *t-source t* ∈ *io-targets M xys q* ∧ *t-input t* = *fst xy* ∧ *t-output t* = *y*
unfolding *io-targets.simps LS.simps*
using *path-append path-append-transition-elim*(*2*) **by** *fastforce*

moreover have ⋀ *xys xy y t* . *t* ∈ *transitions M* ⟹ *t-source t* ∈ *io-targets M xys q* ⟹ *t-input t* = *fst xy* ⟹ *t-output t* = *y* ⟹ ∃ *q′* ∈ (*io-targets M xys q*) . [(*fst xy*,*y*)] ∈ *LS M q′*
unfolding *io-targets.simps LS.simps*
proof −
fix *xys* :: (*′b* × *′c*) *list* **and** *xy* :: *′b* × *′d* **and** *y* :: *′c* **and** *t* :: *′a* × *′b* × *′c* × *′a*
assume *a1*: *t-input t* = *fst xy*
assume *a2*: *t-output t* = *y*
assume *a3*: *t-source t* ∈ {*target q p* |*p*. *path M q p* ∧ *p-io p* = *xys*}
assume *a4*: *t* ∈ *FSM.transitions M*
have ∀ *p f*. [*f* (*p*::*′a* × *′b* × *′c* × *′a*)::*′b* × *′c*] = *map f* [*p*]
by *simp*
then have ∃ *a*. (∃ *ps*. [(*t-input t*, *t-output t*)] = *p-io ps* ∧ *path M a ps*) ∧ *a* ∈ {*target q ps* |*ps*. *path M q ps* ∧ *p-io ps* = *xys*}
using *a4 a3* **by** (*meson single-transition-path*)
then have ∃ *a*. [(*t-input t*, *t-output t*)] ∈ {*p-io ps* |*ps*. *path M a ps*} ∧ *a* ∈ {*target q ps* |*ps*. *path M q ps* ∧ *p-io ps* = *xys*}
by *auto*
then show ∃ *a*∈{*target q ps* |*ps*. *path M q ps* ∧ *p-io ps* = *xys*}. [(*fst xy*, *y*)] ∈ {*p-io ps* |*ps*. *path M a ps*}
using *a2 a1* **by** (*metis* (*no-types*, *lifting*))
qed

ultimately show *?thesis*
unfolding *output-complete-for-FSM-sequences-from-state.simps* **by** *blast*
qed

fun *output-complete-for-FSM-sequences-from-state′* :: (*′a*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ (*′b* × *′c*) *list set* ⇒ *bool* **where**
output-complete-for-FSM-sequences-from-state′ M q P = (∀ *io*∈*P* . ∀ *t* ∈ *transitions M* . *io* = [] ∨ (*t-source t* ∈ *io-targets M* (*butlast io*) *q* ∧ *t-input t* = *fst* (*last io*) ⟶ (*butlast io*)@[(*fst* (*last io*), *t-output t*)] ∈ *P*))

lemma *output-complete-for-FSM-sequences-alt-def′*[*code*] : *output-complete-for-FSM-sequences-from-state M q P* = *output-complete-for-FSM-sequences-from-state′ M q P*
unfolding *output-complete-for-FSM-sequences-from-state.simps output-complete-for-FSM-sequences-from-sta*
by (*metis last-snoc snoc-eq-iff-butlast*)

fun *deadlock-states-sequences* :: (*′a*,*′b*,*′c*) *fsm* ⇒ *′a set* ⇒ (*′b* × *′c*) *list set* ⇒ *bool* **where**
deadlock-states-sequences M Q P = (∀ *xys* ∈ *P* .

$((io\text{-}targets\ M\ xys\ (initial\ M) \subseteq Q$
$\wedge \neg\ (\exists\ xys' \in P\ .\ length\ xys < length\ xys' \wedge take$
$(length\ xys)\ xys' = xys)))$
$\vee\ (\neg\ io\text{-}targets\ M\ xys\ (initial\ M) \cap Q = \{\}$
$\wedge\ (\exists\ xys' \in P\ .\ length\ xys < length\ xys' \wedge take$
$(length\ xys)\ xys' = xys)))$

fun *reachable-states-sequences* :: $('a,'b,'c)\ fsm \Rightarrow 'a\ set \Rightarrow ('b \times 'c)\ list\ set \Rightarrow bool$
where
reachable-states-sequences $M\ Q\ P = (\forall\ q \in Q\ .\ \exists\ xys \in P\ .\ q \in io\text{-}targets\ M\ xys$
$(initial\ M))$

fun *prefix-closed-sequences* :: $('b \times 'c)\ list\ set \Rightarrow bool$ **where**
prefix-closed-sequences $P = (\forall\ xys1\ xys2\ .\ xys1@xys2 \in P \longrightarrow xys1 \in P)$

fun *prefix-closed-sequences'* :: $('b \times 'c)\ list\ set \Rightarrow bool$ **where**
prefix-closed-sequences' $P = (\forall\ io \in P\ .\ io = [] \vee (butlast\ io) \in P)$

lemma *prefix-closed-sequences-alt-def*[*code*] : *prefix-closed-sequences* P = *prefix-closed-sequences'* P
proof
show *prefix-closed-sequences* $P \Longrightarrow$ *prefix-closed-sequences'* P
unfolding *prefix-closed-sequences.simps prefix-closed-sequences'.simps*
by (*metis append-butlast-last-id*)

have $\bigwedge xys1\ xys2.\ \forall io{\in}P.\ io = [] \vee butlast\ io \in P \Longrightarrow xys1\ @\ xys2 \in P \Longrightarrow$
$xys1 \in P$
proof −
fix *xys1 xys2* **assume** $\forall io{\in}P.\ io = [] \vee butlast\ io \in P$ **and** $xys1\ @\ xys2 \in P$
then show $xys1 \in P$
proof (*induction xys2 rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a xys2*)
then show *?case*
by (*metis append.assoc snoc-eq-iff-butlast*)
qed
qed

then show *prefix-closed-sequences'* $P \Longrightarrow$ *prefix-closed-sequences* P
unfolding *prefix-closed-sequences.simps prefix-closed-sequences'.simps* **by** *blast*

qed

8.1 Completions

definition *prefix-completion* :: $'a\ list\ set \Rightarrow 'a\ list\ set$ **where**
prefix-completion $P = \{xs\ .\ \exists\ ys\ .\ xs@ys \in P\}$

lemma *prefix-completion-closed* :
prefix-closed-sequences (*prefix-completion P*)
unfolding *prefix-closed-sequences.simps prefix-completion-def*
by *auto*

lemma *prefix-completion-source-subset* :
$P \subseteq$ *prefix-completion P*
unfolding *prefix-completion-def*
by (*metis* (*no-types*, *lifting*) *append-Nil2 mem-Collect-eq subsetI*)

definition *output-completion-for-FSM* :: ('a,'b,'c) *fsm* $\Rightarrow$ ('b $\times$ 'c) *list set* $\Rightarrow$ ('b $\times$ 'c) *list set* **where**
output-completion-for-FSM M P = $P \cup \{$ *io*@[(*x*,*y'*)] | *io x y'* . (*y'* $\in$ (*outputs M*)) $\wedge$ ($\exists$ *y* . *io*@[(*x*,*y*)] $\in$ *P*)$\}$

lemma *output-completion-for-FSM-complete* :
shows *output-complete-sequences M* (*output-completion-for-FSM M P*)
unfolding *output-completion-for-FSM-def output-complete-sequences.simps*
proof
fix *io* **assume** $*$: *io* $\in P \cup \{$*io* @ [(*x*, *y'*)] |*io x y'*. *y'* $\in$ (*outputs M*) $\wedge$ ($\exists$ *y*. *io* @ [(*x*, *y*)] $\in$ *P*)$\}$
show *io* = [] $\vee$
($\forall$ *y*$\in$(*outputs M*).
butlast io @ [(*fst* (*last io*), *y*)]
$\in P \cup \{$*io* @ [(*x*, *y'*)] |*io x y'*. *y'* $\in$ (*outputs M*) $\wedge$ ($\exists$ *y*. *io* @ [(*x*, *y*)] $\in$ *P*)$\}$)
proof (*cases io rule*: *rev-cases*)
case *Nil*
then show *?thesis* **by** *blast*
next
case (*snoc ys y*)
then show *?thesis* **proof** (*cases io* $\in$ *P*)
case *True*
then have *butlast io* @ [(*fst* (*last io*), (*snd* (*last io*)))] $\in$ *P* **using** *snoc* **by** *auto*
then show *?thesis* **using** *snoc* **by** *blast*
next
case *False*
then show *?thesis*
using $*$ **by** *auto*
qed
qed
qed

lemma *output-completion-for-FSM-length* :
assumes $\forall$ *io* $\in$ *P* . *length io* $\leq$ *k*
shows $\forall$ *io* $\in$ *output-completion-for-FSM M P*. *length io* $\leq$ *k*

using *assms* **unfolding** *output-completion-for-FSM-def*
by *auto*

lemma *output-completion-for-FSM-code*[*code*] :
output-completion-for-FSM M P = *P* ∪ (⋃(*image* (λ(*y*,*io*) . *if length io* = *0 then* {} *else* {((*butlast io*)@[(*fst* (*last io*),*y*)])}) ((*outputs M*) × *P*)))
proof −
let *?OC* = {*io* @ [(*x*, *y*′)] |*io x y*′. *y*′ ∈ *FSM.outputs M* ∧ (∃ *y*. *io* @ [(*x*, *y*)] ∈ *P*)}
let *?OC*′ = (⋃(*y*, *io*)∈*FSM.outputs M* × *P*. *if length io* = *0 then* {} *else* {*butlast io* @ [(*fst* (*last io*), *y*)]})

have *?OC* = *?OC*′
proof −
have *?OC* ⊆ *?OC*′
proof
fix *io*′ **assume** *io*′ ∈ *?OC*
then obtain *io x y y*′ **where** *io*′ = *io* @ [(*x*, *y*′)]
and *y*′ ∈ *FSM.outputs M*
and *io* @ [(*x*, *y*)] ∈ *P*
by *blast*
then have (*y*′,*io* @ [(*x*, *y*)]) ∈ *FSM.outputs M* × *P* **by** *blast*
moreover have *length* (*io* @ [(*x*, *y*)]) ≠ *0* **by** *auto*
ultimately show *io*′ ∈ *?OC*′
unfolding ‹*io*′ = *io* @ [(*x*, *y*′)]› **by** *force*
qed
moreover have *?OC*′ ⊆ *?OC*
proof
fix *io*′ **assume** *io*′ ∈ *?OC*′
then obtain *y io* **where** *y* ∈ *outputs M* **and** *io* ∈ *P*
and *io*′ ∈ (*if length io* = *0 then* {} *else* {*butlast io* @ [(*fst* (*last io*), *y*)]})
by *auto*
then have *io*′ = *butlast io* @ [(*fst* (*last io*), *y*)]
by (*meson empty-iff singletonD*)
have *io* ≠ []
using ‹*io*′ ∈ (*if length io* = *0 then* {} *else* {*butlast io* @ [(*fst* (*last io*), *y*)]})›
by *auto*

then have *butlast io* @ [(*fst* (*last io*), *snd* (*last io*))] ∈ *P*
by (*simp add*: ‹*io* ∈ *P*›)

then show *io*′ ∈ *?OC*
using ‹*y* ∈ *outputs M*› ‹*io* ∈ *P*›
unfolding ‹*io*′ = *butlast io* @ [(*fst* (*last io*), *y*)]› **by** *blast*
qed
ultimately show *?thesis* **by** *blast*
qed

then show *?thesis*
unfolding *output-completion-for-FSM-def*
by *simp*
qed

end

9 Observability

This theory presents the classical algorithm for transforming FSMs into language-equivalent observable FSMs in analogy to the determinisation of nondeterministic finite automata.

theory *Observability*
imports *FSM*
begin

lemma *fPow-Pow* : *Pow* (*fset A*) = *fset* (*fset* |'| *fPow A*)
proof (*induction A*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert x A*)

have *Pow* (*fset* (*finsert x A*)) = *Pow* (*fset A*) ∪ (*insert x*) ' *Pow* (*fset A*)
by (*simp add*: *Pow-insert*)
moreover have *fset* (*fset* |'| *fPow* (*finsert x A*)) = *fset* (*fset* |'| *fPow A*) ∪ (*insert x*) ' *fset* (*fset* |'| *fPow A*)
proof −
have *fset* |'| ((*fPow A*) |∪| (*finsert x*) |'| (*fPow A*)) = (*fset* |'| *fPow A*) |∪| (*insert x*) |'| (*fset* |'| *fPow A*)
unfolding *fimage-funion*
by *fastforce*
moreover have (*fPow* (*finsert x A*)) = (*fPow A*) |∪| (*finsert x*) |'| (*fPow A*)
by (*simp add*: *fPow-finsert*)
ultimately show *?thesis*
by *auto*
qed
ultimately show *?case*
using *insert.IH* **by** *simp*
qed

lemma *fcard-fsubset*: ¬ *fcard* (*A* |−| (*B* |∪| *C*)) < *fcard* (*A* |−| *B*) ⟹ *C* |⊆| *A* ⟹ *C* |⊆| *B*
proof (*induction C*)
case *empty*
then show *?case* **by** *auto*
next

case (*insert x C*)
then show *?case*
unfolding *finsert-fsubset funion-finsert-right not-less*
proof −
assume *a1*: *fcard* (*A* |−| *B*) ≤ *fcard* (*A* |−| *finsert x* (*B* |∪| *C*))
assume ⟦*fcard* (*A* |−| *B*) ≤ *fcard* (*A* |−| (*B* |∪| *C*)); *C* |⊆| *A*⟧ ⟹ *C* |⊆| *B*
assume *a2*: *x* |∈| *A* ∧ *C* |⊆| *A*
have *A* |−| (*C* |∪| *finsert x B*) = *A* |−| *B* ∨ ¬ *A* |−| (*C* |∪| *finsert x B*) |⊆| *A* |−| *B*
using *a1* **by** (*metis* (*no-types*) *fcard-seteq funion-commute funion-finsert-right*)
then show *x* |∈| *B* ∧ *C* |⊆| *B*
using *a2* **by** *blast*
qed
qed

lemma *make-observable-transitions-qtrans-helper*:
assumes *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *A*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *nexts*)
shows ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ' {*t'* . *t'* |∈| *A* ∧ *t-source t'* |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*}
proof −
have *fset qtrans* = { (*q*,*x*,*y*,*q'*) | *q x y q'* . *q* |∈| *nexts* ∧ *q'* ≠ {||} ∧ *fset q'* = *t-target* ' {*t'* . *t'* |∈| *A* ∧ *t-source t'* |∈| *q* ∧ *t-input t'* = *x* ∧ *t-output t'* = *y*}}
proof −
have ⋀ *q* . *fset* (*ffilter* (λ*t* . *t-source t* |∈| *q*) *A*) = *Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)
using *ffilter.rep-eq assms*(*1*) **by** *auto*
then have ⋀ *q* . *fset* (*fimage* (λ *t* . (*t-input t*, *t-output t*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) *A*)) = *image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*))
by *simp*
then have ∗:⋀ *q* . *fset* (*fimage* (λ(*x*,*y*) . (*q*,*x*,*y*, (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))))) (*fimage* (λ *t* . (*t-input t*, *t-output t*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))
= *image* (λ(*x*,*y*) . (*q*,*x*,*y*, (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))))) (*image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)))
by (*metis* (*no-types*, *lifting*) *ffilter.rep-eq fset.set-map*)

have ∗∗: ⋀ *f1 f2 xs ys ys'* . (⋀ *x* . *fset* (*f1 x ys*) = (*f2 x ys'*)) ⟹
fset (*ffUnion* (*fimage* (λ *x* . (*f1 x ys*)) *xs*)) = (⋃ *x* ∈ *fset xs* . (*f2 x ys'*))
unfolding *ffUnion.rep-eq fimage.rep-eq* **by** *force*

have *fset* (*ffUnion* (*fimage* (λ *q* . (*fimage* (λ(*x*,*y*) . (*q*,*x*,*y*, (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))))) (*fimage* (λ *t* . (*t-input t*, *t-output t*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))) *nexts*))
= (⋃ *q* ∈ *fset nexts* . *image* (λ(*x*,*y*) . (*q*,*x*,*y*, (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))))) (*image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*))))
unfolding *ffUnion.rep-eq fimage.rep-eq*
using ∗ **by** *force*

also have ... = { (*q*,*x*,*y*,*q*′) | *q x y q*′ . *q* |∈| *nexts* ∧ *q*′ ≠ {||} ∧ *fset q*′ = *t-target* ' {*t*′ . *t*′ |∈| *A* ∧ *t-source t*′ |∈| *q* ∧ *t-input t*′ = *x* ∧ *t-output t*′ = *y*}}
(**is** *?A* = *?B*) **proof** −
have ⋀ *t* . *t* ∈ *?A* ⟹ *t* ∈ *?B*
proof −
fix *t* **assume** *t* ∈ *?A*
then obtain *q* **where** *q* ∈ *fset nexts*
and *t* ∈ *image* (λ(*x*,*y*) . (*q*,*x*,*y*, (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))))) (*image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)))
by *blast*
then obtain *x y q*′ **where** ∗: (*x*,*y*) ∈ (*image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)))
and *t* = (*q*,*x*,*y*,*q*′)
and ∗∗:*q*′ = (*t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*)))))
by *force*

have *q* |∈| *nexts*
using ‹*q* ∈ *fset nexts*›
by *simp*
moreover have *q*′ ≠ {||}
proof −
have ∗∗∗:(*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)) = *fset* (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))
by *auto*
have ∃ *t* . *t* |∈| (*ffilter* (λ*t*. *t-source t* |∈| *q*) *A*) ∧ (*t-input t*, *t-output t*) = (*x*,*y*)
using ∗
by (*metis* (*no-types*, *lifting*) ∗∗∗ *image-iff*)
then show *?thesis* **unfolding** ∗∗
by *force*
qed
moreover have *fset q*′ = *t-target* ' {*t*′ . *t*′ |∈| *A* ∧ *t-source t*′ |∈| *q* ∧ *t-input t*′ = *x* ∧ *t-output t*′ = *y*}
proof −
have {*t*′ . *t*′ |∈| *A* ∧ *t-source t*′ |∈| *q* ∧ *t-input t*′ = *x* ∧ *t-output t*′ = *y*} = ((*Set.filter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*fset* (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*)))))

by *fastforce*
also have ... = *fset* ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))
by *fastforce*
finally have {*t′* . *t′* |∈| *A* ∧ *t-source t′* |∈| *q* ∧ *t-input t′* = *x* ∧ *t-output t′* = *y*} = *fset* ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*)))) **.**
then show *?thesis*
unfolding ∗∗
by *simp*
qed
ultimately show *t* ∈ *?B*
unfolding ‹*t* = (*q*,*x*,*y*,*q′*)›
by *blast*
qed
moreover have ⋀ *t* . *t* ∈ *?B* ⟹ *t* ∈ *?A*
proof −
fix *t* **assume** *t* ∈ *?B*
then obtain *q x y q′* **where** *t* = (*q*,*x*,*y*,*q′*) **and** (*q*,*x*,*y*,*q′*) ∈ *?B* **by** *force*
then have *q* |∈| *nexts*
and *q′* ≠ {||}
and ∗: *fset q′* = *t-target* ‘ {*t′* . *t′* |∈| *A* ∧ *t-source t′* |∈| *q* ∧ *t-input t′* = *x* ∧ *t-output t′* = *y*}
by *force*+
then have *fset q′* ≠ {}
by (*metis bot-fset.rep-eq fset-inject*)

have (*x*,*y*) ∈ (*image* (λ *t* . (*t-input t*, *t-output t*)) (*Set.filter* (λ*t* . *t-source t* |∈| *q*) (*fset A*)))
using ‹*fset q′* ≠ {}› **by** (*auto simp add*: ∗ *image-iff*)
moreover have *q′* = *t-target* |‘| *ffilter* (λ*t*. (*t-input t*, *t-output t*) = (*x*, *y*)) (*ffilter* (λ*t*. *t-source t* |∈| *q*) *A*)
proof −
have {*t′* . *t′* |∈| *A* ∧ *t-source t′* |∈| *q* ∧ *t-input t′* = *x* ∧ *t-output t′* = *y*} = ((*Set.filter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*fset* (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*)))))
by *fastforce*
also have ... = *fset* ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*))))
by *fastforce*
finally have ∗∗∗:{*t′* . *t′* |∈| *A* ∧ *t-source t′* |∈| *q* ∧ *t-input t′* = *x* ∧ *t-output t′* = *y*} = *fset* ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) (*ffilter* (λ*t* . *t-source t* |∈| *q*) (*A*)))) **.**

show *?thesis*
using ∗
unfolding ∗∗∗
by (*metis* (*no-types*, *lifting*) *fimage.rep-eq fset-inject*)
qed

```
      ultimately show t ∈ ?A
        using ‹q |∈| nexts›
        by (simp add: ‹t = (q,x,y,q')› image-iff)
    qed
    ultimately show ?thesis
      by (metis (no-types, lifting) Collect-cong Sup-set-def mem-Collect-eq)
  qed
  finally show ?thesis
    unfolding assms Let-def by blast
 qed
 then show ⋀ t . t |∈| qtrans ⟷ t-source t |∈| nexts ∧ t-target t ≠ {||} ∧ fset
(t-target t) = t-target ‘ {t' . t' |∈| A ∧ t-source t' |∈| t-source t ∧ t-input t' =
t-input t ∧ t-output t' = t-output t}
   by force
qed
```

```
function make-observable-transitions :: ('a,'b,'c) transition fset ⇒ 'a fset fset ⇒
'a fset fset ⇒ ('a fset × 'b × 'c × 'a fset) fset ⇒ ('a fset × 'b × 'c × 'a fset) fset
where
 make-observable-transitions base-trans nexts dones ts = (let
          qtrans = ffUnion (fimage (λ q . (let qts = ffilter (λt . t-source t |∈| q)
base-trans;
                                   ios = fimage (λ t . (t-input t, t-output t)) qts
                                 in fimage (λ(x,y) . (q,x,y, t-target |'| (ffilter (λt .
(t-input t, t-output t) = (x,y)) qts))) ios)) nexts);
          dones' = dones |∪| nexts;
          ts' = ts |∪| qtrans;
          nexts' = (fimage t-target qtrans) |−| dones'
        in if nexts' = {||}
          then ts'
          else make-observable-transitions base-trans nexts' dones' ts')
 by auto
termination
proof −
 {
   fix base-trans :: ('a,'b,'c) transition fset
   fix nexts :: 'a fset fset
   fix dones :: 'a fset fset
   fix ts    :: ('a fset × 'b × 'c × 'a fset) fset
   fix q x y q'

   assume assm1: ¬ fcard
           (fPow (t-source |'| base-trans |∪| t-target |'| base-trans) |−|
            (dones |∪| nexts |∪|
             t-target |'|
             ffUnion
```

((λq. let qts = ffilter (λt. t-source t |∈| q) base-trans
in ((λ(x, y). (q, x, y, t-target |`| ffilter (λt. t-input t = x ∧ t-output t = y) qts)) ∘ (λt. (t-input t, t-output t))) |`|
qts) |`|
nexts)))
< fcard (fPow (t-source |`| base-trans |∪| t-target |`| base-trans) |−| (dones |∪| nexts))

and *assm2*: (q, x, y, q′) |∈|
ffUnion
((λq. let qts = ffilter (λt. t-source t |∈| q) base-trans
in ((λ(x, y). (q, x, y, t-target |`| ffilter (λt. t-input t = x ∧ t-output t = y) qts)) ∘ (λt. (t-input t, t-output t))) |`| qts) |`|
nexts)

and *assm3*: q′ |∉| nexts

define *qtrans* **where** *qtrans-def*: qtrans = ffUnion (fimage (λ q . (let qts = ffilter (λt . t-source t |∈| q) base-trans;
ios = fimage (λ t . (t-input t, t-output t)) qts
in fimage (λ(x,y) . (q,x,y, t-target |`| ((ffilter (λt . (t-input t, t-output t) = (x,y)) qts)))) ios)) nexts)

have *qtrans-prop*: ⋀ t . t |∈| qtrans ⟷ t-source t |∈| nexts ∧ t-target t ≠ {||} ∧ fset (t-target t) = t-target ` {t′ | t′ . t′ |∈| base-trans ∧ t-source t′ |∈| t-source t ∧ t-input t′ = t-input t ∧ t-output t′ = t-output t}
using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
by *presburger*

have ⋀ t . t |∈| qtrans ⟹ t-target t |∈| fPow (t-target |`| base-trans)
proof −
fix t **assume** t |∈| qtrans
then have ∗: fset (t-target t) = t-target ` {t′ . t′ |∈| base-trans ∧ t-source t′ |∈| t-source t ∧ t-input t′ = t-input t ∧ t-output t′ = t-output t}
using *qtrans-prop* **by** *blast*
then have fset (t-target t) ⊆ t-target ` (fset base-trans)
by (*metis* (*mono-tags*, *lifting*) *imageI image-Collect-subsetI*)
then show t-target t |∈| fPow (t-target |`| base-trans)
by (*simp add*: *less-eq-fset.rep-eq*)
qed
then have t-target |`| qtrans |⊆| (fPow (t-source |`| base-trans |∪| t-target |`| base-trans))
by *fastforce*
moreover have ¬ fcard (fPow (t-source |`| base-trans |∪| t-target |`| base-trans) |−| (dones |∪| nexts |∪| t-target |`| qtrans))
< fcard (fPow (t-source |`| base-trans |∪| t-target |`| base-trans) |−| (dones |∪| nexts))

using *assm1* **unfolding** *qtrans-def* **by** *force*
ultimately have *t-target* |`| *qtrans* |⊆| *dones* |∪| *nexts*
using *fcard-fsubset* **by** *fastforce*
moreover have *q′* |∈| *t-target* |`| *qtrans*
using *assm2* **unfolding** *qtrans-def* **by** *force*
ultimately have *q′* |∈| *dones*
using ‹*q′* |∉| *nexts*› **by** *blast*
} **note** *t* = *this*

show *?thesis*
apply (*relation measure* (λ (*base-trans*, *nexts*, *dones*, *ts*) . *fcard* ((*fPow* (*t-source* |`| *base-trans* |∪| *t-target* |`| *base-trans*)) |−| (*dones* |∪| *nexts*))))
apply *auto*
by (*erule t*)
qed

lemma *make-observable-transitions-mono*: *ts* |⊆| (*make-observable-transitions base-trans nexts dones ts*)
proof (*induction rule*: *make-observable-transitions.induct*[*of* λ *base-trans nexts dones ts* . *ts* |⊆| (*make-observable-transitions base-trans nexts dones ts*)])
case (*1 base-trans nexts dones ts*)

define *qtrans* **where** *qtrans-def*: *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *base-trans*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |`| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *nexts*)

have *qtrans-prop*: ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* | *t′* . *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
by *presburger*

let *?dones′* = *dones* |∪| *nexts*
let *?ts′* = *ts* |∪| *qtrans*
let *?nexts′* = (*fimage t-target qtrans*) |−| *?dones′*

have *res-cases*: *make-observable-transitions base-trans nexts dones ts* = (*if ?nexts′* = {||}
then ?ts′
else make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′)
unfolding *make-observable-transitions.simps*[*of base-trans nexts dones ts*] *qtrans-def Let-def* **by** *simp*

show *?case* **proof** (*cases ?nexts′* = {||})
case *True*
then show *?thesis* **using** *res-cases* **by** *simp*
next
case *False*
then have *make-observable-transitions base-trans nexts dones ts* = *make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′*
using *res-cases* **by** *simp*
moreover have *ts* |∪| *qtrans* |⊆| *make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′*
using *1*[*OF qtrans-def - - - False*, *of ?dones′ ?ts′*] **by** *blast*
ultimately show *?thesis*
by *blast*
qed
qed

inductive *pathlike* :: (*′state*, *′input*, *′output*) *transition fset* ⇒ *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *bool*
where
nil[*intro*!] : *pathlike ts q* [] |
cons[*intro*!] : *t* |∈| *ts* ⟹ *pathlike ts* (*t-target t*) *p* ⟹ *pathlike ts* (*t-source t*) (*t*#*p*)

inductive-cases *pathlike-nil-elim*[*elim*!]: *pathlike ts q* []
inductive-cases *pathlike-cons-elim*[*elim*!]: *pathlike ts q* (*t*#*p*)

lemma *make-observable-transitions-t-source* :
assumes ⋀ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* . *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
and ⋀ *q t′* . *q* |∈| *dones* ⟹ *t′* |∈| *base-trans* ⟹ *t-source t′* |∈| *q* ⟹ ∃ *t* . *t* |∈| *ts* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t′* ∧ *t-output t* = *t-output t′*
and *t* |∈| *make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*
and *t-source t* |∈| *dones*
shows *t* |∈| *ts*
using *assms* **proof** (*induction base-trans* (*fimage t-target ts*) |−| *dones dones ts rule*: *make-observable-transitions.induct*)
case (*1 base-trans dones ts*)

let *?nexts* = (*fimage t-target ts*) |−| *dones*

define *qtrans* **where** *qtrans-def*: *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *base-trans*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*

in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *?nexts*)

have *qtrans-prop*: $\bigwedge$ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *?nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ' {*t'* . *t'* |∈| *base-trans* ∧ *t-source t'* |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*}
  using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
  by *presburger*

let *?dones'* = *dones* |∪| *?nexts*
let *?ts'* = *ts* |∪| *qtrans*
let *?nexts'* = (*fimage t-target qtrans*) |−| *?dones'*

have *res-cases*: *make-observable-transitions base-trans ?nexts dones ts* = (*if ?nexts'* = {||}
    then ?ts'
    else make-observable-transitions base-trans ?nexts' ?dones' ?ts')
  unfolding *make-observable-transitions.simps*[*of base-trans ?nexts dones ts*] *qtrans-def Let-def* **by** *simp*

show *?case* **proof** (*cases ?nexts'* = {||})
  case *True*

  then have *make-observable-transitions base-trans ?nexts dones ts* = *?ts'*
    using *res-cases* **by** *auto*
  then have *t* |∈| *ts* |∪| *qtrans*
    using ‹*t* |∈| *make-observable-transitions base-trans ?nexts dones ts*› ‹*t-source t* |∈| *dones*› **by** *blast*
  then show *?thesis*
    using *qtrans-prop 1.prems*(*3*,*4*) **by** *blast*
next
  case *False*
  then have *make-observable-transitions base-trans ?nexts dones ts* = *make-observable-transitions base-trans ?nexts' ?dones' ?ts'*
    using *res-cases* **by** *simp*

  have *i1*: ($\bigwedge$*t*. *t* |∈| *ts* |∪| *qtrans* ⟹
      t-source t |∈| *dones* |∪| *?nexts* ∧
      t-target t ≠ {||} ∧
      fset (*t-target t*) =
      t-target '
      {*t'* . *t'* |∈| *base-trans* ∧
        t-source t' |∈| *t-source t* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*})
    using *1.prems*(*1*) *qtrans-prop* **by** *blast*

  have *i3*: *t-target* |'| *qtrans* |−| (*dones* |∪| *?nexts*) = *t-target* |'| (*ts* |∪| *qtrans*) |−| (*dones* |∪| *?nexts*)
    unfolding *1.prems*(*3*) **by** *blast*

have *i2*: ($\bigwedge q\ t'$.
q |∈| *dones* |∪| *?nexts* $\Longrightarrow$
t' |∈| *base-trans* $\Longrightarrow$
t-source t' |∈| q $\Longrightarrow$
$\exists\, t.$ t |∈| *ts* |∪| *qtrans* $\wedge$ *t-source* $t = q$ $\wedge$ *t-input* t = *t-input* t' $\wedge$ *t-output* t = *t-output* t')
proof −
fix q t' **assume** q |∈| *dones* |∪| *?nexts*
and $*$:t' |∈| *base-trans*
and $**$:*t-source* t' |∈| q

then consider (a) q |∈| *dones* | (b) q |∈| *?nexts* **by** *blast*
then show $\exists\, t.$ t |∈| *ts* |∪| *qtrans* $\wedge$ *t-source* $t = q$ $\wedge$ *t-input* t = *t-input* t' $\wedge$ *t-output* t = *t-output* t'
proof *cases*
case *a*
then show *?thesis* **using** $*$ $**$
using *1.prems*(*2*) **by** *blast*
next
case *b*

let *?tgts* = $\{t''$. t'' |∈| *base-trans* $\wedge$ *t-source* t'' |∈| q $\wedge$ *t-input* t'' = *t-input* t' $\wedge$ *t-output* t'' = *t-output* $t'\}$
define *tgts* **where** *tgts*: *tgts* = *Abs-fset* (*t-target* ‘ *?tgts*)

have *?tgts* ⊆ *fset base-trans*
by *fastforce*
then have *finite* (*t-target* ‘ *?tgts*)
by (*meson finite-fset finite-imageI finite-subset*)
then have *fset tgts* = (*t-target* ‘ *?tgts*)
unfolding *tgts*
using *Abs-fset-inverse*
by *blast*

have *?tgts* ≠ {}
using $*$ $**$ **by** *blast*
then have *t-target* ‘ *?tgts* ≠ {}
by *blast*
then have *tgts* ≠ {||}
using ‹*fset tgts* = (*t-target* ‘ *?tgts*)›
by *force*

then have (q, *t-input* t', *t-output* t', *tgts*) |∈| *qtrans*
using *b*
unfolding *qtrans-prop*[*of* (q,*t-input* t',*t-output* t',*tgts*)]
unfolding *fst-conv snd-conv*
unfolding ‹*fset tgts* = (*t-target* ‘ *?tgts*)›[*symmetric*]
by *blast*

then show *?thesis*
by *auto*
qed
qed

have t |∈| *make-observable-transitions base-trans ?nexts dones ts* ⟹ *t-source* t |∈| *dones* |∪| *?nexts* ⟹ t |∈| *ts* |∪| *qtrans*
unfolding ‹*make-observable-transitions base-trans ?nexts dones ts* = *make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′*›
using *1.hyps*[*OF qtrans-def - - - - i1 i2*] *False i3* **by** *force*
then have t |∈| *ts* |∪| *qtrans*
using ‹t |∈| *make-observable-transitions base-trans ?nexts dones ts*› ‹*t-source* t |∈| *dones*› **by** *blast*

then show *?thesis*
using *qtrans-prop 1.prems*(*3*,*4*) **by** *blast*
qed
qed

lemma *make-observable-transitions-path* :
assumes ⋀ t . t |∈| *ts* ⟹ *t-source* t |∈| *dones* ∧ *t-target* t ≠ {||} ∧ *fset* (*t-target* t) = *t-target* ‘ {t' ∈ *transitions* M . *t-source* t' |∈| *t-source* t ∧ *t-input* t' = *t-input* t ∧ *t-output* t' = *t-output* t}
and ⋀ q t' . q |∈| *dones* ⟹ t' ∈ *transitions* M ⟹ *t-source* t' |∈| q ⟹ ∃ t . t |∈| *ts* ∧ *t-source* t = q ∧ *t-input* t = *t-input* t' ∧ *t-output* t = *t-output* t'
and ⋀ q . q |∈| (*fimage t-target ts*) |−| *dones* ⟹ q |∈| *fPow* (*t-source* |‘| *ftransitions* M |∪| *t-target* |‘| *ftransitions* M)
and ⋀ q . q |∈| *dones* ⟹ q |∈| *fPow* (*t-source* |‘| *ftransitions* M |∪| *t-target* |‘| *ftransitions* M |∪| {|*initial* M|})
and {||} |∉| *dones*
and q |∈| *dones*
shows (∃ q' p . q' |∈| q ∧ *path* M q' p ∧ *p-io* p = *io*) ⟷ (∃ p'. *pathlike* (*make-observable-transitions* (*ftransitions* M) ((*fimage t-target ts*) |−| *dones*) *dones ts*) q p' ∧ *p-io* p' = *io*)

using *assms* **proof** (*induction ftransitions M* (*fimage t-target ts*) |−| *dones dones ts arbitrary*: *q io rule*: *make-observable-transitions.induct*)
case (*1 dones ts q*)

let *?obs* = (*make-observable-transitions* (*ftransitions M*) ((*fimage t-target ts*) |−| *dones*) *dones ts*)

let *?nexts* = (*fimage t-target ts*) |−| *dones*

show *?case* **proof** (*cases io*)
case *Nil*

have *scheme*: ⋀ *q q′ X* . *q′* |∈| *q* ⟹ *q* |∈| *fPow X* ⟹ *q′* |∈| *X*
by (*simp add*: *fsubsetD*)

obtain *q′* **where** *q′* |∈| *q*
using ‹{||} |∉| *dones*› ‹*q* |∈| *dones*›
by (*metis all-not-in-conv bot-fset.rep-eq fset-cong*)
have *q′* |∈| *t-source* |`| *ftransitions M* |∪| *t-target* |`| *ftransitions M* |∪| {|*FSM.initial M*|}
using *scheme*[*OF* ‹*q′* |∈| *q*› *1.prems*(*4*)[*OF* ‹*q* |∈| *dones*›]] **.**
then have *q′* ∈ *states M*
using *ftransitions-source*[*of q′ M*]
using *ftransitions-target*[*of q′ M*]
by *force*
then have ∃ *q′ p* . *q′* |∈| *q* ∧ *path M q′ p* ∧ *p-io p* = *io*
using ‹*q′* |∈| *q*› *Nil* **by** *auto*
moreover have (∃ *p′*. *pathlike ?obs q p′* ∧ *p-io p′* = *io*)
using *Nil* **by** *auto*
ultimately show *?thesis*
by *simp*
next
case (*Cons ioT ioP*)

define *qtrans* **where** *qtrans-def*: *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) (*ftransitions M*);
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |`| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *?nexts*)

have *qtrans-prop*: ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *?nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* . *t′* |∈| (*ftransitions M*) ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
by *presburger*

let *?dones′* = *dones* |∪| *?nexts*
let *?ts′* = *ts* |∪| *qtrans*
let *?nexts′* = (*fimage t-target qtrans*) |−| *?dones′*

have *res-cases*: *make-observable-transitions* (*ftransitions M*) *?nexts dones ts* = (*if ?nexts′* = {||}
then ?ts′
else make-observable-transitions (*ftransitions M*) *?nexts′ ?dones′ ?ts′*)
unfolding *make-observable-transitions.simps*[*of ftransitions M ?nexts dones*

ts] *qtrans-def Let-def* **by** *simp*

have *i1*: ($\bigwedge$*t*. *t* |∈| *ts* |∪| *qtrans* $\Longrightarrow$
t-source t |∈| *dones* |∪| *?nexts* ∧
t-target t ≠ {||} ∧
fset (*t-target t*) =
t-target ‘
{*t′* ∈ *FSM.transitions M* .
t-source t′ |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*})
using *1.prems*(*1*) *qtrans-prop*
using *ftransitions-set*[*of M*]
by (*metis* (*mono-tags*, *lifting*) *Collect-cong funion-iff*)

have *i2*: ($\bigwedge$*q t′*.
q |∈| *dones* |∪| *?nexts* $\Longrightarrow$
t′ ∈ *FSM.transitions M* $\Longrightarrow$
t-source t′ |∈| *q* $\Longrightarrow$
∃ *t*. *t* |∈| *ts* |∪| *qtrans* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t′* ∧ *t-output t* = *t-output t′*)
proof −
fix *q t′* **assume** *q* |∈| *dones* |∪| *?nexts*
and ∗:*t′* ∈ *FSM.transitions M*
and ∗∗:*t-source t′* |∈| *q*

then consider (*a*) *q* |∈| *dones* | (*b*) *q* |∈| *?nexts* **by** *blast*
then show ∃ *t*. *t* |∈| *ts* |∪| *qtrans* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t′* ∧ *t-output t* = *t-output t′*
proof *cases*
case *a*
then show *?thesis* **using** *1.prems*(*2*) ∗ ∗∗ **by** *blast*
next
case *b*

let *?tgts* = {*t″* ∈ *FSM.transitions M*. *t-source t″* |∈| *q* ∧ *t-input t″* = *t-input t′* ∧ *t-output t″* = *t-output t′*}
have *?tgts* ≠ {}
using ∗ ∗∗ **by** *blast*

let *?tgts* = {*t″* . *t″* |∈| *ftransitions M* ∧ *t-source t″* |∈| *q* ∧ *t-input t″* = *t-input t′* ∧ *t-output t″* = *t-output t′*}
define *tgts* **where** *tgts*: *tgts* = *Abs-fset* (*t-target* ‘ *?tgts*)

have *?tgts* ⊆ *transitions M*
using *ftransitions-set*[*of M*]
by (*metis* (*no-types*, *lifting*) *mem-Collect-eq subsetI*)
then have *finite* (*t-target* ‘ *?tgts*)

by (*meson finite-imageI finite-subset fsm-transitions-finite*)
then have *fset tgts* = (*t-target* ‘ *?tgts*)
unfolding *tgts*
using *Abs-fset-inverse*
by *blast*

have *?tgts* ≠ {}
using ∗ ∗∗
by (*metis* (*mono-tags*, *lifting*) *empty-iff ftransitions-set mem-Collect-eq*)
then have *t-target* ‘ *?tgts* ≠ {}
by *blast*
then have *tgts* ≠ {||}
using ‹*fset tgts* = (*t-target* ‘ *?tgts*)›
by *force*

then have (*q*, *t-input t′*, *t-output t′*, *tgts*) |∈| *qtrans*
using *b*
unfolding *qtrans-prop*[*of* (*q*,*t-input t′*,*t-output t′*,*tgts*)]
unfolding *fst-conv snd-conv*
unfolding ‹*fset tgts* = (*t-target* ‘ *?tgts*)›[*symmetric*]
by *blast*
then show *?thesis*
by *auto*
qed
qed

have *i3*: *t-target* |‘| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*)) = *t-target* |‘| *qtrans* |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*))
by *blast*

have *i4*: (⋀*q*. *q* |∈| *t-target* |‘| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*)) ⟹
q |∈| *fPow* (*t-source* |‘| *ftransitions M* |∪| *t-target* |‘| *ftransitions M*))
proof −
fix *q* **assume** *q* |∈| *t-target* |‘| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*))
then have *q* |∈| *t-target* |‘| *qtrans*
by *auto*
then obtain *t* **where** *t* |∈| *qtrans* **and** *t-target t* = *q*
by *auto*
then have *fset q* = *t-target* ‘ {*t′*. *t′* |∈| *ftransitions M* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
unfolding *qtrans-prop* **by** *auto*
then have *fset q* ⊆ *t-target* ‘ *transitions M*
by (*metis* (*no-types*, *lifting*) *ftransitions-set image-Collect-subsetI image-eqI*)
then show *q* |∈| *fPow* (*t-source* |‘| *ftransitions M* |∪| *t-target* |‘| *ftransitions M*)
by (*metis* (*no-types*, *lifting*) *fPowI fset.set-map fset-inject ftransitions-set le-supI2 sup.orderE sup.orderI sup-fset.rep-eq*)

qed

have *i5*: ($\bigwedge q$. q $|\in|$ *dones* $|\cup|$ *?nexts* $\Longrightarrow$ q $|\in|$ *fPow* (*t-source* $|`|$ *ftransitions* $M$ $|\cup|$ *t-target* $|`|$ *ftransitions* M $|\cup|$ {|*initial* M|}))
using *1.prems*(*4*,*3*) *qtrans-prop*
by *auto*

have *i7*: {||} $|\notin|$ *dones* $|\cup|$ *?nexts*
using *1.prems* **by** *fastforce*

show *?thesis*
proof (*cases ?nexts'* $\neq$ {||})
case *False*
then have *?obs* = *?ts'*
using *res-cases* **by** *auto*

have $\bigwedge$ q *io* . q $|\in|$ *?dones'* $\Longrightarrow$ $q \neq$ {||} $\Longrightarrow$ ($\exists$ q' p. q' $|\in|$ q $\wedge$ *path* M q' p $\wedge$ *p-io* p = *io*) $\longleftrightarrow$ ($\exists$ p'. *pathlike ?obs* q p' $\wedge$ *p-io* p' = *io*)
proof −
fix q *io* **assume** q $|\in|$ *?dones'* **and** $q \neq$ {||}
then show ($\exists$ q' p. q' $|\in|$ q $\wedge$ *path* M q' p $\wedge$ *p-io* p = *io*) $\longleftrightarrow$ ($\exists$ p'. *pathlike ?obs* q p' $\wedge$ *p-io* p' = *io*)
proof (*induction io arbitrary*: q)
case *Nil*

have *scheme*: $\bigwedge$ q q' X . q' $|\in|$ q $\Longrightarrow$ q $|\in|$ *fPow* X $\Longrightarrow$ q' $|\in|$ X
by (*simp add*: *fsubsetD*)

obtain q' **where** q' $|\in|$ q
using ‹$q \neq$ {||}› **by** *fastforce*
have q' $|\in|$ *t-source* $|`|$ *ftransitions* $M$ $|\cup|$ *t-target* $|`|$ *ftransitions* M $|\cup|$ {|*FSM.initial* M|}
using *scheme*[*OF* ‹q' $|\in|$ q› *i5*[*OF* ‹q $|\in|$ *?dones'*›]] **.**
then have $q' \in$ *states* M
using *ftransitions-source*[*of* q' M]
using *ftransitions-target*[*of* q' M]
by *force*
then have $\exists$ q' p . q' $|\in|$ q $\wedge$ *path* M q' p $\wedge$ *p-io* p = []
using ‹q' $|\in|$ q› **by** *auto*
moreover have ($\exists$ p'. *pathlike ?obs* q p' $\wedge$ *p-io* p' = [])
by *auto*
ultimately show *?case*
by *simp*
next
case (*Cons ioT ioP*)

have ($\exists$ q' p. q' $|\in|$ q $\wedge$ *path* M q' p $\wedge$ *p-io* p = *ioT* # *ioP*) $\Longrightarrow$ ($\exists$ p'. *pathlike ?obs* q p' $\wedge$ *p-io* p' = *ioT* # *ioP*)

proof −
assume $\exists\, q'\ p.\ q' \mid\in\mid q \wedge path\ M\ q'\ p \wedge p\text{-}io\ p = ioT\ \#\ ioP$
then obtain $q'\ p$ **where** $q' \mid\in\mid q$ **and** $path\ M\ q'\ p$ **and** $p\text{-}io\ p = ioT\ \#\ ioP$
by *meson*

then obtain $tM\ pM$ **where** $p = tM\ \#\ pM$
by *auto*
then have $tM \in transitions\ M$ **and** $t\text{-}source\ tM \mid\in\mid q$
using ‹$path\ M\ q'\ p$› ‹$q' \mid\in\mid q$› **by** *blast+*
then obtain tP **where** $tP \mid\in\mid ts \mid\cup\mid qtrans$
and $t\text{-}source\ tP = q$
and $t\text{-}input\ tP = t\text{-}input\ tM$
and $t\text{-}output\ tP = t\text{-}output\ tM$
using *Cons.prems i2* **by** *blast*

have $path\ M\ (t\text{-}target\ tM)\ pM$ **and** $p\text{-}io\ pM = ioP$
using ‹$path\ M\ q'\ p$› ‹$p\text{-}io\ p = ioT\ \#\ ioP$› **unfolding** ‹$p = tM\ \#\ pM$› **by** *auto*
moreover have $t\text{-}target\ tM \mid\in\mid t\text{-}target\ tP$
using *i1*[*OF* ‹$tP \mid\in\mid ts \mid\cup\mid qtrans$›]
using ‹$p = tM\ \#\ pM$› ‹$path\ M\ q'\ p$› ‹$q' \mid\in\mid q$›
unfolding ‹$t\text{-}input\ tP = t\text{-}input\ tM$› ‹$t\text{-}output\ tP = t\text{-}output\ tM$› ‹$t\text{-}source\ tP = q$›
by *fastforce*
ultimately have $\exists\, q'\ p.\ q' \mid\in\mid t\text{-}target\ tP \wedge path\ M\ q'\ p \wedge p\text{-}io\ p = ioP$
using ‹$p\text{-}io\ pM = ioP$› ‹$path\ M\ (t\text{-}target\ tM)\ pM$› **by** *blast*

have $t\text{-}target\ tP \mid\in\mid dones \mid\cup\mid (t\text{-}target \mid`\mid ts \mid-\mid dones)$
using *False* ‹$tP \mid\in\mid ts \mid\cup\mid qtrans$› **by** *blast*
moreover have $t\text{-}target\ tP \neq \{||\}$
using *i1*[*OF* ‹$tP \mid\in\mid ts \mid\cup\mid qtrans$›] **by** *blast*
ultimately obtain pP **where** $pathlike\ ?obs\ (t\text{-}target\ tP)\ pP$ **and** $p\text{-}io\ pP = ioP$
using *Cons.IH* ‹$\exists\, q'\ p.\ q' \mid\in\mid t\text{-}target\ tP \wedge path\ M\ q'\ p \wedge p\text{-}io\ p = ioP$› **by** *blast*
then have $pathlike\ ?obs\ q\ (tP\#pP)$
using ‹$t\text{-}source\ tP = q$› ‹$tP \mid\in\mid ts \mid\cup\mid qtrans$› ‹$?obs = ?ts'$› **by** *auto*
moreover have $p\text{-}io\ (tP\#pP) = ioT\ \#\ ioP$
using ‹$t\text{-}input\ tP = t\text{-}input\ tM$› ‹$t\text{-}output\ tP = t\text{-}output\ tM$› ‹$p\text{-}io\ p = ioT\ \#\ ioP$› ‹$p = tM\ \#\ pM$› ‹$p\text{-}io\ pP = ioP$› **by** *simp*
ultimately show *?thesis*
by *auto*
qed

moreover have $(\exists\, p'.\ pathlike\ ?obs\ q\ p' \wedge p\text{-}io\ p' = ioT\ \#\ ioP) \Longrightarrow (\exists\, q'\ p.\ q' \mid\in\mid q \wedge path\ M\ q'\ p \wedge p\text{-}io\ p = ioT\ \#\ ioP)$
proof −
assume $\exists\, p'.\ pathlike\ ?obs\ q\ p' \wedge p\text{-}io\ p' = ioT\ \#\ ioP$

then obtain p' **where** *pathlike ?ts'* q p' **and** *p-io* p' = *ioT* # *ioP*
unfolding ‹*?obs* = *?ts'*› **by** *meson*
then obtain *tP pP* **where** p' = *tP#pP*
by *auto*

then have *t-source tP* = q **and** *tP* |∈| *?ts'*
using ‹*pathlike ?ts'* q p'› **by** *auto*

have *pathlike ?ts'* (*t-target tP*) *pP* **and** *p-io pP* = *ioP*
using ‹*pathlike ?ts'* q p'› ‹*p-io* p' = *ioT* # *ioP*› ‹p' = *tP#pP*› **by** *auto*
then have ∃ p'. *pathlike ?ts'* (*t-target tP*) p' ∧ *p-io* p' = *ioP*
by *auto*
moreover have *t-target tP* |∈| *dones* |∪| (*t-target* |'| *ts* |−| *dones*)
using *False* ‹*tP* |∈| *ts* |∪| *qtrans*› **by** *blast*
moreover have *t-target tP* ≠ {||}
using *i1*[*OF* ‹*tP* |∈| *ts* |∪| *qtrans*›] **by** *blast*

ultimately obtain q'' *pM* **where** q'' |∈| *t-target tP* **and** *path M* q'' *pM* **and** *p-io pM* = *ioP*
using *Cons.IH* **unfolding** ‹*?obs* = *?ts'*› **by** *blast*

then obtain *tM* **where** *t-source tM* |∈| q **and** *tM* ∈ *transitions M* **and** *t-input tM* = *t-input tP* **and** *t-output tM* = *t-output tP* **and** *t-target tM* = q''
using *i1*[*OF* ‹*tP* |∈| *ts* |∪| *qtrans*›]
using ‹q'' |∈| *t-target tP*›
unfolding ‹*t-source tP* = q› **by** *force*

have *path M* (*t-source tM*) (*tM#pM*)
using ‹*tM* ∈ *transitions M*› ‹*t-target tM* = q''› ‹*path M* q'' *pM*› **by** *auto*
moreover have *p-io* (*tM#pM*) = *ioT* # *ioP*
using ‹*p-io pM* = *ioP*› ‹*t-input tM* = *t-input tP*› ‹*t-output tM* = *t-output tP*› ‹*p-io* p' = *ioT* # *ioP*› ‹p' = *tP#pP*› **by** *auto*
ultimately show *?thesis*
using ‹*t-source tM* |∈| q› **by** *meson*
qed
ultimately show *?case*
by *meson*
qed
qed

then show *?thesis*
using ‹q |∈| *dones*› ‹{||} |∉| *dones*› **by** *blast*
next
case *True*

have *make-observable-transitions* (*ftransitions M*) *?nexts' ?dones' ?ts'* = *make-observable-transitions* (*ftransitions M*) *?nexts dones ts*

proof (*cases ?nexts′* = {||})
case *True*
then have *?obs* = *?ts′*
using *qtrans-def* **by** *auto*
moreover have *make-observable-transitions* (*ftransitions M*) *?nexts′ ?dones′ ?ts′* = *?ts′*
unfolding *make-observable-transitions.simps*[*of ftransitions M ?nexts′ ?dones′ ?ts′*]
unfolding *True Let-def* **by** *auto*
ultimately show *?thesis*
by *blast*
next
case *False*
then show *?thesis*
unfolding *make-observable-transitions.simps*[*of ftransitions M ?nexts dones ts*] *qtrans-def Let-def* **by** *auto*
qed

then have *IStep*: ⋀ *q io* . *q* |∈| *?dones′* ⟹
(∃ *q′ p*. *q′* |∈| *q* ∧ *path M q′ p* ∧ *p-io p* = *io*) =
(∃ *p′*. *pathlike* (*make-observable-transitions* (*ftransitions M*) *?nexts dones ts*) *q p′* ∧ *p-io p′* = *io*)
using *1.hyps*[*OF qtrans-def - - - - i1 i2 i4 i5 i7*] *True*
unfolding *i3*
by *presburger*

show *?thesis*
unfolding ‹*io* = *ioT* # *ioP*›
proof
show ∃ *q′ p*. *q′* |∈| *q* ∧ *path M q′ p* ∧ *p-io p* = *ioT* # *ioP* ⟹ ∃ *p′*. *pathlike ?obs q p′* ∧ *p-io p′* = *ioT* # *ioP*
proof −
assume ∃ *q′ p*. *q′* |∈| *q* ∧ *path M q′ p* ∧ *p-io p* = *ioT* # *ioP*
then obtain *q′ p* **where** *q′* |∈| *q* **and** *path M q′ p* **and** *p-io p* = *ioT* # *ioP*
by *meson*

then obtain *tM pM* **where** *p* = *tM* # *pM*
by *auto*
then have *tM* ∈ *transitions M* **and** *t-source tM* |∈| *q*
using ‹*path M q′ p*› ‹*q′* |∈| *q*› **by** *blast+*

then obtain *tP* **where** *tP* |∈| *ts*
and *t-source tP* = *q*
and *t-input tP* = *t-input tM*
and *t-output tP* = *t-output tM*
using *1.prems*(*2*,*6*) **by** *blast*

then have *i9*: *t-target tP* |∈| *dones* |∪| *?nexts*
by *simp*

have *path M* (*t-target tM*) *pM* **and** *p-io pM* = *ioP*
using ‹*path M q′ p*› ‹*p-io p* = *ioT* # *ioP*› **unfolding** ‹*p* = *tM* # *pM*› **by** *auto*
moreover have *t-target tM* |∈| *t-target tP*
using *1.prems*(*1*)[*OF* ‹*tP* |∈| *ts*›] ‹*p* = *tM* # *pM*› ‹*path M q′ p*› ‹*q′* |∈| *q*›
unfolding ‹*t-input tP* = *t-input tM*› ‹*t-output tP* = *t-output tM*› ‹*t-source tP* = *q*›
by *fastforce*
ultimately have ∃ *q′ p*. *q′* |∈| *t-target tP* ∧ *path M q′ p* ∧ *p-io p* = *ioP*
using ‹*p-io pM* = *ioP*› ‹*path M* (*t-target tM*) *pM*› **by** *blast*

obtain *pP* **where** *pathlike ?obs* (*t-target tP*) *pP* **and** *p-io pP* = *ioP*
using ‹∃ *q′ p*. *q′* |∈| *t-target tP* ∧ *path M q′ p* ∧ *p-io p* = *ioP*› **unfolding** *IStep*[*OF i9*]
using *that* **by** *blast*

then have *pathlike ?obs q* (*tP*#*pP*)
using ‹*t-source tP* = *q*› ‹*tP* |∈| *ts*› *make-observable-transitions-mono* **by** *blast*
moreover have *p-io* (*tP*#*pP*) = *ioT* # *ioP*
using ‹*t-input tP* = *t-input tM*› ‹*t-output tP* = *t-output tM*› ‹*p-io p* = *ioT* # *ioP*› ‹*p* = *tM* # *pM*› ‹*p-io pP* = *ioP*› **by** *simp*
ultimately show *?thesis*
by *auto*
qed

show ∃ *p′*. *pathlike ?obs q p′* ∧ *p-io p′* = *ioT* # *ioP* ⟹ ∃ *q′ p*. *q′* |∈| *q* ∧ *path M q′ p* ∧ *p-io p* = *ioT* # *ioP*
proof −
assume ∃ *p′*. *pathlike ?obs q p′* ∧ *p-io p′* = *ioT* # *ioP*
then obtain *p′* **where** *pathlike ?obs q p′* **and** *p-io p′* = *ioT* # *ioP*
by *meson*
then obtain *tP pP* **where** *p′* = *tP*#*pP*
by *auto*

have ⋀ *t′* . *t′* |∈| *ftransitions M* = (*t′* ∈ *transitions M*)
using *ftransitions-set*
by *metis*

from ‹*p′* = *tP*#*pP*› **have** *t-source tP* = *q* **and** *tP* |∈| *?obs*
using ‹*pathlike ?obs q p′*› **by** *auto*
then have *tP* |∈| *ts*

using *1.prems*(*6*) *make-observable-transitions-t-source*[*of ts dones ftransitions M*] *1.prems*(*1*,*2*)
unfolding ‹$\bigwedge$ *t′* . *t′* |∈| *ftransitions M* = (*t′* ∈ *transitions M*)›
by *blast*
then have *i9*: *t-target tP* |∈| *dones* |∪| *?nexts*
by *simp*

have *pathlike ?obs* (*t-target tP*) *pP* **and** *p-io pP* = *ioP*
using ‹*pathlike ?obs q p′*› ‹*p-io p′* = *ioT* # *ioP*› ‹*p′* = *tP*#*pP*› **by** *auto*
then have ∃ *p′*. *pathlike ?obs* (*t-target tP*) *p′* ∧ *p-io p′* = *ioP*
by *auto*
then obtain *q″ pM* **where** *q″* |∈| *t-target tP* **and** *path M q″ pM* **and** *p-io pM* = *ioP*
using *IStep*[*OF i9*] **by** *blast*

obtain *tM* **where** *t-source tM* |∈| *q* **and** *tM* ∈ *transitions M* **and** *t-input tM* = *t-input tP* **and** *t-output tM* = *t-output tP* **and** *t-target tM* = *q″*
using *1.prems*(*1*)[*OF* ‹*tP* |∈| *ts*›] ‹*q″* |∈| *t-target tP*›
unfolding ‹*t-source tP* = *q*›
by *force*

have *path M* (*t-source tM*) (*tM*#*pM*)
using ‹*tM* ∈ *transitions M*› ‹*t-target tM* = *q″*› ‹*path M q″ pM*› **by** *auto*
moreover have *p-io* (*tM*#*pM*) = *ioT* # *ioP*
using ‹*p-io pM* = *ioP*› ‹*t-input tM* = *t-input tP*› ‹*t-output tM* = *t-output tP*› ‹*p-io p′* = *ioT* # *ioP*› ‹*p′* = *tP*#*pP*› **by** *auto*
ultimately show *?thesis*
using ‹*t-source tM* |∈| *q*› **by** *meson*
qed
qed
qed
qed
qed

fun *observable-fset* :: (*′a*,*′b*,*′c*) *transition fset* ⇒ *bool* **where**
observable-fset ts = (∀ *t1 t2* . *t1* |∈| *ts* ⟶ *t2* |∈| *ts* ⟶
t-source t1 = *t-source t2* ⟶ *t-input t1* = *t-input t2* ⟶ *t-output t1* = *t-output t2*
⟶ *t-target t1* = *t-target t2*)

lemma *make-observable-transitions-observable* :
assumes $\bigwedge$ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target*

t) = *t-target* ‘ {*t′* . *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
and *observable-fset ts*
shows *observable-fset* (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*)
using *assms* **proof** (*induction base-trans* (*fimage t-target ts*) |−| *dones dones ts rule*: *make-observable-transitions.induct*)
case (*1 base-trans dones ts*)

let *?nexts* = (*fimage t-target ts*) |−| *dones*

define *qtrans* **where** *qtrans-def*: *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *base-trans*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |‘| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *?nexts*)

have *qtrans-prop*: ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *?nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* . *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
by *presburger*

let *?dones′* = *dones* |∪| *?nexts*
let *?ts′* = *ts* |∪| *qtrans*
let *?nexts′* = (*fimage t-target qtrans*) |−| *?dones′*

have *observable-fset qtrans*
using *qtrans-prop*
unfolding *observable-fset.simps*
by (*metis* (*mono-tags*, *lifting*) *Collect-cong fset-inject*)
moreover have *t-source* |‘| *qtrans* |∩| *t-source* |‘| *ts* = {||}
using *1.prems*(*1*) *qtrans-prop* **by** *force*
ultimately have *observable-fset ?ts′*
using *1.prems*(*2*) **unfolding** *observable-fset.simps*
by *blast*

have *res-cases*: *make-observable-transitions base-trans ?nexts dones ts* = (*if ?nexts′* = {||}
then ?ts′
else make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′)
unfolding *make-observable-transitions.simps*[*of base-trans ?nexts dones ts*] *qtrans-def Let-def* **by** *simp*

show *?case* **proof** (*cases ?nexts′* = {||})

case *True*
then have *make-observable-transitions base-trans ?nexts dones ts* = *?ts′*
using *res-cases* **by** *simp*
then show *?thesis*
using ‹*observable-fset ?ts′*› **by** *simp*
next
case *False*
then have ∗: *make-observable-transitions base-trans ?nexts dones ts* = *make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′*
using *res-cases* **by** *simp*

have *i1*: (⋀*t*. *t* |∈| *ts* |∪| *qtrans* ⟹
t-source t |∈| *dones* |∪| *?nexts* ∧
t-target t ≠ {||} ∧
fset (*t-target t*) =
t-target ‘
{*t′* . *t′* |∈| *base-trans* ∧
t-source t′ |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*})
using *1.prems*(*1*) *qtrans-prop* **by** *blast*

have *i3*: *t-target* |‘| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*)) = *t-target* |‘| *qtrans* |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*))
by *auto*

have *i4*: *t-target* |‘| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |‘| *ts* |−| *dones*)) ≠ {||}
using *False* **by** *auto*

show *?thesis*
using *1.hyps*[*OF qtrans-def - - i3 i4 i1* ‹*observable-fset ?ts′*›] **unfolding** ∗ *i3* **by** *metis*
qed
qed

lemma *make-observable-transitions-transition-props* :
assumes ⋀ *t* . *t* |∈| *ts* ⟹ *t-source t* |∈| *dones* ∧ *t-target t* |∈| *dones* |∪| ((*fimage t-target ts*) |−| *dones*) ∧ *t-input t* |∈| *t-input* |‘| *base-trans* ∧ *t-output t* |∈| *t-output* |‘| *base-trans*
assumes *t* |∈| *make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*
shows *t-source t* |∈| *dones* |∪| (*t-target* |‘| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-target t* |∈| *dones* |∪| (*t-target* |‘| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-input t* |∈| *t-input* |‘| *base-trans*
and *t-output t* |∈| *t-output* |‘| *base-trans*
proof −

have *t-source t* |∈| *dones* |∪| (*t-target* |`| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
∧ *t-target t* |∈| *dones* |∪| (*t-target* |`| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
∧ *t-input t* |∈| *t-input* |`| *base-trans*
∧ *t-output t* |∈| *t-output* |`| *base-trans*
using *assms*(*1*,*2*)
proof (*induction base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts rule*: *make-observable-transitions.induct*)
case (*1 base-trans dones ts*)

let *?nexts* = ((*fimage t-target ts*) |−| *dones*)

define *qtrans* **where** *qtrans-def*: *qtrans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λ*t* . *t-source t* |∈| *q*) *base-trans*;
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . (*q*,*x*,*y*, *t-target* |`| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)) *?nexts*)

have *qtrans-prop*: ⋀ *t* . *t* |∈| *qtrans* ⟷ *t-source t* |∈| *?nexts* ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ` {*t′* . *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using *make-observable-transitions-qtrans-helper*[*OF qtrans-def*]
by *presburger*

let *?dones′* = *dones* |∪| *?nexts*
let *?ts′* = *ts* |∪| *qtrans*
let *?nexts′* = (*fimage t-target qtrans*) |−| *?dones′*

have *res-cases*: *make-observable-transitions base-trans ?nexts dones ts* = (*if ?nexts′* = {||}
then ?ts′
else make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′)
unfolding *make-observable-transitions.simps*[*of base-trans ?nexts dones ts*] *qtrans-def Let-def* **by** *simp*

have *qtrans-trans-prop*: (⋀*t*. *t* |∈| *qtrans* ⟹
t-source t |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*) ∧
t-target t |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*) |∪| (*t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*))) ∧
t-input t |∈| *t-input* |`| *base-trans* ∧ *t-output t* |∈| *t-output* |`| *base-trans*) (**is** ⋀ *t* . *t* |∈| *qtrans* ⟹ *?P t*)
proof −
fix *t* **assume** *t* |∈| *qtrans*

then have *t-source t* |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*)
using ‹*t* |∈| *qtrans*› **unfolding** *qtrans-prop*[*of t*] **by** *blast*
moreover have *t-target t* |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*) |∪| (*t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*)))

using ‹*t* |∈| *qtrans*› *1.prems*(*1*) **by** *blast*
moreover have *t-input t* |∈| *t-input* |`| *base-trans* ∧ *t-output t* |∈| *t-output* |`| *base-trans*
proof −
obtain *t′* **where** *t′* ∈ {*t′*. *t′* |∈| *base-trans* ∧ *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using ‹*t* |∈| *qtrans*› **unfolding** *qtrans-prop*[*of t*]
by (*metis* (*mono-tags*, *lifting*) *Collect-empty-eq bot-fset.rep-eq empty-is-image fset-inject mem-Collect-eq*)
then show *?thesis*
by *force*
qed
ultimately show *?P t*
by *blast*
qed

show *?case* **proof** (*cases ?nexts′* = {||})
case *True*
then have *t* |∈| *?ts′*
using *1.prems*(*2*) *res-cases* **by** *force*
then show *?thesis*
using *1.prems*(*1*) *qtrans-trans-prop*
by (*metis True fimage-funion funion-fminus-cancel funion-iff res-cases*)
next
case *False*
then have ∗: *make-observable-transitions base-trans ?nexts dones ts* = *make-observable-transitions base-trans ?nexts′ ?dones′ ?ts′*
using *res-cases* **by** *simp*

have *i1*: *t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*)) = *t-target* |`| *qtrans* |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*))
by *blast*

have *i2*: *t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*)) ≠ {||}
using *False* **by** *blast*

have *i3*: (⋀*t*. *t* |∈| *ts* |∪| *qtrans* ⟹
t-source t |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*) ∧
t-target t |∈| *dones* |∪| (*t-target* |`| *ts* |−| *dones*) |∪| (*t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*))) ∧
t-input t |∈| *t-input* |`| *base-trans* ∧ *t-output t* |∈| *t-output* |`| *base-trans*)
using *1.prems*(*1*) *qtrans-trans-prop* **by** *blast*

have *i4*: *t* |∈| *make-observable-transitions base-trans* (*t-target* |`| (*ts* |∪| *qtrans*) |−| (*dones* |∪| (*t-target* |`| *ts* |−| *dones*))) (*dones* |∪| (*t-target* |`| *ts* |−| *dones*)) (*ts* |∪| *qtrans*)

using *1.prems(2)* **unfolding** ∗ *i1* **by** *assumption*

show *?thesis*
using *1.hyps*[*OF qtrans-def - - i1 i2 i3 i4*] **unfolding** *i1* **unfolding** ∗[*symmetric*]
using *make-observable-transitions-mono*[*of ts base-trans ?nexts dones*] **by** *blast*
qed
qed
then show *t-source t* |∈| *dones* |∪| (*t-target* |'| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-target t* |∈| *dones* |∪| (*t-target* |'| (*make-observable-transitions base-trans* ((*fimage t-target ts*) |−| *dones*) *dones ts*))
and *t-input t* |∈| *t-input* |'| *base-trans*
and *t-output t* |∈| *t-output* |'| *base-trans*
by *blast*+
qed

fun *make-observable* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* ⇒ ('*a fset*,'*b*,'*c*) *fsm* **where**
make-observable M = (*let*
initial-trans = (*let qts* = *ffilter* (λ*t* . *t-source t* = *initial M*) (*ftransitions M*);
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage (λ(*x*,*y*) . ({|*initial M*|},*x*,*y*, *t-target* |'| ((*ffilter* (λ*t* . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*);
nexts = *fimage t-target initial-trans* |−| {|{|*initial M*|}|};
ptransitions = *make-observable-transitions* (*ftransitions M*) *nexts* {|{|*initial M*|}|} *initial-trans*;
pstates = *finsert* {|*initial M*|} (*t-target* |'| *ptransitions*);
M' = *create-unconnected-fsm-from-fsets* {|*initial M*|} *pstates* (*finputs M*) (*foutputs M*)
in add-transitions M' (*fset ptransitions*))

lemma *make-observable-language-observable* :
shows *L* (*make-observable M*) = *L M*
and *observable* (*make-observable M*)
and *initial* (*make-observable M*) = {|*initial M*|}
and *inputs* (*make-observable M*) = *inputs M*
and *outputs* (*make-observable M*) = *outputs M*
proof −

define *initial-trans* **where** *initial-trans* = (*let qts* = *ffilter* (λt . *t-source t* = *initial M*) (*ftransitions M*);
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage ($\lambda(x,y)$. ({|*initial M*|},*x*,*y*, *t-target* |'| ((*ffilter* (λt . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*)))) *ios*)
moreover define *ptransitions* **where** *ptransitions* = *make-observable-transitions* (*ftransitions M*) (*fimage t-target initial-trans* |−| {|{|*initial M*|}|}) {|{|*initial M*|}|} *initial-trans*
moreover define *pstates* **where** *pstates* = *finsert* {|*initial M*|} (*t-target* |'| *ptransitions*)
moreover define *M′* **where** *M′* = *create-unconnected-fsm-from-fsets* {|*initial M*|} *pstates* (*finputs M*) (*foutputs M*)

ultimately have *make-observable M* = *add-transitions M′* (*fset ptransitions*)
unfolding *make-observable.simps Let-def* **by** *blast*

have {|*initial M*|} |∈| *pstates*
unfolding *pstates-def* **by** *blast*
have *inputs M′* = *inputs M*
unfolding *M′-def create-unconnected-fsm-from-fsets-simps*(*3*)[*OF* ‹{|*initial M*|} |∈| *pstates*›, *of finputs M foutputs M*]
using *fset-of-list.rep-eq inputs-as-list-set* **by** *fastforce*
have *outputs M′* = *outputs M*
unfolding *M′-def create-unconnected-fsm-from-fsets-simps*(*4*)[*OF* ‹{|*initial M*|} |∈| *pstates*›, *of finputs M foutputs M*]
using *fset-of-list.rep-eq outputs-as-list-set* **by** *fastforce*
have *states M′* = *fset pstates* **and** *transitions M′* = {} **and** *initial M′* = {|*initial M*|}
unfolding *M′-def create-unconnected-fsm-from-fsets-simps*(*1*,*2*,*5*)[*OF* ‹{|*initial M*|} |∈| *pstates*›] **by** *simp*+

have *initial-trans-prop*: $\bigwedge$ *t* . *t* |∈| *initial-trans* ⟷ *t-source t* |∈| {|{|*FSM.initial M*|}|} ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* ∈ *transitions M* . *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
proof −
have ∗:$\bigwedge$ *t′* . *t′* |∈| *ftransitions M* = (*t′* ∈ *transitions M*)
using *ftransitions-set*
by *metis*
have ∗∗: *initial-trans* = *ffUnion* (*fimage* (λ *q* . (*let qts* = *ffilter* (λt . *t-source t* |∈| *q*) (*ftransitions M*);
ios = *fimage* (λ *t* . (*t-input t*, *t-output t*)) *qts*
in fimage ($\lambda(x,y)$. (*q*,*x*,*y*, *t-target* |'| (*ffilter* (λt . (*t-input t*, *t-output t*) = (*x*,*y*)) *qts*))) *ios*)) {|{|*initial M*|}|})
unfolding *initial-trans-def* **by** *auto*
show $\bigwedge$ *t* . *t* |∈| *initial-trans* ⟷ *t-source t* |∈| {|{|*FSM.initial M*|}|} ∧ *t-target t* ≠ {||} ∧ *fset* (*t-target t*) = *t-target* ‘ {*t′* ∈ *transitions M* . *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}

unfolding *make-observable-transitions-qtrans-helper*[*OF* ∗∗] ∗
by *presburger*
qed

have *well-formed-transitions*: $\bigwedge$ *t* . *t* ∈ (*fset ptransitions*) ⟹ *t-source t* ∈ *states M′* ∧ *t-input t* ∈ *inputs M′* ∧ *t-output t* ∈ *outputs M′* ∧ *t-target t* ∈ *states M′*
(**is** $\bigwedge$ *t* . *t* ∈ (*fset ptransitions*) ⟹ *?P1 t* ∧ *?P2 t* ∧ *?P3 t* ∧ *?P4 t*)
proof −
fix *t* **assume** *t* ∈ (*fset ptransitions*)

then have *i2*: *t* |∈| *make-observable-transitions* (*ftransitions M*) (*fimage t-target initial-trans* |−| {|{|*initial M*|}|}) {|{|*initial M*|}|} *initial-trans*
using *ptransitions-def*
by *metis*

have *i1*: (⋀*t*. *t* |∈| *initial-trans* ⟹
t-source t |∈| {|{|*FSM.initial M*|}|} ∧
t-target t |∈| {|{|*FSM.initial M*|}|} |∪| (*t-target* |‘| *initial-trans* |−| {|{|*FSM.initial M*|}|}) ∧
t-input t |∈| *t-input* |‘| *ftransitions M* ∧ *t-output t* |∈| *t-output* |‘| *ftransitions M*) (**is** $\bigwedge$ *t* . *t* |∈| *initial-trans* ⟹ *?P t*)
proof −
fix *t* **assume** ∗: *t* |∈| *initial-trans*
then have *t-source t* |∈| {|{|*FSM.initial M*|}|}
and *t-target t* ≠ {||}
and *fset* (*t-target t*) = *t-target* ‘ {*t′* ∈ *FSM.transitions M*. *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}
using *initial-trans-prop* **by** *blast+*

have *t-target t* |∈| {|{|*FSM.initial M*|}|} |∪| (*t-target* |‘| *initial-trans* |−| {|{|*FSM.initial M*|}|})
using ∗ **by** *blast*

moreover have *t-input t* |∈| *t-input* |‘| *ftransitions M* ∧ *t-output t* |∈| *t-output* |‘| *ftransitions M*
proof −
obtain *t′* **where** *t′* ∈ *transitions M* **and** *t-input t* = *t-input t′* **and** *t-output t* = *t-output t′*
using ‹*t-target t* ≠ {||}› ‹*fset* (*t-target t*) = *t-target* ‘ {*t′* ∈ *FSM.transitions M*. *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*}›
by (*metis* (*mono-tags*, *lifting*) *bot-fset.rep-eq empty-Collect-eq fset-inject image-empty*)

have *fset* (*ftransitions M*) = *transitions M*
by (*simp add*: *fset-of-list.rep-eq fsm-transitions-finite*)
then show *?thesis*
unfolding ‹*t-input t* = *t-input t′*› ‹*t-output t* = *t-output t′*›
using ‹*t′* ∈ *transitions M*›
by *auto*

```
    qed

    ultimately show ?P t
      using ‹t-source t |∈| {|{|FSM.initial M|}|}› by blast
  qed


  have ?P1 t
     using make-observable-transitions-transition-props(1)[OF i1 i2] unfolding
pstates-def ptransitions-def ‹states M′ = fset pstates›
    by (metis finsert-is-funion)
  moreover have ?P2 t
  proof−
    have t-input t |∈| t-input |`| ftransitions M
      using make-observable-transitions-transition-props(3)[OF i1 i2] by blast
    then have t-input t ∈ t-input ` transitions M
      using ftransitions-set by (metis (mono-tags, lifting) fset.set-map)
    then show ?thesis
     using finputs-set fsm-transition-input ‹inputs M′ = inputs M› by fastforce
  qed
  moreover have ?P3 t
  proof−
    have t-output t |∈| t-output |`| ftransitions M
      using make-observable-transitions-transition-props(4)[OF i1 i2] by blast
    then have t-output t ∈ t-output ` transitions M
      using ftransitions-set by (metis (mono-tags, lifting) fset.set-map)
    then show ?thesis
        using foutputs-set fsm-transition-output ‹outputs M′ = outputs M› by
fastforce
  qed
  moreover have ?P4 t
     using make-observable-transitions-transition-props(2)[OF i1 i2] unfolding
pstates-def ptransitions-def ‹states M′ = fset pstates›
    by (metis finsert-is-funion)

  ultimately show ?P1 t ∧ ?P2 t ∧ ?P3 t ∧ ?P4 t
    by blast
 qed

 have initial (make-observable M) = {|initial M|}
  and states (make-observable M) = fset pstates
  and inputs (make-observable M) = inputs M
  and outputs (make-observable M) = outputs M
  and transitions (make-observable M) = fset ptransitions
   using add-transitions-simps[OF well-formed-transitions, of fset ptransitions]
  unfolding ‹make-observable M = add-transitions M′ (fset ptransitions)›[symmetric]
         ‹inputs M′ = inputs M› ‹outputs M′ = outputs M› ‹initial M′ = {|initial
M|}› ‹states M′ = fset pstates› ‹transitions M′ = {}›
   by blast+
```

then show *initial* (*make-observable M*) = {|*initial M*|} **and** *inputs* (*make-observable M*) = *inputs M* **and** *outputs* (*make-observable M*) = *outputs M*
by *presburger+*

have *i1*: (⋀*t*. *t* |∈| *initial-trans* ⟹
t-source t |∈| {|{|*FSM.initial M*|}|} ∧
t-target t ≠ {||} ∧
fset (*t-target t*) = *t-target* ‘ {*t′* ∈ *FSM.transitions M*. *t-source t′* |∈| *t-source t* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*})
using *initial-trans-prop* **by** *blast*

have *i2*: (⋀*q t′*.
q |∈| {|{|*FSM.initial M*|}|} ⟹
t′ ∈ *FSM.transitions M* ⟹ *t-source t′* |∈| *q* ⟹ ∃ *t*. *t* |∈| *initial-trans* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t′* ∧ *t-output t* = *t-output t′*)
proof −
fix *q t′* **assume** *q* |∈| {|{|*FSM.initial M*|}|} **and** *t′* ∈ *FSM.transitions M* **and** *t-source t′* |∈| *q*
then have *q* = {|*FSM.initial M*|} **and** *t-source t′* = *initial M*
by *auto*

define *tgt* **where** *tgt* = *t-target* ‘ {*t″* ∈ *FSM.transitions M*. *t-source t″* |∈| {|*FSM.initial M*|} ∧ *t-input t″* = *t-input t′* ∧ *t-output t″* = *t-output t′*}
have *t-target t′* ∈ *tgt*
unfolding *tgt-def* **using** ‹*t′* ∈ *FSM.transitions M*› ‹*t-source t′* = *initial M*› **by** *auto*
then have *tgt* ≠ {}
by *auto*

have *finite tgt*
using *fsm-transitions-finite*[*of M*] **unfolding** *tgt-def* **by** *auto*
then have *fset* (*Abs-fset tgt*) = *tgt*
by (*simp add*: *Abs-fset-inverse*)
then have *Abs-fset tgt* ≠ {||}
using ‹*tgt* ≠ {}› **by** *auto*

let *?t* = ({|*FSM.initial M*|}, *t-input t′*, *t-output t′*, *Abs-fset tgt*)
have *?t* |∈| *initial-trans*
unfolding *initial-trans-prop fst-conv snd-conv* ‹*fset* (*Abs-fset tgt*) = *tgt*›
unfolding ‹*tgt* = *t-target* ‘ {*t″* ∈ *FSM.transitions M*. *t-source t″* |∈| {|*FSM.initial M*|} ∧ *t-input t″* = *t-input t′* ∧ *t-output t″* = *t-output t′*}›[*symmetric*]
using ‹*Abs-fset tgt* ≠ {||}›
by *blast*
then show ∃ *t*. *t* |∈| *initial-trans* ∧ *t-source t* = *q* ∧ *t-input t* = *t-input t′* ∧ *t-output t* = *t-output t′*
using ‹*q* = {|*FSM.initial M*|}› **by** *auto*
qed

have *i3*: ($\bigwedge q$. q |∈| *t-target* |`| *initial-trans* |−| {|{|*FSM.initial M*|}|} $\Longrightarrow$ $q$ |∈| *fPow* (*t-source* |`| *ftransitions M* |∪| *t-target* |`| *ftransitions M*))
proof −
fix q **assume** q |∈| *t-target* |`| *initial-trans* |−| {|{|*FSM.initial M*|}|}
then obtain t **where** t |∈| *initial-trans* **and** *t-target* $t = q$
by *auto*

have *fset* $q \subseteq$ *t-target* ` (*transitions M*)
using ‹t |∈| *initial-trans*›
unfolding *initial-trans-prop* ‹*t-target* $t = q$›
by *auto*
then have q |⊆| (*t-target* |`| *ftransitions M*)
using *ftransitions-set*[*of M*]
by (*simp add*: *less-eq-fset.rep-eq*)
then show q |∈| *fPow* (*t-source* |`| *ftransitions M* |∪| *t-target* |`| *ftransitions M*)
by *auto*
qed

have *i4*: ($\bigwedge q$. q |∈| {|{|*FSM.initial M*|}|} $\Longrightarrow$ q |∈| *fPow* (*t-source* |`| *ftransitions M* |∪| *t-target* |`| *ftransitions M* |∪| {|*FSM.initial M*|}))
and *i5*: {||} |∉| {|{|*FSM.initial M*|}|}
and *i6*: {|*FSM.initial M*|} |∈| {|{|*FSM.initial M*|}|}
by *blast*+

show *L* (*make-observable M*) = *L M*
proof −
have ∗: $\bigwedge$ p . *pathlike ptransitions* {|*initial M*|} p = *path* (*make-observable M*) {|*initial M*|} p
proof
have $\bigwedge$ q p . $p \neq []$ $\Longrightarrow$ *pathlike ptransitions* q p $\Longrightarrow$ *path* (*make-observable M*) q p
proof −
fix q p **assume** $p \neq []$ **and** *pathlike ptransitions* q p
then show *path* (*make-observable M*) q p
proof (*induction p arbitrary*: q)
case *Nil*
then show *?case* **by** *blast*
next
case (*Cons t p*)
then have t |∈| *ptransitions* **and** *pathlike ptransitions* (*t-target t*) p **and** *t-source* $t = q$
by *blast*+

have $t \in$ *transitions* (*make-observable M*)
using ‹t |∈| *ptransitions*› ‹*transitions* (*make-observable M*) = *fset ptransitions*›

by *metis*
moreover have *path* (*make-observable M*) (*t-target t*) *p*
using *Cons.IH*[*OF* - ‹*pathlike ptransitions* (*t-target t*) *p*›] *calculation* **by** *blast*
ultimately show *?case*
using ‹*t-source t* = *q*› **by** *blast*
qed
qed
then show $\bigwedge$ *p* . *pathlike ptransitions* {|*initial M*|} *p* $\Longrightarrow$ *path* (*make-observable M*) {|*initial M*|} *p*
by (*metis* ‹*FSM.initial* (*make-observable M*) = {|*FSM.initial M*|}› *fsm-initial path.nil*)

show $\bigwedge$ *q p* . *path* (*make-observable M*) *q p* $\Longrightarrow$ *pathlike ptransitions q p*
proof −
fix *q p* **assume** *path* (*make-observable M*) *q p*
then show *pathlike ptransitions q p*
proof (*induction p arbitrary*: *q rule*: *list.induct*)
case *Nil*
then show *?case* **by** *blast*
next
case (*Cons t p*)
then have *t* ∈ *transitions* (*make-observable M*) **and** *path* (*make-observable M*) (*t-target t*) *p* **and** *t-source t* = *q*
by *blast*+

have *t* |∈| *ptransitions*
using ‹*t* ∈ *transitions* (*make-observable M*)› ‹*transitions* (*make-observable M*) = *fset ptransitions*›
by *metis*
then show *?case*
using *Cons.IH*[*OF* ‹*path* (*make-observable M*) (*t-target t*) *p*›] ‹*t-source t* = *q*› **by** *blast*
qed
qed
qed

have $\bigwedge$ *io* . (∃ *q′ p*. *q′* |∈| {|*FSM.initial M*|} ∧ *path M q′ p* ∧ *p-io p* = *io*) = (∃ *p′*. *pathlike ptransitions* {|*FSM.initial M*|} *p′* ∧ *p-io p′* = *io*)
using *make-observable-transitions-path*[*OF i1 i2 i3 i4 i5 i6*] **unfolding** *ptransitions-def*[*symmetric*] **by** *blast*
then have $\bigwedge$ *io* . (∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *p-io p* = *io*) = (∃ *p′* . *path* (*make-observable M*) {|*initial M*|} *p′* ∧ *p-io p′* = *io*)
unfolding ∗
by (*metis* (*no-types*, *lifting*) *fempty-iff finsert-iff*)
then show *?thesis*
unfolding *LS.simps* ‹*initial* (*make-observable M*) = {|*initial M*|}› **by** (*metis* (*no-types*, *lifting*))
qed

```
  show observable (make-observable M)
  proof −

    have i2: observable-fset initial-trans
      unfolding observable-fset.simps
      unfolding initial-trans-prop
      using fset-inject
      by metis

    have ⋀ t′ . t′ |∈| ftransitions M = (t′ ∈ transitions M)
      using ftransitions-set
      by metis
    have observable-fset ptransitions
      using make-observable-transitions-observable[OF - i2, of {| {|initial M|} |}
ftransitions M] i1
      unfolding ptransitions-def ‹⋀ t′ . t′ |∈| ftransitions M = (t′ ∈ transitions
M)›
      by blast
    then show ?thesis
    unfolding observable.simps observable-fset.simps ‹transitions (make-observable
M) = fset ptransitions›
      by metis
  qed
qed

end
```

10 Prefix Tree

This theory introduces a tree to efficiently store prefix-complete sets of lists. Several functions to lookup or merge subtrees are provided.

```
theory Prefix-Tree
imports Util HOL−Library.Mapping HOL−Library.List-Lexorder
begin

datatype ′a prefix-tree = PT ′a ⇀ ′a prefix-tree

definition empty :: ′a prefix-tree where
  empty = PT Map.empty

fun isin :: ′a prefix-tree ⇒ ′a list ⇒ bool where
  isin t [] = True |
  isin (PT m) (x # xs) = (case m x of None ⇒ False | Some t ⇒ isin t xs)

lemma isin-prefix :
  assumes isin t (xs@xs′)
```

```
  shows isin t xs
proof −
  obtain m where t = PT m
    by (metis prefix-tree.exhaust)

  show ?thesis using assms unfolding ‹t = PT m›
  proof (induction xs arbitrary: m)
    case Nil
    then show ?case by auto
  next
    case (Cons x xs)
    then have isin (PT m) (x # (xs @ xs'))
      by auto
    then obtain m' where m x = Some (PT m')
                    and isin (PT m') (xs@xs')
      unfolding isin.simps
     by (metis option.exhaust option.simps(4) option.simps(5) prefix-tree.exhaust)

    then show ?case
      using Cons.IH[of m'] by auto
  qed
qed


fun set :: 'a prefix-tree ⇒ 'a list set where
  set t = {xs . isin t xs}

lemma set-empty : set empty = ({[]} :: 'a list set)
proof
  show set empty ⊆ ({[]} :: 'a list set)
  proof
    fix xs :: 'a list
    assume xs ∈ set empty
    then have isin empty xs
      by auto

    have xs = []
    proof (rule ccontr)
      assume xs ≠ []
      then obtain x xs' where xs = x#xs'
        using list.exhaust by auto
      then have Map.empty x ≠ None
        using ‹isin empty xs› unfolding empty-def
        by simp
      then show False
        by auto
    qed
    then show xs ∈ {[]}
      by blast
```

qed
show $(\{[]\} :: 'a\ list\ set) \subseteq set\ empty$
unfolding *set.simps empty-def*
by *simp*
qed

lemma *set-Nil* : $[] \in set\ t$
by *auto*

fun *insert* :: $'a\ prefix\text{-}tree \Rightarrow 'a\ list \Rightarrow 'a\ prefix\text{-}tree$ **where**
$insert\ t\ [] = t$ |
$insert\ (PT\ m)\ (x\#xs) = PT\ (m(x \mapsto insert\ (case\ m\ x\ of\ None \Rightarrow empty \mid Some\ t' \Rightarrow t')\ xs))$

lemma *insert-isin-prefix* : $isin\ (insert\ t\ (xs@xs')\ xs$
proof (*induction xs arbitrary*: *t*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs*)
moreover obtain *m* **where** $t = PT\ m$
using *prefix-tree.exhaust* **by** *auto*
ultimately obtain t' **where** $(m(x \mapsto insert\ (case\ m\ x\ of\ None \Rightarrow empty \mid Some\ t' \Rightarrow t')\ xs))\ x = Some\ t'$
by *simp*
then have $isin\ (insert\ t\ ((x\#xs)@xs'))\ (x\#xs) = isin\ (insert\ (case\ m\ x\ of\ None \Rightarrow empty \mid Some\ t' \Rightarrow t')\ (xs@xs'))\ xs$
unfolding ‹$t = PT\ m$›
by *simp*
then show *?case*
using *Cons.IH* **by** *auto*
qed

lemma *insert-isin-other* :
assumes *isin t xs*
shows $isin\ (insert\ t\ xs')\ xs$
proof $(cases\ xs = xs')$
case *True*
then show *?thesis* **using** *insert-isin-prefix*[*of t xs* []] **by** *simp*
next
case *False*

have $*$: $\bigwedge\ i\ xs\ xs'\ .\ take\ i\ xs = take\ i\ xs' \Longrightarrow take\ (Suc\ i)\ xs \neq take\ (Suc\ i)\ xs' \Longrightarrow isin\ t\ xs \Longrightarrow isin\ (insert\ t\ xs')\ xs$
proof −

```
  fix i xs xs' assume take i xs = take i xs'
                 and take (Suc i) xs ≠ take (Suc i) xs'
                 and isin t xs
  then show isin (insert t xs') xs
  proof (induction i arbitrary: xs xs' t)
    case 0
    then consider (a) xs = [] ∧ xs' ≠ [] |
                  (b) xs' = [] ∧ xs ≠ [] |
                  (c) xs ≠ [] ∧ xs' ≠ [] ∧ hd xs ≠ hd xs'
      by (metis take-Suc take-eq-Nil)
    then show ?case proof cases
      case a
      then show ?thesis by auto
    next
      case b
      then show ?thesis
        by (simp add: 0.prems(3))
    next
      case c
      then obtain b bs c cs where xs = b#bs and xs' = c#cs and b ≠ c
        using list.exhaust-sel by blast
      obtain m where t = PT m
        using prefix-tree.exhaust by auto
      have isin (Prefix-Tree.insert t xs') xs = isin t xs
        unfolding ‹t = PT m› ‹xs = b#bs› ‹xs' = c#cs› insert.simps isin.simps
using ‹b ≠ c›
        by simp
      then show ?thesis
        using ‹isin t xs› by simp
    qed
  next
    case (Suc i)

    define hxs where hxs: hxs = hd xs
    define txs where txs: txs = tl xs
    define txs' where txs': txs' = tl xs'

    have xs = hxs#txs
      unfolding hxs txs
       using ‹take (Suc i) xs = take (Suc i) xs'› ‹take (Suc (Suc i)) xs ≠ take
(Suc (Suc i)) xs'›
      by (metis Zero-not-Suc hd-Cons-tl take-eq-Nil)
    moreover have xs' = hxs#txs'
      unfolding hxs txs txs'
       using ‹take (Suc i) xs = take (Suc i) xs'› ‹take (Suc (Suc i)) xs ≠ take
(Suc (Suc i)) xs'›
      by (metis hd-Cons-tl hd-take take-Nil take-Suc-Cons take-tl zero-less-Suc)
    ultimately have take (Suc i) txs ≠ take (Suc i) txs'
       using ‹take (Suc (Suc i)) xs ≠ take (Suc (Suc i)) xs'›
```

```
        by (metis take-Suc-Cons)
      moreover have take i txs = take i txs'
        using ‹take (Suc i) xs = take (Suc i) xs'› unfolding txs txs'
        by (simp add: take-tl)
      ultimately have ⋀ t . isin t txs ⟹ isin (Prefix-Tree.insert t txs') txs
        using Suc.IH by blast

      obtain m where t = PT m
        using prefix-tree.exhaust by auto

      obtain t' where m hxs = Some t'
                and isin t' txs
        using case-optionE by (metis Suc.prems(3) ‹t = PT m› ‹xs = hxs # txs›
isin.simps(2))

      have isin (Prefix-Tree.insert t xs') xs = isin (Prefix-Tree.insert t' txs') txs
         using ‹m hxs = Some t'› unfolding ‹t = PT m› ‹xs = hxs#txs› ‹xs' =
hxs#txs'› by auto
      then show ?case
        using ‹⋀ t . isin t txs ⟹ isin (Prefix-Tree.insert t txs') txs› ‹isin t' txs›
        by simp
    qed
  qed

  show ?thesis
    using different-lists-shared-prefix[OF False] *[OF - - assms] by blast
qed


lemma insert-isin-rev :
  assumes isin (insert t xs') xs
shows isin t xs ∨ (∃ xs'' . xs' = xs@xs'')
proof (cases xs = xs')
  case True
  then show ?thesis using insert-isin-prefix[of t xs []] by simp
next
  case False

  have *: ⋀ i xs xs' . take i xs = take i xs' ⟹ take (Suc i) xs ≠ take (Suc i) xs'
⟹ isin (insert t xs') xs ⟹ isin t xs ∨ (∃ xs'' . xs' = xs@xs'')
  proof −
    fix i xs xs' assume take i xs = take i xs'
                  and take (Suc i) xs ≠ take (Suc i) xs'
                  and isin (insert t xs') xs
    then show isin t xs ∨ (∃ xs'' . xs' = xs@xs'')
    proof (induction i arbitrary: xs xs' t)
      case 0
      then consider (a) xs = [] ∧ xs' ≠ [] |
                    (b) xs' = [] ∧ xs ≠ [] |
```

```
                (c) xs ≠ [] ∧ xs' ≠ [] ∧ hd xs ≠ hd xs'
      by (metis take-Suc take-eq-Nil)
    then show ?case proof cases
      case a
      then show ?thesis
        by (metis isin.simps(1) )
    next
      case b
      then show ?thesis
        using 0.prems(3) by auto
    next
      case c
      then obtain b bs c cs where xs = b#bs and xs' = c#cs and b ≠ c
        using list.exhaust-sel by blast
      obtain m where t = PT m
        using prefix-tree.exhaust by auto
      have isin (Prefix-Tree.insert t xs') xs = isin t xs
        unfolding ‹t = PT m› ‹xs = b#bs› ‹xs' = c#cs› insert.simps isin.simps
using ‹b ≠ c›
        by simp
      then show ?thesis
        using ‹isin (insert t xs') xs› by simp
    qed
  next
    case (Suc i)

    define hxs where hxs: hxs = hd xs
    define txs where txs: txs = tl xs
    define txs' where txs': txs' = tl xs'

    have xs = hxs#txs
      unfolding hxs txs
       using ‹take (Suc i) xs = take (Suc i) xs'› ‹take (Suc (Suc i)) xs ≠ take
(Suc (Suc i)) xs'›
      by (metis Zero-not-Suc hd-Cons-tl take-eq-Nil)
    moreover have xs' = hxs#txs'
      unfolding hxs txs txs'
       using ‹take (Suc i) xs = take (Suc i) xs'› ‹take (Suc (Suc i)) xs ≠ take
(Suc (Suc i)) xs'›
      by (metis hd-Cons-tl hd-take take-Nil take-Suc-Cons take-tl zero-less-Suc)
    ultimately have take (Suc i) txs ≠ take (Suc i) txs'
       using ‹take (Suc (Suc i)) xs ≠ take (Suc (Suc i)) xs'›
       by (metis take-Suc-Cons)
    moreover have take i txs = take i txs'
      using ‹take (Suc i) xs = take (Suc i) xs'› unfolding txs txs'
      by (simp add: take-tl)
     ultimately have ⋀ t . isin (Prefix-Tree.insert t txs') txs ⟹ isin t txs ∨
(∃ xs''. txs' = txs @ xs'')
      using Suc.IH by blast
```

```
    obtain m where t = PT m
      using prefix-tree.exhaust by auto

    obtain t' where (m(hxs ↦ insert (case m hxs of None ⇒ empty | Some t'
⇒ t') txs')) hxs = Some t'
                and isin t' txs
      using case-optionE ‹isin (Prefix-Tree.insert t xs') xs›
        unfolding ‹t = PT m› ‹xs = hxs#txs› ‹xs' = hxs#txs'› insert.simps
isin.simps by blast
    then have t' = insert (case m hxs of None ⇒ empty | Some t' ⇒ t') txs'
      by auto
    then have ∗: isin (case m hxs of None ⇒ empty | Some t' ⇒ t') txs ∨ (∃ xs''.
txs' = txs @ xs'')
      using ‹⋀ t . isin (Prefix-Tree.insert t txs') txs ⟹ isin t txs ∨ (∃ xs''. txs'
= txs @ xs'')›
            ‹isin t' txs›
      by auto

    show ?case proof (cases m hxs)
      case None
      then have isin empty txs ∨ (∃ xs''. txs' = txs @ xs'')
        using ∗ by auto
      then have txs = [] ∨ (∃ xs''. txs' = txs @ xs'')
          by (metis Prefix-Tree.empty-def case-optionE isin.elims(2) option.discI
prefix-tree.inject)
      then have (∃ xs''. txs' = txs @ xs'')
        by auto
      then show ?thesis
        unfolding ‹xs = hxs#txs› ‹xs' = hxs#txs'› by auto
    next
      case (Some t'')
      then consider isin t'' txs | (∃ xs''. txs' = txs @ xs'')
        using ∗ by auto
      then show ?thesis proof cases
        case 1
        moreover have isin t xs = isin t'' txs
          unfolding ‹t = PT m› ‹xs = hxs#txs› ‹xs' = hxs#txs'› using Some by
auto
        ultimately show ?thesis by simp
      next
        case 2
        then show ?thesis
          unfolding ‹xs = hxs#txs› ‹xs' = hxs#txs'› by auto
      qed
    qed
  qed
qed
```

```
  show ?thesis
    using different-lists-shared-prefix[OF False] *[OF - - assms] by blast
qed


lemma insert-set : set (insert t xs) = set t ∪ {xs' . ∃ xs'' . xs = xs'@xs''}
proof −
  have set t ⊆ set (insert t xs)
    using insert-isin-other by auto
  moreover have {xs' . ∃ xs'' . xs = xs'@xs''} ⊆ set (insert t xs)
    using insert-isin-prefix
    by auto
  moreover have set (insert t xs) ⊆ set t ∪ {xs' . ∃ xs'' . xs = xs'@xs''}
    using insert-isin-rev[of t xs] unfolding set.simps by blast
  ultimately show ?thesis
    by blast
qed

lemma insert-isin : xs ∈ set (insert t xs)
  unfolding insert-set by auto

lemma set-prefix :
  assumes xs@ys ∈ set T
  shows xs ∈ set T
  using assms isin-prefix by auto


fun after :: 'a prefix-tree ⇒ 'a list ⇒ 'a prefix-tree where
  after t [] = t |
  after (PT m) (x # xs) = (case m x of None ⇒ empty | Some t ⇒ after t xs)

lemma after-set : set (after t xs) = Set.insert [] {xs' . xs@xs' ∈ set t}
  (is ?A t xs = ?B t xs)
proof
  show ?A t xs ⊆ ?B t xs
  proof
    fix xs' assume xs' ∈ ?A t xs
    then show xs' ∈ ?B t xs
    proof (induction xs arbitrary: t)
      case Nil
      then show ?case by auto
    next
      case (Cons x xs)
      obtain m where t = PT m
        using prefix-tree.exhaust by auto
      show ?case proof (cases m x)
        case None
```

```
      then have after t (x#xs) = empty
        unfolding ‹t = PT m› by auto
      then have xs' = []
        using Cons.prems set-empty by auto
      then show ?thesis by blast
    next
      case (Some t')
      then have after t (x#xs) = after t' xs
        unfolding ‹t = PT m› by auto
      then have xs' ∈ set (after t' xs)
        using Cons.prems by simp
      then have xs' ∈ ?B t' xs
        using Cons.IH by auto

      show ?thesis proof (cases xs' = [])
        case True
        then show ?thesis by auto
      next
        case False
        then have isin t' (xs@xs')
          using ‹xs' ∈ ?B t' xs› by auto
        then have isin t (x#(xs@xs'))
          unfolding ‹t = PT m› using Some by auto
        then show ?thesis by auto
      qed
    qed
  qed
qed

show ?B t xs ⊆ ?A t xs
proof
  fix xs' assume xs' ∈ ?B t xs
  then show xs' ∈ ?A t xs
  proof (induction xs arbitrary: t)
    case Nil
    then show ?case by (cases xs'; auto)
  next
    case (Cons x xs)
    obtain m where t = PT m
      using prefix-tree.exhaust by auto

    show ?case proof (cases xs' = [])
      case True
      then show ?thesis by (cases xs'; auto)
    next
      case False
      then have x # (xs @ xs') ∈ set t
        using Cons.prems by auto
      then have isin t (x # (xs @ xs'))
```

```
        by auto
      then obtain t' where m x = Some t'
                    and isin t' (xs@xs')
        unfolding ‹t = PT m›
        by (metis case-optionE isin.simps(2))
      then have xs' ∈ ?B t' xs
        by auto
      then have xs' ∈ ?A t' xs
        using Cons.IH by blast
      moreover have after t (x#xs) = after t' xs
        using ‹m x = Some t'› unfolding ‹t = PT m› by auto
      ultimately show ?thesis
        by simp
    qed
  qed
 qed
qed

lemma after-set-Cons :
 assumes γ ∈ set (after T α)
 and     γ ≠ []
shows α ∈ set T
 using assms unfolding after-set
 by (metis insertE isin-prefix mem-Collect-eq set.simps)


function (domintros) combine :: 'a prefix-tree ⇒ 'a prefix-tree ⇒ 'a prefix-tree
where
 combine (PT m1) (PT m2) = (PT (λ x . case m1 x of
   None ⇒ m2 x |
   Some t1 ⇒ (case m2 x of
     None ⇒ Some t1 |
     Some t2 ⇒ Some (combine t1 t2))))
 by pat-completeness auto
termination
proof −
 {
   fix a b :: 'a prefix-tree

   have combine-dom (a,b)
   proof (induction a arbitrary: b)
     case (PT m1)

     obtain m2 where b = PT m2
       by (metis prefix-tree.exhaust)

     have (⋀x a' b'. m1 x = Some a' ⟹ m2 x = Some b' ⟹ combine-dom (a',
b'))
     proof −
```

```
      fix x a′ b′ assume m1 x = Some a′ and m2 x = Some b′

      have Some a′ ∈ range m1
        by (metis ‹m1 x = Some a′› range-eqI)

      show combine-dom (a′, b′)
        using PT(1)[OF ‹Some a′ ∈ range m1›, of a′]
        by simp
    qed

    then show ?case
      using combine.domintros unfolding ‹b = PT m2› by blast
  qed
} note t = this

then show ?thesis by auto
qed

lemma combine-alt-def :
  combine (PT m1) (PT m2) = PT (λx . combine-options combine (m1 x) (m2
x))
  unfolding combine.simps
  by (simp add: combine-options-def)


lemma combine-set :
  set (combine t1 t2) = set t1 ∪ set t2
proof

  show set (combine t1 t2) ⊆ set t1 ∪ set t2
  proof
    fix xs assume xs ∈ set (combine t1 t2)
    then show xs ∈ set t1 ∪ set t2
    proof (induction xs arbitrary: t1 t2)
      case Nil
      show ?case
        using set-Nil by auto
    next
      case (Cons x xs)

      obtain m1 m2 where t1 = PT m1 and t2 = PT m2
        by (meson prefix-tree.exhaust)

      obtain t′ where combine-options combine (m1 x) (m2 x) = Some t′
                 and isin t′ xs
       using Cons.prems unfolding ‹t1 = PT m1› ‹t2 = PT m2› combine-alt-def
set.simps
        by (metis (no-types, lifting) case-optionE isin.simps(2) mem-Collect-eq)
```

```
    show ?case proof (cases m1 x)
      case None
      show ?thesis proof (cases m2 x)
        case None
        then have False
          using ‹m1 x = None› ‹combine-options combine (m1 x) (m2 x) = Some
t'›
          by simp
        then show ?thesis
          by simp
      next
        case (Some t'')
        then have m2 x = Some t'
          using ‹m1 x = None› ‹combine-options combine (m1 x) (m2 x) = Some
t'›
          by simp
        then have isin t2 (x#xs)
          using ‹isin t' xs› unfolding ‹t2 = PT m2› by auto
        then show ?thesis
          by simp
      qed
    next
      case (Some t1')
      show ?thesis proof (cases m2 x)
        case None
        then have m1 x = Some t'
           using ‹m1 x = Some t1'› ‹combine-options combine (m1 x) (m2 x) =
Some t'›
          by simp
        then have isin t1 (x#xs)
          using ‹isin t' xs› unfolding ‹t1 = PT m1› by auto
        then show ?thesis
          by simp
      next
        case (Some t2')
        then have t' = combine t1' t2'
           using ‹m1 x = Some t1'› ‹combine-options combine (m1 x) (m2 x) =
Some t'›
          by simp
        then have xs ∈ Prefix-Tree.set (combine t1' t2')
          using ‹isin t' xs›
          by simp
        then have xs ∈ Prefix-Tree.set t1' ∪ Prefix-Tree.set t2'
          using Cons.IH by blast
        then have isin t1' xs ∨ isin t2' xs
          by simp
        then have isin t1 (x#xs) ∨ isin t2 (x#xs)
          using ‹m1 x = Some t1'› ‹m2 x = Some t2'› unfolding ‹t1 = PT m1›
‹t2 = PT m2› by auto
```

```
        then show ?thesis
          by simp
      qed
    qed
  qed
qed

show (set t1 ∪ set t2) ⊆ set (combine t1 t2)
proof −
  have set t1 ⊆ set (combine t1 t2)
  proof
    fix xs assume xs ∈ set t1
    then have isin t1 xs
      by auto
    then show xs ∈ set (combine t1 t2)
    proof (induction xs arbitrary: t1 t2)
      case Nil
      then show ?case using set-Nil by auto
    next
      case (Cons x xs)

      obtain m1 m2 where t1 = PT m1 and t2 = PT m2
        by (meson prefix-tree.exhaust)

      obtain t1' where m1 x = Some t1'
                  and isin t1' xs
        using Cons.prems unfolding ‹t1 = PT m1› isin.simps
        using case-optionE by blast

      show ?case proof (cases m2 x)
        case None
        then have combine-options combine (m1 x) (m2 x) = Some t1'
          by (simp add: ‹m1 x = Some t1'›)
        then have isin (combine t1 t2) (x#xs)
          using combine-alt-def
           by (metis (no-types, lifting) Cons.prems ‹m1 x = Some t1'› ‹t1 = PT
m1› ‹t2 = PT m2› isin.simps(2))
        then show ?thesis
          by simp
      next
        case (Some t2')
        then have combine-options combine (m1 x) (m2 x) = Some (combine t1'
t2')
          by (simp add: ‹m1 x = Some t1'›)
        moreover have isin (combine t1' t2') xs
          using Cons.IH[OF ‹isin t1' xs›]
          by simp
        ultimately have isin (combine t1 t2) (x#xs)
          unfolding ‹t1 = PT m1› ‹t2 = PT m2› using isin.simps(2)[of - x xs]
```

```
        by (metis (no-types, lifting) combine-alt-def option.simps(5))
      then show ?thesis by simp
    qed
  qed
qed
moreover have set t2 ⊆ set (combine t1 t2)
proof
  fix xs assume xs ∈ set t2
  then have isin t2 xs
    by auto
  then show xs ∈ set (combine t1 t2)
  proof (induction xs arbitrary: t1 t2)
    case Nil
    then show ?case using set-Nil by auto
  next
    case (Cons x xs)

    obtain m1 m2 where t1 = PT m1 and t2 = PT m2
      by (meson prefix-tree.exhaust)

    obtain t2' where m2 x = Some t2'
              and isin t2' xs
      using Cons.prems unfolding ‹t2 = PT m2› isin.simps
      using case-optionE by blast

    show ?case proof (cases m1 x)
      case None
      then have combine-options combine (m1 x) (m2 x) = Some t2'
        by (simp add: ‹m2 x = Some t2'›)
      then have isin (combine t1 t2) (x#xs)
        using combine-alt-def
         by (metis (no-types, lifting) Cons.prems ‹m2 x = Some t2'› ‹t1 = PT
m1› ‹t2 = PT m2› isin.simps(2))
      then show ?thesis
        by simp
    next
      case (Some t1')
      then have combine-options combine (m1 x) (m2 x) = Some (combine t1'
t2')
        by (simp add: ‹m2 x = Some t2'›)
      moreover have isin (combine t1' t2') xs
        using Cons.IH[OF ‹isin t2' xs›]
        by simp
      ultimately have isin (combine t1 t2) (x#xs)
        unfolding ‹t1 = PT m1› ‹t2 = PT m2› using isin.simps(2)[of - x xs]
        by (metis (no-types, lifting) combine-alt-def option.simps(5))
      then show ?thesis by simp
    qed
  qed
```

```
    qed
    ultimately show ?thesis
      by blast
  qed
qed
```

```
fun combine-after :: 'a prefix-tree ⇒ 'a list ⇒ 'a prefix-tree ⇒ 'a prefix-tree where
  combine-after t1 [] t2 = combine t1 t2 |
  combine-after (PT m) (x#xs) t2 = PT (m(x ↦ combine-after (case m x of None
⇒ empty | Some t' ⇒ t') xs t2))

lemma combine-after-set : set (combine-after t1 xs t2) = set t1 ∪ {xs' . ∃ xs'' .
xs = xs'@xs''} ∪ {xs@xs' | xs' . xs' ∈ set t2}
proof
  show set (combine-after t1 xs t2) ⊆ set t1 ∪ {xs' . ∃ xs'' . xs = xs'@xs''} ∪
{xs@xs' | xs' . xs' ∈ set t2}
  proof
    fix ys assume ys ∈ set (combine-after t1 xs t2)
    then show ys ∈ set t1 ∪ {xs' . ∃ xs'' . xs = xs'@xs''} ∪ {xs@xs' | xs' . xs' ∈
set t2}
    proof (induction ys arbitrary: xs t1)
      case Nil
      show ?case using set-Nil by auto
    next
      case (Cons y ys)

      obtain m1 where t1 = PT m1
        by (meson prefix-tree.exhaust)

      show ?case proof (cases xs)
        case Nil
        then show ?thesis using combine-set Cons.prems by auto
      next
        case (Cons x xs')

        show ?thesis proof (cases x = y)
          case True
          then have isin (combine-after t1 (x#xs') t2) (x#ys)
            using Cons Cons.prems by auto
          then have isin (combine-after (case m1 x of None ⇒ empty | Some t' ⇒
t') xs' t2) ys
            unfolding ‹t1 = PT m1› by auto
          then consider ys ∈ set (case m1 x of None ⇒ empty | Some t' ⇒ t') |
ys ∈ {xs'' . ∃ xs''' . xs' = xs''@xs'''} | ys ∈ {xs' @ xs'' |xs''. xs'' ∈ set t2}
            using Cons.IH by auto
          then show ?thesis proof cases
```

```
          case 1
          then show ?thesis proof (cases m1 x)
            case None
            then have ys = []
              using 1 set-empty by auto
            then show ?thesis unfolding True Cons by auto
          next
            case (Some t')
            then have isin t' ys
              using 1 by auto
            then have y # ys ∈ Prefix-Tree.set (PT m1)
              using Some by (simp add: True)
            then show ?thesis unfolding ‹t1 = PT m1› by auto
          qed
        next
          case 2
          then show ?thesis unfolding True ‹t1 = PT m1› Cons by auto
        next
          case 3
          then show ?thesis unfolding True ‹t1 = PT m1› Cons by auto
        qed
      next
        case False
        then have (m1(x ↦ combine-after (case m1 x of None ⇒ empty | Some
t' ⇒ t') xs' t2)) y = m1 y
          by auto
        then have isin t1 (y#ys)
          using Cons Cons.prems unfolding ‹t1 = PT m1›
          by simp
        then show ?thesis by auto
      qed
    qed
  qed
qed

show set t1 ∪ {xs' . ∃ xs'' . xs = xs'@xs''} ∪ {xs@xs' | xs' . xs' ∈ set t2} ⊆ set
(combine-after t1 xs t2)
proof −
  have set t1 ⊆ set (combine-after t1 xs t2)
  proof
    fix ys assume ys ∈ set t1
    then show ys ∈ set (combine-after t1 xs t2)
    proof (induction ys arbitrary: t1 xs)
      case Nil
      then show ?case using set-Nil by auto
    next
      case (Cons y ys)
      then have isin t1 (y#ys)
        by auto
```

```
      show ?case proof (cases xs)
        case Nil
        then show ?thesis using Cons.prems combine-set by auto
      next
        case (Cons x xs')

        obtain m1 where t1 = PT m1
          by (meson prefix-tree.exhaust)
        obtain t' where m1 y = Some t'
                  and isin t' ys
          using ‹isin t1 (y#ys)› unfolding ‹t1 = PT m1› isin.simps
          using case-optionE by blast
        then have ys ∈ set t'
          by auto
        then have isin (combine-after t' xs' t2) ys
          using Cons.IH by auto

        show ?thesis proof (cases x=y)
          case True
          show ?thesis
            using ‹isin (combine-after t' xs' t2) ys› ‹m1 y = Some t'›
            unfolding Cons True ‹t1 = PT m1› by auto
        next
          case False
           then have isin (combine-after (PT m1) (x # xs') t2) (y#ys) = isin
(PT m1) (y#ys)
            unfolding combine-after.simps by auto
          then show ?thesis
            using ‹y # ys ∈ Prefix-Tree.set t1›
            unfolding Cons ‹t1 = PT m1›
            by auto
        qed
      qed
    qed
  qed
  moreover have {xs' . ∃ xs'' . xs = xs'@xs''} ∪ {xs@xs' | xs' . xs' ∈ set t2} ⊆
set (combine-after t1 xs t2)
  proof −
    have {xs@xs' | xs' . xs' ∈ set t2} ⊆ set (combine-after t1 xs t2) ⟹ {xs' . ∃
xs'' . xs = xs'@xs''} ⊆ set (combine-after t1 xs t2)
    proof
      fix ys assume *:{xs@xs' | xs' . xs' ∈ set t2} ⊆ set (combine-after t1 xs t2)
              and ys ∈ {xs' . ∃ xs'' . xs = xs'@xs''}
      then obtain xs' where xs = ys@xs'
        by blast
      then have **: isin (combine-after t1 xs t2) (ys@xs')
        using * set-Nil[of t2] by force
      show ys ∈ set (combine-after t1 xs t2)
```

```
        using isin-prefix[OF **] by auto
      qed
      moreover have {xs@xs' | xs' . xs' ∈ set t2} ⊆ set (combine-after t1 xs t2)
      proof
        fix ys assume ys ∈ {xs@xs' | xs' . xs' ∈ set t2}
        then obtain xs' where ys = xs@xs' and xs' ∈ set t2
          by auto



        show ys ∈ set (combine-after t1 xs t2)
          unfolding ‹ys = xs@xs'›
        proof (induction xs arbitrary: t1)
          case Nil
          then show ?case using combine-set ‹xs' ∈ set t2› by auto
        next
          case (Cons x xs)

          obtain m1 where t1 = PT m1
            by (meson prefix-tree.exhaust)

            have isin (combine-after t1 (x # xs) t2) ((x # xs) @ xs') = isin
(combine-after (case m1 x of None ⇒ empty | Some t' ⇒ t') xs t2) (xs @ xs')
            unfolding ‹t1 = PT m1› by auto
          then have *:(x # xs) @ xs' ∈ Prefix-Tree.set (combine-after t1 (x # xs)
t2) = isin (combine-after (case m1 x of None ⇒ empty | Some t' ⇒ t') xs t2) (xs
@ xs')
            by auto

          show ?case
            using ‹xs' ∈ set t2› Cons
            unfolding *
            by (cases m1 x; simp)
        qed
      qed
      ultimately show ?thesis
        by blast
    qed
    ultimately show ?thesis
      by blast
  qed
qed


fun from-list :: 'a list list ⇒ 'a prefix-tree where
  from-list xs = foldr (λ x t . insert t x) xs empty

lemma from-list-set : set (from-list xs) = Set.insert [] {xs'' . ∃ xs' xs''' . xs' ∈
list.set xs ∧ xs' = xs''@xs'''}
```

proof (*induction xs*)
  case *Nil*
  have *from-list* [] = *empty*
    by *auto*
  then have *set* (*from-list* []) = {[]}
    using *set-empty* **by** *auto*
  moreover have *Set.insert* [] {*xs''* . ∃ *xs' xs'''* . *xs'* ∈ *list.set* [] ∧ *xs'* = *xs''*@*xs'''*}
= {[]}
    by *auto*
  ultimately show *?case*
    by *blast*
next
  case (*Cons x xs*)

  have *from-list* (*x*#*xs*) = *insert* (*from-list xs*) *x*
    by *auto*
  then have *set* (*from-list* (*x*#*xs*)) = *set* (*from-list xs*) ∪ {*xs'*. ∃*xs''*. *x* = *xs'* @
xs''}
    using *insert-set* **by** *auto*
  then show *?case*
    unfolding *Cons* **by** *force*
qed

lemma *from-list-subset* : *list.set xs* ⊆ *set* (*from-list xs*)
  unfolding *from-list-set* **by** *auto*

lemma *from-list-set-elem* :
  assumes *x* ∈ *list.set xs*
  shows *x* ∈ *set* (*from-list xs*)
  using *assms* **unfolding** *from-list-set* **by** *force*

function (*domintros*) *finite-tree* :: *'a prefix-tree* ⇒ *bool* **where**
  finite-tree (*PT m*) = (*finite* (*dom m*) ∧ (∀ *t* ∈ *ran m* . *finite-tree t*))
  by *pat-completeness auto*
termination
proof −
  { fix *a* :: *'a prefix-tree*

    have *finite-tree-dom a*
    proof (*induction a*)
      case (*PT m*)

      have (⋀*x*. *x* ∈ *ran m* ⟹ *finite-tree-dom x*)
      proof −
        fix *x* :: *'a prefix-tree*
        assume *x* ∈ *ran m*
        then have ∃*a*. *m a* = *Some x*
          by (*simp add*: *ran-def*)
        then show *finite-tree-dom x*

```
        using PT.IH by blast
      qed
      then show ?case
        using finite-tree.domintros
        by blast
    qed
  }
  then show ?thesis by auto
qed

lemma combine-after-after-subset :
  set T2 ⊆ set (after (combine-after T1 xs T2) xs)
  unfolding combine-after-set after-set
  by auto

lemma subset-after-subset :
  set T2 ⊆ set T1 ⟹ set (after T2 xs) ⊆ set (after T1 xs)
  unfolding after-set by auto

lemma set-alt-def :
  set (PT m) = Set.insert [] (⋃ x ∈ dom m . (Cons x) ‘ (set (the (m x))))
  (is ?A m = ?B m)
proof
  show ?A m ⊆ ?B m
  proof
    fix xs assume xs ∈ ?A m
    then have isin (PT m) xs
      by auto
    then show xs ∈ ?B m
    proof (induction xs arbitrary: m)
      case Nil
      then show ?case by auto
    next
      case (Cons x xs)
      then obtain t where m x = Some t
                    and isin t xs
        by (metis (no-types, lifting) case-optionE isin.simps(2))

      obtain m′ where t = PT m′
        using prefix-tree.exhaust by blast
      then have xs ∈ ?B m′
        using ‹isin t xs› Cons.IH by blast
      moreover have x ∈ dom m
        using ‹m x = Some t›
        by auto
      ultimately show ?case
        using ‹m x = Some t›
        using ‹isin t xs› ‹t = PT m′›
        by fastforce
```

```
    qed
  qed

  show ?B m ⊆ ?A m
  proof
    fix xs assume xs ∈ ?B m
    then show xs ∈ ?A m
    proof (induction xs arbitrary: m)
      case Nil
      show ?case
        by auto
    next
      case (Cons x xs)
      then have x#xs ∈ (⋃ x ∈ dom m . (Cons x) ‘ (set (the (m x))))
        by auto
      then have x ∈ dom m
            and xs ∈ (set (the (m x)))
        by auto
      then obtain t where m x = Some t and isin t xs
        unfolding keys-is-none-rep
        by auto
      then show ?case
        by auto
    qed
  qed
qed



lemma finite-tree-iff :
  finite-tree t = finite (set t)
  (is ?P1 = ?P2)
proof
  show ?P1 ⟹ ?P2
  proof induction
    case (PT m)

    have set (PT m) = Set.insert [] (⋃x∈dom m. (#) x ‘ set (the (m x)))
      unfolding set-alt-def by simp
    moreover have finite (dom m)
      using PT.prems by auto
    moreover have ⋀ x . x ∈ dom m ⟹ finite ((#) x ‘ set (the (m x)))
    proof −
      fix x assume x ∈ dom m
      then obtain y where m x = Some y
        by auto
      then have y ∈ ran m
        by (meson ranI)
      then have finite-tree y
```

```
      using PT.prems by auto
    then have finite (set y)
      using PT.IH[of Some y y] ‹m x = Some y›
      by (metis option.set-intros rangeI)
    moreover have (the (m x)) = y
      using ‹m x = Some y› by auto
    ultimately show finite ((#) x ‘ set (the (m x)))
      by blast
  qed
  ultimately show ?case
    by simp
qed

show ?P2 ⟹ ?P1
proof (induction t)
  case (PT m)

  have finite (dom m)
  proof −
    have ⋀ x . x ∈ dom m ⟹ [x] ∈ set (PT m)
      using image-eqI by auto
    then have (λx . [x]) ‘ dom m ⊆ set (PT m)
      by auto
    have inj (λx . [x])
      by (meson inj-onI list.inject)
    show ?thesis
     by (meson PT.prems UNIV-I ‹(λx. [x]) ‘ dom m ⊆ Prefix-Tree.set (PT m)›
‹inj (λx. [x])› inj-on-finite inj-on-subset subsetI)
  qed
  moreover have ⋀ t . t ∈ ran m ⟹ finite-tree t
  proof −
    fix t assume t ∈ ran m
    then obtain x where m x = Some t
      unfolding ran-def by blast
    then have (#) x ‘ set t ⊆ set (PT m)
      unfolding set-alt-def
      by auto
    then have finite ((#) x ‘ set t)
      using PT.prems
      by (simp add: finite-subset)
    moreover have inj ((#) x)
      by auto
    ultimately have finite (set t)
      by (simp add: finite-image-iff)
    then show finite-tree t
      using PT.IH[of Some t t] ‹m x = Some t›
      by (metis option.set-intros rangeI)
  qed
  ultimately show ?case
```

```
    by simp
  qed
qed

lemma empty-finite-tree :
  finite-tree empty
  unfolding finite-tree-iff set-empty by auto

lemma insert-finite-tree :
  assumes finite-tree t
  shows finite-tree (insert t xs)
proof −
  have {xs'. ∃ xs''. xs = xs' @ xs''} = list.set (prefixes xs)
    unfolding prefixes-set by blast
  then have finite {xs'. ∃ xs''. xs = xs' @ xs''}
    using List.finite-set by simp
  then show ?thesis
    using assms unfolding finite-tree-iff insert-set
    by blast
qed

lemma from-list-finite-tree :
  finite-tree (from-list xs)
  using insert-finite-tree empty-finite-tree by (induction xs; auto)

lemma combine-after-finite-tree :
  assumes finite-tree t1
  and     finite-tree t2
shows finite-tree (combine-after t1 α t2)
proof −
  have finite (Prefix-Tree.set t2) and finite (Prefix-Tree.set t1)
    using assms unfolding finite-tree-iff by auto
  then have finite (Prefix-Tree.set (Prefix-Tree.insert t1 α) ∪ {α @ as |as. as ∈
Prefix-Tree.set t2})
    using finite-tree-iff insert-finite-tree by fastforce
  then show ?thesis
    unfolding finite-tree-iff combine-after-set
    by (metis insert-set)
qed

lemma combine-finite-tree :
  assumes finite-tree t1
  and     finite-tree t2
shows finite-tree (combine t1 t2)
  using assms unfolding finite-tree-iff combine-set
  by blast


function (domintros) sorted-list-of-maximal-sequences-in-tree :: ('a :: linorder) pre-
```

fix-tree ⇒ *'a list list* **where**
sorted-list-of-maximal-sequences-in-tree (*PT m*) =
(*if dom m* = {}
then [[]]
else concat (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-maximal-sequences-in-tree* (*the* (*m k*)))) (*sorted-list-of-set* (*dom m*))))
by *pat-completeness auto*
termination
proof −
{ fix *a* :: *'a prefix-tree*

have *sorted-list-of-maximal-sequences-in-tree-dom a*
proof (*induction a*)
case (*PT m*)
then show *?case*
by (*metis List.set-empty domIff empty-iff option.set-sel range-eqI set-sorted-list-of-set sorted-list-of-maximal-sequences-in-tree.domintros sorted-list-of-set.fold-insort-key.infinite*)
qed
}
then show *?thesis* **by** *auto*
qed

lemma *sorted-list-of-maximal-sequences-in-tree-Nil* :
assumes [] ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree t*)
shows *t* = *empty*
proof −
obtain *m* **where** *t* = *PT m*
using *prefix-tree.exhaust* **by** *blast*

show *?thesis* **proof** (*cases dom m* = {})
case *True*
then have *m* = *Map.empty*
using *True* **by** *blast*
then show *?thesis*
unfolding ‹*t* = *PT m*›
by (*simp add*: *Prefix-Tree.empty-def*)
next
case *False*
then have [] ∈ *list.set* (*concat* (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-maximal-sequences-in-tree* (*the* (*m k*)))) (*sorted-list-of-set* (*dom m*))))
using *assms* **unfolding** ‹*t* = *PT m*› **by** *auto*
then show *?thesis*
by *auto*
qed
qed

lemma *sorted-list-of-maximal-sequences-in-tree-set* :
assumes *finite-tree t*

```
  shows list.set (sorted-list-of-maximal-sequences-in-tree t) = {y. y ∈ set t ∧ ¬(∃
y' . y' ≠ [] ∧ y@y' ∈ set t)}
    (is ?S1 = ?S2)
proof
  show ?S1 ⊆ ?S2
  proof
    fix xs assume xs ∈ ?S1
    then show xs ∈ ?S2
    proof (induction xs arbitrary: t)
      case Nil
      then have t = empty
        using sorted-list-of-maximal-sequences-in-tree-Nil by auto
      then show ?case
        using set-empty by auto
    next
      case (Cons x xs)

      obtain m where t = PT m
        using prefix-tree.exhaust by blast
     have x#xs ∈ list.set (concat (map (λk . map ((#) k) (sorted-list-of-maximal-sequences-in-tree
(the (m k)))) (sorted-list-of-set (dom m))))
          by (metis (no-types) Cons.prems(1) ‹t = PT m› empty-iff list.set(1)
list.simps(3) set-ConsD sorted-list-of-maximal-sequences-in-tree.simps)
      then have x ∈ list.set (sorted-list-of-set (dom m))
           and xs ∈ list.set (sorted-list-of-maximal-sequences-in-tree (the (m x)))
        by auto

      have x ∈ dom m
        using ‹x ∈ list.set (sorted-list-of-set (dom m))› unfolding ‹t = PT m›
      by (metis equals0D list.set(1) sorted-list-of-set.fold-insort-key.infinite sorted-list-of-set.set-sorted-key-list-
      then obtain t' where m x = Some t'
        by auto
      then have xs ∈ list.set (sorted-list-of-maximal-sequences-in-tree t')
        using ‹xs ∈ list.set (sorted-list-of-maximal-sequences-in-tree (the (m x)))›
        by auto
      then have xs ∈ set t' and ¬(∃ y' . y' ≠ [] ∧ xs@y' ∈ set t')
        using Cons.IH by blast+

      have x#xs ∈ set t
        unfolding ‹t = PT m› using ‹xs ∈ set t'› ‹m x = Some t'› by auto
      moreover have ¬(∃ y' . y' ≠ [] ∧ (x#xs)@y' ∈ set t)
      proof
        assume ∃y'. y' ≠ [] ∧ (x # xs) @ y' ∈ Prefix-Tree.set t
        then obtain y' where y' ≠ [] and (x # xs) @ y' ∈ Prefix-Tree.set t
          by blast
        then have isin (PT m) (x # (xs @ y'))
          unfolding ‹t = PT m› by auto
        then have isin t' (xs @ y')
          using ‹m x = Some t'› by auto
```

```
      then have ∃ y' . y' ≠ [] ∧ xs@y' ∈ set t'
        using ‹y' ≠ []› by auto
      then show False
        using ‹¬(∃ y' . y' ≠ [] ∧ xs@y' ∈ set t')› by simp
    qed
    ultimately show ?case by blast
  qed
qed

show ?S2 ⊆ ?S1
proof
  fix xs assume xs ∈ ?S2
  then show xs ∈ ?S1
  using assms proof (induction xs arbitrary: t)
    case Nil
    then have set t = {[]}
      by auto
    moreover obtain m where t = PT m
      using prefix-tree.exhaust by blast
    ultimately have ⋀ x . ¬ isin (PT m) [x]
      by force
    moreover have ⋀ x . x ∈ dom m ⟹ isin (PT m) [x]
      by auto
    ultimately have dom m = {}
      by blast
    then show ?case
      unfolding ‹t = PT m› by auto
  next
    case (Cons x xs)

    obtain m where t = PT m
      using prefix-tree.exhaust by blast
    then have isin (PT m) (x#xs)
      using Cons.prems(1) by auto
    then obtain t' where m x = Some t'
                    and isin t' xs
      by (metis case-optionE isin.simps(2))
    then have x ∈ dom m
      by auto
    then have dom m ≠ {}
      by auto

    have finite-tree t'
      using ‹finite-tree t› ‹m x = Some t'› unfolding ‹t = PT m›
      by (meson finite-tree.simps ranI)
      moreover have xs ∈ {y ∈ Prefix-Tree.set t'. ∄y'. y' ≠ [] ∧ y @ y' ∈
Prefix-Tree.set t'}
    proof −
      have xs ∈ set t'
```

```
        using ‹isin t′ xs› by auto
      moreover have (∄ y′. y′ ≠ [] ∧ xs @ y′ ∈ Prefix-Tree.set t′)
      proof
        assume ∃ y′. y′ ≠ [] ∧ xs @ y′ ∈ Prefix-Tree.set t′
        then obtain y′ where y′ ≠ [] and xs @ y′ ∈ Prefix-Tree.set t′
          by blast
        then have isin t′ (xs@y′)
          by auto
        then have isin (PT m) (x#(xs@y′))
          using ‹m x = Some t′› by auto
        then show False
          using Cons.prems(1) ‹y′ ≠ []› unfolding ‹t = PT m› by auto
      qed
      ultimately show ?thesis
        by blast
    qed
    ultimately have xs ∈ list.set (sorted-list-of-maximal-sequences-in-tree t′)
      using Cons.IH by blast
    moreover have x ∈ list.set (sorted-list-of-set (dom m))
      using ‹x ∈ dom m› ‹finite-tree t› unfolding ‹t = PT m›
      by simp
    ultimately show ?case
     using ‹finite-tree t› ‹dom m ≠ {}› ‹m x = Some t′› unfolding ‹t = PT m›

      by force
  qed
 qed
qed


lemma sorted-list-of-maximal-sequences-in-tree-ob :
 assumes finite-tree T
 and     xs ∈ set T
obtains xs′ where xs@xs′ ∈ list.set (sorted-list-of-maximal-sequences-in-tree T)
proof −
 let ?xs = {xs@xs′ | xs′ . xs@xs′ ∈ set T}

 let ?xs′ = arg-max-on length ?xs

 have xs ∈ ?xs
   using assms(2) by auto
 then have ?xs ≠ {}
   by blast
 moreover have finite ?xs
   using finite-subset[of ?xs set T]
   using assms(1) unfolding finite-tree-iff
   by blast
 ultimately obtain xs′ where xs′ ∈ ?xs and ⋀ xs′′ . xs′′ ∈ ?xs ⟹ length xs′′
≤ length xs′
```

```
    using max-length-elem[of ?xs]
    by force

  obtain xs'' where xs' = xs@xs'' and xs@xs'' ∈ set T
    using ‹xs' ∈ ?xs› by auto
  have ⋀ xs''' . xs@xs''' ∈ set T ⟹ length xs''' ≤ length xs''
  proof −
    fix xs''' assume xs@xs''' ∈ set T
    then have xs@xs''' ∈ ?xs
      by auto
    then have length (xs@xs''')  ≤ length xs'
      using ‹⋀ xs'' . xs'' ∈ ?xs ⟹ length xs'' ≤ length xs'›
      by blast
    then show length xs''' ≤ length xs''
      unfolding ‹xs' = xs@xs''› by auto
  qed
  then have ¬(∃  y' . y' ≠ [] ∧ (xs@xs'')@y' ∈ set T)
    by fastforce
  then have xs@xs'' ∈ list.set (sorted-list-of-maximal-sequences-in-tree T)
    using ‹xs@xs'' ∈ set T›
    unfolding sorted-list-of-maximal-sequences-in-tree-set[OF assms(1)]
    by blast
  then show ?thesis using that by blast
qed


function (domintros) sorted-list-of-sequences-in-tree :: ('a :: linorder) prefix-tree
⇒ 'a list list where
  sorted-list-of-sequences-in-tree (PT m) =
    (if dom m = {}
      then [[]]
      else [] # concat (map (λk . map ((#) k) (sorted-list-of-sequences-in-tree (the
(m k)))) (sorted-list-of-set (dom m))))
  by pat-completeness auto
termination
proof −
  {
    fix a :: 'a prefix-tree

    have sorted-list-of-sequences-in-tree-dom a
    proof (induction a)
      case (PT m)
      then show ?case
      by (metis List.set-empty domIff emptyE option.set-sel rangeI sorted-list-of-sequences-in-tree.domintros
sorted-list-of-set.fold-insort-key.infinite sorted-list-of-set.set-sorted-key-list-of-set)
    qed
  }
  then show ?thesis by auto
qed
```

```
lemma sorted-list-of-sequences-in-tree-set :
  assumes finite-tree t
  shows list.set (sorted-list-of-sequences-in-tree t) = set t
    (is ?S1 = ?S2)
proof
  show ?S1 ⊆ ?S2
  proof
    fix xs assume xs ∈ ?S1
    then show xs ∈ ?S2
    proof (induction xs arbitrary: t)
      case Nil
      then show ?case
        using set-empty by auto
    next
      case (Cons x xs)

      obtain m where t = PT m
        using prefix-tree.exhaust by blast
     have x#xs ∈ list.set (concat (map (λk . map ((#) k) (sorted-list-of-sequences-in-tree
(the (m k)))) (sorted-list-of-set (dom m))))
          by (metis (no-types) Cons.prems(1) ‹t = PT m› empty-iff list.set(1)
list.simps(3) set-ConsD sorted-list-of-sequences-in-tree.simps)
      then have x ∈ list.set (sorted-list-of-set (dom m))
           and xs ∈ list.set (sorted-list-of-sequences-in-tree (the (m x)))
        by auto

      have x ∈ dom m
        using ‹x ∈ list.set (sorted-list-of-set (dom m))› unfolding ‹t = PT m›
      by (metis emptyE empty-set sorted-list-of-set.fold-insort-key.infinite sorted-list-of-set.set-sorted-key-list-of
      then obtain t′ where m x = Some t′
        by auto
      then have xs ∈ list.set (sorted-list-of-sequences-in-tree t′)
        using ‹xs ∈ list.set (sorted-list-of-sequences-in-tree (the (m x)))›
        by auto
      then have xs ∈ set t′
        using Cons.IH by blast+

      show x#xs ∈ set t
        unfolding ‹t = PT m› using ‹xs ∈ set t′› ‹m x = Some t′› by auto
    qed
  qed

  show ?S2 ⊆ ?S1
  proof
    fix xs assume xs ∈ ?S2
    then show xs ∈ ?S1
    using assms proof (induction xs arbitrary: t)
      case Nil
```

```
      obtain m where t = PT m
        using prefix-tree.exhaust by blast
      then show ?case
        by auto
    next
      case (Cons x xs)

      obtain m where t = PT m
        using prefix-tree.exhaust by blast
      then have isin (PT m) (x#xs)
        using Cons.prems(1) by auto
      then obtain t' where m x = Some t'
                    and isin t' xs
        by (metis case-optionE isin.simps(2))
      then have x ∈ dom m
        by auto
      then have dom m ≠ {}
        by auto

      have finite-tree t'
        using ‹finite-tree t› ‹m x = Some t'› unfolding ‹t = PT m›
        by (meson finite-tree.simps ranI)
      moreover have xs ∈ set t'
        using ‹isin t' xs› by auto
      ultimately have xs ∈ list.set (sorted-list-of-sequences-in-tree t')
        using Cons.IH by blast
      moreover have x ∈ list.set (sorted-list-of-set (dom m))
        using ‹x ∈ dom m› ‹finite-tree t› unfolding ‹t = PT m›
        by simp
      ultimately show ?case
       using ‹finite-tree t› ‹dom m ≠ {}› ‹m x = Some t'› unfolding ‹t = PT m›

        by force
    qed
  qed
qed
```

```
fun difference-list :: ('a::linorder) prefix-tree ⇒ 'a prefix-tree ⇒ 'a list list where
  difference-list t1 t2 = filter (λ xs . ¬ isin t2 xs) (sorted-list-of-sequences-in-tree
t1)

lemma difference-list-set :
  assumes finite-tree t1
shows List.set (difference-list t1 t2) = (set t1 − set t2)
  unfolding difference-list.simps
```

```
        filter-set[symmetric]
        sorted-list-of-sequences-in-tree-set[OF assms]
        set.simps
  by fastforce

fun is-leaf :: 'a prefix-tree ⇒ bool where
  is-leaf t = (t = empty)

fun is-maximal-in :: 'a prefix-tree ⇒ 'a list ⇒ bool where
  is-maximal-in T α = (isin T α ∧ is-leaf (after T α))

function (domintros) height :: 'a prefix-tree ⇒ nat where
  height (PT m) = (if (is-leaf (PT m)) then 0 else 1 + Max (height ‘ ran m))
  by pat-completeness auto
termination
proof −
  { fix a :: 'a prefix-tree

    have height-dom a
    proof (induction a)
      case (PT m)

      have (⋀x. x ∈ ran m ⟹ height-dom x)
      proof −
        fix x :: 'a prefix-tree
        assume x ∈ ran m
        then have ∃ a. m a = Some x
          by (simp add: ran-def)
        then show height-dom x
          using PT.IH by blast
      qed
      then show ?case
        using height.domintros
        by blast
    qed
  }
  then show ?thesis by auto
qed

function (domintros) height-over :: 'a list ⇒ 'a prefix-tree ⇒ nat where
  height-over xs (PT m) = 1 + foldr (λ x maxH . case m x of Some t' ⇒ max
(height-over xs t') maxH | None ⇒ maxH) xs 0
  by pat-completeness auto
termination
proof −
  {
    fix a :: 'a prefix-tree
    fix xs :: 'a list
```

```
    have height-over-dom (xs, a)
    proof (induction a)
      case (PT m)

      have (⋀x. x ∈ ran m ⟹ height-over-dom (xs, x))
      proof −
        fix x :: 'a prefix-tree
        assume x ∈ ran m
        then have ∃ a. m a = Some x
          by (simp add: ran-def)
        then show height-over-dom (xs, x)
          using PT.IH by blast
      qed
      then show ?case
        using height-over.domintros
        by (simp add: height-over.domintros ranI)
    qed
  }
  then show ?thesis by auto
qed

lemma height-over-empty :
  height-over xs empty = 1
proof −
  define xs' where xs' = xs
  have foldr (λ x maxH . case Map.empty x of Some t' ⇒ max (height-over xs' t')
maxH | None ⇒ maxH) xs 0 = 0
    by (induction xs; auto)
  then show ?thesis
    unfolding xs'-def empty-def
    by auto
qed


lemma height-over-subtree-less :
  assumes m x = Some t'
  and     x ∈ list.set xs
shows height-over xs t' < height-over xs (PT m)
proof −

  define xs' where xs' = xs

   have height-over xs' t' ≤ foldr (λ x maxH . case m x of Some t' ⇒ max
(height-over xs' t') maxH | None ⇒ maxH) xs 0
    using assms(2) proof (induction xs)
    case Nil
    then show ?case by auto
  next
    case (Cons x' xs)
```

```
  define f where f = foldr (λ x maxH . case m x of Some t' ⇒ max (height-over
xs' t') maxH | None ⇒ maxH) xs 0

  have ∗: foldr (λ x maxH . case m x of Some t' ⇒ max (height-over xs' t') maxH
| None ⇒ maxH) (x'#xs) 0
            = (case m x' of Some t' ⇒ max (height-over xs' t') f | None ⇒ f)
    unfolding f-def by auto

  show ?case proof (cases x=x')
    case True
    show ?thesis
      using ‹m x = Some t'›
      unfolding ∗ True by auto
  next
    case False
    then have x ∈ list.set xs
      using Cons.prems(1) by auto
    show ?thesis
      using Cons.IH[OF ‹x ∈ list.set xs›]
      unfolding ∗ f-def[symmetric]
      by (cases m x'; auto)
  qed
 qed
 then show ?thesis
  unfolding xs'-def by auto
qed


fun maximum-prefix :: 'a prefix-tree ⇒ 'a list ⇒ 'a list where
 maximum-prefix t [] = [] |
 maximum-prefix (PT m) (x # xs) = (case m x of None ⇒ [] | Some t ⇒ x #
maximum-prefix t xs)

lemma maximum-prefix-isin :
 isin t (maximum-prefix t xs)
proof (induction xs arbitrary: t)
 case Nil
 show ?case
  by auto
next
 case (Cons x xs)

 obtain m where ∗:t = PT m
  using finite-tree.cases by blast

 show ?case proof (cases m x)
  case None
  then have maximum-prefix t (x#xs) = []
```

```
      unfolding ∗ by auto
    then show ?thesis
      by auto
  next
    case (Some t′)
    then have maximum-prefix t (x#xs) = x # maximum-prefix t′ xs
      unfolding ∗ by auto
    moreover have isin t′ (maximum-prefix t′ xs)
      using Cons.IH by auto
    ultimately show ?thesis
      by (simp add: ∗ Some)
  qed
qed


lemma maximum-prefix-maximal :
  maximum-prefix t xs = xs
    ∨ (∃ x′ xs′ . xs = (maximum-prefix t xs)@[x′]@xs′ ∧ ¬ isin t ((maximum-prefix
t xs)@[x′]))
proof (induction xs arbitrary: t)
  case Nil
  show ?case by auto
next
  case (Cons x xs)
  obtain m where ∗:t = PT m
    using finite-tree.cases by blast

  show ?case proof (cases m x)
    case None
    then have maximum-prefix t (x#xs) = []
      unfolding ∗ by auto
    moreover have ¬ isin t ([]@[x]@xs)
      using isin-prefix[of t [x] xs]
      by (simp add: ∗ None)
    ultimately show ?thesis
      by (simp add: ∗ None)
  next
    case (Some t′)
    then have maximum-prefix t (x#xs) = x # maximum-prefix t′ xs
      unfolding ∗ by auto
    moreover note Cons.IH[of t′]
    ultimately show ?thesis
      by (simp add: ∗ Some)
  qed
qed
```

fun *maximum-fst-prefixes* :: $('a \times 'b)$ *prefix-tree* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'b$ *list* $\Rightarrow$ $('a \times 'b)$ *list list* **where**
maximum-fst-prefixes t [] *ys* = (*if is-leaf t then* [[]] *else* []) |
maximum-fst-prefixes (*PT m*) (*x* # *xs*) *ys* = (*if is-leaf* (*PT m*) *then* [[]] *else concat* (*map* (λ *y* . *map* ((#) (*x*,*y*)) (*maximum-fst-prefixes* (*the* (*m* (*x*,*y*))) *xs ys*)) (*filter* (λ *y* . (*m* (*x*,*y*) $\neq$ *None*)) *ys*)))

lemma *maximum-fst-prefixes-set* :
list.set (*maximum-fst-prefixes t xs ys*) $\subseteq$ *set t*
proof (*induction xs arbitrary*: *t*)
case *Nil*
show *?case*
by *auto*
next
case (*Cons x xs*)

obtain *m* **where** $*$:*t* = *PT m*
using *finite-tree.cases* **by** *blast*

show *list.set* (*maximum-fst-prefixes t* (*x* # *xs*) *ys*) $\subseteq$ *set t*
proof
fix *p* **assume** *p* $\in$ *list.set* (*maximum-fst-prefixes t* (*x* # *xs*) *ys*)

show *p* $\in$ *set t* **proof** (*cases is-leaf* (*PT m*))
case *True*
then have *p* = []
using ‹*p* $\in$ *list.set* (*maximum-fst-prefixes t* (*x* # *xs*) *ys*)› **unfolding** $*$ *maximum-fst-prefixes.simps* **by** *force*
then show *?thesis*
using *set-Nil*[*of t*]
by *blast*
next
case *False*
then obtain *y* **where** *y* $\in$ *list.set* (*filter* (λ *y* . (*m* (*x*,*y*) $\neq$ *None*)) *ys*)
and *p* $\in$ *list.set* (*map* ((#) (*x*,*y*)) (*maximum-fst-prefixes* (*the* (*m* (*x*,*y*))) *xs ys*))
using ‹*p* $\in$ *list.set* (*maximum-fst-prefixes t* (*x* # *xs*) *ys*)›
unfolding $*$ **by** *auto*

then have *m* (*x*,*y*) $\neq$ *None*
by *auto*
then obtain t' **where** *m* (*x*,*y*) = *Some* t'
by *auto*
moreover obtain p' **where** *p* = (*x*,*y*)#p' **and** p' $\in$ *list.set* (*maximum-fst-prefixes* (*the* (*m* (*x*,*y*))) *xs ys*)
using ‹*p* $\in$ *list.set* (*map* ((#) (*x*,*y*)) (*maximum-fst-prefixes* (*the* (*m* (*x*,*y*))) *xs ys*))›

```
      by auto
    ultimately have isin t' p'
      using Cons.IH
      by auto
    then have isin t p
      unfolding * ‹p = (x,y)#p'› using ‹m (x,y) = Some t'› by auto
    then show p ∈ set t
      by auto
  qed
 qed
qed

lemma maximum-fst-prefixes-are-prefixes :
 assumes xys ∈ list.set (maximum-fst-prefixes t xs ys)
 shows map fst xys = take (length xys) xs
using assms proof (induction xys arbitrary: t xs)
 case Nil
 then show ?case by auto
next
 case (Cons xy xys)
 then have xs ≠ []
  by auto
 then obtain x xs' where xs = x#xs'
  using list.exhaust by auto

 obtain m where *:t = PT m
  using finite-tree.cases by blast
 have is-leaf (PT m) = False
  using Cons.prems unfolding * ‹xs = x#xs'›
  by auto
 have (xy#xys) ∈ list.set (concat (map (λ y . map ((#) (x,y)) (maximum-fst-prefixes
(the (m (x,y))) xs' ys)) (filter (λ y . (m (x,y) ≠ None)) ys)))
  using Cons.prems unfolding * ‹xs = x#xs'› ‹is-leaf (PT m) = False› maxi-
mum-fst-prefixes.simps by auto
 then obtain y where y ∈ list.set (filter (λ y . (m (x,y) ≠ None)) ys)
              and (xy#xys) ∈ list.set (map ((#) (x,y)) (maximum-fst-prefixes
(the (m (x,y))) xs' ys))
  by auto
 then have xy = (x,y) and xys ∈ list.set (maximum-fst-prefixes (the (m (x,y)))
xs' ys)
  by auto

 have **: take (length ((x, y) # xys)) (x # xs') = x # (take (length xys) xs')
  by auto

 show ?case
  using Cons.IH[OF ‹xys ∈ list.set (maximum-fst-prefixes (the (m (x,y))) xs'
ys)›]
  unfolding ‹xy = (x,y)› ‹xs = x#xs'› ** by auto
```

```
qed


lemma finite-tree-set-eq :
  assumes set t1 = set t2
  and     finite-tree t1
  shows t1 = t2
using assms proof (induction height t1 arbitrary: t1 t2 rule: less-induct)
  case less

  obtain m1 m2 where t1 = PT m1 and t2 = PT m2
    by (metis finite-tree.cases)

  show ?case proof (cases height t1)
    case 0

    have t1 = empty
      using 0
      unfolding ‹t1 = PT m1› height.simps is-leaf.simps
      by (metis add-is-0 zero-neq-one)
    then have set t2 = {[]}
      using less Prefix-Tree.set-empty by auto
    have m2 = Map.empty
    proof
      show ⋀x. m2 x = None
      proof −
        fix x show m2 x = None
        proof (rule ccontr)
          assume m2 x ≠ None
          then obtain t′ where m2 x = Some t′
            by blast
          then have [x] ∈ set t2
            unfolding ‹t2 = PT m2› set.simps by auto
          then show False
            using ‹set t2 = {[]}› by auto
        qed
      qed
    qed
    then show ?thesis
      unfolding ‹t1 = empty› ‹t2 = PT m2› empty-def by simp
  next
    case (Suc k)


    show ?thesis proof (rule ccontr)
      assume t1 ≠ t2

      then have m1 ≠ m2
```

```
    using ‹t1 = PT m1› ‹t2 = PT m2› by auto
  then obtain x where m1 x ≠ m2 x
    by (meson ext)

  then consider m1 x ≠ None ∧ m2 x ≠ None | m1 x = None ⟷ m2 x ≠
None
    by fastforce
  then show False proof cases
    case 1
    then obtain t1' t2' where m1 x = Some t1' and m2 x = Some t2'
      by auto
    then have t1' ≠ t2'
      using ‹m1 x ≠ m2 x› by auto
    moreover have set t1' = set t2'
    proof −
      have ⋀ io . isin t1' io = isin t1 (x#io)
        unfolding ‹t1 = PT m1› using ‹m1 x = Some t1'› by auto
      moreover have ⋀ io . isin t2' io = isin t2 (x#io)
        unfolding ‹t2 = PT m2› using ‹m2 x = Some t2'› by auto
      ultimately show ?thesis
        using less.prems(1)
        by (metis Collect-cong mem-Collect-eq set.simps)
    qed
    moreover have height t1' < height t1
    proof −
      have height t1 = 1 + Max (height ‘ ran m1)
        using Suc
        unfolding ‹t1 = PT m1› height.simps
        by (meson Zero-not-Suc)
      moreover have height t1' ∈ height ‘ ran m1
        using ‹m1 x = Some t1'›
        by (meson image-eqI ranI)
      moreover have finite (ran m1)
        using less.prems(2)
        unfolding ‹t1 = PT m1› finite-tree.simps
        by (simp add: finite-ran)
      ultimately have height t1 ≥ 1 + height t1'
        by simp
      then show ?thesis by auto
    qed
    moreover have finite-tree t1'
      using less.prems(2)
      unfolding ‹t1 = PT m1› finite-tree.simps
      by (meson ‹m1 x = Some t1'› ranI)
    ultimately show False
      using less.hyps[of t1' t2']
      by blast
  next
    case 2
```

```
      then have isin t1 [x] ≠ isin t2 [x]
        unfolding ‹t1 = PT m1› ‹t2 = PT m2› by auto
      then show False using less.prems(1) by auto
    qed
  qed
 qed
qed
```

```
fun after-fst :: ('a × 'b) prefix-tree ⇒ 'a list ⇒ 'b list ⇒ ('a × 'b) prefix-tree where
  after-fst t [] ys = t |
  after-fst (PT m) (x # xs) ys = foldr (λ y t . case m (x,y) of None ⇒ t | Some
t' ⇒ combine t (after-fst t' xs ys)) ys empty
```

10.1 Alternative characterization for code generation

In order to generate code for the prefix trees, we represent the map inside each prefix tree by Mapping.

```
definition MPT :: ('a,'a prefix-tree) mapping ⇒ 'a prefix-tree where
  MPT m = PT (Mapping.lookup m)

code-datatype MPT

lemma equals-MPT[code]: equal-class.equal (MPT m1) (MPT m2) = (m1 = m2)

proof −
 have equal-class.equal (MPT m1) (MPT m2) = equal-class.equal (PT (Mapping.lookup
m1)) (PT (Mapping.lookup m2))
    unfolding MPT-def by simp
  also have ... = ((Mapping.lookup m1) = (Mapping.lookup m2))
    using prefix-tree.eq.simps by auto
  also have ... = (m1 = m2)
    by (simp add: Mapping.lookup.rep-eq rep-inject)
  finally show ?thesis .
qed

lemma empty-MPT[code] :
  empty = MPT Mapping.empty
  unfolding MPT-def empty-def
  by (metis lookup-empty)

lemma insert-MPT[code] :
  insert (MPT m) xs = (case xs of
    [] ⇒ (MPT m) |
    (x#xs) ⇒ MPT (Mapping.update x (insert (case Mapping.lookup m x of None
⇒ empty | Some t' ⇒ t') xs) m))
```

apply (*cases xs*; *simp*)
by (*simp add*: *MPT-def lookup.rep-eq update.rep-eq*)

lemma *isin-MPT*[*code*] :
isin (*MPT m*) *xs* = (*case xs of*
[] ⇒ *True* |
(*x*#*xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ *False* | *Some t* ⇒ *isin t xs*))
unfolding *MPT-def* **by** (*cases xs*; *auto*)

lemma *after-MPT*[*code*] :
after (*MPT m*) *xs* = (*case xs of*
[] ⇒ *MPT m* |
(*x*#*xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ *empty* | *Some t* ⇒ *after t xs*))
unfolding *MPT-def* **by** (*cases xs*; *auto*)

lemma *PT-Mapping-ob* :
fixes *t* :: *'a prefix-tree*
obtains *m* **where** *t* = *MPT m*
proof −
obtain *m'* **where** *t* = *PT m'*
using *prefix-tree.exhaust* **by** *blast*
then have *t* = *MPT* (*Mapping m'*)
unfolding *MPT-def*
by (*simp add*: *Mapping-inverse lookup.rep-eq*)
then show *?thesis* **using** *that* **by** *blast*
qed

lemma *set-MPT*[*code*] :
set (*MPT m*) = *Set.insert* [] (⋃ *x* ∈ *Mapping.keys m* . (*Cons x*) ‘ (*set* (*the* (*Mapping.lookup m x*))))
unfolding *MPT-def set-alt-def keys-dom-lookup* **by** *simp*

lemma *combine-MPT*[*code*] :
combine (*MPT m1*) (*MPT m2*) = *MPT* (*Mapping.combine combine m1 m2*)
proof −
have *combine* (*MPT m1*) (*MPT m2*) = *combine* (*PT* (*Mapping.lookup m1*)) (*PT* (*Mapping.lookup m2*))
unfolding *MPT-def* **by** *simp*
also have ... = *PT* (λ*x* . *combine-options combine* ((*Mapping.lookup m1*) *x*) ((*Mapping.lookup m2*) *x*))
unfolding *combine.simps*
by (*simp add*: *combine-options-def*)
ultimately show *?thesis*
by (*metis MPT-def combine.abs-eq lookup.abs-eq rep-inverse*)
qed

lemma *combine-after-MPT*[*code*] :
combine-after (*MPT m*) *xs t* = (*case xs of*
[] ⇒ *combine* (*MPT m*) *t* |
(*x*#*xs*) ⇒ *MPT* (*Mapping.update x* (*combine-after* (*case Mapping.lookup m x of None* ⇒ *empty* | *Some t′* ⇒ *t′*) *xs t*) *m*))
apply (*cases xs*; *simp*)
by (*simp add*: *MPT-def lookup.rep-eq update.rep-eq*)

lemma *finite-tree-MPT*[*code*] :
finite-tree (*MPT m*) = (*finite* (*Mapping.keys m*) ∧ (∀ *x* ∈ *Mapping.keys m* . *finite-tree* (*the* (*Mapping.lookup m x*))))
unfolding *MPT-def finite-tree.simps keys-dom-lookup ran-dom-the-eq*[*symmetric*]
by *blast*

lemma *sorted-list-of-maximal-sequences-in-tree-MPT*[*code*] :
sorted-list-of-maximal-sequences-in-tree (*MPT m*) =
(*if Mapping.keys m* = {}
then [[]]
else concat (*map* (λ*k* . *map* ((#) *k*) (*sorted-list-of-maximal-sequences-in-tree* (*the* (*Mapping.lookup m k*)))) (*sorted-list-of-set* (*Mapping.keys m*))))
unfolding *MPT-def sorted-list-of-maximal-sequences-in-tree.simps keys-dom-lookup*
by *simp*

lemma *is-leaf-MPT*[*code*]:
is-leaf (*MPT m*) = (*Mapping.is-empty m*)
by (*simp add*: *MPT-def Mapping.is-empty-def Prefix-Tree.empty-def keys-dom-lookup*)

lemma *height-MPT*[*code*] :
height (*MPT m*) = (*if* (*is-leaf* (*MPT m*)) *then 0 else 1* + *Max* ((*height* ∘ *the* ∘ *Mapping.lookup m*) ‘ *Mapping.keys m*))
proof −
have *height* (*MPT m*) = (*if* (*is-leaf* (*MPT m*)) *then 0 else 1* + *Max* (*height* ‘ ((λ*k* . *the* (*Mapping.lookup m k*)) ‘ *Mapping.keys m*)))
by (*simp add*: *MPT-def keys-dom-lookup ran-dom-the-eq*)
moreover have (*height* ‘ ((λ*k* . *the* (*Mapping.lookup m k*)) ‘ *Mapping.keys m*)) = ((*height* ∘ *the* ∘ *Mapping.lookup m*) ‘ *Mapping.keys m*)
by *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *maximum-prefix-MPT*[*code*]:
maximum-prefix (*MPT m*) *xs* = (*case xs of*
[] ⇒ [] |
(*x*#*xs*) ⇒ (*case Mapping.lookup m x of None* ⇒ [] | *Some t* ⇒ *x* # *maximum-prefix t xs*))

```
  apply (cases xs; simp)
  by (simp add: MPT-def lookup.rep-eq)

lemma sorted-list-of-in-tree-MPT[code] :
  sorted-list-of-sequences-in-tree (MPT m) =
    (if Mapping.keys m = {}
      then [[]]
      else [] # concat (map (λk . map ((#) k) (sorted-list-of-sequences-in-tree (the
(Mapping.lookup m k)))) (sorted-list-of-set (Mapping.keys m))))
   unfolding MPT-def sorted-list-of-sequences-in-tree.simps keys-dom-lookup by
simp

lemma maximum-fst-prefixes-leaf:
  fixes xs :: 'a list and ys :: 'b list
shows maximum-fst-prefixes empty xs ys  = [[]]
proof −
  have is-leaf (empty :: ('a×'b)prefix-tree) by auto

  obtain m where (empty :: ('a×'b)prefix-tree) = PT m
    using prefix-tree.exhaust by blast

  show ?thesis proof (cases xs)
    case Nil
    then show ?thesis by auto
  next
    case (Cons x xs)
    show ?thesis
      using ‹is-leaf (empty :: ('a×'b)prefix-tree) ›
     unfolding ‹(empty :: ('a×'b)prefix-tree) = PT m› Cons maximum-fst-prefixes.simps
by force
  qed
qed

lemma maximum-fst-prefixes-MPT[code]:
  maximum-fst-prefixes (MPT m) xs ys = (case xs of
    [] ⇒ (if is-leaf (MPT m) then [[]] else []) |
     (x # xs) ⇒ (if is-leaf (MPT m) then [[]] else concat (map (λ y . map ((#)
(x,y)) (maximum-fst-prefixes (the (Mapping.lookup m (x,y))) xs ys)) (filter (λ y .
(Mapping.lookup m (x,y) ≠ None)) ys))))
  using maximum-fst-prefixes-leaf
  apply (cases xs)
    apply auto[1]
  by (simp add: MPT-def lookup.rep-eq)
```

end

11 Refined Code Generation for Prefix Trees

This theory provides alternative code equations for selected functions on prefix trees. Currently only Mapping via RBT is supported.

theory *Prefix-Tree-Refined*
imports *Prefix-Tree Containers.Containers*
begin

lemma *combine-refined*[*code*] :
  fixes *m1* :: (*'a* :: *ccompare*, *'a prefix-tree*) *mapping-rbt*
  shows *Prefix-Tree.combine* (*MPT* (*RBT-Mapping m1*)) (*MPT* (*RBT-Mapping m2*))
        = (*case ID CCOMPARE*(*'a*) *of*
          None ⇒ *Code.abort* (*STR ''combine-MPT-RBT-Mapping*: *ccompare* = *None''*) (λ-. *Prefix-Tree.combine* (*MPT* (*RBT-Mapping m1*)) (*MPT* (*RBT-Mapping m2*)))
            | *Some* - ⇒ *MPT* (*RBT-Mapping* (*RBT-Mapping2.join* (λ *a t1 t2* . *Prefix-Tree.combine t1 t2*) *m1 m2*)))
    (**is** *?PT1* = *?PT2*)
proof (*cases ID CCOMPARE*(*'a*))
  case *None*
  then show *?thesis* **by** *simp*
next
  case (*Some a*)
  then have ∗: *?PT2* = *MPT* (*RBT-Mapping* (*RBT-Mapping2.join* (λ *a t1 t2* . *Prefix-Tree.combine t1 t2*) *m1 m2*))
    by *auto*

  have *ID CCOMPARE*(*'a*) ≠ *None*
    using *Some* **by** *auto*

  have *Mapping.lookup* (*Mapping.combine Prefix-Tree.combine* (*RBT-Mapping m1*) (*RBT-Mapping m2*)) = *Mapping.lookup* (*RBT-Mapping* (*RBT-Mapping2.join* (λ *a b c* . *Prefix-Tree.combine b c*) *m1 m2*))
  proof
    fix *x*

    show *Mapping.lookup* (*Mapping.combine Prefix-Tree.combine* (*RBT-Mapping m1*) (*RBT-Mapping m2*)) *x* =
      Mapping.lookup (*RBT-Mapping* (*RBT-Mapping2.join* (λ*a. Prefix-Tree.combine*)

```
m1 m2)) x
  (is ?M1 = ?M2)
  proof (cases RBT-Mapping2.lookup m1 x)
    case None
    show ?thesis proof (cases RBT-Mapping2.lookup m2 x)
      case None

      have ?M1 = None
        using ‹RBT-Mapping2.lookup m1 x = None› None
        by (simp add: lookup-combine Mapping.lookup.abs-eq)
      moreover have ?M2 = None
        using ‹RBT-Mapping2.lookup m1 x = None› None
       by (simp add: Mapping.lookup.abs-eq ‹ID ccompare ≠ None› lookup-join)
      ultimately show ?thesis
        by simp
    next
      case (Some a)
      have ?M1 = Some a
        using ‹RBT-Mapping2.lookup m1 x = None› Some
        by (simp add: lookup-combine Mapping.lookup.abs-eq)
      moreover have ?M2 = Some a
        using ‹RBT-Mapping2.lookup m1 x = None› Some
       by (simp add: Mapping.lookup.abs-eq ‹ID ccompare ≠ None› lookup-join)
      ultimately show ?thesis
        by simp
    qed
  next
    case (Some a)
    show ?thesis proof (cases RBT-Mapping2.lookup m2 x)
      case None

      have ?M1 = Some a
        using None Some
        by (simp add: lookup-combine Mapping.lookup.abs-eq)
      moreover have ?M2 = Some a
        using None Some
       by (simp add: Mapping.lookup.abs-eq ‹ID ccompare ≠ None› lookup-join)
      ultimately show ?thesis
        by simp
    next
      case (Some b)

      have ?M1 = Some (Prefix-Tree.combine a b)
        using ‹RBT-Mapping2.lookup m1 x = Some a› Some
        by (simp add: lookup-combine Mapping.lookup.abs-eq)
      moreover have ?M2 = Some (Prefix-Tree.combine a b)
        using ‹RBT-Mapping2.lookup m1 x = Some a› Some
       by (simp add: Mapping.lookup.abs-eq ‹ID ccompare ≠ None› lookup-join)
      ultimately show ?thesis
```

```
        by simp
      qed
    qed
  qed
 then have (Mapping.combine Prefix-Tree.combine (RBT-Mapping m1) (RBT-Mapping
m2)) = (RBT-Mapping (RBT-Mapping2.join (λ a b c . Prefix-Tree.combine b c)
m1 m2))
    by (metis Mapping.lookup.rep-eq rep-inverse)
  then show ?thesis
    unfolding ∗ unfolding combine-MPT by simp
qed

lemma is-leaf-refined[code] :
  fixes m :: ('a :: ccompare, 'a prefix-tree) mapping-rbt
  shows Prefix-Tree.is-leaf (MPT (RBT-Mapping m))
          = (case ID CCOMPARE('a) of
              None ⇒ Code.abort (STR ''is-leaf-MPT-RBT-Mapping: ccompare =
None'') (λ-. Prefix-Tree.is-leaf (MPT (RBT-Mapping m)))
             | Some - ⇒ RBT-Mapping2.is-empty m)
    (is ?PT1 = ?PT2)
proof (cases ID CCOMPARE('a))
  case None
  then show ?thesis by simp
next
  case (Some a)
  then have ∗: ?PT2 = RBT-Mapping2.is-empty m
    by auto
  show ?thesis
    by (simp add: ∗ MPT-def Mapping.lookup.abs-eq Prefix-Tree.empty-def Some)
qed

end
```

12 State Cover

This theory introduces a simple depth-first strategy for computing state covers.

```
theory State-Cover
imports FSM
begin
```

12.1 Basic Definitions

```
type-synonym ('a,'b) state-cover = ('a × 'b) list set
type-synonym ('a,'b,'c) state-cover-assignment = 'a ⇒ ('b × 'c) list

fun is-state-cover :: ('a,'b,'c) fsm ⇒ ('b,'c) state-cover ⇒ bool where
```

is-state-cover M SC = ([] ∈ *SC* ∧ (∀ *q* ∈ *reachable-states M* . ∃ *io* ∈ *SC* . *q* ∈ *io-targets M io* (*initial M*)))

fun *is-state-cover-assignment* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ *bool* **where**
is-state-cover-assignment M f = (*f* (*initial M*) = [] ∧ (∀ *q* ∈ *reachable-states M* . *q* ∈ *io-targets M* (*f q*) (*initial M*)))

lemma *state-cover-assignment-from-state-cover* :
assumes *is-state-cover M SC*
obtains *f* **where** *is-state-cover-assignment M f*
and ⋀ *q* . *q* ∈ *reachable-states M* ⟹ *f q* ∈ *SC*
proof −
define *f* **where** *f*: *f* = (λ *q* . (*if q* = *initial M then* [] *else* (*SOME io* . *io* ∈ *SC* ∧ *q* ∈ *io-targets M io* (*initial M*))))

have *f* (*initial M*) = []
using *f* **by** *auto*
moreover have ⋀ *q* . *q* ∈ *reachable-states M* ⟹ *f q* ∈ *SC* ∧ *q* ∈ *io-targets M* (*f q*) (*initial M*)
proof −
fix *q* **assume** *q* ∈ *reachable-states M*
show *f q* ∈ *SC* ∧ *q* ∈ *io-targets M* (*f q*) (*initial M*)
proof (*cases q* = *initial M*)
case *True*
have *q* ∈ *io-targets M* (*f q*) (*FSM.initial M*)
unfolding *True* ‹*f* (*initial M*) = []› **by** *auto*
then show *?thesis*
using *True assms* ‹*f* (*initial M*) = []› **by** *auto*
next
case *False*
then have *f q* = (*SOME io* . *io* ∈ *SC* ∧ *q* ∈ *io-targets M io* (*initial M*))
using *f* **by** *auto*
moreover have ∃ *io* . *io* ∈ *SC* ∧ *q* ∈ *io-targets M io* (*initial M*)
using *assms* ‹*q* ∈ *reachable-states M*›
by (*meson is-state-cover.simps*)
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *someI-ex*)
qed
qed
ultimately show *?thesis* **using** *that*[*of f*]
by (*meson is-state-cover-assignment.elims*(*3*))
qed

lemma *is-state-cover-assignment-language* :
assumes *is-state-cover-assignment M V*
and *q* ∈ *reachable-states M*
shows *V q* ∈ *L M*
using *assms io-targets-language*

by (*metis is-state-cover-assignment.simps*)

lemma *is-state-cover-assignment-observable-after* :
assumes *observable M*
and *is-state-cover-assignment M V*
and $q \in$ *reachable-states M*
shows *after-initial M* (*V q*) = *q*
proof −
have $q \in$ *io-targets M* (*V q*) (*initial M*)
using *assms*(*2*,*3*)
by *auto*
then have *io-targets M* (*V q*) (*initial M*) = {*q*}
using *observable-io-targets*[*OF assms*(*1*) *io-targets-language*[*OF* ‹$q \in$ *io-targets M* (*V q*) (*initial M*)›]]
by (*metis singletonD*)

then obtain *p* **where** *path M* (*initial M*) *p* **and** *p-io p* = *V q* **and** *target* (*initial M*) *p* = *q*
unfolding *io-targets.simps*
by *blast*
then show *after-initial M* (*V q*) = *q*
using *after-path*[*OF assms*(*1*), *of initial M p*]
by *simp*
qed

lemma *non-initialized-state-cover-assignment-from-non-initialized-state-cover* :
assumes $\bigwedge$ *q* . $q \in$ *reachable-states M* $\Longrightarrow \exists$ *io* $\in$ *L M* $\cap$ *SC* . $q \in$ *io-targets M io* (*initial M*)
obtains *f* **where** $\bigwedge$ *q* . $q \in$ *reachable-states M* $\Longrightarrow q \in$ *io-targets M* (*f q*) (*initial M*)
and $\bigwedge$ *q* . $q \in$ *reachable-states M* $\Longrightarrow$ *f q* $\in$ *L M* $\cap$ *SC*
proof −
define *f* **where** *f*: *f* = (λ *q* . (*SOME io* . *io* $\in$ *L M* $\cap$ *SC* $\wedge$ *q* $\in$ *io-targets M io* (*initial M*)))

have $\bigwedge$ *q* . $q \in$ *reachable-states M* $\Longrightarrow$ *f q* $\in$ *L M* $\cap$ *SC* $\wedge$ *q* $\in$ *io-targets M* (*f q*) (*initial M*)
proof −
fix *q* **assume** $q \in$ *reachable-states M*
show *f q* $\in$ *L M* $\cap$ *SC* $\wedge$ *q* $\in$ *io-targets M* (*f q*) (*initial M*)
proof −
have *f q* = (*SOME io* . *io* $\in$ *L M* $\cap$ *SC* $\wedge$ *q* $\in$ *io-targets M io* (*initial M*))
using *f* **by** *auto*
moreover have $\exists$ *io* . *io* $\in$ *L M* $\cap$ *SC* $\wedge$ *q* $\in$ *io-targets M io* (*initial M*)
using *assms* ‹$q \in$ *reachable-states M*›
by (*meson Int-iff*)
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *someI-ex*)
qed

qed
then show *?thesis* **using** *that*[*of f*]
by *blast*
qed

lemma *state-cover-assignment-inj* :
assumes *is-state-cover-assignment M V*
and *observable M*
and *q1* ∈ *reachable-states M*
and *q2* ∈ *reachable-states M*
and *q1* ≠ *q2*
shows *V q1* ≠ *V q2*
proof (*rule ccontr*)
assume ¬ *V q1* ≠ *V q2*

then have *io-targets M* (*V q1*) (*initial M*) = *io-targets M* (*V q2*) (*initial M*)
by *auto*
then have *q1* = *q2*
using *assms*(*2*)
proof −
have *f1*: ∀ *a f*. *a* ∉ *FSM.states* (*f*::('*a*, '*b*, '*c*) *fsm*) ∨ *FSM.initial* (*FSM.from-FSM f a*) = *a*
by (*meson from-FSM-simps*(*1*))
obtain *ff* :: ('*a* ⇒ ('*b* × '*c*) *list*) ⇒ ('*a*, '*b*, '*c*) *fsm* ⇒ ('*a*, '*b*, '*c*) *fsm* **and** *pps* :: ('*a* ⇒ ('*b* × '*c*) *list*) ⇒ ('*a*, '*b*, '*c*) *fsm* ⇒ '*a* ⇒ ('*b* × '*c*) *list* **where**
f2: *M* = *ff V M* ∧ *V* = *pps V M* ∧ *pps V M* (*FSM.initial* (*ff V M*)) = [] ∧ (∀ *a*. *a* ∉ *reachable-states* (*ff V M*) ∨ *a* ∈ *io-targets* (*ff V M*) (*pps V M a*) (*FSM.initial* (*ff V M*)))
using *assms*(*1*) **by** *fastforce*
then have *f3*: *q2* ∈ *FSM.states* (*ff V M*)
by (*simp add*: ‹*q2* ∈ *reachable-states M*› *reachable-state-is-state*)
then have *f4*: ∃ *ps*. *FSM.initial* (*FSM.from-FSM M q2*) = *target* (*FSM.initial* (*ff V M*)) *ps* ∧ *path* (*ff V M*) (*FSM.initial* (*ff V M*)) *ps* ∧ *p-io ps* = *V q2*
using *f2* ‹*q2* ∈ *reachable-states M*› *assms*(*1*) **by** *auto*
have *q1* ∈ {*target* (*FSM.initial M*) *ps* |*ps*. *path M* (*FSM.initial M*) *ps* ∧ *p-io ps* = *V q2*}
by (*metis* (*no-types*) ‹*io-targets M* (*V q1*) (*FSM.initial M*) = *io-targets M* (*V q2*) (*FSM.initial M*)› ‹*q1* ∈ *reachable-states M*› *assms*(*1*) *io-targets.simps is-state-cover-assignment.simps*)
then have ∃ *ps*. *FSM.initial* (*FSM.from-FSM M q1*) = *target* (*FSM.initial* (*ff V M*)) *ps* ∧ *path* (*ff V M*) (*FSM.initial* (*ff V M*)) *ps* ∧ *p-io ps* = *V q2*
using *f2* **by** (*simp add*: ‹*q1* ∈ *reachable-states M*› *reachable-state-is-state*)
then show *?thesis*
using *f4 f3 f2 f1* **by** (*metis* (*no-types*) ‹*observable M*› ‹*q1* ∈ *reachable-states M*› *observable-path-io-target reachable-state-is-state singletonD singletonI*)
qed
then show *False*
using ‹*q1* ≠ *q2*› **by** *blast*
qed

lemma *state-cover-assignment-card* :
  assumes *is-state-cover-assignment M V*
  and *observable M*
shows *card* (*V* ‘ *reachable-states M*) = *card* (*reachable-states M*)
proof −
  have *inj-on V* (*reachable-states M*)
    using *state-cover-assignment-inj*[*OF assms*] **by** (*meson inj-onI*)

  then have *card* (*reachable-states M*) ≤ *card* (*V* ‘ *reachable-states M*)
    using *fsm-states-finite restrict-to-reachable-states-simps*(*2*)
    by (*simp add*: *card-image*)
  moreover have *card* (*V* ‘ *reachable-states M*) ≤ *card* (*reachable-states M*)
    using *fsm-states-finite*
    using *card-image-le*
    by (*metis restrict-to-reachable-states-simps*(*2*))
  ultimately show *?thesis* **by** *simp*
qed

lemma *state-cover-assignment-language* :
  assumes *is-state-cover-assignment M V*
  shows *V* ‘ *reachable-states M* ⊆ *L M*
  using *assms* **unfolding** *is-state-cover-assignment.simps*
  using *language-io-target-append* **by** *fastforce*

fun *is-minimal-state-cover* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'b*,*'c*) *state-cover* ⇒ *bool* **where**
is-minimal-state-cover M SC = (∃ *f* . (*SC* = *f* ‘ *reachable-states M*) ∧ (*is-state-cover-assignment M f*))

lemma *minimal-state-cover-is-state-cover* :
  assumes *is-minimal-state-cover M SC*
  shows *is-state-cover M SC*
proof −
  obtain *f* **where** *f* (*initial M*) = [] **and** (*SC* = *f* ‘ *reachable-states M*) **and** (⋀ *q* . *q* ∈ *reachable-states M* ⟹ *q* ∈ *io-targets M* (*f q*) (*initial M*))
    using *assms* **by** *auto*

  show *?thesis* **unfolding** *is-state-cover.simps* ‹(*SC* = *f* ‘ *reachable-states M*)›
  proof −
    have *f* ‘ *FSM.reachable-states M* ⊆ *L M*
    proof
      fix *io* **assume** *io* ∈ *f* ‘ *FSM.reachable-states M*
      then obtain *q* **where** *q* ∈ *reachable-states M* **and** *io* = *f q*
        by *blast*
      then have *q* ∈ *io-targets M* (*f q*) (*initial M*)
        using ‹(⋀ *q* . *q* ∈ *reachable-states M* ⟹ *q* ∈ *io-targets M* (*f q*) (*initial*

M))› **by** *blast*
then show $io \in L\ M$
unfolding ‹$io = f\ q$ › **by** *force*
qed

moreover have $\forall q \in FSM.reachable\text{-}states\ M.\ \exists io \in f\ `\ FSM.reachable\text{-}states\ M.\ q \in io\text{-}targets\ M\ io\ (FSM.initial\ M)$
using ‹$(\bigwedge q\ .\ q \in reachable\text{-}states\ M \Longrightarrow q \in io\text{-}targets\ M\ (f\ q)\ (initial\ M))$› **by** *blast*

ultimately show $[] \in f\ `\ FSM.reachable\text{-}states\ M \wedge (\forall q \in FSM.reachable\text{-}states\ M.\ \exists io \in f\ `\ FSM.reachable\text{-}states\ M.\ q \in io\text{-}targets\ M\ io\ (FSM.initial\ M))$
using ‹$f\ (initial\ M) = []$› *reachable-states-initial* **by** *force*
qed
qed

lemma *state-cover-assignment-after* :
assumes *observable M*
and *is-state-cover-assignment M V*
and $q \in reachable\text{-}states\ M$
shows $V\ q \in L\ M$ **and** $after\text{-}initial\ M\ (V\ q) = q$
proof −
have $V\ q \in L\ M \wedge after\text{-}initial\ M\ (V\ q) = q$
using *assms*(*3*) **proof** (*induct rule*: *reachable-states-induct*)
case *init*
have $V\ (FSM.initial\ M) = []$
using *assms*(*2*)
by *auto*
then show *?case*
by *auto*
next
case (*transition t*)
then have $t\text{-}target\ t \in reachable\text{-}states\ M$
using *reachable-states-next*
by *metis*
then have $t\text{-}target\ t \in io\text{-}targets\ M\ (V\ (t\text{-}target\ t))\ (FSM.initial\ M)$
using *assms*(*2*)
unfolding *is-state-cover-assignment.simps*
by *auto*
then obtain p **where** $path\ M\ (initial\ M)\ p$ **and** $target\ (initial\ M)\ p = t\text{-}target\ t$ **and** $p\text{-}io\ p = V\ (t\text{-}target\ t)$
by *auto*
then have $V\ (t\text{-}target\ t) \in L\ M$
by *force*
then show *?case*
using *after-path*[*OF assms*(*1*) ‹$path\ M\ (initial\ M)\ p$›]
unfolding ‹$p\text{-}io\ p = V\ (t\text{-}target\ t)$› ‹$target\ (initial\ M)\ p = t\text{-}target\ t$›
by *simp*
qed

then show $V\ q \in L\ M$ **and** *after-initial* $M\ (V\ q) = q$
by *simp+*
qed

definition *covered-transitions* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $('a,'b,'c)$ *transition set* **where**
covered-transitions $M\ V\ \alpha$ = (*let*
ts = *the-elem* (*paths-for-io* M (*initial* M) α)
in
List.set (*filter* (λt . ((V (*t-source* t)) @ [(*t-input* t, *t-output* t)]) = (V (*t-target* t))) *ts*))

12.2 State Cover Computation

fun *reaching-paths-up-to-depth* :: $('a::linorder,'b::linorder,'c::linorder)$ *fsm* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $('a \Rightarrow ('a,'b,'c)$ *path option*$)$ $\Rightarrow$ *nat* $\Rightarrow$ $('a \Rightarrow ('a,'b,'c)$ *path option*$)$ **where**
reaching-paths-up-to-depth M nexts dones assignment 0 = *assignment* |
reaching-paths-up-to-depth M nexts dones assignment (*Suc k*) = (*let*
usable-transitions = *filter* (λ t . *t-source* $t \in$ *nexts* $\wedge$ *t-target* $t \notin$ *dones* $\wedge$ *t-target* $t \notin$ *nexts*) (*transitions-as-list* M);
targets = *map t-target usable-transitions*;
transition-choice = *Map.empty*(*targets* [$\mapsto$] *usable-transitions*);
assignment' = *assignment*(*targets* [$\mapsto$] (*map* ($\lambda q'$. *case transition-choice* q' *of Some* $t \Rightarrow$ (*case assignment* (*t-source* t) *of Some* $p \Rightarrow p@[t]$)) *targets*));
nexts' = *set targets*;
dones' = *nexts* $\cup$ *dones*
in reaching-paths-up-to-depth M nexts' dones' assignment' k)

lemma *reaching-paths-up-to-depth-set* :
assumes *nexts* = $\{q$. ($\exists$ p . *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p = n$) $\wedge$ ($\nexists$ p . *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p < n$)$\}$
and *dones* = $\{q$. $\exists$ p . *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p < n\}$
and $\bigwedge$ q . *assignment* q = *None* = ($\nexists p$. *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p \leq n$)
and $\bigwedge$ $q\ p$. *assignment* q = *Some* $p \Longrightarrow$ *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p \leq n$
and *dom assignment* = *nexts* $\cup$ *dones*
shows ((*reaching-paths-up-to-depth M nexts dones assignment k*) q = *None*) = ($\nexists p$. *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p \leq n+k$)
and ((*reaching-paths-up-to-depth M nexts dones assignment k*) q = *Some* p) $\Longrightarrow$ *path* M (*initial* M) p $\wedge$ *target* (*initial* M) $p = q$ $\wedge$ *length* $p \leq n+k$
and $q \in$ *nexts* $\cup$ *dones* $\Longrightarrow$ (*reaching-paths-up-to-depth M nexts dones assignment k*) q = *assignment* q

proof −
have (((*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *None*) = (∄ *p* . *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *q* ∧ *length p* ≤ *n*+*k*))
∧ (((*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *Some p*) ⟶ *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *q* ∧ *length p* ≤ *n*+*k*)
∧ (*q* ∈ *nexts* ∪ *dones* ⟶ (*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *assignment q*)
using *assms* **proof** (*induction k arbitrary*: *n q nexts dones assignment*)
case *0*

have ∗:((*reaching-paths-up-to-depth M nexts dones assignment 0*) *q*) = *assignment q*
by *auto*
show *?case*
unfolding ∗ **using** *0.prems*(*3*,*4*)[*of q*] **by** *simp*
next
case (*Suc k*)

define *usable-transitions* **where** *d1*: *usable-transitions* = *filter* (λ *t* . *t-source t* ∈ *nexts* ∧ *t-target t* ∉ *dones* ∧ *t-target t* ∉ *nexts*) (*transitions-as-list M*)
moreover define *targets* **where** *d2*: *targets* = *map t-target usable-transitions*
moreover define *transition-choice* **where** *d3*: *transition-choice* = *Map.empty*(*targets* [↦] *usable-transitions*)
moreover define *assignment′* **where** *d4*: *assignment′* = *assignment*(*targets* [↦] (*map* (λ*q′* . *case transition-choice q′ of Some t* ⇒ (*case assignment* (*t-source t*) *of Some p* ⇒ *p*@[*t*])) *targets*))
ultimately have *d5*: *reaching-paths-up-to-depth M nexts dones assignment* (*Suc k*) = *reaching-paths-up-to-depth M* (*set targets*) (*nexts* ∪ *dones*) *assignment′ k*
unfolding *reaching-paths-up-to-depth.simps Let-def* **by** *force*

let *?nexts′* = (*set targets*)
let *?dones′* = (*nexts* ∪ *dones*)

have *p1*: *?nexts′* = {*q*. (∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* = *Suc n*) ∧
(∄ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *Suc n*)} (**is** *?nexts′* = *?PS*)
proof −
have ⋀ *q* . *q* ∈ *?nexts′* ⟹ *q* ∈ *?PS*
proof −
fix *q* **assume** *q* ∈ *?nexts′*
then obtain *t* **where** *t* ∈ *transitions M*
and *t-source t* ∈ *nexts*
and *t-target t* = *q*
and *t-target t* ∉ *dones*
and *t-target t* ∉ *nexts*
unfolding *d2 d1* **using** *transitions-as-list-set*[*of M*] **by** *force*

obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *t-source t* **and** *length p* = *n*
using ‹*t-source t* ∈ *nexts*› **unfolding** *Suc.prems* **by** *blast*
then have *path M* (*initial M*) (*p*@[*t*]) **and** *target* (*initial M*) (*p*@[*t*]) = *q*
unfolding ‹*t-target t* = *q*›[*symmetric*] **using** ‹*t* ∈ *transitions M*› **by** *auto*
then have (∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* = *Suc n*)
using ‹*length p* = *n*› **by** (*metis length-append-singleton*)
moreover have (∄ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *Suc n*)
using ‹*t-target t* ∉ *dones*› ‹*t-target t* ∉ *nexts*› **unfolding** ‹*t-target t* = *q*› *Suc.prems*
using *less-antisym* **by** *blast*
ultimately show *q* ∈ *?PS*
by *blast*
qed
moreover have ⋀ *q* . *q* ∈ *?PS* ⟹ *q* ∈ *?nexts′*
proof −
fix *q* **assume** *q* ∈ *?PS*
then obtain *p* **where** *path M* (*initial M*) *p* **and** *target* (*initial M*) *p* = *q* **and** *length p* = *Suc n*
by *auto*

let *?p* = *butlast p*
let *?t* = *last p*

have *p* = *?p*@[*?t*]
using ‹*length p* = *Suc n*›
by (*metis append-butlast-last-id list.size*(*3*) *nat.simps*(*3*))
then have *path M* (*initial M*) (*?p*@[*?t*])
using ‹*path M* (*initial M*) *p*› **by** *auto*

have *path M* (*FSM.initial M*) *?p*
?t ∈ *FSM.transitions M*
t-source ?t = *target* (*FSM.initial M*) *?p*
using *path-append-transition-elim*[*OF* ‹*path M* (*initial M*) (*?p*@[*?t*])›] **by** *blast*+

have *t-target ?t* = *q*
using ‹*target* (*initial M*) *p* = *q*› ‹*p* = *?p*@[*?t*]› **unfolding** *target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) *last-ConsR last-map map-is-Nil-conv snoc-eq-iff-butlast*)

moreover have *t-source ?t* ∈ *nexts*
proof −
have *length ?p* = *n*
using ‹*p* = *?p*@[*?t*]› ‹*length p* = *Suc n*› **by** *auto*
then have (∃ *p* . *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *t-source*

?t ∧ *length p* = *n*)
using ‹*path M* (*FSM.initial M*) *?p*› ‹*t-source ?t* = *target* (*FSM.initial M*) *?p*›
by *metis*
moreover have (∄ *p* . *path M* (*initial M*) *p* ∧ *target* (*initial M*) *p* = *t-source ?t* ∧ *length p* < *n*)
proof
assume ∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *t-source ?t* ∧ *length p* < *n*
then obtain *p′* **where** *path M* (*FSM.initial M*) *p′* **and** *target* (*FSM.initial M*) *p′* = *t-source ?t* **and** *length p′* < *n*
by *blast*
then have *path M* (*initial M*) (*p′*@[*?t*]) **and** *length* (*p′*@[*?t*]) < *Suc n*
using ‹*?t* ∈ *FSM.transitions M*› **by** *auto*
moreover have *target* (*initial M*) (*p′*@[*?t*]) = *q*
using ‹*t-target ?t* = *q*› **by** *auto*
ultimately show *False*
using ‹*q* ∈ *?PS*›
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
qed
ultimately show *?thesis*
unfolding *Suc.prems* **by** *blast*
qed
moreover have *q* ∉ *dones* **and** *q* ∉ *nexts*
unfolding *Suc.prems* **using** ‹*q* ∈ *?PS*›
using *less-SucI* **by** *blast+*
ultimately have *t-source ?t* ∈ *nexts* ∧ *t-target ?t* ∉ *dones* ∧ *t-target ?t* ∉ *nexts*
by *simp*
then show *q* ∈ *?nexts′*
unfolding *d2 d1* **using** *transitions-as-list-set*[*of M*] ‹*?t* ∈ *FSM.transitions M*› ‹*t-target ?t* = *q*›
by *auto*
qed
ultimately show *?thesis*
by *blast*
qed

have *p2*: *?dones′* = {*q*. ∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *Suc n*} (**is** *?dones′* = *?PS*)
proof −
have ⋀ *q* . *q* ∈ *?dones′* ⟹ *q* ∈ *?PS*
unfolding *Suc.prems*
using *less-SucI* **by** *blast*
moreover have ⋀ *q* . *q* ∈ *?PS* ⟹ *q* ∈ *?dones′*
proof −
fix *q* **assume** *q* ∈ *?PS*
show *q* ∈ *?dones′* **proof** (*cases* ∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *n*)

case *True*
then show *?thesis* **unfolding** *Suc.prems* **by** *blast*
next
case *False*

obtain p **where** $*$: *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q$ **and** *length* $p <$ *Suc n*
using ‹$q \in$ *?PS*› **by** *blast*
then have *length* $p = n$
using *False* **by** *force*

then show *?thesis*
using $*$ *False* **unfolding** *Suc.prems* **by** *blast*
qed
qed
ultimately show *?thesis*
by *blast*
qed

have *p3*: ($\bigwedge q$. (*assignment′ q* = *None*) = ($\nexists p$. *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q \wedge$ *length* $p \leq$ *Suc n*))
and *p4*: ($\bigwedge q\ p$. *assignment′ q* = *Some p* $\Longrightarrow$ *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q \wedge$ *length* $p \leq$ *Suc n*)
and *p5*: *dom assignment′* = *?nexts′* $\cup$ *?dones′*
proof −

have *dom transition-choice* = *set targets*
unfolding *d3 d2* **by** *auto*

show *dom assignment′* = *?nexts′* $\cup$ *?dones′*
by (*simp add*: ‹*dom assignment* = *nexts* $\cup$ *dones*› *d4*)

have *helper*: $\bigwedge f\ P$ (*n*::*nat*) . $\{x$. ($\exists$ y . $P\ x\ y \wedge f\ y = n$) $\wedge$ ($\nexists$ y . $P\ x\ y \wedge f\ y < n$)$\}$ $\cup$ $\{x$. ($\exists$ y . $P\ x\ y \wedge f\ y < n$)$\}$= $\{x$. ($\exists$ y . $P\ x\ y \wedge f\ y \leq n$)$\}$
by *force*

have *dom′*: *dom assignment′* = $\{q$. $\exists p$. *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q \wedge$ *length* $p \leq$ *Suc n*$\}$
unfolding ‹*dom assignment′* = *?nexts′* $\cup$ *?dones′*› *p1 p2*
using *helper*[*of* λ *q p* . *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q$ *length Suc n*] **by** *force*

have $*$: $\bigwedge$ q . $q \in$ *?nexts′* $\Longrightarrow$ $\exists$ p . *assignment′ q* = *Some p* $\wedge$ *path M* (*FSM.initial M*) $p \wedge$ *target* (*FSM.initial M*) $p = q \wedge$ *length* $p \leq$ *Suc n*
proof −
fix q **assume** $q \in$ *?nexts′*
then obtain t **where** *transition-choice q* = *Some t*
using ‹*dom transition-choice* = *set targets*› *d2 d3* **by** *blast*

then have $t \in$ *set usable-transitions*
and *t-target* $t = q$
and $q \in$ *set targets*
unfolding *d3 d2* **using** *map-upds-map-set-left*[*of t-target usable-transitions q t*] **by** *auto*
then have *t-source* $t \in$ *nexts* **and** $t \in$ *transitions M*
unfolding *d1* **using** *transitions-as-list-set*[*of M*] **by** *auto*
then obtain *p* **where** *assignment* (*t-source t*) = *Some p*
using *Suc.prems*(*1*,*3*,*4*)
by *fastforce*
then have *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *t-source t* ∧ *length p* ≤ *n*
using *Suc.prems*(*4*) **by** *blast*
then have *path M* (*FSM.initial M*) (*p*@[*t*]) ∧ *target* (*FSM.initial M*) (*p*@[*t*]) = *q* ∧ *length* (*p*@[*t*]) ≤ *Suc n*
using ‹*t* ∈ *transitions M*› ‹*t-target t* = *q*› **by** *auto*
moreover have *assignment′ q* = *Some* (*p*@[*t*])
proof −
have *assignment′ q* = [*targets* [↦] (*map* (λ*q′* . *case transition-choice q′ of Some t* ⇒ (*case assignment* (*t-source t*) *of Some p* ⇒ *p*@[*t*])) *targets*)] *q*
unfolding *d4* **using** *map-upds-overwrite*[*OF* ‹*q* ∈ *set targets*›, *of map* (λ*q′* . *case transition-choice q′ of Some t* ⇒ (*case assignment* (*t-source t*) *of Some p* ⇒ *p*@[*t*])) *targets assignment*]
by *auto*
also have . . . = *Some* (*case transition-choice q of Some t* ⇒ *case assignment* (*t-source t*) *of Some p* ⇒ *p* @ [*t*])
using *map-upds-map-set-right*[*OF* ‹*q* ∈ *set targets*›] **by** *auto*
also have . . . = *Some* (*p*@[*t*])
using ‹*transition-choice q* = *Some t*› ‹*assignment* (*t-source t*) = *Some p*› **by** *simp*
finally show *?thesis* **.**
qed
ultimately show ∃ *p* . *assignment′ q* = *Some p* ∧ *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *Suc n*
by *simp*
qed

show (⋀*q*. (*assignment′ q* = *None*) = (∄*p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *Suc n*))
using *dom′* **by** *blast*

show (⋀*q p*. *assignment′ q* = *Some p* ⟹ *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *Suc n*)
proof −
fix *q p* **assume** *assignment′ q* = *Some p*

show *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *Suc n*
proof (*cases q* ∈ *?nexts′*)

case *True*
show *?thesis* **using** $*$[*OF True*] ‹*assignment′ q = Some p*›
by *simp*
next
case *False*
moreover have $\bigwedge$ *q . assignment q* $\neq$ *assignment′ q* $\Longrightarrow$ *q* $\in$ *?nexts′*
unfolding *d4*
by (*metis* (*no-types*) *map-upds-apply-nontin*)
ultimately have *assignment′ q = assignment q*
by *force*
then show *?thesis*
using *Suc.prems*(*4*) ‹*assignment′ q = Some p*›
by (*simp add*: *le-SucI*)
qed
qed
qed

have $\bigwedge$ *q* . (*reaching-paths-up-to-depth M* (*set targets*) (*nexts* $\cup$ *dones*) *assignment′ k q = None*) =
($\nexists$*p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p = q* $\wedge$ *length p* $\leq$ *n + Suc k*) $\wedge$
(*reaching-paths-up-to-depth M* (*set targets*) (*nexts* $\cup$ *dones*) *assignment′ k q = Some p* $\longrightarrow$
path M (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p = q* $\wedge$ *length p* $\leq$ *n + Suc k*)
using *Suc.IH*[*OF p1 p2 p3 p4 p5*] **by** *auto*

moreover have (*q* $\in$ *nexts* $\cup$ *dones* $\longrightarrow$ *reaching-paths-up-to-depth M nexts dones assignment* (*Suc k*) *q = assignment q*)
proof −
have $\bigwedge$ *q* . (*q* $\in$ *set targets* $\cup$ (*nexts* $\cup$ *dones*) $\Longrightarrow$ *reaching-paths-up-to-depth M* (*set targets*) (*nexts* $\cup$ *dones*) *assignment′ k q = assignment′ q*)
using *Suc.IH*[*OF p1 p2 p3 p4 p5*] **by** *auto*
moreover have $\bigwedge$ *q . assignment q* $\neq$ *assignment′ q* $\Longrightarrow$ *q* $\in$ *?nexts′*
unfolding *d4*
by (*metis* (*no-types*) *map-upds-apply-nontin*)
ultimately show *?thesis*
unfolding *d5*
by (*metis* (*mono-tags*, *lifting*) *Un-iff mem-Collect-eq p1 p2*)
qed
ultimately show *?case*
unfolding *d5* **by** *blast*
qed

then show ((*reaching-paths-up-to-depth M nexts dones assignment k*) *q = None*) = ($\nexists$*p . path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p = q* $\wedge$ *length p* $\leq$ *n*+*k*)
and ((*reaching-paths-up-to-depth M nexts dones assignment k*) *q = Some p*) $\Longrightarrow$ *path M* (*initial M*) *p* $\wedge$ *target* (*initial M*) *p = q* $\wedge$ *length p* $\leq$ *n*+*k*

and $q \in$ *nexts* $\cup$ *dones* $\Longrightarrow$ (*reaching-paths-up-to-depth M nexts dones assignment k*) *q* = *assignment q*
by *blast+*
qed

fun *get-state-cover-assignment* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* **where**
get-state-cover-assignment M = (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* $\mapsto$ []] (*size M* −*1*)
in (λ *q* . *case path-assignments q of Some p* $\Rightarrow$ *p-io p* | *None* $\Rightarrow$ []))

lemma *get-state-cover-assignment-is-state-cover-assignment* :
is-state-cover-assignment M (*get-state-cover-assignment M*)
unfolding *is-state-cover-assignment.simps*
proof

define *path-assignments* **where** *path-assignments* = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* $\mapsto$ []] (*size M* −*1*)
then have $*$:$\bigwedge$ *q* . *get-state-cover-assignment M q* = (*case path-assignments q of Some p* $\Rightarrow$ *p-io p* | *None* $\Rightarrow$ [])
by *auto*

have *c1*: {*FSM.initial M*} =
{*q*. ($\exists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* = *0*) $\wedge$
($\nexists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* < *0*)}
by *auto*
have *c2*: {} = {*q*. $\exists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* < *0*}
by *auto*
have *c3*: ($\bigwedge$*q*. ([*FSM.initial M* $\mapsto$ []] *q* = *None*) =
($\nexists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *0*)) **by** *auto*
have *c4*: ($\bigwedge$*q p*. [*FSM.initial M* $\mapsto$ []] *q* = *Some p* $\Longrightarrow$
path M (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *0*)
by (*metis* (*no-types*, *lifting*) *c3 le-zero-eq length-0-conv map-upd-Some-unfold option.discI*)
have *c5*: *dom* [*FSM.initial M* $\mapsto$ []] = {*FSM.initial M*} $\cup$ {}
by *simp*

have *p1*: $\bigwedge$ *q* . (*path-assignments q* = *None*) =

$(\nexists p.\ path\ M\ (FSM.initial\ M)\ p \wedge target\ (FSM.initial\ M)\ p = q \wedge$
$length\ p \leq (FSM.size\ M - 1))$
and *p2*: $\bigwedge q\ p\ .\ path\text{-}assignments\ q = Some\ p \Longrightarrow$
$path\ M\ (FSM.initial\ M)\ p \wedge target\ (FSM.initial\ M)\ p = q \wedge$
$length\ p \leq (FSM.size\ M - 1)$
and *p3*: $path\text{-}assignments\ (initial\ M) = Some\ []$
unfolding ‹$path\text{-}assignments = reaching\text{-}paths\text{-}up\text{-}to\text{-}depth\ M\ \{initial\ M\}\ \{\}\ [initial\ M \mapsto []]\ (size\ M - 1)$›
using *reaching-paths-up-to-depth-set*[*OF c1 c2 c3 c4 c5*] **by** *auto*

show $get\text{-}state\text{-}cover\text{-}assignment\ M\ (FSM.initial\ M) = []$
unfolding $*$ *p3* **by** *auto*

show $\forall q \in reachable\text{-}states\ M.\ q \in io\text{-}targets\ M\ (get\text{-}state\text{-}cover\text{-}assignment\ M\ q)\ (FSM.initial\ M)$
proof
fix q **assume** $q \in reachable\text{-}states\ M$
then have $q \in reachable\text{-}k\ M\ (FSM.initial\ M)\ (FSM.size\ M - 1)$
using *reachable-k-states* **by** *metis*
then obtain p **where** $target\ (initial\ M)\ p = q$ **and** $path\ M\ (initial\ M)\ p$ **and** $length\ p \leq size\ M - 1$
by *auto*
then have $path\text{-}assignments\ q \neq None$
using *p1* **by** *fastforce*
then obtain p' **where** $get\text{-}state\text{-}cover\text{-}assignment\ M\ q = p\text{-}io\ p'$
and $path\ M\ (FSM.initial\ M)\ p'$ **and** $target\ (FSM.initial\ M)\ p' = q$
using *p2* **unfolding** $*$ **by** *force*
then show $q \in io\text{-}targets\ M\ (get\text{-}state\text{-}cover\text{-}assignment\ M\ q)\ (initial\ M)$
unfolding *io-targets.simps* **unfolding** ‹$get\text{-}state\text{-}cover\text{-}assignment\ M\ q = p\text{-}io\ p'$› **by** *blast*
qed
qed

12.3 Computing Reachable States via State Cover Computation

lemma *restrict-to-reachable-states*[*code*]:
$restrict\text{-}to\text{-}reachable\text{-}states\ M = (let$
$path\text{-}assignments = reaching\text{-}paths\text{-}up\text{-}to\text{-}depth\ M\ \{initial\ M\}\ \{\}\ [initial\ M \mapsto []]\ (size\ M - 1)$
$in\ filter\text{-}states\ M\ (\lambda\ q\ .\ path\text{-}assignments\ q \neq None))$
proof −
define $path\text{-}assignments$ **where** $path\text{-}assignments = reaching\text{-}paths\text{-}up\text{-}to\text{-}depth\ M\ \{initial\ M\}\ \{\}\ [initial\ M \mapsto []]\ (size\ M - 1)$
then have $*$: $(let$
$path\text{-}assignments = reaching\text{-}paths\text{-}up\text{-}to\text{-}depth\ M\ \{initial\ M\}\ \{\}\ [initial\ M \mapsto []]\ (size\ M - 1)$
$in\ filter\text{-}states\ M\ (\lambda\ q\ .\ path\text{-}assignments\ q \neq None)) = filter\text{-}states\ M\ (\lambda\ q\ .$

path-assignments q $\neq$ *None*)
by *simp*

have *c1*: $\{$*FSM.initial M*$\}$ =
$\{$*q*. ($\exists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* = *0*) $\wedge$
($\nexists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* < *0*)$\}$
by *auto*
have *c2*: $\{\}$ = $\{$*q*. $\exists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* < *0*$\}$
by *auto*
have *c3*: ($\bigwedge$*q*. ([*FSM.initial M* $\mapsto$ []] *q* = *None*) =
($\nexists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *0*)) **by** *auto*
have *c4*: ($\bigwedge$*q p*. [*FSM.initial M* $\mapsto$ []] *q* = *Some p* $\Longrightarrow$
path M (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ *0*)
by (*metis* (*no-types*, *lifting*) *c3 le-zero-eq length-0-conv map-upd-Some-unfold option.discI*)
have *c5*: *dom* [*FSM.initial M* $\mapsto$ []] = $\{$*FSM.initial M*$\}$ $\cup$ $\{\}$
by *simp*

have *p1*: $\bigwedge$ *q* . (*path-assignments q* = *None*) =
($\nexists$ *p*. *path M* (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ (*FSM.size M* − *1*))
and *p2*: $\bigwedge$ *q p* . *path-assignments q* = *Some p* $\Longrightarrow$
path M (*FSM.initial M*) *p* $\wedge$ *target* (*FSM.initial M*) *p* = *q* $\wedge$ *length p* $\leq$ (*FSM.size M* − *1*)
and *p3*: *path-assignments* (*initial M*) = *Some* []
unfolding ‹*path-assignments* = *reaching-paths-up-to-depth M* $\{$*initial M*$\}$ $\{\}$ [*initial M* $\mapsto$ []] (*size M* −*1*)›
using *reaching-paths-up-to-depth-set*[*OF c1 c2 c3 c4 c5*] **by** *auto*

have $\bigwedge$ *q* . *path-assignments q* $\neq$ *None* $\longleftrightarrow$ *q* $\in$ *reachable-states M*
proof
show $\bigwedge$*q*. *path-assignments q* $\neq$ *None* $\Longrightarrow$ *q* $\in$ *reachable-states M*
using *p2* **unfolding** *reachable-states-def*
by *blast*
show $\bigwedge$*q*. *q* $\in$ *reachable-states M* $\Longrightarrow$ *path-assignments q* $\neq$ *None*
proof −
fix *q* **assume** *q* $\in$ *reachable-states M*
then have *q* $\in$ *reachable-k M* (*FSM.initial M*) (*FSM.size M* − *1*)
using *reachable-k-states* **by** *metis*
then obtain *p* **where** *target* (*initial M*) *p* = *q* **and** *path M* (*initial M*) *p* **and** *length p* $\leq$ *size M* − *1*
by *auto*
then show *path-assignments q* $\neq$ *None*

using *p1* **by** *fastforce*
qed
qed
then show *?thesis*
unfolding *restrict-to-reachable-states.simps* $*$ **by** *simp*
qed

lemma *reachable-states-refined*[*code*] :
reachable-states M = (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
in Set.filter (λ *q* . *path-assignments q* ≠ *None*) (*states M*))
proof −
define *path-assignments* **where** *path-assignments* = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
then have $*$: (*let*
path-assignments = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)
in Set.filter (λ *q* . *path-assignments q* ≠ *None*) (*states M*)) = *Set.filter* (λ *q* . *path-assignments q* ≠ *None*) (*states M*)
by *simp*

have *c1*: {*FSM.initial M*} =
{*q*. (∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* = *0*) ∧
(∄ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *0*)}
by *auto*
have *c2*: {} = {*q*. ∃ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* < *0*}
by *auto*
have *c3*: (⋀*q*. ([*FSM.initial M* ↦ []] *q* = *None*) =
(∄ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *0*)) **by** *auto*
have *c4*: (⋀*q p*. [*FSM.initial M* ↦ []] *q* = *Some p* ⟹
path M (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ *0*)
by (*metis* (*no-types*, *lifting*) *c3 le-zero-eq length-0-conv map-upd-Some-unfold option.discI*)
have *c5*: *dom* [*FSM.initial M* ↦ []] = {*FSM.initial M*} ∪ {}
by *simp*

have *p1*: ⋀ *q* . (*path-assignments q* = *None*) =
(∄ *p*. *path M* (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ (*FSM.size M* − *1*))
and *p2*: ⋀ *q p* . *path-assignments q* = *Some p* ⟹
path M (*FSM.initial M*) *p* ∧ *target* (*FSM.initial M*) *p* = *q* ∧ *length p* ≤ (*FSM.size M* − *1*)
and *p3*: *path-assignments* (*initial M*) = *Some* []

unfolding ‹*path-assignments* = *reaching-paths-up-to-depth M* {*initial M*} {} [*initial M* ↦ []] (*size M* −*1*)›
using *reaching-paths-up-to-depth-set*[*OF c1 c2 c3 c4 c5*] **by** *auto*

have ⋀ *q* . *path-assignments q* ≠ *None* ⟷ *q* ∈ *reachable-states M*
proof
show ⋀*q*. *path-assignments q* ≠ *None* ⟹ *q* ∈ *reachable-states M*
using *p2* **unfolding** *reachable-states-def*
by *blast*
show ⋀*q*. *q* ∈ *reachable-states M* ⟹ *path-assignments q* ≠ *None*
proof −
fix *q* **assume** *q* ∈ *reachable-states M*
then have *q* ∈ *reachable-k M* (*FSM.initial M*) (*FSM.size M* − *1*)
using *reachable-k-states* **by** *metis*
then obtain *p* **where** *target* (*initial M*) *p* = *q* **and** *path M* (*initial M*) *p* **and** *length p* ≤ *size M* − *1*
by *auto*
then show *path-assignments q* ≠ *None*
using *p1* **by** *fastforce*
qed
qed
then show *?thesis*
unfolding ∗ **using** *reachable-state-is-state* **by** *force*
qed

lemma *minimal-sequence-to-failure-from-state-cover-assignment-ob* :
assumes *L M* ≠ *L I*
and *is-state-cover-assignment M V*
and (*L M* ∩ (*V* ‘ *reachable-states M*)) = (*L I* ∩ (*V* ‘ *reachable-states M*))
obtains *ioT ioX* **where** *ioT* ∈ (*V* ‘ *reachable-states M*)
and *ioT* @ *ioX* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*)
and ⋀ *io q* . *q* ∈ *reachable-states M* ⟹ (*V q*)@*io* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*) ⟹ *length ioX* ≤ *length io*
proof −

let *?exts* = {*io* . ∃ *q* ∈ *reachable-states M* . (*V q*)@*io* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*)}
define *exMin* **where** *exMin*: *exMin* = *arg-min length* (λ *io* . *io* ∈ *?exts*)

have *V* (*initial M*) = []
using *assms*(*2*) **by** *auto*
moreover have ∃ *io* . *io* ∈ (*L M* − *L I*) ∪ (*L I* − *L M*)
using *assms*(*1*) **by** *blast*
ultimately have *?exts* ≠ {}
using *reachable-states-initial* **by** (*metis* (*mono-tags*, *lifting*) *append-self-conv2 empty-iff mem-Collect-eq*)
then have *exMin* ∈ *?exts* ∧ (∀ *io′* . *io′* ∈ *?exts* ⟶ *length exMin* ≤ *length io′*)

using *exMin arg-min-nat-lemma* **by** (*metis* (*no-types*, *lifting*) *all-not-in-conv*)
then show *?thesis*
using *that* **by** *blast*
qed

end

13 Alternative OFSM Table Computation

The approach to computing OFSM tables presented in the imported theories is easy to use in proofs but inefficient in practice due to repeated recomputation of the same tables. Thus, in the following we present a more efficient method for computing and storing tables.

theory *OFSM-Tables-Refined*
imports *Minimisation Distinguishability*
begin

13.1 Computing a List of all OFSM Tables

type-synonym ('a,'b,'c) *ofsm-table* = ('a, 'a *set*) *mapping*

fun *initial-ofsm-table* :: ('a::*linorder*,'b,'c) *fsm* ⇒ ('a,'b,'c) *ofsm-table* **where**
initial-ofsm-table M = *Mapping.tabulate* (*states-as-list M*) (λ*q* . *states M*)

abbreviation *ofsm-lookup* ≡ *Mapping.lookup-default* {}

lemma *initial-ofsm-table-lookup-invar*: *ofsm-lookup* (*initial-ofsm-table M*) *q* = *ofsm-table M* (λ*q* . *states M*) *0 q*
proof (*cases q* ∈ *states M*)
case *True*
then have *q* ∈ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *auto*
then have *Mapping.lookup* (*initial-ofsm-table M*) *q* = *Some* (*states M*)
unfolding *initial-ofsm-table.simps*
by (*simp add*: *lookup-tabulate*)
then have *ofsm-lookup* (*initial-ofsm-table M*) *q* = *states M*
by (*simp add*: *lookup-default-def*)
then show *?thesis*
using *True* **by** *auto*
next
case *False*
then have *q* ∉ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *auto*
then have *Mapping.lookup* (*initial-ofsm-table M*) *q* = *None*
unfolding *initial-ofsm-table.simps*
by (*simp add*: *lookup-tabulate*)

then have *ofsm-lookup* (*initial-ofsm-table M*) *q* = {}
by (*simp add*: *lookup-default-def*)
then show *?thesis*
using *False* **by** *auto*
qed

lemma *initial-ofsm-table-keys-invar*: *Mapping.keys* (*initial-ofsm-table M*) = *states M*
using *states-as-list-set*[*of M*]
by *simp*

fun *next-ofsm-table* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *ofsm-table* ⇒ ('*a*,'*b*,'*c*) *ofsm-table* **where**
next-ofsm-table M prev-table = *Mapping.tabulate* (*states-as-list M*) (λ *q* . {*q*′ ∈ *ofsm-lookup prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q*′ *x y of Some qT*′ ⇒ *ofsm-lookup prev-table qT* = *ofsm-lookup prev-table qT*′ | *None* ⇒ *False*) | *None* ⇒ *h-obs M q*′ *x y* = *None*) })

lemma *h-obs-non-state* :
assumes *q* ∉ *states M*
shows *h-obs M q x y* = *None*
proof −
have ∗:⋀ *x* . *h M* (*q*,*x*) = {}
using *assms fsm-transition-source*
unfolding *h-simps*
by *force*
show *?thesis*
unfolding *h-obs-simps Let-def* ∗
by (*simp add*:)
qed

lemma *next-ofsm-table-lookup-invar*:
assumes ⋀ *q* . *ofsm-lookup prev-table q* = *ofsm-table M* (λ*q* . *states M*) *k q*
shows *ofsm-lookup* (*next-ofsm-table M prev-table*) *q* = *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q*
proof (*cases q* ∈ *states M*)
case *True*

let *?prev-table* = *ofsm-table M* (λ*q* . *states M*) *k*

from *True* **have** *q* ∈ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *auto*
then have *Mapping.lookup* (*next-ofsm-table M prev-table*) *q* = *Some* {*q*′ ∈ *ofsm-lookup prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q*′ *x y of Some qT*′ ⇒ *ofsm-lookup prev-table qT* = *ofsm-lookup prev-table qT*′ | *None* ⇒ *False*) | *None* ⇒ *h-obs M q*′ *x y* =

None) }
unfolding *next-ofsm-table.simps*
by (*meson lookup-tabulate states-as-list-distinct*)
then have *ofsm-lookup* (*next-ofsm-table M prev-table*) *q* = {*q'* ∈ *ofsm-lookup prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q' x y of Some qT'* ⇒ *ofsm-lookup prev-table qT* = *ofsm-lookup prev-table qT'* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q' x y* = *None*) }
by (*simp add*: *lookup-default-def*)
also have ... = {*q'* ∈ *?prev-table q* . ∀ *x* ∈ *inputs M* . ∀ *y* ∈ *outputs M* . (*case h-obs M q x y of Some qT* ⇒ (*case h-obs M q' x y of Some qT'* ⇒ *?prev-table qT* = *?prev-table qT'* | *None* ⇒ *False*) | *None* ⇒ *h-obs M q' x y* = *None*) }
unfolding *assms* **by** *presburger*
also have ... = *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q*
unfolding *ofsm-table.simps Let-def* **by** *presburger*
finally show *?thesis* **.**
next
case *False*
then have *q* ∉ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *auto*
then have *Mapping.lookup* (*next-ofsm-table M prev-table*) *q* = *None*
by (*simp add*: *lookup-tabulate*)
then have *ofsm-lookup* (*next-ofsm-table M prev-table*) *q* = {}
by (*simp add*: *lookup-default-def*)
then show *?thesis*
unfolding *ofsm-table-non-state*[*OF False*] **.**
qed

lemma *next-ofsm-table-keys-invar*: *Mapping.keys* (*next-ofsm-table M prev-table*) = *states M*
using *states-as-list-set*[*of M*]
by *simp*

fun *compute-ofsm-table-list* :: (*'a::linorder,'b,'c*) *fsm* ⇒ *nat* ⇒ (*'a,'b,'c*) *ofsm-table list* **where**
compute-ofsm-table-list M k = *rev* (*foldr* (λ - *prev* . (*next-ofsm-table M* (*hd prev*)) # *prev*) [*0..<k*] [*initial-ofsm-table M*])

lemma *compute-ofsm-table-list-props*:
length (*compute-ofsm-table-list M k*) = *Suc k*
⋀ *i q* . *i* < *Suc k* ⟹ *ofsm-lookup* ((*compute-ofsm-table-list M k*) ! *i*) *q* = *ofsm-table M* (λ*q* . *states M*) *i q*
⋀ *i* . *i* < *Suc k* ⟹ *Mapping.keys* ((*compute-ofsm-table-list M k*) ! *i*) = *states M*
proof −

define *t* **where** *t* = (λ *k* . (*foldr* (λ - *prev* . (*next-ofsm-table M* (*hd prev*)) # *prev*) (*rev* [*0..<k*]) [*initial-ofsm-table M*]))

```
  have t-props:length (t k) = Suc k
        ∧ (∀ i q . i < Suc k ⟶ ofsm-lookup (t k ! (k−i)) q = ofsm-table M (λq
. states M) i q)
        ∧ (∀ i . i < Suc k ⟶ Mapping.keys (t k ! i) = states M)
  proof (induction k)
    case 0
    have t 0 = [initial-ofsm-table M]
      unfolding t-def by auto
    show ?case
      unfolding ‹t 0 = [initial-ofsm-table M]›
      using initial-ofsm-table-lookup-invar[of M]
      using initial-ofsm-table-keys-invar[of M]
      by auto
  next
    case (Suc k)

    have rev [0..<Suc k] = k # (rev [0..<k])
      by auto
    have ∗: t (Suc k) = (next-ofsm-table M (hd (t k))) # (t k)
      unfolding t-def ‹rev [0..<Suc k] = k # (rev [0..<k])›
      by auto

    have IH1: length (t k) = Suc k
     and IH2: ⋀i q . i < Suc k ⟹ ofsm-lookup (t k ! (k−i)) q = ofsm-table M
(λq. FSM.states M) i q
    and IH3: ⋀i . i < Suc k ⟹ Mapping.keys (t k ! i) = FSM.states M
      using Suc.IH by blast+

    have length (t (Suc k)) = Suc (Suc k)
      using IH1 unfolding ∗ by auto
     moreover have ⋀i q . i < Suc (Suc k) ⟹ ofsm-lookup (t (Suc k) ! ((Suc
k)−i)) q = ofsm-table M (λq. FSM.states M) i q
    proof −
      fix i q assume i < Suc (Suc k)
      then consider i = Suc k | i < Suc k
        using less-Suc-eq by blast
       then show ofsm-lookup (t (Suc k) ! ((Suc k)−i)) q = ofsm-table M (λq.
FSM.states M) i q proof cases
        case 1
        then have (t (Suc k) ! ((Suc k)−i)) = hd (t (Suc k))
          by (metis ∗ diff-self-eq-0 list.sel(1) nth-Cons-0)
        then have (t (Suc k) ! ((Suc k)−i)) = next-ofsm-table M (hd (t k))
          unfolding ∗ by (metis list.sel(1))
      then have ofsm-lookup (t (Suc k) ! ((Suc k)−i)) q = ofsm-lookup (next-ofsm-table
M (hd (t k))) q
          by auto

        have (hd (t k)) = (t k ! (k−k))
          by (metis IH1 diff-self-eq-0 hd-conv-nth list.size(3) nat.simps(3))
```

moreover have $k < Suc\ k$ **by** *auto*
ultimately have *ofsm-lookup* (*next-ofsm-table M* (*hd* (*t k*))) *q* = *ofsm-table M* (λ*q*. *FSM.states M*) *i q*
by (*metis 1 IH2 next-ofsm-table-lookup-invar*)
then show *?thesis*
unfolding ‹*ofsm-lookup* (*t* (*Suc k*) ! ((*Suc k*)−*i*)) *q* = *ofsm-lookup* (*next-ofsm-table M* (*hd* (*t k*))) *q*› **.**
next
case *2*
then have ((*Suc k*)−*i*) > *0*
by *auto*
then have (*t* (*Suc k*) ! ((*Suc k*)−*i*)) = *t k* ! (((*Suc k*)−*i*) − *1*)
unfolding ∗ **by** (*meson nth-Cons-pos*)
then have (*t* (*Suc k*) ! ((*Suc k*)−*i*)) = *t k* ! (*k*−*i*)
by *auto*
show *ofsm-lookup* (*t* (*Suc k*) ! ((*Suc k*)−*i*)) *q* = *ofsm-table M* (λ*q*. *FSM.states M*) *i q*
using *IH2*[*OF 2*]
unfolding ‹(*t* (*Suc k*) ! ((*Suc k*)−*i*)) = *t k* ! (*k*−*i*)› **by** *metis*
qed
qed
moreover have ⋀*i* . *i* < *Suc* (*Suc k*) ⟹ *Mapping.keys* (*t* (*Suc k*) ! *i*) = *FSM.states M*
by (*metis* ∗ *IH3 Suc-diff-1 Suc-less-eq less-Suc-eq-0-disj next-ofsm-table-keys-invar nth-Cons′*)
ultimately show *?case*
by *blast*
qed

have ∗:(*compute-ofsm-table-list M k*) = *rev* (*t k*)
unfolding *compute-ofsm-table-list.simps t-def*
using *foldr-length-helper*[*of rev* [*0*..<*k*] [*0*..<*k*] (λ *prev* . (*next-ofsm-table M* (*hd prev*)) # *prev*), *OF length-rev*]
by *metis*

show *length* (*compute-ofsm-table-list M k*) = *Suc k*
using *t-props* **unfolding** ∗ *length-rev* **by** *blast*

have ⋀ *i* . *i* < *Suc k* ⟹ (*rev* (*t k*) ! *i*) = *t k* ! (*k* − *i*)
by (*simp add*: *rev-nth t-props*)
then show ⋀*i q*. *i* < *Suc k* ⟹
ofsm-lookup (*compute-ofsm-table-list M k* ! *i*) *q* = *ofsm-table M* (λ*q*. *FSM.states M*) *i q*
unfolding ∗ **using** *t-props*
by *presburger*

show ⋀*i*. *i* < *Suc k* ⟹ *Mapping.keys* (*compute-ofsm-table-list M k* ! *i*) = *FSM.states M*

unfolding ∗ **using** *t-props* ‹$\bigwedge$ *i* . *i* < *Suc k* ⟹ (*rev* (*t k*) ! *i*) = *t k* ! (*k* − *i*)›
by *simp*
qed

fun *compute-ofsm-tables* :: ('*a*::*linorder*,'*b*,'*c*) *fsm* ⇒ *nat* ⇒ (*nat*, ('*a*,'*b*,'*c*) *ofsm-table*) *mapping* **where**
compute-ofsm-tables M k = *Mapping.bulkload* (*compute-ofsm-table-list M k*)

lemma *compute-ofsm-tables-entries* :
assumes *i* < *Suc k*
shows (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) = ((*compute-ofsm-table-list M k*) ! *i*)
using *assms*
unfolding *compute-ofsm-tables.simps bulkload-def*
by (*metis bulkload.rep-eq bulkload-def compute-ofsm-table-list-props*(*1*) *lookup.rep-eq option.sel*)

lemma *compute-ofsm-tables-lookup-invar* :
assumes *i* < *Suc k*
shows *ofsm-lookup* (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) *q* = *ofsm-table M* (λ*q* . *states M*) *i q*
using *compute-ofsm-table-list-props*(*2*)[*OF assms*]
unfolding *compute-ofsm-tables-entries*[*OF assms*] **by** *metis*

lemma *compute-ofsm-tables-keys-invar* :
assumes *i* < *Suc k*
shows *Mapping.keys* (*the* (*Mapping.lookup* (*compute-ofsm-tables M k*) *i*)) = *states M*
using *compute-ofsm-table-list-props*(*3*)[*OF assms*]
unfolding *compute-ofsm-tables-entries*[*OF assms*] **by** *metis*

13.2 Finding Diverging Tables

lemma *ofsm-table-fix-from-compute-ofsm-tables* :
assumes *q* ∈ *states M*
shows *ofsm-lookup* (*the* (*Mapping.lookup* (*compute-ofsm-tables M* (*size M* − *1*)) (*size M* − *1*))) *q* = *ofsm-table-fix M* (λ*q*. *FSM.states M*) *0 q*
proof −
have ((λ*q*. *FSM.states M*) ‘ *FSM.states M*) = {*states M*}
using *fsm-initial*[*of M*] **by** *auto*
then have *card* ((λ*q*. *FSM.states M*) ‘ *FSM.states M*) = *1*
by *auto*

have *ofsm-lookup* (*the* (*Mapping.lookup* (*compute-ofsm-tables M* (*size M* − *1*)) (*size M* − *1*))) *q* = *ofsm-table M* (λ*q*. *FSM.states M*) (*FSM.size M* − *1*) *q*
using *compute-ofsm-tables-lookup-invar*[*of* (*size M* − *1*) (*size M* − *1*) *M q*]

by *linarith*
also have ... = *ofsm-table-fix M* (λq. *FSM.states M*) *0 q*
using *ofsm-table-fix-partition-fixpoint*[*OF minimise-initial-partition - assms*(*1*), *of size M*]
unfolding ‹*card* ((λq. *FSM.states M*) ‘ *FSM.states M*) = *1*›
by *blast*
finally show *?thesis* **.**
qed

fun *find-first-distinct-ofsm-table′* :: (*′a*::*linorder*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ *nat* **where**
find-first-distinct-ofsm-table′ M q1 q2 = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*))
in if (*q1* ∈ *states M*
∧ *q2* ∈ *states M*
∧ (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q1*
≠ *ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))) *q2*))
then the (*find-index* (λ *i* . *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* ≠ *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q2*) [*0*..<*size M*])
else 0)

lemma *find-first-distinct-ofsm-table-is-first′* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*
and *ofsm-table-fix M* (λq . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λq . *states M*) *0 q2*
shows (*find-first-distinct-ofsm-table M q1 q2*) = *Min* {*k* . *ofsm-table M* (λq . *states M*) *k q1* ≠ *ofsm-table M* (λq . *states M*) *k q2*
∧ (∀ *k′* . *k′* < *k* ⟶ *ofsm-table M* (λq . *states M*) *k′ q1* = *ofsm-table M* (λq . *states M*) *k′ q2*)}
(**is** *find-first-distinct-ofsm-table M q1 q2* = *Min ?ks*)
proof −
have *find-first-distinct-ofsm-table M q1 q2* ∈ *?ks*
using *find-first-distinct-ofsm-table-is-first*[*OF assms*]
by *blast*
moreover have ⋀ *k* . *k* ∈ *?ks* ⟹ *k* = *find-first-distinct-ofsm-table M q1 q2*
using *calculation linorder-neqE-nat* **by** *blast*
ultimately have *?ks* = {*find-first-distinct-ofsm-table M q1 q2*}
by *blast*
then show *?thesis*
by *fastforce*
qed

lemma *find-first-distinct-ofsm-table′-is-first′* :
assumes *q1* ∈ *FSM.states M*
and *q2* ∈ *FSM.states M*

and *ofsm-table-fix M* (λq . *states M*) *0 q1* $\neq$ *ofsm-table-fix M* (λq . *states M*) *0 q2*
shows (*find-first-distinct-ofsm-table′ M q1 q2*) = *Min* {*k* . *ofsm-table M* (λq . *states M*) *k q1* $\neq$ *ofsm-table M* (λq . *states M*) *k q2*
$\wedge$ ($\forall k'$. $k' < k \longrightarrow$ *ofsm-table M* (λq . *states M*) *k′ q1* = *ofsm-table M* (λq . *states M*) *k′ q2*)}
(**is** *find-first-distinct-ofsm-table′ M q1 q2* = *Min ?ks*)
and *find-first-distinct-ofsm-table′ M q1 q2* $\leq$ *size M* − *1*
proof −

define *tables* **where** *tables* = *compute-ofsm-tables M* (*FSM.size M* − *1*)

have *ofsm-lookup* (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q2*
unfolding *tables-def*
unfolding *ofsm-table-fix-from-compute-ofsm-tables*[*OF assms*(*1*)]
unfolding *ofsm-table-fix-from-compute-ofsm-tables*[*OF assms*(*2*)]
using *assms*(*3*) **.**

then have *find-first-distinct-ofsm-table′ M q1 q2* = *the* (*find-index*
(λi. *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables i*)) *q2*)
[*0*..<*FSM.size M*])
unfolding *find-first-distinct-ofsm-table′.simps*
using *assms*(*1*,*2*,*3*)
unfolding *Let-def tables-def*[*symmetric*]
by *presburger*

have *FSM.size M* − *1* $\in$ *set* [*0*..<*FSM.size M*]
using *fsm-size-Suc*[*of M*] **by** *auto*
then have $*$:$\exists$ *k* $\in$ *set* [*0*..<*FSM.size M*] . (λi. *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables i*)) *q2*) *k*
using ‹*ofsm-lookup* (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q2*›
by *blast*
have *find-index*
(λi. *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables i*)) *q2*)
[*0*..<*FSM.size M*] $\neq$ *None*
using *find-index-exhaustive*[*OF* $*$] **.**
then obtain *k* **where** $*$:*find-index*
(λi. *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* $\neq$
ofsm-lookup (*the* (*Mapping.lookup tables i*)) *q2*)
[*0*..<*FSM.size M*] = *Some k*
by *blast*
then have *find-first-distinct-ofsm-table′ M q1 q2* = *k*
unfolding ‹*find-first-distinct-ofsm-table′ M q1 q2* = *the* (*find-index*
(λi. *ofsm-lookup* (*the* (*Mapping.lookup tables i*)) *q1* $\neq$

ofsm-lookup (*the* (*Mapping.lookup tables i*)) *q2*)
[*0..<FSM.size M*])›
by *auto*

have $\bigwedge$ *k′* . *k′* ≤ *k* ⟹ [*0..<FSM.size M*] ! *k′* = *k′*
using *find-index-index*(*1*)[*OF* ∗]
by (*metis add.left-neutral diff-zero dual-order.trans length-upt not-le nth-upt*)
then have [*0..<FSM.size M*] ! *k* = *k* **and** $\bigwedge$ *k′* . *k′* < *k* ⟹ [*0..<FSM.size M*] ! *k′* = *k′*
by *auto*
have *k* < *Suc* (*size M* − *1*)
using *find-index-index*(*1*)[*OF* ∗]
by *auto*

have *ofsm-lookup* (*the* (*Mapping.lookup tables k*)) *q1* ≠ *ofsm-lookup* (*the* (*Mapping.lookup tables k*)) *q2*
using *find-index-index*(*2*)[*OF* ∗]
unfolding ‹[*0..<FSM.size M*] ! *k* = *k*› **.**
then have *p1*: *ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
unfolding *tables-def*
unfolding *compute-ofsm-tables-lookup-invar*[*OF* ‹*k* < *Suc* (*size M* − *1*)›] **.**

have $\bigwedge$ *k′* . *k′* < *k* ⟹ *ofsm-lookup* (*the* (*Mapping.lookup tables k′*)) *q1* = *ofsm-lookup* (*the* (*Mapping.lookup tables k′*)) *q2*
using ‹$\bigwedge$ *k′* . *k′* < *k* ⟹ [*0..<FSM.size M*] ! *k′* = *k′*›
using *find-index-index*(*3*)[*OF* ∗]
by *auto*
then have *p2*: (∀ *k′* . *k′* < *k* ⟶ *ofsm-table M* (λ*q* . *states M*) *k′ q1* = *ofsm-table M* (λ*q* . *states M*) *k′ q2*)
unfolding *tables-def*
using *compute-ofsm-tables-lookup-invar*[*of* - (*size M* − *1*) *M*] ‹*k* < *Suc* (*size M* − *1*)›
using *less-trans* **by** *blast*

have *k* ∈ *?ks*
using *p1 p2* **by** *blast*
moreover have $\bigwedge$ *k′* . *k′* ∈ *?ks* ⟹ *k′* = *k*
using *calculation linorder-neqE-nat* **by** *blast*
ultimately have *?ks* = {*k*}
by *blast*
then show *find-first-distinct-ofsm-table′ M q1 q2* = *Min ?ks*
unfolding ‹*find-first-distinct-ofsm-table′ M q1 q2* = *k*›
by *fastforce*

show *find-first-distinct-ofsm-table′ M q1 q2* ≤ *FSM.size M* − *1*
unfolding ‹*find-first-distinct-ofsm-table′ M q1 q2* = *k*›
using ‹*k* < *Suc* (*size M* − *1*)›

```
    by auto
qed

lemma find-first-distinct-ofsm-table'-max :
  find-first-distinct-ofsm-table' M q1 q2 ≤ size M − 1
proof (cases q1 ∈ states M
      ∧ q2 ∈ states M
      ∧ (ofsm-lookup (the (Mapping.lookup (compute-ofsm-tables M (size M − 1))
(size M − 1))) q1
        ≠ ofsm-lookup (the (Mapping.lookup (compute-ofsm-tables M (size M − 1))
(size M − 1))) q2))
  case True
  then show ?thesis using find-first-distinct-ofsm-table'-is-first'(2)[of q1 M q2]
    using ofsm-table-fix-from-compute-ofsm-tables by blast
next
  case False
  then have find-first-distinct-ofsm-table' M q1 q2 = 0
    unfolding find-first-distinct-ofsm-table'.simps Let-def by meson
  then show ?thesis
    by linarith
qed



lemma find-first-distinct-ofsm-table-alt-def:
  find-first-distinct-ofsm-table M q1 q2 = find-first-distinct-ofsm-table' M q1 q2
proof (cases q1 ∈ states M ∧ q2 ∈ states M ∧ ((ofsm-table-fix M (λq . states M)
0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2)))
  case True
  then have **: q1 ∈ states M
      and ***: q2 ∈ states M
      and ****: (ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq .
states M) 0 q2)
    by blast+
  show ?thesis
    unfolding find-first-distinct-ofsm-table'-is-first'[OF ** *** ****]
    unfolding find-first-distinct-ofsm-table-is-first'[OF ** *** ****]
    by presburger
next
  case False

  have find-first-distinct-ofsm-table M q1 q2 = 0
    by (meson False find-first-distinct-ofsm-table-gt.simps)
  moreover have find-first-distinct-ofsm-table' M q1 q2 = 0
  proof (cases q1 ∈ states M ∧ q2 ∈ states M)
    case True
    then have **: q1 ∈ states M
        and ***: q2 ∈ states M
      by blast+
```

then have ****:((*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* = *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*))
using *False* **by** *blast*

define *tables* **where** *tables* = *compute-ofsm-tables M* (*FSM.size M* − *1*)

have *ofsm-lookup* (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q1* =
ofsm-lookup (*the* (*Mapping.lookup tables* (*FSM.size M* − *1*))) *q2*
unfolding *tables-def*
unfolding *ofsm-table-fix-from-compute-ofsm-tables*[*OF* **]
unfolding *ofsm-table-fix-from-compute-ofsm-tables*[*OF* ***]
using **** **.**
then show *?thesis*
unfolding *find-first-distinct-ofsm-table′.simps Let-def tables-def*[*symmetric*]
by *auto*
next
case *False*
then show *?thesis*
unfolding *find-first-distinct-ofsm-table′.simps Let-def*
by *meson*
qed
ultimately show *?thesis*
by *presburger*
qed

13.3 Refining the Computation of Distinguishing Traces via OFSM Tables

```
fun select-diverging-ofsm-table-io′ :: ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'a
⇒ 'a ⇒ nat ⇒ ('b × 'c) × ('a option × 'a option) where
  select-diverging-ofsm-table-io′ M q1 q2 k = (let
      tables = (compute-ofsm-tables M (size M − 1));
      ins = inputs-as-list M;
      outs = outputs-as-list M;
      table = ofsm-lookup (the (Mapping.lookup tables (k−1)));
      f = (λ (x,y) . case (h-obs M q1 x y, h-obs M q2 x y)
                 of
                    (Some q1′, Some q2′)  ⇒ if table q1′ ≠ table q2′
                                            then Some ((x,y),(Some q1′, Some q2′))
                                            else None |
                    (None,None)           ⇒ None |
                    (Some q1′, None)      ⇒ Some ((x,y),(Some q1′, None)) |
                    (None, Some q2′)      ⇒ Some ((x,y),(None, Some q2′)))
      in
        hd (List.map-filter f (List.product ins outs)))

lemma select-diverging-ofsm-table-io-alt-def :
  assumes k ≤ size M − 1
  shows select-diverging-ofsm-table-io M q1 q2 k = select-diverging-ofsm-table-io′
```

M q1 q2 k
proof −
  define *tables* **where** *tables* = *compute-ofsm-tables M* (*FSM.size M* − *1*)
  define *table* **where** *table* = *ofsm-lookup* (*the* (*Mapping.lookup tables* (*k*−*1*)))

  have *k* − *1* < *Suc* (*size M* − *1*)
    using *assms* **by** *auto*
  have *ofsm-table M* (λ*q* . *states M*) (*k*−*1*) = *table*
    unfolding *table-def tables-def*
    unfolding *compute-ofsm-tables-lookup-invar*[*OF* ‹*k* − *1* < *Suc* (*size M* − *1*)›]
    by *presburger*

  show *?thesis*
    unfolding *select-diverging-ofsm-table-io′.simps*
          select-diverging-ofsm-table-io.simps
          Let-def
    unfolding *tables-def*[*symmetric*] *table-def*[*symmetric*]
    unfolding ‹*ofsm-table M* (λ*q* . *states M*) (*k*−*1*) = *table*›
    by *meson*
qed

fun *assemble-distinguishing-sequence-from-ofsm-table′* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ (′*b* × ′*c*) *list* **where**
  assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 0 = [] |
  assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 (*Suc k*) = (*case*
    select-diverging-ofsm-table-io′ M q1 q2 (*Suc k*)
   of
   ((*x*,*y*),(*Some q1′*,*Some q2′*)) ⇒ (*x*,*y*) # (*assemble-distinguishing-sequence-from-ofsm-table′ M q1′ q2′ k*) |
    ((*x*,*y*),-) ⇒ [(*x*,*y*)])

lemma *assemble-distinguishing-sequence-from-ofsm-table-alt-def* :
  assumes *k* ≤ *size M* − *1*
  shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* = *assemble-distinguishing-sequence-from-ofsm-table′ M q1 q2 k*
using *assms* **proof** (*induction k arbitrary*: *q1 q2*)
  case *0*
  show *?case*
    unfolding *assemble-distinguishing-sequence-from-ofsm-table.simps*
    unfolding *assemble-distinguishing-sequence-from-ofsm-table′.simps*
    by *presburger*
next
  case (*Suc k*)
  then have *k* ≤ *FSM.size M* − *1*
    by *auto*
  show *?case*
    unfolding *assemble-distinguishing-sequence-from-ofsm-table.simps*
    unfolding *assemble-distinguishing-sequence-from-ofsm-table′.simps*
    unfolding *select-diverging-ofsm-table-io-alt-def*[*OF* ‹*Suc k* ≤ *FSM.size M* −

1›]
  unfolding *Suc.IH*[*OF* ‹*k* ≤ *FSM.size M* − *1*›]
  by *meson*
qed

fun *get-distinguishing-sequence-from-ofsm-tables-refined* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list* **where**
  get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = (*let*
    k = *find-first-distinct-ofsm-table' M q1 q2*
  in assemble-distinguishing-sequence-from-ofsm-table' M q1 q2 k)

lemma *get-distinguishing-sequence-from-ofsm-tables-refined-alt-def* :
 get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = *get-distinguishing-sequence-from-ofsm-tables M q1 q2*
proof −
  define *k* **where** *k* = *find-first-distinct-ofsm-table' M q1 q2*
  then have *k* ≤ *size M* − *1*
    using *find-first-distinct-ofsm-table'-max* **by** *metis*
  have *find-first-distinct-ofsm-table M q1 q2* = *k*
    unfolding *k-def find-first-distinct-ofsm-table-alt-def*
    by *meson*

  show *?thesis*
    unfolding *get-distinguishing-sequence-from-ofsm-tables-refined.simps*
    unfolding *get-distinguishing-sequence-from-ofsm-tables.simps*
    unfolding *Let-def*
    unfolding *k-def*[*symmetric*] ‹*find-first-distinct-ofsm-table M q1 q2* = *k*›
     unfolding *assemble-distinguishing-sequence-from-ofsm-table-alt-def*[*OF* ‹*k* ≤ *size M* − *1*›]
    by *meson*
qed

lemma *get-distinguishing-sequence-from-ofsm-tables-refined-distinguishes* :
  assumes *observable M*
  and *minimal M*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *q1* ≠ *q2*
shows *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2*)
  unfolding *get-distinguishing-sequence-from-ofsm-tables-refined-alt-def*
  using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*] **.**

fun *select-diverging-ofsm-table-io-with-provided-tables* :: (*nat*, (*'a*,*'b*,*'c*) *ofsm-table*) *mapping* ⇒ (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ *nat* ⇒ (*'b* × *'c*) × (*'a option* × *'a option*) **where**
  select-diverging-ofsm-table-io-with-provided-tables tables M q1 q2 k = (*let*

```
    ins = inputs-as-list M;
    outs = outputs-as-list M;
    table = ofsm-lookup (the (Mapping.lookup tables (k−1)));
    f = (λ (x,y) . case (h-obs M q1 x y, h-obs M q2 x y)
              of
                (Some q1', Some q2')  ⇒ if table q1' ≠ table q2'
                                        then Some ((x,y),(Some q1', Some q2'))
                                        else None |
                (None,None)           ⇒ None |
                (Some q1', None)      ⇒ Some ((x,y),(Some q1', None)) |
                (None, Some q2')      ⇒ Some ((x,y),(None, Some q2')))
    in
      hd (List.map-filter f (List.product ins outs)))
```

lemma *select-diverging-ofsm-table-io-with-provided-tables-simp* :
select-diverging-ofsm-table-io-with-provided-tables (*compute-ofsm-tables M* (*size M − 1*)) *M* = *select-diverging-ofsm-table-io' M*
unfolding *select-diverging-ofsm-table-io-with-provided-tables.simps*
select-diverging-ofsm-table-io'.simps
Let-def
by *meson*

fun *assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables* :: (*nat*, ('*a*,'*b*,'*c*) *ofsm-table*) *mapping* ⇒ ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ *nat* ⇒ ('*b* × '*c*) *list* **where**
assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1 q2 0 = [] |
assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M q1 q2 (*Suc k*) = (*case*
select-diverging-ofsm-table-io-with-provided-tables tables M q1 q2 (*Suc k*)
of
((*x*,*y*),(*Some q1'*,*Some q2'*)) ⇒ (*x*,*y*) # (*assemble-distinguishing-sequence-from-ofsm-table-with-provided-ta*
tables M q1' q2' k) |
((*x*,*y*),-) ⇒ [(*x*,*y*)])

lemma *assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables-simp*
:
assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables (*compute-ofsm-tables M* (*size M − 1*)) *M q1 q2 k*= *assemble-distinguishing-sequence-from-ofsm-table' M q1 q2 k*
proof (*induction k arbitrary*: *q1 q2*)
case *0*
show *?case*
unfolding *assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables.simps*
assemble-distinguishing-sequence-from-ofsm-table'.simps
Let-def
by *meson*
next
case (*Suc k'*)

```
  show ?case
   unfolding assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables.simps
    unfolding assemble-distinguishing-sequence-from-ofsm-table'.simps
   unfolding Let-def select-diverging-ofsm-table-io-with-provided-tables-simp Suc.IH
    by meson
qed


lemma get-distinguishing-sequence-from-ofsm-tables-refined-code[code] :
  get-distinguishing-sequence-from-ofsm-tables-refined M q1 q2 = (let
      tables = (compute-ofsm-tables M (size M − 1));
      k = (if (q1 ∈ states M
                ∧ q2 ∈ states M
                ∧ (ofsm-lookup (the (Mapping.lookup tables (size M − 1))) q1
                   ≠ ofsm-lookup (the (Mapping.lookup tables (size M − 1))) q2))
           then the (find-index (λ i . ofsm-lookup (the (Mapping.lookup tables i)) q1
≠ ofsm-lookup (the (Mapping.lookup tables i)) q2) [0..<size M])
           else 0)
  in assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M
q1 q2 k)
  unfolding get-distinguishing-sequence-from-ofsm-tables-refined.simps
           find-first-distinct-ofsm-table'.simps
           Let-def
         assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables-simp
  by meson

fun get-distinguishing-sequence-from-ofsm-tables-with-provided-tables :: (nat, ('a,'b,'c)
ofsm-table) mapping ⇒ ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ 'a ⇒ ('b
× 'c) list where
   get-distinguishing-sequence-from-ofsm-tables-with-provided-tables tables M q1 q2
= (let
      k = (if (q1 ∈ states M
                ∧ q2 ∈ states M
                ∧ (ofsm-lookup (the (Mapping.lookup tables (size M − 1))) q1
                   ≠ ofsm-lookup (the (Mapping.lookup tables (size M − 1))) q2))
           then the (find-index (λ i . ofsm-lookup (the (Mapping.lookup tables i)) q1
≠ ofsm-lookup (the (Mapping.lookup tables i)) q2) [0..<size M])
           else 0)
  in assemble-distinguishing-sequence-from-ofsm-table-with-provided-tables tables M
q1 q2 k)

lemma get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-simp :
  get-distinguishing-sequence-from-ofsm-tables-with-provided-tables (compute-ofsm-tables
M (size M − 1)) M = get-distinguishing-sequence-from-ofsm-tables-refined M
  unfolding get-distinguishing-sequence-from-ofsm-tables-with-provided-tables.simps
           get-distinguishing-sequence-from-ofsm-tables-refined-code
           Let-def
  by meson
```

lemma *get-distinguishing-sequence-from-ofsm-tables-precomputed*:
get-distinguishing-sequence-from-ofsm-tables M = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2)
in distHelper)
proof −
define *distStates* **where** *distStates* = (*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product*
(*states-as-list M*) (*states-as-list M*)))

define *distMap* **where** *distMap-orig*: *distMap* = *mapping-of* (*map* (λ (*q1*,*q2*) .
((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables* (*compute-ofsm-tables*
M (*size M* − *1*)) *M q1 q2*))
distStates)

have *distinct distStates*
unfolding *distStates-def* **using** *states-as-list-distinct*
using *distinct-filter distinct-product* **by** *blast*
then have *distinct* (*map fst* (*map* (λ(*q1*, *q2*). ((*q1*, *q2*), *get-distinguishing-sequence-from-ofsm-tables*
M q1 q2)) *distStates*))
unfolding *map-pair-fst-helper* **.**
then have *distMap-def*: *Mapping.lookup distMap* = *map-of* (*map* (λ (*q1*,*q2*) .
((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables M q1 q2*))
distStates)
unfolding *distMap-orig get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-simp*

get-distinguishing-sequence-from-ofsm-tables-refined-alt-def
using *mapping-of-map-of*
by *blast*

define *distHelper* **where** *distHelper* = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M*
∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2)

have *distHelper* = *get-distinguishing-sequence-from-ofsm-tables M*
proof −
have ⋀ *q1 q2* . *distHelper q1 q2* = *get-distinguishing-sequence-from-ofsm-tables*
M q1 q2
proof −
fix *q1 q2*
show *distHelper q1 q2* = *get-distinguishing-sequence-from-ofsm-tables M q1*
q2
proof (*cases q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2*)

```
      case False
      then show ?thesis
        unfolding distHelper-def by metis
    next
      case True
      then have *:(q1,q2) ∈ list.set distStates
        using states-as-list-set unfolding distStates-def by fastforce

    have distinct (map fst (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables
M q1 q2)) distStates))
      proof −
     have **: (map fst (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables
M q1 q2)) distStates)) = distStates
        proof (induction distStates)
          case Nil
          then show ?case by auto
        next
          case (Cons a distStates)
          obtain x y where a = (x,y)
            using surjective-pairing by blast
          show ?case
            using Cons unfolding ‹a = (x,y)› by auto
        qed

        show ?thesis
          unfolding **
          unfolding distStates-def
          by (simp add: distinct-product)
      qed

        have ((q1,q2), get-distinguishing-sequence-from-ofsm-tables M q1 q2) ∈
list.set (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables M
q1 q2)) distStates)
     using Util.map-set[OF *, of (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables
M q1 q2))]
        by force
    then have the (Mapping.lookup distMap (q1,q2)) = get-distinguishing-sequence-from-ofsm-tables
M q1 q2
        unfolding distMap-def
       unfolding Map.map-of-eq-Some-iff[OF ‹distinct (map fst (map (λ (q1,q2)
. ((q1,q2), get-distinguishing-sequence-from-ofsm-tables M q1 q2)) distStates)›,
symmetric]
        by (metis option.sel)
      moreover have distHelper q1 q2 = the (Mapping.lookup distMap (q1,q2))
        using True unfolding distHelper-def by metis
      ultimately show ?thesis
        by presburger
    qed
  qed
```

then show *?thesis*
by *blast*
qed

then show *?thesis*
unfolding *distHelper-def distMap-orig distStates-def Let-def*
by *presburger*
qed

lemma *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-distinguishes*
:
assumes *observable M*
and *minimal M*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *q1 ≠ q2*
shows *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables-with-provided-tables* (*compute-ofsm-tables M* (*size M* − *1*)) *M q1 q2*)
unfolding *get-distinguishing-sequence-from-ofsm-tables-with-provided-tables-simp*
using *get-distinguishing-sequence-from-ofsm-tables-refined-distinguishes*[*OF assms*]
.

13.4 Refining Minimisation

fun *minimise-refined* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*'a set*,*'b*,*'c*) *fsm* **where**
minimise-refined M = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
eq-class = (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))));
ts = (λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*);
q0 = *eq-class* (*initial M*);
eq-states = *eq-class* |‘| *fstates M*;
M' = *create-unconnected-fsm-from-fsets q0 eq-states* (*finputs M*) (*foutputs M*)
in add-transitions M' ts)

lemma *minimise-refined-is-minimise*[*code*] : *minimise M* = *minimise-refined M*
proof −
define *tables* **where** *tables* = *compute-ofsm-tables M* (*FSM.size M* − *1*)
define *eq-class-refined* **where** *eq-class-refined* = (*ofsm-lookup* (*the* (*Mapping.lookup tables* (*size M* − *1*))))
define *eq-class* **where** *eq-class* = *ofsm-table-fix M* (λ*q* . *states M*) *0*

have (*size M* − *1*) < *Suc* (*size M* − *1*)
by *auto*
have ⋀ *q* . *q* ∈ *states M* ⟹ *eq-class q* = *eq-class-refined q*
unfolding *eq-class-def eq-class-refined-def tables-def*
unfolding *compute-ofsm-tables-lookup-invar*[*OF* ‹(*size M* − *1*) < *Suc* (*size M*

– *1*)›]
by (*metis ofsm-table-fix-partition-fixpoint-trivial-partition*)

have *ts*: (λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*)
= (λ *t* . (*eq-class-refined* (*t-source t*), *t-input t*, *t-output t*, *eq-class-refined* (*t-target t*))) ‘ (*transitions M*)
using ‹⋀ *q* . *q* ∈ *states M* ⟹ *eq-class q* = *eq-class-refined q*›[*OF fsm-transition-source*]
using ‹⋀ *q* . *q* ∈ *states M* ⟹ *eq-class q* = *eq-class-refined q*›[*OF fsm-transition-target*]
by *auto*

have *q0*: *eq-class* (*initial M*) = *eq-class-refined* (*initial M*)
using ‹⋀ *q* . *q* ∈ *states M* ⟹ *eq-class q* = *eq-class-refined q*›[*OF fsm-initial*] **.**

have *eq-states*: *eq-class* |‘| *fstates M* = *eq-class-refined* |‘| *fstates M*
using *fstates-set*[*of M*]
using ‹⋀ *q* . *q* ∈ *states M* ⟹ *eq-class q* = *eq-class-refined q*›
by (*metis fset.map-cong*)

have *M′*: *create-unconnected-fsm-from-fsets* (*eq-class* (*initial M*)) (*eq-class* |‘| *fstates M*) (*finputs M*) (*foutputs M*)
= *create-unconnected-fsm-from-fsets* (*eq-class-refined* (*initial M*)) (*eq-class-refined* |‘| *fstates M*) (*finputs M*) (*foutputs M*)
unfolding *q0 eq-states* **by** *meson*

have *res*: *add-transitions* (*create-unconnected-fsm-from-fsets* (*eq-class* (*initial M*)) (*eq-class* |‘| *fstates M*) (*finputs M*) (*foutputs M*)) ((λ *t* . (*eq-class* (*t-source t*), *t-input t*, *t-output t*, *eq-class* (*t-target t*))) ‘ (*transitions M*))
= *add-transitions* (*create-unconnected-fsm-from-fsets* (*eq-class-refined* (*initial M*)) (*eq-class-refined* |‘| *fstates M*) (*finputs M*) (*foutputs M*)) ((λ *t* . (*eq-class-refined* (*t-source t*), *t-input t*, *t-output t*, *eq-class-refined* (*t-target t*))) ‘ (*transitions M*))
unfolding *M′ ts* **by** *meson*

show *?thesis*
unfolding *minimise.simps minimise-refined.simps Let-def*
unfolding *eq-class-def*[*symmetric*]
unfolding *tables-def*[*symmetric*] *eq-class-refined-def*[*symmetric*]
unfolding *res*
by *meson*
qed

end

14 Transformation to Language-Equivalent Prime FSMs

This theory describes the transformation of FSMs into language-equivalent FSMs that are prime, that is: observable, minimal and initially connected.

theory *Prime-Transformation*
imports *Minimisation Observability State-Cover OFSM-Tables-Refined HOL−Library.List-Lexorder Native-Word.Uint64*
begin

14.1 Helper Functions

The following functions transform FSMs whose states are Sets or FSets into language-equivalent fsms whose states are lists. These steps are required in the chosen implementation of the transformation function, as Sets or FSets are not instances of linorder.

lemma *linorder-fset-list-bij* : *bij-betw sorted-list-of-fset xs* (*sorted-list-of-fset* ‘ *xs*)
  unfolding *bij-betw-def inj-on-def*
  by (*metis sorted-list-of-fset-simps*(*2*))

lemma *linorder-set-list-bij* :
  assumes $\bigwedge$ *x* . *x* ∈ *xs* ⟹ *finite x*
  shows *bij-betw sorted-list-of-set xs* (*sorted-list-of-set* ‘ *xs*)
proof −
  have $\bigwedge$ *x* . *x* ∈ *xs* ⟹ *set* (*sorted-list-of-set x*) = *x*
    by (*simp add*: *assms*)
  then show *?thesis*
    unfolding *bij-betw-def inj-on-def*
    by *metis*
qed

definition *fset-states-to-list-states* :: ((*'a*::*linorder*) *fset*,*'b*,*'c*) *fsm* ⇒ (*'a list*,*'b*,*'c*) *fsm* **where**
  fset-states-to-list-states M = *rename-states M sorted-list-of-fset*

definition *set-states-to-list-states* :: ((*'a*::*linorder*) *set*,*'b*,*'c*) *fsm* ⇒ (*'a list*,*'b*,*'c*) *fsm* **where**
  set-states-to-list-states M = *rename-states M sorted-list-of-set*

lemma *fset-states-to-list-states-language* :
  L (*fset-states-to-list-states M*) = *L M*
  using *rename-states-isomorphism-language*[*OF linorder-fset-list-bij*]
  unfolding *fset-states-to-list-states-def* **.**

lemma *set-states-to-list-states-language* :
  assumes $\bigwedge$ *x* . *x* ∈ *states M* ⟹ *finite x*
  shows *L* (*set-states-to-list-states M*) = *L M*

using *rename-states-isomorphism-language*[*OF linorder-set-list-bij*[*OF assms*]]
unfolding *set-states-to-list-states-def* **.**

lemma *fset-states-to-list-states-observable* :
assumes *observable M*
shows *observable* (*fset-states-to-list-states M*)
using *rename-states-observable*[*OF linorder-fset-list-bij assms*]
unfolding *fset-states-to-list-states-def* **.**

lemma *set-states-to-list-states-observable* :
assumes $\bigwedge$ *x* . *x* ∈ *states M* ⟹ *finite x*
assumes *observable M*
shows *observable* (*set-states-to-list-states M*)
using *rename-states-observable*[*OF linorder-set-list-bij*[*OF assms*(*1*)] *assms*(*2*)]
unfolding *set-states-to-list-states-def* **by** *blast*

lemma *fset-states-to-list-states-minimal* :
assumes *minimal M*
shows *minimal* (*fset-states-to-list-states M*)
using *rename-states-minimal*[*OF linorder-fset-list-bij assms*]
unfolding *fset-states-to-list-states-def* **.**

lemma *set-states-to-list-states-minimal* :
assumes $\bigwedge$ *x* . *x* ∈ *states M* ⟹ *finite x*
assumes *minimal M*
shows *minimal* (*set-states-to-list-states M*)
using *rename-states-minimal*[*OF linorder-set-list-bij*[*OF assms*(*1*)] *assms*(*2*)]
unfolding *set-states-to-list-states-def* **by** *blast*

14.2 The Transformation Algorithm

definition *to-prime* :: (*'a* :: *linorder*, *'b* :: *linorder*, *'c* :: *linorder*) *fsm* ⇒ (*integer*, *'b*, *'c*) *fsm* **where**

```
to-prime M = restrict-to-reachable-states (
               index-states-integer (
                 set-states-to-list-states (
                   minimise-refined (
                     index-states (
                       fset-states-to-list-states (
                         make-observable (
                           restrict-to-reachable-states M)))))))
```

lemma *to-prime-props* :
L (*to-prime M*) = *L M*
observable (*to-prime M*)
minimal (*to-prime M*)
reachable-states (*to-prime M*) = *states* (*to-prime M*)
inputs (*to-prime M*) = *inputs M*
outputs (*to-prime M*) = *outputs M*

proof −

define *M1* **where** *M1*: *M1* = *restrict-to-reachable-states M*
define *M2* **where** *M2*: *M2* = *make-observable M1*
define *M3* **where** *M3*: *M3* = *fset-states-to-list-states M2*
define *M4* **where** *M4*: *M4* = *index-states M3*
define *M5* **where** *M5*: *M5* = *minimise-refined M4*
define *M6* **where** *M6*: *M6* = *set-states-to-list-states M5*
define *M7* **where** *M7*: *M7* = *index-states-integer M6*
define *M8* **where** *M8*: *M8* = *restrict-to-reachable-states M7*

have *to-prime M* = *M8*
  unfolding *M8 M7 M6 M5 M4 M3 M2 M1 to-prime-def* **by** *presburger*

have *observable M2*
  unfolding *M2*
  using *make-observable-language-observable*(*2*) **by** *blast*
then have *observable M3*
  unfolding *M3*
  using *fset-states-to-list-states-observable* **by** *blast*
then have *observable M4*
  unfolding *M4*
  using *index-states-observable* **by** *blast*
then have *observable M5*
  unfolding *M5*
  unfolding *minimise-refined-is-minimise*[*symmetric*]
  using *minimise-observable* **by** *blast*
then have *observable M6*
  unfolding *M6 M5*
  unfolding *minimise-refined-is-minimise*[*symmetric*]
  using *minimise-states-finite*[*OF* ‹*observable M4*›]
  using *set-states-to-list-states-observable*
  by *metis*
then have *observable M7*
  unfolding *M7*
  using *index-states-integer-observable* **by** *blast*
then show *observable* (*to-prime M*)
  unfolding ‹*to-prime M* = *M8*› *M8*
  using *restrict-to-reachable-states-observable* **by** *blast*

have *L M* = *L M1*
  unfolding *M1 restrict-to-reachable-states-language* **by** *simp*
also have ... = *L M2*
  unfolding *M2 make-observable-language-observable*(*1*) **by** *simp*
also have ... = *L M3*
  unfolding *M3 fset-states-to-list-states-language* **by** *simp*
also have ... = *L M4*
  unfolding *M4 index-states-language* **by** *simp*

also have ... = *L M5*
unfolding *M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *minimise-language*[*OF* ‹*observable M4*›] **by** *blast*
also have ... = *L M6*
unfolding *M6 M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *set-states-to-list-states-language*[*OF minimise-states-finite*[*OF* ‹*observable M4*›]] **by** *blast*
also have ... = *L M7*
unfolding *M7* **using** *index-states-integer-language* **by** *blast*
also have ... = *L M8*
unfolding *M8 restrict-to-reachable-states-language* **by** *simp*
finally show *L* (*to-prime M*) = *L M*
unfolding ‹*to-prime M* = *M8*› **by** *blast*

have *minimal M5*
unfolding *M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *minimise-minimal*[*OF* ‹*observable M4*›] **.**
then have *minimal M6*
unfolding *M6 M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *set-states-to-list-states-minimal*[*OF minimise-states-finite*[*OF* ‹*observable M4*›]] **by** *blast*
then have *minimal M7*
unfolding *M7* **using** *index-states-integer-minimal* **by** *blast*
then show *minimal* (*to-prime M*)
unfolding ‹*to-prime M* = *M8*› *M8*
using *restrict-to-reachable-states-minimal* **by** *blast*

show *reachable-states* (*to-prime M*) = *states* (*to-prime M*)
unfolding ‹*to-prime M* = *M8*› *M8 restrict-to-reachable-states-reachable-states* **by** *presburger*

have *inputs M* = *inputs M1*
unfolding *M1 restrict-to-reachable-states-simps* **by** *simp*
also have ... = *inputs M2*
unfolding *M2 make-observable-language-observable Let-def add-transitions-simps create-unconnected-fsm-simps* **by** *blast*
also have ... = *inputs M3*
unfolding *M3 fset-states-to-list-states-def* **by** *simp*
also have ... = *inputs M4*
unfolding *M4 index-states.simps* **by** *simp*
also have ... = *inputs M5*
unfolding *M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *minimise-props*[*OF* ‹*observable M4*›] **by** *blast*
also have ... = *inputs M6*
unfolding *M6 M5 set-states-to-list-states-def* **by** *simp*
also have ... = *inputs M7*
unfolding *M7 index-states.simps* **by** *simp*

also have ... = *inputs M8*
unfolding *M8 restrict-to-reachable-states-simps* **by** *simp*
finally show *inputs* (*to-prime M*) = *inputs M*
unfolding ‹*to-prime M = M8*› **by** *blast*

have *outputs M = outputs M1*
unfolding *M1 restrict-to-reachable-states-simps* **by** *simp*
also have ... = *outputs M2*
unfolding *M2 make-observable-language-observable Let-def add-transitions-simps create-unconnected-fsm-simps* **by** *blast*
also have ... = *outputs M3*
unfolding *M3 fset-states-to-list-states-def* **by** *simp*
also have ... = *outputs M4*
unfolding *M4 index-states.simps* **by** *simp*
also have ... = *outputs M5*
unfolding *M5* **unfolding** *minimise-refined-is-minimise*[*symmetric*]
using *minimise-props*[*OF* ‹*observable M4*›] **by** *blast*
also have ... = *outputs M6*
unfolding *M6 M5 set-states-to-list-states-def* **by** *simp*
also have ... = *outputs M7*
unfolding *M7 index-states.simps* **by** *simp*
also have ... = *outputs M8*
unfolding *M8 restrict-to-reachable-states-simps* **by** *simp*
finally show *outputs* (*to-prime M*) = *outputs M*
unfolding ‹*to-prime M = M8*› **by** *blast*
qed

14.3 Renaming states to Words

lemma *uint64-nat-bij* : (*x* :: *nat*) < *2^64* ⟹ *nat-of-uint64* (*uint64-of-nat x*) = *x*
by *transfer* (*simp add*: *unsigned-of-nat take-bit-nat-eq-self*)

fun *index-states-uint64* :: (*'a::linorder,'b,'c*) *fsm* ⇒ (*uint64,'b,'c*) *fsm* **where**
index-states-uint64 M = rename-states M (*uint64-of-nat* ∘ *assign-indices* (*states M*))

lemma *assign-indices-uint64-bij-betw* :
assumes *size M < 2^64*
shows *bij-betw* (*uint64-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*) ((*uint64-of-nat* ∘ *assign-indices* (*states M*)) ‘ *FSM.states M*)
proof −
have ∗: *inj-on* (*assign-indices* (*FSM.states M*)) (*FSM.states M*)
using *assign-indices-bij*[*OF fsm-states-finite*[*of M*]]
unfolding *bij-betw-def*
by *auto*
moreover have ⋀ *q* . *q* ∈ *states M* ⟹ *assign-indices* (*states M*) *q* < *2^64*
using *assms assign-indices-bij*[*OF fsm-states-finite*[*of M*]]

unfolding *size-def*
by (*meson bij-betwE lessThan-iff less-imp-le less-le-trans*)
ultimately have *inj-on* (*uint64-of-nat* ∘ *assign-indices* (*states M*)) (*FSM.states M*)
unfolding *inj-on-def*
by (*metis comp-apply uint64-nat-bij*)
then show *?thesis*
unfolding *bij-betw-def*
by *auto*
qed

lemma *index-states-uint64-language* :
assumes *size M* < *2^64*
shows *L* (*index-states-uint64 M*) = *L M*
using *rename-states-isomorphism-language*[*of uint64-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-uint64-bij-betw*[*OF assms*]]
by *auto*

lemma *index-states-uint64-observable* :
assumes *size M* < *2^64* **and** *observable M*
shows *observable* (*index-states-uint64 M*)
using *rename-states-observable*[*of uint64-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-uint64-bij-betw*[*OF assms*(*1*)] *assms*(*2*)]
unfolding *index-states-uint64.simps* **.**

lemma *index-states-uint64-minimal* :
assumes *size M* < *2^64* **and** *minimal M*
shows *minimal* (*index-states-uint64 M*)
using *rename-states-minimal*[*of uint64-of-nat* ∘ *assign-indices* (*states M*) *M*, *OF assign-indices-uint64-bij-betw*[*OF assms*(*1*)] *assms*(*2*)]
unfolding *index-states-uint64.simps* **.**

definition *to-prime-uint64* :: (*'a* :: *linorder*,*'b* :: *linorder*,*'c* :: *linorder*) *fsm* ⇒ (*uint64*,*'b*,*'c*) *fsm* **where**
to-prime-uint64 M = *restrict-to-reachable-states* (*index-states-uint64* (*to-prime M*))

lemma *to-prime-uint64-props* :
assumes *size* (*to-prime M*) < *2^64*
shows
L (*to-prime-uint64 M*) = *L M*
observable (*to-prime-uint64 M*)
minimal (*to-prime-uint64 M*)
reachable-states (*to-prime-uint64 M*) = *states* (*to-prime-uint64 M*)
inputs (*to-prime-uint64 M*) = *inputs M*
outputs (*to-prime-uint64 M*) = *outputs M*
using *restrict-to-reachable-states-reachable-states*[*of index-states-uint64* (*to-prime*

M)]
unfolding *to-prime-uint64-def*
using *index-states-uint64-language*[*OF assms*]
unfolding *restrict-to-reachable-states-language*
using *restrict-to-reachable-states-observable*[*OF index-states-uint64-observable*[*OF assms to-prime-props*(*2*)]]
using *restrict-to-reachable-states-minimal*[*OF index-states-uint64-minimal*[*OF assms to-prime-props*(*3*)]]
unfolding *index-states-uint64.simps*
unfolding *restrict-to-reachable-states-simps*
unfolding *rename-states-simps*(*3*,*4*)
unfolding *to-prime-props*(*1*,*5*,*6*)
by *blast*+

end

15 Convergence of Traces

This theory defines convergence of traces in observable FSMs and provides results on sufficient conditions to establish that two traces converge. Furthermore it is shown how convergence can be employed in proving language equivalence.

theory *Convergence*
imports *../Minimisation ../Distinguishability ../State-Cover HOL−Library.List-Lexorder*
begin

15.1 Basic Definitions

fun *converge* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ $('b \times 'c)$ *list* $\Rightarrow$ *bool* **where**
converge M π τ = ($\pi \in L\ M \wedge \tau \in L\ M \wedge$ (*LS M* (*after-initial M* π) = *LS M* (*after-initial M* τ)))

fun *preserves-divergence* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
preserves-divergence M1 M2 A = ($\forall\ \alpha \in L\ M1 \cap A\ .\ \forall\ \beta \in L\ M1 \cap A\ .\ \neg$ *converge M1* α $\beta \longrightarrow \neg$ *converge M2* α β)

fun *preserves-convergence* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
preserves-convergence M1 M2 A = ($\forall\ \alpha \in L\ M1 \cap A\ .\ \forall\ \beta \in L\ M1 \cap A\ .$ *converge M1* α $\beta \longrightarrow$ *converge M2* α β)

lemma *converge-refl* :
assumes $\alpha \in L\ M$
shows *converge M* α α
using *assms* **by** *auto*

lemma *convergence-minimal* :
  assumes *minimal M*
  and *observable M*
  and $\alpha \in L\ M$
  and $\beta \in L\ M$
shows *converge M* α β = ((*after-initial M* α) = (*after-initial M* β))
proof
  have $*$:(*after-initial M* α) $\in$ *states M*
    using ‹$\alpha \in L\ M$› **by** (*meson after-is-state assms*(*2*))
  have $**$:(*after-initial M* β) $\in$ *states M*
    using ‹$\beta \in L\ M$› **by** (*meson after-is-state assms*(*2*))

  show *converge M* α β $\Longrightarrow$ ((*after-initial M* α) = (*after-initial M* β))
    using $*$ $**$ ‹*minimal M*› **unfolding** *minimal.simps converge.simps*
    by *blast*

  show ((*after-initial M* α) = (*after-initial M* β)) $\Longrightarrow$ *converge M* α β
    unfolding *converge.simps* **using** *assms*(*3*,*4*) **by** *simp*
qed

lemma *state-cover-assignment-diverges* :
  assumes *observable M*
  and *minimal M*
  and *is-state-cover-assignment M f*
  and *q1* $\in$ *reachable-states M*
  and *q2* $\in$ *reachable-states M*
  and *q1* $\neq$ *q2*
shows $\neg$ *converge M* (*f q1*) (*f q2*)
proof $-$
  have *f q1* $\in$ *L M*
    using *assms*(*3*,*4*)
   by (*metis from-FSM-language is-state-cover-assignment.simps language-contains-empty-sequence language-io-target-append language-prefix reachable-state-is-state*)
  moreover have *q1* $\in$ *io-targets M* (*f q1*) (*initial M*)
    using *assms*(*3*,*4*) **unfolding** *is-state-cover-assignment.simps* **by** *blast*
  ultimately have (*after-initial M* (*f q1*)) = *q1*
    using *assms*(*1*)
    by (*metis* (*no-types*, *lifting*) *observable-after-path observable-path-io-target singletonD*)

  have *f q2* $\in$ *L M*
    using *assms*(*3*,*5*)
   by (*metis from-FSM-language is-state-cover-assignment.simps language-contains-empty-sequence language-io-target-append language-prefix reachable-state-is-state*)
  moreover have *q2* $\in$ *io-targets M* (*f q2*) (*initial M*)
    using *assms*(*3*,*5*) **unfolding** *is-state-cover-assignment.simps* **by** *blast*
  ultimately have (*after-initial M* (*f q2*)) = *q2*
    using *assms*(*1*)

by (*metis* (*no-types*, *lifting*) *observable-after-path observable-path-io-target singletonD*)

show *?thesis*
using *convergence-minimal*[*OF assms*(*2*,*1*) ‹*f q1* ∈ *L M*› ‹*f q2* ∈ *L M*›] ‹*q1* ≠ *q2*›
unfolding ‹(*after-initial M* (*f q1*)) = *q1*› ‹(*after-initial M* (*f q2*)) = *q2*›
by *simp*
qed

lemma *converge-extend* :
assumes *observable M*
and *converge M* α β
and α@γ ∈ *L M*
and β ∈ *L M*
shows β@γ ∈ *L M*
by (*metis after-io-targets assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *converge.simps language-io-target-append language-prefix observable-io-targets observable-io-targets-language singletonI the-elem-eq*)

lemma *converge-append* :
assumes *observable M*
and *converge M* α β
and α@γ ∈ *L M*
and β ∈ *L M*
shows *converge M* (α@γ) (β@γ)
using *after-language-append-iff*[*OF assms*(*1*,*3*)]
using *after-language-append-iff*[*OF assms*(*1*) *converge-extend*[*OF assms*]]
using *assms converge-extend*
unfolding *converge.simps*
by *blast*

lemma *non-initialized-state-cover-assignment-diverges* :
assumes *observable M*
and *minimal M*
and ⋀ *q* . *q* ∈ *reachable-states M* ⟹ *q* ∈ *io-targets M* (*f q*) (*initial M*)
and ⋀ *q* . *q* ∈ *reachable-states M* ⟹ *f q* ∈ *L M* ∩ *SC*
and *q1* ∈ *reachable-states M*
and *q2* ∈ *reachable-states M*
and *q1* ≠ *q2*
shows ¬ *converge M* (*f q1*) (*f q2*)
proof −

have *f q1* ∈ *L M*
using *assms*(*4*,*5*) **by** *blast*
moreover have *q1* ∈ *io-targets M* (*f q1*) (*initial M*)

using *assms*(*3*,*5*) **by** *blast*
ultimately have (*after-initial M* (*f q1*)) = *q1*
using *assms*(*1*)
by (*metis* (*no-types*, *lifting*) *observable-after-path observable-path-io-target singletonD*)

have *f q2* ∈ *L M*
using *assms*(*4*,*6*) **by** *blast*
moreover have *q2* ∈ *io-targets M* (*f q2*) (*initial M*)
using *assms*(*3*,*6*) **by** *blast*
ultimately have (*after-initial M* (*f q2*)) = *q2*
using *assms*(*1*)
by (*metis* (*no-types*, *lifting*) *observable-after-path observable-path-io-target singletonD*)

show *?thesis*
using *convergence-minimal*[*OF assms*(*2*,*1*) ‹*f q1* ∈ *L M*› ‹*f q2* ∈ *L M*›] ‹*q1* ≠ *q2*›
unfolding ‹(*after-initial M* (*f q1*)) = *q1*› ‹(*after-initial M* (*f q2*)) = *q2*›
by *simp*
qed

lemma *converge-trans-2* :
assumes *observable M* **and** *minimal M* **and** *converge M u v*
shows *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*v@w2*)
converge M (*u@w1*) (*u@w2*) = *converge M* (*u@w1*) (*v@w2*)
converge M (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*u@w2*)
proof −
have *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*v@w2*) ∧ *converge M* (*u@w1*) (*u@w2*) = *converge M* (*u@w1*) (*v@w2*) ∧ *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*u@w2*)
proof (*cases u@w1* ∈ *L M* ∧ *u@w2* ∈ *L M*)
case *False*
then consider *u@w1* ∉ *L M* | *u@w2* ∉ *L M*
by *blast*
then have *v@w1* ∉ *L M* ∨ *v@w2* ∉ *L M*
using *after-language-iff*[*OF assms*(*1*), *of u initial M w1*]
after-language-iff[*OF assms*(*1*), *of u initial M w2*]
after-language-iff[*OF assms*(*1*), *of v initial M w1*]
after-language-iff[*OF assms*(*1*), *of v initial M w2*]
by (*metis assms*(*3*) *converge.elims*(*2*))
then show *?thesis*
by (*meson assms*(*1*) *assms*(*3*) *converge.elims*(*2*) *converge-extend*)
next
case *True*
then have *u@w1* ∈ *L M* **and** *u@w2* ∈ *L M* **by** *auto*
then have *v@w1* ∈ *L M* **and** *v@w2* ∈ *L M*
by (*meson assms*(*1*) *assms*(*3*) *converge.simps converge-extend*)+

have $u \in L\ M$ **using** ‹$u@w1 \in L\ M$› *language-prefix* **by** *metis*
have $v \in L\ M$ **using** ‹$v@w1 \in L\ M$› *language-prefix* **by** *metis*

have *after-initial M u* = *after-initial M v*
using ‹$u \in L\ M$› ‹$v \in L\ M$› *assms*(*1*) *assms*(*2*) *assms*(*3*) *convergence-minimal* **by** *blast*
moreover have *after-initial M* (*u* @ *w1*) = *after-initial M* (*v* @ *w1*)
by (*metis calculation True* ‹*v* @ *w1* ∈ *L M*› *after-split assms*(*1*))
ultimately have *after-initial M* (*u* @ *w2*) = *after-initial M* (*v* @ *w2*)
by (*metis* (*no-types*) *True* ‹*v* @ *w2* ∈ *L M*› *after-split assms*(*1*))

have *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*v@w2*)
using *True* ‹*after-initial M* (*u* @ *w1*) = *after-initial M* (*v* @ *w1*)› ‹*after-initial M* (*u* @ *w2*) = *after-initial M* (*v* @ *w2*)› ‹*v* @ *w1* ∈ *L M*› ‹*v* @ *w2* ∈ *L M*›
by *auto*
moreover have *converge M* (*u@w1*) (*u@w2*) = *converge M* (*u@w1*) (*v@w2*)
using *True* ‹*after-initial M* (*u* @ *w2*) = *after-initial M* (*v* @ *w2*)› ‹*v* @ *w2* ∈ *L M*› **by** *auto*
moreover have *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*u@w2*)
using *True* ‹*after-initial M* (*u* @ *w1*) = *after-initial M* (*v* @ *w1*)› ‹*v* @ *w1* ∈ *L M*› **by** *auto*
ultimately show *?thesis*
by *blast*
qed
then show *converge M* (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*v@w2*)
converge M (*u@w1*) (*u@w2*) = *converge M* (*u@w1*) (*v@w2*)
converge M (*u@w1*) (*u@w2*) = *converge M* (*v@w1*) (*u@w2*)
by *blast+*
qed

lemma *preserves-divergence-converge-insert* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1 u v*
and *converge M2 u v*
and *preserves-divergence M1 M2 X*
and $u \in X$
shows *preserves-divergence M1 M2* (*Set.insert v X*)
proof −
have $\bigwedge w\ .\ w \in L\ M1 \cap X \Longrightarrow \neg$*converge M1 v w* $\Longrightarrow \neg$*converge M2 v w*
proof −
fix *w*
assume $w \in L\ M1 \cap X$ **and** $\neg$*converge M1 v w*

then have $\neg$*converge M1 u w*

```
      using assms(5)
      using converge.simps by blast
    then have ¬converge M2 u w
      using assms(5−8)
     by (meson IntI ‹w ∈ L M1 ∩ X› converge.elims(2) preserves-divergence.simps)

    then show ¬converge M2 v w
      using assms(6) converge.simps by blast
  qed
  then show ?thesis
    using assms(7)
    unfolding preserves-divergence.simps
     by (metis (no-types, lifting) Int-insert-right-if1 assms(1) assms(2) assms(3)
assms(4) assms(5) converge.elims(2) convergence-minimal insert-iff)
qed

lemma preserves-divergence-converge-replace :
  assumes observable M1
      and observable M2
      and minimal M1
      and minimal M2
      and converge M1 u v
      and converge M2 u v
      and preserves-divergence M1 M2 (Set.insert u X)
shows preserves-divergence M1 M2 (Set.insert v X)
proof −
  have u ∈ L M1 and v ∈ L M1
    using assms(5) by auto
  then have after-initial M1 u = after-initial M1 v
    using assms(1) assms(3) assms(5) convergence-minimal by blast

  have ⋀ w . w ∈ L M1 ∩ X ⟹ ¬converge M1 v w ⟹ ¬converge M2 v w
  proof −
    fix w
    assume w ∈ L M1 ∩ X and ¬converge M1 v w

    then have ¬converge M1 u w
      using assms(5)
      using converge.simps by blast
    then have ¬converge M2 u w
      using assms(5−7)
       by (meson IntD1 IntD2 IntI ‹w ∈ L M1 ∩ X› converge.elims(2) insertCI
preserves-divergence.elims(1))
    then show ¬converge M2 v w
      using assms(6) converge.simps by blast
  qed

  have ⋀ α β . α ∈ L M1 ⟹ α ∈ insert v X ⟹ β ∈ L M1 ⟹ β ∈ insert v X
⟹ ¬ converge M1 α β ⟹ ¬ converge M2 α β
```

proof −
fix α β **assume** $\alpha \in L\ M1$ $\alpha \in insert\ v\ X$ $\beta \in L\ M1$ $\beta \in insert\ v\ X$ $\neg\ converge\ M1\ \alpha\ \beta$

then consider $\alpha = v \wedge \beta = v$ |
$\alpha = v \wedge \beta \in X$ |
$\alpha \in X \wedge \beta = v$ |
$\alpha \in X \wedge \beta \in X$
by *blast*
then show $\neg\ converge\ M2\ \alpha\ \beta$
proof *cases*
case *1*
then show *?thesis*
using ‹$\alpha \in L\ M1$› ‹$\beta \in L\ M1$› ‹$\neg\ converge\ M1\ \alpha\ \beta$› **by** *auto*
next
case *2*
then show *?thesis*
by (*metis IntI* ‹$\bigwedge w.\ [\![w \in L\ M1 \cap X;\ \neg\ converge\ M1\ v\ w]\!] \Longrightarrow \neg\ converge\ M2\ v\ w$› ‹$\beta \in L\ M1$› ‹$\neg\ converge\ M1\ \alpha\ \beta$›)
next
case *3*
then show *?thesis*
by (*metis IntI* ‹$\bigwedge w.\ [\![w \in L\ M1 \cap X;\ \neg\ converge\ M1\ v\ w]\!] \Longrightarrow \neg\ converge\ M2\ v\ w$› ‹$\alpha \in L\ M1$› ‹$\neg\ converge\ M1\ \alpha\ \beta$› *converge.simps*)
next
case *4*
then show *?thesis*
using *assms*(*7*) **unfolding** *preserves-divergence.simps*
using ‹$\alpha \in L\ M1$› ‹$\beta \in L\ M1$› ‹$\neg\ converge\ M1\ \alpha\ \beta$› **by** *blast*
qed
qed
then show *?thesis*
unfolding *preserves-divergence.simps* **by** *blast*
qed

lemma *preserves-divergence-converge-replace-iff* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1 u v*
and *converge M2 u v*
shows *preserves-divergence M1 M2* (*Set.insert u X*) = *preserves-divergence M1 M2* (*Set.insert v X*)
proof −
have $*$: *converge M1 v u* **using** *assms*(*5*) **by** *auto*
have $**$: *converge M2 v u* **using** *assms*(*6*) **by** *auto*

show *?thesis*

using *preserves-divergence-converge-replace*[*OF assms*]
preserves-divergence-converge-replace[*OF assms*(*1*−*4*) ∗ ∗∗]
by *blast*
qed

lemma *preserves-divergence-subset* :
assumes *preserves-divergence M1 M2 B*
and $A \subseteq B$
shows *preserves-divergence M1 M2 A*
using *assms* **unfolding** *preserves-divergence.simps* **by** *blast*

lemma *preserves-divergence-insertI* :
assumes *preserves-divergence M1 M2 X*
and $\bigwedge \alpha \,.\, \alpha \in L\ M1 \cap X \Longrightarrow \beta \in L\ M1 \Longrightarrow \neg converge\ M1\ \alpha\ \beta \Longrightarrow \neg converge\ M2\ \alpha\ \beta$
shows *preserves-divergence M1 M2* (*Set.insert* β *X*)
using *assms* **unfolding** *preserves-divergence.simps*
by (*metis Int-insert-right converge.elims*(*2*) *converge.elims*(*3*) *insertE*)

lemma *preserves-divergence-insertE* :
assumes *preserves-divergence M1 M2* (*Set.insert* β *X*)
shows *preserves-divergence M1 M2 X*
and $\bigwedge \alpha \,.\, \alpha \in L\ M1 \cap X \Longrightarrow \beta \in L\ M1 \Longrightarrow \neg converge\ M1\ \alpha\ \beta \Longrightarrow \neg converge\ M2\ \alpha\ \beta$
using *assms* **unfolding** *preserves-divergence.simps*
by *blast*+

lemma *distinguishes-diverge-prefix* :
assumes *observable M*
and *distinguishes M* (*after-initial M u*) (*after-initial M v*) *w*
and $u \in L\ M$
and $v \in L\ M$
and $w' \in set\ (prefixes\ w)$
and $w' \in LS\ M\ (after\text{-}initial\ M\ u)$
and $w' \in LS\ M\ (after\text{-}initial\ M\ v)$
shows $\neg converge\ M\ (u@w')\ (v@w')$
proof
assume $converge\ M\ (u\ @\ w')\ (v\ @\ w')$

obtain w'' **where** $w = w'@w''$
using *assms*(*5*)
using *prefixes-set-ob* **by** *auto*

have $u@w' \in L\ M$
using *assms*(*3*,*6*) *after-language-iff*[*OF assms*(*1*)]
by *blast*
then have $*$:$(w \in LS\ M\ (after\text{-}initial\ M\ u)) = (w'' \in LS\ M\ (after\text{-}initial\ M\ (u@w')))$
using *after-language-append-iff*[*OF assms*(*1*)]

unfolding ‹$w = w'@w''$›
by *blast*

have $v@w' \in L\ M$
using *assms*(*4*,*7*) *after-language-iff*[*OF assms*(*1*)]
by *blast*
then have $**$:$(w \in LS\ M\ (after\text{-}initial\ M\ v)) = (w'' \in LS\ M\ (after\text{-}initial\ M\ (v@w')))$
using *after-language-append-iff*[*OF assms*(*1*)]
unfolding ‹$w = w'@w''$›
by *blast*

have $(w \in LS\ M\ (after\text{-}initial\ M\ u)) = (w \in LS\ M\ (after\text{-}initial\ M\ v))$
unfolding $* **$
using ‹*converge* $M\ (u\ @\ w')\ (v\ @\ w')$›
by (*metis converge.elims*(*2*))
then show *False*
using ‹*distinguishes* M (*after-initial* $M\ u$) (*after-initial* $M\ v$) w›
unfolding *distinguishes-def*
by *blast*
qed

lemma *converge-distinguishable-helper* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *converge M1* $\pi\ \alpha$
and *converge M2* $\pi\ \alpha$
and *converge M1* $\tau\ \beta$
and *converge M2* $\tau\ \beta$
and *distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) v
and $L\ M1 \cap \{\alpha@v,\beta@v\} = L\ M2 \cap \{\alpha@v,\beta@v\}$
shows (*after-initial M1* π) $\neq$ (*after-initial M1* τ)
proof −
have *LS M1* (*after-initial M1* π) = *LS M1* (*after-initial M1* α)
by (*meson assms*(*5*) *converge.elims*(*2*))
have *LS M1* (*after-initial M1* τ) = *LS M1* (*after-initial M1* β)
by (*meson assms*(*7*) *converge.elims*(*2*))
have *LS M2* (*after-initial M2* π) = *LS M2* (*after-initial M2* α)
by (*meson assms*(*6*) *converge.elims*(*2*))
have *LS M2* (*after-initial M2* τ) = *LS M2* (*after-initial M2* β)
by (*meson assms*(*8*) *converge.elims*(*2*))

have $v \in LS\ M2$ (*after-initial M2* π) $\longleftrightarrow v \notin LS\ M2$ (*after-initial M2* τ)
using *assms*(*9*) **unfolding** *distinguishes-def* **by** *blast*
then have $v \in LS\ M2$ (*after-initial M2* α) $\longleftrightarrow v \notin LS\ M2$ (*after-initial M2* β)
using ‹*LS M2* (*after-initial M2* π) = *LS M2* (*after-initial M2* α)› ‹*LS M2* (*after-initial M2* τ) = *LS M2* (*after-initial M2* β)› **by** *blast*

then have $\alpha@v \in L\ M2 \longleftrightarrow \beta@v \notin L\ M2$
by (*meson after-language-iff assms*(*2*) *assms*(*6*) *assms*(*8*) *converge.elims*(*2*))
then have $\alpha@v \in L\ M1 \longleftrightarrow \beta@v \notin L\ M1$
using *assms*(*10*)
by (*metis* (*no-types*, *lifting*) *Int-insert-right inf-sup-ord*(*1*) *insert-subset*)
then have $v \in LS\ M1\ (after\text{-}initial\ M1\ \alpha) \longleftrightarrow v \notin LS\ M1\ (after\text{-}initial\ M1\ \beta)$
by (*meson after-language-iff assms*(*1*) *assms*(*5*) *assms*(*7*) *converge.elims*(*2*))
then have $v \in LS\ M1\ (after\text{-}initial\ M1\ \pi) \longleftrightarrow v \notin LS\ M1\ (after\text{-}initial\ M1\ \tau)$
using ‹$LS\ M1\ (after\text{-}initial\ M1\ \pi) = LS\ M1\ (after\text{-}initial\ M1\ \alpha)$› ‹$LS\ M1\ (after\text{-}initial\ M1\ \tau) = LS\ M1\ (after\text{-}initial\ M1\ \beta)$› **by** *blast*
then show *?thesis*
by *metis*
qed

lemma *converge-append-language-iff* :
assumes *observable M*
and *converge M* α β
shows $(\alpha@\gamma \in L\ M) = (\beta@\gamma \in L\ M)$
by (*metis* (*no-types*) *assms*(*1*) *assms*(*2*) *converge.simps converge-extend*)

lemma *converge-append-iff* :
assumes *observable M*
and *converge M* α β
shows *converge M* γ $(\alpha@\omega)$ = *converge M* γ $(\beta@\omega)$
proof (*cases* $(\alpha@\omega) \in L\ M$)
case *True*
then show *?thesis*
using *converge-append-language-iff*[*OF assms*] *language-prefix*[*of* β ω *M initial M*]
using *converge-append*[*OF assms True*]
by *auto*
next
case *False*
then show *?thesis*
using *converge-append-language-iff*[*OF assms*]
using *converge.simps* **by** *blast*
qed

lemma *after-distinguishes-language* :
assumes *observable M1*
and $\alpha \in L\ M1$
and $\beta \in L\ M1$
and *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) γ
shows $(\alpha@\gamma \in L\ M1) \neq (\beta@\gamma \in L\ M1)$
unfolding *after-language-iff*[*OF assms*(*1*,*2*),*symmetric*]
after-language-iff[*OF assms*(*1*,*3*),*symmetric*]
using *assms*(*4*)
unfolding *distinguishes-def*
by *blast*

lemma *distinguish-diverge* :
assumes *observable M1*
and *observable M2*
and *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) γ
and u @ $\gamma \in T$
and v @ $\gamma \in T$
and $u \in L$ *M1*
and $v \in L$ *M1*
and L *M1* $\cap T = L$ *M2* $\cap T$
shows $\neg$ *converge M2 u v*
proof
assume *converge M2 u v*
then have $u@\gamma \in L$ *M2* $\longleftrightarrow v@\gamma \in L$ *M2*
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
moreover have $u@\gamma \in L$ *M1* $\longleftrightarrow v@\gamma \notin L$ *M1*
using *assms*(*1*,*3*,*6*,*7*)
using *after-distinguishes-language*
by *blast*
ultimately show *False*
using *assms*(*4*,*5*,*8*) **by** *blast*
qed

lemma *distinguish-converge-diverge* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and $u' \in L$ *M1*
and $v' \in L$ *M1*
and *converge M1 u u'*
and *converge M1 v v'*
and *converge M2 u u'*
and *converge M2 v v'*
and *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) γ
and u' @ $\gamma \in T$
and v' @ $\gamma \in T$
and L *M1* $\cap T = L$ *M2* $\cap T$
shows $\neg$ *converge M2 u v*
proof −
have $*$:*distinguishes M1* (*after-initial M1 u'*) (*after-initial M1 v'*) γ
by (*metis* (*mono-tags*, *opaque-lifting*) *assms*(*1*) *assms*(*10*) *assms*(*3*) *assms*(*6*) *assms*(*7*) *converge.simps convergence-minimal*)

show *?thesis*
using *distinguish-diverge*[*OF assms*(*1*−*2*) $*$]
by (*metis* (*mono-tags*, *lifting*) *assms*(*9*) *assms*(*11*) *assms*(*12*) *assms*(*13*) *assms*(*4*) *assms*(*5*) *assms*(*8*) *converge.simps*)

qed

lemma *diverge-prefix* :
  assumes *observable M*
  and $\alpha@\gamma \in L\ M$
  and $\beta@\gamma \in L\ M$
  and $\neg$ *converge M* $(\alpha@\gamma)$ $(\beta@\gamma)$
shows $\neg$ *converge M* α β
  by (*meson assms converge-append language-prefix*)

lemma *converge-sym*: *converge M u v* = *converge M v u*
  by *auto*

lemma *state-cover-transition-converges* :
  assumes *observable M*
  and *is-state-cover-assignment M V*
  and *t* $\in$ *transitions M*
  and *t-source t* $\in$ *reachable-states M*
shows *converge M* ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) (*V* (*t-target t*))
proof −
  have *t-target t* $\in$ *reachable-states M*
    using *assms*(*3*,*4*) *reachable-states-next*
    by *metis*

  have *V* (*t-source t*) $\in$ *L M* **and** *after-initial M* (*V* (*t-source t*)) = (*t-source t*)
    using *state-cover-assignment-after*[*OF assms*(*1*,*2*,*4*)]
    by *simp*+

  have ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\in$ *L M*
    using *after-language-iff*[*OF assms*(*1*) ‹*V* (*t-source t*) $\in$ *L M*›, *of* [(*t-input t*,*t-output t*)]]
      assms(*3*)
    unfolding *LS-single-transition* ‹*after-initial M* (*V* (*t-source t*)) = (*t-source t*)›
    by *force*

  have *FSM.after M* (*t-source t*) [(*t-input t*, *t-output t*)] = *t-target t*
    using *after-transition*[*OF assms*(*1*)] *assms*(*3*)
    by *auto*
  then have *after-initial M* ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) = *t-target t*
    using ‹*after-initial M* (*V* (*t-source t*)) = (*t-source t*)›
    using *after-split*[*OF assms*(*1*) ‹((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\in$ *L M*›]
    by *force*
  then show *?thesis*
    using ‹((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\in$ *L M*›
    using *state-cover-assignment-after*[*OF assms*(*1*,*2*) ‹*t-target t* $\in$ *reachable-states M*›]
    by *auto*

qed

lemma *equivalence-preserves-divergence* :
assumes *observable M*
and *observable I*
and $L\ M = L\ I$
shows *preserves-divergence M I A*
proof −
have $\bigwedge \alpha\ \beta\ .\ \alpha \in L\ M \cap A \Longrightarrow \beta \in L\ M \cap A \Longrightarrow \neg$ *converge M* $\alpha\ \beta \Longrightarrow \neg$ *converge I* $\alpha\ \beta$
proof −
fix $\alpha\ \beta$ **assume** $\alpha \in L\ M \cap A$ **and** $\beta \in L\ M \cap A$ **and** $\neg$ *converge M* $\alpha\ \beta$
then have *after-initial M* $\alpha \in$ *states M* **and** *after-initial M* $\beta \in$ *states M* **and** *LS M* (*after-initial M* α) $\neq$ *LS M* (*after-initial M* β)
using *after-is-state*[*OF assms*(*1*)] **unfolding** *converge.simps*
by *auto*
then obtain γ **where** ($\gamma \in$ *LS M* (*after-initial M* α)) $\neq$ ($\gamma \in$ *LS M* (*after-initial M* β))
by *blast*
then have ($\alpha@\gamma \in L\ M$) $\neq$ ($\beta@\gamma \in L\ M$)
using *after-language-iff*[*OF assms*(*1*)] ‹$\alpha \in L\ M \cap A$› ‹$\beta \in L\ M \cap A$› **by** *blast*
then have ($\alpha@\gamma \in L\ I$) $\neq$ ($\beta@\gamma \in L\ I$)
using *assms*(*3*) **by** *blast*
then show $\neg$ *converge I* $\alpha\ \beta$
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
qed
then show *?thesis*
unfolding *preserves-divergence.simps* **by** *blast*
qed

15.2 Sufficient Conditions for Convergence

The following lemma provides a condition for convergence that assumes the existence of a single state cover covering all extensions of length up to (m - |M1|). This is too restrictive for the SPYH method but could be used in the SPY method. The proof idea has been developed by Wen-ling Huang and adapted by the author to avoid requiring the SC to cover traces that contain a proper prefix already not in the language of FSM M1.

lemma *sufficient-condition-for-convergence-in-SPY-method* :
fixes *M1* :: ('*a*,'*b*,'*c*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq m$
and *size M2* $\leq m$

and $L\ M1 \cap T = L\ M2 \cap T$
and $\pi \in L\ M1 \cap T$
and $\tau \in L\ M1 \cap T$
and *converge M1* $\pi\ \tau$
and $SC \subseteq T$
and $\bigwedge q\ .\ q \in$ *reachable-states M1* $\Longrightarrow \exists\ io \in L\ M1 \cap SC\ .\ q \in$ *io-targets M1 io* (*initial M1*)
and *preserves-divergence M1 M2 SC*
and $\bigwedge \gamma\ x\ y\ .$ *length* $\gamma < m -$ *size-r M1* $\Longrightarrow$
$\gamma \in LS\ M1$ (*after-initial M1* π) $\Longrightarrow$
$x \in$ *inputs M1* $\Longrightarrow$
$y \in$ *outputs M1* $\Longrightarrow$
$\exists\ \alpha\ \beta\ .$ *converge M1* α (π@γ) $\wedge$
converge M2 α (π@γ) $\wedge$
converge M1 β (τ@γ) $\wedge$
converge M2 β (τ@γ) $\wedge$
$\alpha \in SC \wedge$
α@[(x,y)] $\in SC \wedge$
$\beta \in SC \wedge$
β@[(x,y)] $\in SC$
and $\exists\ \alpha\ \beta\ .$ *converge M1* $\alpha\ \pi \wedge$
converge M2 $\alpha\ \pi \wedge$
converge M1 $\beta\ \tau \wedge$
converge M2 $\beta\ \tau \wedge$
$\alpha \in SC \wedge$
$\beta \in SC$
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows *converge M2* $\pi\ \tau$
proof −

obtain f **where** *f1*: $\bigwedge q\ .\ q \in$ *reachable-states M1* $\Longrightarrow q \in$ *io-targets M1* ($f\ q$) (*initial M1*)
and *f2*: $\bigwedge q\ .\ q \in$ *reachable-states M1* $\Longrightarrow f\ q \in L\ M1 \cap SC$
using *non-initialized-state-cover-assignment-from-non-initialized-state-cover*[*OF* ‹$\bigwedge q\ .\ q \in$ *reachable-states M1* $\Longrightarrow \exists\ io \in L\ M1 \cap SC\ .\ q \in$ *io-targets M1 io* (*initial M1*)›]
by *blast*

define A **where** A: $A = (\lambda\ q\ .$ *Set.filter* (*converge M1* ($f\ q$)) ($L\ M1 \cap SC$))

define Q **where** Q: $Q = (\lambda\ q\ . \bigcup\ \alpha \in A\ q\ .$ *io-targets M2* α (*initial M2*))

have $\bigwedge q\ .\ q \in$ *reachable-states M1* $\Longrightarrow Q\ q \neq \{\}$
proof −
fix q **assume** $q \in$ *reachable-states M1*
then have $f\ q \in A\ q$
using A
using *f2* **by** *auto*

moreover have $f\ q \in L\ M2$
proof −
have $f\ q \in L\ M1 \cap SC$
using ‹$q \in$ *reachable-states M1*› *f2* **by** *blast*
then show *?thesis*
using ‹$SC \subseteq T$› ‹$L\ M1 \cap T = L\ M2 \cap T$› **by** *blast*
qed
ultimately show $Q\ q \neq \{\}$
unfolding Q
by *auto*
qed

have *states M2* $= (\bigcup\ q \in$ *reachable-states M1* $.\ Q\ q) \cup ($*states M2* $- (\bigcup\ q \in$ *reachable-states M1* $.\ Q\ q))$
proof −
have $(\bigcup\ q \in$ *reachable-states M1* $.\ Q\ q) \subseteq$ *reachable-states M2*
proof
fix q **assume** $q \in (\bigcup\ q \in$ *reachable-states M1* $.\ Q\ q)$
then obtain α **where** $q \in$ *io-targets M2* α (*initial M2*)
unfolding Q **by** *blast*
then show $q \in$ *reachable-states M2*
unfolding *io-targets.simps reachable-states-def* **by** *blast*
qed
then show *?thesis*
by (*metis Diff-partition reachable-state-is-state subset-iff*)
qed

have $\bigwedge$ *q1 q2* . *q1* $\in$ *reachable-states M1* $\Longrightarrow$ *q2* $\in$ *reachable-states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow Q\ q1 \cap Q\ q2 = \{\}$
proof −
fix *q1 q2*
assume *q1* $\in$ *reachable-states M1* **and** *q2* $\in$ *reachable-states M1* **and** *q1* $\neq$ *q2*

have $\bigwedge\ \alpha\ \beta\ .\ \alpha \in A\ q1 \Longrightarrow \beta \in A\ q2 \Longrightarrow$ *io-targets M2* α (*initial M2*) $\cap$ *io-targets M2* β (*initial M2*) $= \{\}$
proof −
fix $\alpha\ \beta$ **assume** $\alpha \in A\ q1$ **and** $\beta \in A\ q2$

then have *converge M1* ($f\ q1$) α **and** *converge M1* ($f\ q2$) β
by (*simp-all add*: A)
moreover have $\neg$ *converge M1* ($f\ q1$) ($f\ q2$)
using *non-initialized-state-cover-assignment-diverges*[*OF assms*(*1*,*3*) *f1 f2* ‹*q1* $\in$ *reachable-states M1*› ‹*q2* $\in$ *reachable-states M1*› ‹*q1* $\neq$ *q2*›] **.**
ultimately have $\neg$ *converge M1* $\alpha\ \beta$
unfolding *converge.simps* **by** *blast*
moreover have $\alpha \in L\ M1 \cap SC$
using ‹$\alpha \in A\ q1$› **by** (*simp add*: A)
moreover have $\beta \in L\ M1 \cap SC$
using ‹$\beta \in A\ q2$› **by** (*simp add*: A)

ultimately have $\neg$ *converge M2* α β
using ‹*preserves-divergence M1 M2 SC*›
unfolding *preserves-divergence.simps*
by *blast*

have $\alpha \in L$ *M2* **and** $\beta \in L$ *M2*
using ‹$\alpha \in L$ *M1* $\cap$ *SC*› ‹$\beta \in L$ *M1* $\cap$ *SC*› ‹*SC* $\subseteq$ *T*› ‹*L M1* $\cap$ *T* = *L M2* $\cap$ *T*› **by** *blast+*

have *io-targets M2* α (*initial M2*) = {*after-initial M2* α}
using *observable-io-targets*[*OF* ‹*observable M2*› ‹$\alpha \in L$ *M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹$\alpha \in L$ *M2*›]
by (*metis the-elem-eq*)
moreover have *io-targets M2* β (*initial M2*) = {*after-initial M2* β}
using *observable-io-targets*[*OF* ‹*observable M2*› ‹$\beta \in L$ *M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹$\beta \in L$ *M2*›]
by (*metis the-elem-eq*)
ultimately show *io-targets M2* α (*initial M2*) $\cap$ *io-targets M2* β (*initial M2*) = {}
using ‹$\neg$ *converge M2* α β› **unfolding** *convergence-minimal*[*OF assms*(*4*,*2*) ‹$\alpha \in L$ *M2*› ‹$\beta \in L$ *M2*›]
by (*metis Int-insert-right-if0 inf-bot-right singletonD*)
qed

then show *Q q1* $\cap$ *Q q2* = {}
unfolding *Q* **by** *blast*
qed
then have $\bigwedge$ *q* . *Uniq* ($\lambda q'$. $q' \in$ *reachable-states M1* $\wedge$ $q \in$ *Q* q')
unfolding *Uniq-def*
by *blast*

define *partition* **where** *partition*: *partition* = (λ *q* . *if* $\exists$ $q' \in$ *reachable-states M1* . $q \in$ *Q* q'
then Q (*THE* q' . $q' \in$ *reachable-states M1* $\wedge$ $q \in$ *Q* q')
else (*states M2* − ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)))

have *is-eq*: *equivalence-relation-on-states M2 partition*
proof−
let *?r* = {(*q1*,*q2*) | *q1 q2* . *q1* $\in$ *states M2* $\wedge$ *q2* $\in$ *partition q1*}

have $\bigwedge$ *q* .*partition q* $\subseteq$ *states M2*
proof −
fix *q* **show** *partition q* $\subseteq$ *states M2*
proof (*cases* $\exists$ $q' \in$ *reachable-states M1* . $q \in$ *Q* q')
case *True*

then have *partition q* = *Q* (*THE q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)
unfolding *partition* **by** *simp*
then show *?thesis*
using *True* ‹⋀ *q* . *Uniq* (λ*q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)›
by (*metis* (*no-types*, *lifting*) *Q SUP-least io-targets-states*)
next
case *False*
then show *?thesis* **unfolding** *partition*
by *auto*
qed
qed

have ⋀ *q* . *q* ∈ *states M2* ⟹ *q* ∈ *partition q*
proof −
fix *q* **assume** *q* ∈ *states M2*
show *q* ∈ *partition q*
proof (*cases* ∃ *q′* ∈ *reachable-states M1* . *q* ∈ *Q q′*)
case *True*
then have *partition q* = *Q* (*THE q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)
unfolding *partition* **by** *simp*
then show *?thesis*
using *True* ‹⋀ *q* . *Uniq* (λ*q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)›
using *the1-equality′* **by** *fastforce*
next
case *False*
then show *?thesis* **unfolding** *partition*
using ‹*q* ∈ *states M2*›
by *simp*
qed
qed

have ⋀ *q q′* . *q* ∈ *states M2* ⟹ *q′* ∈ *partition q* ⟹ *q* ∈ *partition q′*
proof −
fix *q q′* **assume** *q* ∈ *states M2* **and** *q′* ∈ *partition q*
show *q* ∈ *partition q′*
proof (*cases* ∃ *q′* ∈ *reachable-states M1* . *q* ∈ *Q q′*)
case *True*
then have *partition q* = *Q* (*THE q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)
unfolding *partition* **by** *simp*
then obtain *q1* **where** *partition q* = *Q q1* **and** *q1* ∈ *reachable-states M1* **and** *q* ∈ *Q q1*
using *True* ‹⋀ *q* . *Uniq* (λ*q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)›
using *the1-equality′* **by** *fastforce*
then have *q′* ∈ *Q q1*
using ‹*q′* ∈ *partition q*› **by** *auto*
then have *partition q′* = *Q q1*
using ‹*q1* ∈ *reachable-states M1*›
using *the1-equality′*[*OF* ‹⋀ *q* . *Uniq* (λ*q′* . *q′* ∈ *reachable-states M1* ∧ *q* ∈ *Q q′*)›]

unfolding *partition*
by *auto*
then show *?thesis*
using ‹$q \in Q\ q1$› ‹$q' \in partition\ q$› ‹$partition\ q = Q\ q1$› **by** *blast*
next
case *False*
then show *?thesis*
using ‹$q \in states\ M2$› ‹$q' \in partition\ q$›
by (*simp add*: *partition*)
qed
qed

have $\bigwedge q\ q'\ q''\ .\ q \in states\ M2 \Longrightarrow q' \in partition\ q \Longrightarrow q'' \in partition\ q' \Longrightarrow q'' \in partition\ q$
proof −
fix $q\ q'\ q''$
assume $q \in states\ M2$ **and** $q' \in partition\ q$ **and** $q'' \in partition\ q'$
show $q'' \in partition\ q$
proof (*cases* $\exists\ q' \in reachable\text{-}states\ M1\ .\ q \in Q\ q'$)
case *True*
then have $partition\ q = Q\ (THE\ q'\ .\ q' \in reachable\text{-}states\ M1 \wedge q \in Q\ q')$
unfolding *partition* **by** *simp*
then obtain *q1* **where** $partition\ q = Q\ q1$ **and** $q1 \in reachable\text{-}states\ M1$ **and** $q \in Q\ q1$
using *True* ‹$\bigwedge q\ .\ Uniq\ (\lambda q'\ .\ q' \in reachable\text{-}states\ M1 \wedge q \in Q\ q')$›
using *the1-equality'* **by** *fastforce*
then have $q' \in Q\ q1$
using ‹$q' \in partition\ q$› **by** *auto*
then have $partition\ q' = Q\ q1$
using ‹$q1 \in reachable\text{-}states\ M1$›
using *the1-equality'*[*OF* ‹$\bigwedge q\ .\ Uniq\ (\lambda q'\ .\ q' \in reachable\text{-}states\ M1 \wedge q \in Q\ q')$›]
unfolding *partition*
by *auto*
then have $q'' \in Q\ q1$
using ‹$q'' \in partition\ q'$› **by** *auto*
then have $partition\ q'' = Q\ q1$
using ‹$q1 \in reachable\text{-}states\ M1$›
using *the1-equality'*[*OF* ‹$\bigwedge q\ .\ Uniq\ (\lambda q'\ .\ q' \in reachable\text{-}states\ M1 \wedge q \in Q\ q')$›]
unfolding *partition*
by *auto*
then show *?thesis*
unfolding ‹$partition\ q = Q\ q1$›
using ‹$q'' \in Q\ q1$› **by** *blast*
next
case *False*
then show *?thesis*

using ‹$q \in$ *states M2*› ‹$q' \in$ *partition q*› ‹$q'' \in$ *partition q'*›
by (*simp add*: *partition*)
qed
qed

have *refl-on* (*states M2*) *?r* **unfolding** *refl-on-def*
proof −
have $\{(q1, q2) \mid q1\ q2.\ q1 \in FSM.states\ M2 \wedge q2 \in partition\ q1\} \subseteq FSM.states\ M2 \times FSM.states\ M2$
using ‹$\bigwedge q\ .partition\ q \subseteq states\ M2$› **by** *blast*
show $\forall x{\in}FSM.states\ M2.\ (x, x) \in \{(q1, q2) \mid q1\ q2.\ q1 \in FSM.states\ M2 \wedge q2 \in partition\ q1\}$
proof
fix *q* **assume** $q \in states\ M2$
then show $(q,q) \in \{(q1, q2) \mid q1\ q2.\ q1 \in FSM.states\ M2 \wedge q2 \in partition\ q1\}$
using ‹$\bigwedge q\ .\ q \in states\ M2 \Longrightarrow q \in partition\ q$›
by *blast*
qed
qed
moreover have *sym ?r*
unfolding *sym-def*
using ‹$\bigwedge q\ q'\ .\ q \in states\ M2 \Longrightarrow q' \in partition\ q \Longrightarrow q \in partition\ q'$› ‹$\bigwedge q\ .partition\ q \subseteq states\ M2$›
by *blast*
moreover have *trans ?r*
unfolding *trans-def*
using ‹$\bigwedge q\ q'\ q''\ .\ q \in states\ M2 \Longrightarrow q' \in partition\ q \Longrightarrow q'' \in partition\ q' \Longrightarrow q'' \in partition\ q$›
by *blast*
ultimately show *?thesis*
unfolding *equivalence-relation-on-states-def equiv-def*
using ‹$\bigwedge q\ .partition\ q \subseteq states\ M2$› **by** *blast*
qed

define *n0* **where** *n0*: $n0 = card\ (partition\ `\ states\ M2)$
have $n0 \leq Suc\ (size\text{-}r\ M1)$ **and** $n0 \geq size\text{-}r\ M1$
proof −
have $partition\ `\ states\ M2 \subseteq insert\ (states\ M2 - (\bigcup\ q \in reachable\text{-}states\ M1\ .\ Q\ q))\ (Q\ `\ reachable\text{-}states\ M1)$
proof
fix *X* **assume** $X \in partition\ `\ states\ M2$
then obtain *q* **where** $q \in states\ M2$ **and** $X = partition\ q$
by *blast*

show $X \in$ *insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*)
proof (*cases* $\exists\, q' \in$*reachable-states M1*. $q \in Q\ q'$)
case *True*
then show *?thesis*
unfolding ‹X = *partition q*› *partition*
using *the1-equality'*[*OF* ‹$\bigwedge$ *q* . *Uniq* ($\lambda q'$. $q' \in$ *reachable-states M1* $\wedge$ $q \in Q\ q'$)›]
by *auto*
next
case *False*
then show *?thesis*
unfolding ‹X = *partition q*› *partition*
by *auto*
qed
qed
moreover have *card* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*)) $\leq$ *Suc* (*size-r M1*)
and *finite* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*))
and *card* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*)) $\geq$ *size-r M1*
proof $-$
have *finite* (*Q* ‘ *reachable-states M1*)
using *fsm-states-finite*[*of M1*]
by (*metis finite-imageI fsm-states-finite restrict-to-reachable-states-simps*(*2*))

moreover have *card* (*Q* ‘ *reachable-states M1*) = *size-r M1*
proof $-$
have *card* (*Q* ‘ *reachable-states M1*) $\leq$ *size-r M1*
by (*metis card-image-le fsm-states-finite restrict-to-reachable-states-simps*(*2*))
moreover have *card* (*Q* ‘ *reachable-states M1*) $\geq$ *size-r M1*
using ‹*finite* (*Q* ‘ *reachable-states M1*)›
by (*metis* (*full-types*) ‹$\bigwedge q.\ q \in$ *reachable-states M1* $\Longrightarrow$ $Q\ q \neq \{\}$› ‹$\bigwedge q2\ q1.\ [\![q1 \in$ *reachable-states M1*; $q2 \in$ *reachable-states M1*; $q1 \neq q2]\!] \Longrightarrow Q\ q1 \cap Q\ q2 = \{\}$› *calculation card-eq-0-iff card-union-of-distinct le-0-eq*)
ultimately show *?thesis*
by *simp*
qed
ultimately show *card* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*)) $\leq$ *Suc* (*size-r M1*)
and *card* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*)) $\geq$ *size-r M1*
by (*simp add*: *card-insert-if*)+
show *finite* (*insert* (*states M2* $-$ ($\bigcup$ $q \in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*))
using ‹*finite* (*Q* ‘ *reachable-states M1*)›
by *blast*
qed

ultimately show *n0* $\leq$ *Suc* (*size-r M1*) **unfolding** *n0*
by (*meson card-mono le-trans*)

have (*Q* ‘ *reachable-states M1*) $\subseteq$ *partition* ‘ *states M2*
proof
fix *x* **assume** *x* $\in$ (*Q* ‘ *reachable-states M1*)
then obtain *q′* **where** *q′* $\in$ *reachable-states M1* **and** *x* = *Q q′*
by *blast*
then obtain *q* **where** *q* $\in$ *Q q′*
using ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *Q q* $\neq$ {}› **by** *blast*
then obtain α **where** α $\in$ *A q′* **and** *q* $\in$ *io-targets M2* α (*initial M2*)
unfolding *Q* **by** *blast*
then have *q* $\in$ *states M2*
by (*meson io-targets-states subset-iff*)

have $\exists$ *q′*$\in$*reachable-states M1*. *q* $\in$ *Q q′*
using ‹*q′* $\in$ *reachable-states M1*› ‹*q* $\in$ *Q q′*› **by** *blast*
then have *partition q* = *Q q′*
unfolding *partition*
using *the1-equality′*[*OF* ‹$\bigwedge$ *q* . *Uniq* (λ*q′* . *q′* $\in$ *reachable-states M1* $\wedge$ *q* $\in$ *Q q′*)›, *of q′ q*] ‹*q* $\in$ *Q q′*› ‹*q′* $\in$ *reachable-states M1*›
by *auto*
then show *x* $\in$ *partition* ‘ *states M2*
using ‹*q* $\in$ *states M2*› ‹*x* = *Q q′*›
by *blast*
qed
then show *n0* $\geq$ *size-r M1*
unfolding *n0*
using ‹*finite* (*insert* (*states M2* − ($\bigcup$ *q* $\in$ *reachable-states M1* . *Q q*)) (*Q* ‘ *reachable-states M1*))›
by (*metis* (*full-types*) ‹$\bigwedge$*q*. *q* $\in$ *reachable-states M1* $\Longrightarrow$ *Q q* $\neq$ {}› ‹$\bigwedge$*q2 q1*. ⟦*q1* $\in$ *reachable-states M1*; *q2* $\in$ *reachable-states M1*; *q1* $\neq$ *q2*⟧ $\Longrightarrow$ *Q q1* $\cap$ *Q q2* = {}› ‹*partition* ‘ *FSM.states M2* $\subseteq$ *insert* (*FSM.states M2* − $\bigcup$ (*Q* ‘ *reachable-states M1*)) (*Q* ‘ *reachable-states M1*)› *card-mono card-union-of-distinct finite-subset fsm-states-finite restrict-to-reachable-states-simps*(*2*))
qed

moreover have *after-initial M2* τ $\in$ *ofsm-table M2 partition* (*m* − *size-r M1*) (*after-initial M2* π)
proof −

define *q1* **where** *q1*: *q1* = (*after-initial M2* π)
define *q2* **where** *q2*: *q2* = (*after-initial M2* τ)

have π $\in$ *L M2* **and** τ $\in$ *L M2*
using *assms*(*7*,*8*,*9*) **by** *blast*+

have *q1* ∈ *states M2*
using ‹$\pi \in L$ *M2*› *after-is-state*[*OF* ‹*observable M2*›] **unfolding** *q1* **by** *blast*
have *q2* ∈ *states M2*
using ‹$\tau \in L$ *M2*› *after-is-state*[*OF* ‹*observable M2*›] **unfolding** *q2* **by** *blast*

moreover have $\bigwedge \gamma$. *length* $\gamma \leq m -$ *size-r M1* $\Longrightarrow$ ($\gamma \in$ *LS M2 q1*) = ($\gamma \in$ *LS M2 q2*) $\wedge$ ($\gamma \in$ *LS M2 q1* $\longrightarrow$ *after M2 q2* $\gamma \in$ *partition* (*after M2 q1* γ))
proof −
fix γ :: ($'b \times 'c$) *list*
assume *length* $\gamma \leq m -$ *size-r M1*

then show (($\gamma \in$ *LS M2 q1*) = ($\gamma \in$ *LS M2 q2*)) $\wedge$ ($\gamma \in$ *LS M2 q1* $\longrightarrow$ *after M2 q2* $\gamma \in$ *partition* (*after M2 q1* γ))
proof (*induction* γ *rule*: *rev-induct*)
case *Nil*

show *?case*
proof

have ([] ∈ *LS M2 q1*) **and** ([] ∈ *LS M2 q2*)
using ‹*q1* ∈ *states M2*› ‹*q2* ∈ *states M2*›
by *auto*
then have *after M2 q1* [] = *q1* **and** *after M2 q2* [] = *q2*
unfolding *Nil*
by *auto*

obtain α β **where** *converge M1* α π
converge M2 α π
converge M1 β τ
converge M2 β τ
$\alpha \in SC$
$\beta \in SC$
using *assms*(*15*) **by** *blast*
then have $\alpha \in L$ *M1* **and** $\beta \in L$ *M1*
by *auto*

have $\alpha \in L$ *M2*
using ‹$\alpha \in L$ *M1*› ‹$\alpha \in SC$› ‹*L M1* ∩ *T* = *L M2* ∩ *T*› ‹*SC* ⊆ *T*› **by** *blast*
have $\beta \in L$ *M2*
using ‹$\beta \in L$ *M1*› ‹$\beta \in SC$› ‹*L M1* ∩ *T* = *L M2* ∩ *T*› ‹*SC* ⊆ *T*› **by** *blast*

have ([] ∈ *LS M2 q1*) = ([] ∈ *LS M2* (*after-initial M2* α))
using ‹*converge M2* α π› **unfolding** *q1 converge.simps* **by** *simp*
also have ... = ([] ∈ *LS M1* (*after-initial M1* α))
using ‹$\alpha \in SC$› ‹*L M1* ∩ *T* = *L M2* ∩ *T*› ‹*SC* ⊆ *T*›

unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\alpha \in L$ *M1*›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\alpha \in L$ *M2*›]
unfolding *Nil*
by *auto*
also have ... = ([] $\in$ *LS M1* (*after-initial M1* β))
using ‹*converge M1* π τ› ‹*converge M1* α π› ‹*converge M1* β τ›
unfolding *converge.simps* **by** *blast*
also have ... = ([] $\in$ *LS M2* (*after-initial M2* β))
using ‹$\beta \in SC$› ‹*L M1* $\cap$ *T* = *L M2* $\cap$ *T*› ‹*SC* $\subseteq$ *T*›
unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\beta \in L$ *M1*›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\beta \in L$ *M2*›]
unfolding *Nil*
by *auto*
also have ... = ([] $\in$ *LS M2 q2*)
using ‹*converge M2* β τ› **unfolding** *q2 converge.simps* **by** *simp*
finally show ([] $\in$ *LS M2 q1*) = ([] $\in$ *LS M2 q2*) **.**

show ([] $\in$ *LS M2 q1* $\longrightarrow$ *after M2 q2* [] $\in$ *partition* (*after M2 q1* []))
proof
assume [] $\in$ *LS M2 q1*
then have [] $\in$ *LS M1* (*after-initial M1* α)
and [] $\in$ *LS M1* (*after-initial M1* β)
unfolding ‹([] $\in$ *LS M2 q1*) = ([] $\in$ *LS M2* (*after-initial M2* α))›
‹([] $\in$ *LS M2* (*after M2* (*FSM.initial M2*) α)) = ([] $\in$ *LS M1* (*after M1* (*FSM.initial M1*) α))›
‹([] $\in$ *LS M1* (*after M1* (*FSM.initial M1*) α)) = ([] $\in$ *LS M1* (*after M1* (*FSM.initial M1*) β))›
by *simp+*

have α@[] $\in$ *L M1*
using ‹[] $\in$ *LS M1* (*after-initial M1* α)› **unfolding** *after-language-iff*[*OF* ‹*observable M1*› ‹$\alpha \in L$ *M1*›] **.**
moreover have β@[] $\in$ *L M1*
using ‹[] $\in$ *LS M1* (*after-initial M1* β)› **unfolding** *after-language-iff*[*OF* ‹*observable M1*› ‹$\beta \in L$ *M1*›] **.**
moreover have *converge M1* α β
using ‹*converge M1* π τ› ‹*converge M1* α π› ‹*converge M1* β τ›
unfolding *converge.simps* **by** *blast*
ultimately have *converge M1* (α@[]) (β@[])
using *converge-append*[*OF* ‹*observable M1*›] *language-prefix*[*of* β [] *M1 initial M1*] **by** *blast*

have (α @ []) $\in$ *L M2* **and** (β @ []) $\in$ *L M2*
using ‹α@[] $\in$ *L M1*› ‹$\alpha \in SC$› ‹β@[] $\in$ *L M1*› ‹$\beta \in SC$ ›‹*L M1* $\cap$ *T* = *L M2* $\cap$ *T*› ‹*SC* $\subseteq$ *T*› **by** *auto*
have *after-initial M1* (α@[]) $\in$ *reachable-states M1*
using *observable-after-path*[*OF* ‹*observable M1*›]
unfolding *reachable-states-def*
proof −

have $\exists$ *ps. after M1* (*FSM.initial M1*) α = *target* (*FSM.initial M1*) *ps* $\wedge$ *path M1* (*FSM.initial M1*) *ps*
by (*metis* (*no-types*) ‹$\bigwedge$*thesis q io.* ⟦*io* $\in$ *LS M1 q*; $\bigwedge$*p.* ⟦*path M1 q p*; *p-io p* = *io*; *target q p* = *after M1 q io*⟧ $\Longrightarrow$ *thesis*⟧ $\Longrightarrow$ *thesis*› ‹$\alpha \in$ *L M1*›)
then show *after M1* (*FSM.initial M1*) (α @ []) $\in$ {*target* (*FSM.initial M1*) *ps* |*ps. path M1* (*FSM.initial M1*) *ps*}
by *auto*
qed
have (α@[]) $\in$ *A* (*after-initial M1* (α@[]))
using *convergence-minimal*[*OF assms*(*3*,*1*) - ‹α@[] $\in$ *L M1*›, *of f* (*after-initial M1* (α@[]))]
using *f2*[*OF* ‹*after-initial M1* (α@[]) $\in$ *reachable-states M1*›]
using ‹$\alpha \in$ *SC*› ‹$\alpha \in$ *L M1*› ‹*after M1* (*FSM.initial M1*) (α @ []) $\in$ *reachable-states M1*›
apply (*auto simp add*: *A Nil*)
apply (*metis* (*lifting*) *ext after-path assms*(*1*) *f1 observable-path-io-target singletonD*)+
done
then have *after M2* (*FSM.initial M2*) (α @ []) $\in$ *Q* (*after-initial M1* (α@[]))
unfolding *Q*
using *observable-io-targets*[*OF* ‹*observable M2*› ‹(α @ []) $\in$ *L M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹(α @ []) $\in$ *L M2*›]
by (*metis UN-iff insertCI the-elem-eq*)
then have $\exists$ *q'*$\in$*reachable-states M1. after M2* (*FSM.initial M2*) (α @ []) $\in$ *Q q'*
using ‹*after-initial M1* (α@[]) $\in$ *reachable-states M1*› **by** *blast*
moreover have (*THE q'. q'* $\in$ *reachable-states M1* $\wedge$ *after M2* (*FSM.initial M2*) (α @ []) $\in$ *Q q'*) = (*after-initial M1* (α@[]))
using ‹*after-initial M1* (α@[]) $\in$ *reachable-states M1*›
using ‹*after M2* (*FSM.initial M2*) (α @ []) $\in$ *Q* (*after-initial M1* (α@[]))›
by (*simp add*: ‹$\bigwedge$*q.* $\exists_{\leq 1}$ *q'. q'* $\in$ *reachable-states M1* $\wedge$ *q* $\in$ *Q q'*› *the1-equality'*)
moreover have *after M2* (*FSM.initial M2*) (β @ []) $\in$ *Q* (*after-initial M1* (α@[]))
proof −
have (β@[]) $\in$ *A* (*after-initial M1* (α@[]))
using *A* ‹α @ [] $\in$ *A* (*after M1* (*FSM.initial M1*) (α @ []))› ‹β @ [] $\in$ *L M1*› ‹$\beta \in$ *SC*› ‹*converge M1* (α @ []) (β @ [])› **unfolding** *Nil* **by** *auto*
then show *?thesis*
unfolding *Q*
using *observable-io-targets*[*OF* ‹*observable M2*› ‹(β @ []) $\in$ *L M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹(β @ []) $\in$ *L M2*›]
by (*metis UN-iff insertCI the-elem-eq*)
qed
ultimately have *after-initial M2* (β@[]) $\in$ *partition* (*after-initial M2* (α@[]))
unfolding *partition*

by *presburger*
moreover have *after-initial M2* (α@[]) = *after-initial M2* (π@[])
using *converge-append*[*OF assms*(*2*) ‹*converge M2* α π› ‹(α @ []) $\in$ *L M2*› ‹π $\in$ *L M2*›]
unfolding *convergence-minimal*[*OF assms*(*4*,*2*) ‹(α @ []) $\in$ *L M2*› *converge-extend*[*OF assms*(*2*) ‹*converge M2* α π› ‹(α @ []) $\in$ *L M2*› ‹π $\in$ *L M2*›]]
.
moreover have *after-initial M2* (β@[]) = *after-initial M2* (τ@[])
using *converge-append*[*OF assms*(*2*) ‹*converge M2* β τ› ‹(β @ []) $\in$ *L M2*› ‹τ $\in$ *L M2*›]
unfolding *convergence-minimal*[*OF assms*(*4*,*2*) ‹(β @ []) $\in$ *L M2*› *converge-extend*[*OF assms*(*2*) ‹*converge M2* β τ› ‹(β @ []) $\in$ *L M2*› ‹τ $\in$ *L M2*›]]
.
ultimately show *after M2 q2* [] $\in$ *partition* (*after M2 q1* [])
unfolding *q1 q2*
unfolding *after-split*[*OF assms*(*2*) *converge-extend*[*OF assms*(*2*) ‹*converge M2* α π› ‹(α @ []) $\in$ *L M2*› ‹π $\in$ *L M2*›]]
unfolding *after-split*[*OF assms*(*2*) *converge-extend*[*OF assms*(*2*) ‹*converge M2* β τ› ‹(β @ []) $\in$ *L M2*› ‹τ $\in$ *L M2*›]]
by *simp*
qed
qed
next
case (*snoc xy* γ)

obtain *x y* **where** *xy* = (*x*,*y*)
by *fastforce*

show *?case* **proof** (*cases* $\forall$ x' y' . (x',y') $\in$ *set* (γ@[(*x*,*y*)]) $\longrightarrow$ x' $\in$ *inputs M1* $\wedge$ y' $\in$ *outputs M1*)
case *False*

have γ@[(*x*,*y*)] $\notin$ *LS M2 q1* **and** γ@[(*x*,*y*)] $\notin$ *LS M2 q2*
using *language-io*[*of* γ@[(*x*,*y*)] *M2* -] *False*
unfolding ‹*inputs M2* = *inputs M1*› ‹*outputs M2* = *outputs M1*›
by *blast*+
then show *?thesis*
unfolding ‹*xy* = (*x*,*y*)›
by *blast*
next
case *True*

define *s1* **where** *s1*: *s1* = (*after-initial M1* π)
define *s2* **where** *s2*: *s2* = (*after-initial M1* τ)

have *s1* $\in$ *states M1*
using ‹π $\in$ *L M1* $\cap$ *T*› *after-is-state*[*OF* ‹*observable M1*›] **unfolding** *s1* **by** *blast*
have *s2* $\in$ *states M1*

using ‹$\tau \in L\ M1 \cap T$› *after-is-state*[*OF* ‹*observable M1*›] **unfolding** *s2* **by** *blast*

show *?thesis* **proof** (*cases* $\gamma \in LS\ M1\ s1$)
case *False*

obtain $io'\ x'\ y'\ io''$ **where** $\gamma = io'\ @\ [(x',\ y')]\ @\ io''$
and $io' \in LS\ M1\ s1$
and $io'\ @\ [(x',\ y')] \notin LS\ M1\ s1$
using *language-maximal-contained-prefix-ob*[*OF False* ‹$s1 \in states\ M1$› ‹*observable M1*›]
by *blast*

have $*$: $length\ io' < m - size\text{-}r\ M1$
using ‹$length\ (\gamma\ @\ [xy]) \leq m - size\text{-}r\ M1$›
unfolding ‹$\gamma = io'\ @\ [(x',\ y')]\ @\ io''$›
by *auto*
have $**$: $io' \in LS\ M1\ (after\ M1\ (FSM.initial\ M1)\ \pi)$
using ‹$io' \in LS\ M1\ s1$› **unfolding** *s1* **.**
have $x' \in inputs\ M1$ **and** $y' \in outputs\ M1$
using *True*
unfolding ‹$\gamma = io'\ @\ [(x',\ y')]\ @\ io''$›
by *auto*

obtain $\alpha\ \beta$ **where** $converge\ M1\ \alpha\ (\pi\ @\ io')$
$converge\ M2\ \alpha\ (\pi\ @\ io')$
$converge\ M1\ \beta\ (\tau\ @\ io')$
$converge\ M2\ \beta\ (\tau\ @\ io')$
$\alpha \in SC$
$\alpha\ @\ [(x',\ y')] \in SC$
$\beta \in SC$
$\beta\ @\ [(x',\ y')] \in SC$
using *assms*(*14*)[*OF* $*$ $**$ ‹$x' \in inputs\ M1$› ‹$y' \in outputs\ M1$›]
by *blast*
then have $\alpha \in L\ M1$ **and** $\beta \in L\ M1$
by *auto*

have $\pi@io' \in L\ M1$
using ‹$io' \in LS\ M1\ s1$› ‹$\pi \in L\ M1 \cap T$›
using *after-language-iff*[*OF* ‹*observable M1*›, *of* π *initial M1* io']
unfolding *s1*
by *blast*
have $converge\ M1\ (\pi\ @\ io')\ (\tau\ @\ io')$
using *converge-append*[*OF* ‹*observable M1*› ‹$converge\ M1\ \pi\ \tau$› ‹$\pi@io' \in L\ M1$›]
using ‹$\tau \in L\ M1 \cap T$›
by *blast*

have $(\pi$ @ $io')$ @ $[(x',\ y')] \notin L\ M1$
using ‹io' @ $[(x',\ y')] \notin LS\ M1\ s1$›
using ‹$\pi \in L\ M1 \cap T$›
using *after-language-iff*[*OF* ‹*observable M1*›, *of* π *initial M1* io'@$[(x',y')]$]
unfolding *s1*
by *auto*
then have $[(x',y')] \notin LS\ M1$ (*after-initial M1* α)
using *after-language-iff*[*OF* ‹*observable M1*› ‹π@$io' \in L\ M1$›, *of* $[(x',y')]$]
using ‹*converge M1* α (π @ io')›
unfolding *converge.simps*
by *blast*
then have $[(x',y')] \notin LS\ M1$ (*after-initial M1* β)
using ‹*converge M1* (π @ io') (τ @ io')› ‹*converge M1* α (π @ io')› ‹*converge M1* β (τ @ io')›
unfolding *converge.simps*
by *blast*

have $\alpha \in L\ M2$
using ‹$\alpha \in L\ M1$› ‹$\alpha \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$› **by** *blast*
have $\beta \in L\ M2$
using ‹$\beta \in L\ M1$› ‹$\beta \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$› **by** *blast*

have $[(x',y')] \notin LS\ M2$ (*after-initial M2* α)
using ‹α @ $[(x',\ y')] \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$›
‹$[(x',y')] \notin LS\ M1$ (*after-initial M1* α)›
unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\alpha \in L\ M1$›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\alpha \in L\ M2$›]
by *blast*
then have io'@$[(x',y')] \notin LS\ M2\ q1$
using ‹*converge M2* α (π @ io')›
unfolding *q1 converge.simps*
using *after-language-append-iff assms*(*2*) **by** *blast*
then have γ@$[xy] \notin LS\ M2\ q1$
unfolding ‹$\gamma = io'$ @ $[(x',\ y')]$ @ io''›
using *language-prefix*
by (*metis append-assoc*)

have $[(x',y')] \notin LS\ M2$ (*after-initial M2* β)
using ‹β @ $[(x',\ y')] \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$›
‹$[(x',y')] \notin LS\ M1$ (*after-initial M1* β)›
unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\beta \in L\ M1$›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\beta \in L\ M2$›]
by *blast*

then have $io'@[(x',y')] \notin LS\ M2\ q2$
using ‹*converge M2* β (τ @ io')›
unfolding *q2 converge.simps*
using *after-language-append-iff assms*(*2*) **by** *blast*
then have $\gamma@[xy] \notin LS\ M2\ q2$
unfolding ‹$\gamma = io'$ @ [(x', y')] @ io''›
using *language-prefix*
by (*metis append-assoc*)

then show *?thesis*
using ‹$\gamma@[xy] \notin LS\ M2\ q1$›
by *blast*
next
case *True*

have $*$: *length* $\gamma < m -$ *size-r M1*
using ‹*length* (γ @ [xy]) $\leq m -$ *size-r M1*›
by *auto*
have $**$: $\gamma \in LS\ M1$ (*after M1* (*FSM.initial M1*) π)
using *True* **unfolding** *s1* **.**
have $x \in$ *inputs M1* **and** $y \in$ *outputs M1*
using ‹$\forall\ x'\ y'$. (x',y') $\in$ *set* ($\gamma@[(x,y)]$) $\longrightarrow x' \in$ *inputs M1* $\wedge\ y' \in$ *outputs M1*›
by *auto*

obtain α β **where** *converge M1* α (π @ γ)
converge M2 α (π @ γ)
converge M1 β (τ @ γ)
converge M2 β (τ @ γ)
$\alpha \in SC$
α @ [xy] $\in SC$
$\beta \in SC$
β @ [xy] $\in SC$
using *assms*(*14*)[*OF* $*$ $**$ ‹$x \in$ *inputs M1*› ‹$y \in$ *outputs M1*›]
unfolding ‹$xy = (x,y)$›
by *blast*
then have $\alpha \in L\ M1$ **and** $\beta \in L\ M1$
by *auto*

have $\alpha \in L\ M2$
using ‹$\alpha \in L\ M1$› ‹$\alpha \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$› **by** *blast*
have $\beta \in L\ M2$
using ‹$\beta \in L\ M1$› ‹$\beta \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$› **by** *blast*

have (π @ γ) $\in L\ M2$
using ‹*converge M2* α (π @ γ)› **by** *auto*

have $(\tau$ @ $\gamma) \in L\ M2$
using ‹*converge M2* β $(\tau$ @ $\gamma)$› **by** *auto*

have *converge M1* $(\pi$ @ $\gamma)$ $(\tau$ @ $\gamma)$
using *converge-append*[*OF* ‹*observable M1*› ‹*converge M1* π τ›, *of* γ]
using ‹*converge M1* α $(\pi$ @ $\gamma)$› ‹$\tau \in L\ M1 \cap T$›
by *auto*

have $(\gamma@[xy] \in LS\ M2\ q1) = ([xy] \in LS\ M2$ (*after-initial M2* $(\pi@\gamma)))$
unfolding *q1*
using *after-language-append-iff*[*OF* ‹*observable M2*› ‹$(\pi$ @ $\gamma) \in L\ M2$›]
by *auto*
also have ... $= ([xy] \in LS\ M2$ (*after-initial M2* α))
using ‹*converge M2* α $(\pi$ @ $\gamma)$› **unfolding** *q1 converge.simps*
by *blast*
also have ... $= ([xy] \in LS\ M1$ (*after-initial M1* α))
using ‹$\alpha@[xy] \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$›
unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\alpha \in L\ M1$›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\alpha \in L\ M2$›]
by *blast*
also have ... $= ([xy] \in LS\ M1$ (*after-initial M1* β))
using ‹*converge M1* $(\pi$ @ $\gamma)$ $(\tau$ @ $\gamma)$› ‹*converge M1* α $(\pi$ @ $\gamma)$› ‹*converge M1* β $(\tau$ @ $\gamma)$›
unfolding *converge.simps*
by *blast*
also have ... $= ([xy] \in LS\ M2$ (*after-initial M2* β))
using ‹$\beta@[xy] \in SC$› ‹$L\ M1 \cap T = L\ M2 \cap T$› ‹$SC \subseteq T$›
unfolding *after-language-iff*[*OF* ‹*observable M1*› ‹$\beta \in L\ M1$›]
unfolding *after-language-iff*[*OF* ‹*observable M2*› ‹$\beta \in L\ M2$›]
by *blast*
also have ... $= ([xy] \in LS\ M2$ (*after-initial M2* $(\tau@\gamma)))$
using ‹*converge M2* β $(\tau$ @ $\gamma)$› **unfolding** *q1 converge.simps*
by *blast*
also have ... $= (\gamma@[xy] \in LS\ M2\ q2)$
unfolding *q2*
using *after-language-append-iff*[*OF* ‹*observable M2*› ‹$(\tau$ @ $\gamma) \in L\ M2$›]
by *auto*
finally have *p1*: $(\gamma@[xy] \in LS\ M2\ q1) = (\gamma@[xy] \in LS\ M2\ q2)$
.

moreover have $(\gamma@[xy] \in LS\ M2\ q1 \longrightarrow$ *after M2 q2* $(\gamma@[xy]) \in$ *partition* (*after M2 q1* $(\gamma@[xy])))$
proof
assume $\gamma@[xy] \in LS\ M2\ q1$
then have $[xy] \in LS\ M1$ (*after-initial M1* α)
and $[xy] \in LS\ M1$ (*after-initial M1* β)
unfolding ‹$(\gamma@[xy] \in LS\ M2\ q1) = ([xy] \in LS\ M2$ (*after-initial M2* $(\pi@\gamma)))$›
‹$([xy] \in LS\ M2$ (*after-initial M2* $(\pi@\gamma))) = ([xy] \in LS\ M2$

(*after-initial M2* α))›
‹([*xy*] ∈ *LS M2* (*after M2* (*FSM.initial M2*) α)) = ([*xy*] ∈ *LS M1* (*after M1* (*FSM.initial M1*) α))›
‹([*xy*] ∈ *LS M1* (*after M1* (*FSM.initial M1*) α)) = ([*xy*] ∈ *LS M1* (*after M1* (*FSM.initial M1*) β))›
by *simp+*

have α@[*xy*] ∈ *L M1*
using ‹[*xy*] ∈ *LS M1* (*after-initial M1* α)› **unfolding** *after-language-iff*[*OF* ‹*observable M1*› ‹α ∈ *L M1*›] **.**
moreover have β@[*xy*] ∈ *L M1*
using ‹[*xy*] ∈ *LS M1* (*after-initial M1* β)› **unfolding** *after-language-iff*[*OF* ‹*observable M1*› ‹β ∈ *L M1*›] **.**
moreover have *converge M1* α β
using ‹*converge M1* (π @ γ) (τ @ γ)› ‹*converge M1* α (π @ γ)› ‹*converge M1* β (τ @ γ)›
unfolding *converge.simps*
by *blast*
ultimately have *converge M1* (α@[*xy*]) (β@[*xy*])
using *converge-append*[*OF* ‹*observable M1*›] *language-prefix*[*of* β [*xy*] *M1 initial M1*] **by** *blast*

have (α @ [*xy*]) ∈ *L M2* **and** (β @ [*xy*]) ∈ *L M2*
using ‹α@[*xy*] ∈ *L M1*› ‹α@[*xy*] ∈ *SC*› ‹β@[*xy*] ∈ *L M1*› ‹β@[*xy*] ∈ *SC* ›‹*L M1* ∩ *T* = *L M2* ∩ *T*› ‹*SC* ⊆ *T*› **by** *blast+*
have *after-initial M1* (α@[*xy*]) ∈ *reachable-states M1*
using *observable-after-path*[*OF* ‹*observable M1*› ‹α@[*xy*] ∈ *L M1*›]
unfolding *reachable-states-def*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
have (α@[*xy*]) ∈ *A* (*after-initial M1* (α@[*xy*]))
using *convergence-minimal*[*OF assms*(*3*,*1*) - ‹α@[*xy*] ∈ *L M1*›, *of f* (*after-initial M1* (α@[*xy*]))]
using *f2*[*OF* ‹*after-initial M1* (α@[*xy*]) ∈ *reachable-states M1*›]
using ‹α@[*xy*] ∈ *SC*› ‹α @ [*xy*] ∈ *L M1*› ‹*after M1* (*FSM.initial M1*) (α @ [*xy*]) ∈ *reachable-states M1*› *assms*(*1*)
apply (*auto simp add*: *A*)
apply (*metis* (*no-types*, *lifting*) *ext after-path assms*(*1*) *f1 observable-path-io-target*
singletonD)+
done
then have *after M2* (*FSM.initial M2*) (α @ [*xy*]) ∈ *Q* (*after-initial M1* (α@[*xy*]))
unfolding *Q*
using *observable-io-targets*[*OF* ‹*observable M2*› ‹(α @ [*xy*]) ∈ *L M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹(α @ [*xy*]) ∈ *L M2*›]
by (*metis UN-iff insertCI the-elem-eq*)
then have ∃ *q'*∈*reachable-states M1*. *after M2* (*FSM.initial M2*) (α @ [*xy*]) ∈ *Q q'*
using ‹*after-initial M1* (α@[*xy*]) ∈ *reachable-states M1*› **by** *blast*

moreover have (*THE* q'. $q' \in$ *reachable-states M1* $\wedge$ *after M2* (*FSM.initial M2*) (α @ [*xy*]) $\in$ *Q* q') = (*after-initial M1* (α@[*xy*]))
using ‹*after-initial M1* (α@[*xy*]) $\in$ *reachable-states M1*›
using ‹*after M2* (*FSM.initial M2*) (α @ [*xy*]) $\in$ *Q* (*after-initial M1* (α@[*xy*]))›
by (*simp add*: ‹$\bigwedge q$. $\exists_{\leq 1}$ q'. $q' \in$ *reachable-states M1* $\wedge$ $q \in$ *Q* q'› *the1-equality'*)
moreover have *after M2* (*FSM.initial M2*) (β @ [*xy*]) $\in$ *Q* (*after-initial M1* (α@[*xy*]))
proof −
have (β@[*xy*]) $\in$ *A* (*after-initial M1* (α@[*xy*]))
using *A* ‹α @ [*xy*] $\in$ *A* (*after M1* (*FSM.initial M1*) (α @ [*xy*]))› ‹β @ [*xy*] $\in$ *L M1*› ‹β @ [*xy*] $\in$ *SC*› ‹*converge M1* (α @ [*xy*]) (β @ [*xy*])› **by** *auto*
then show *?thesis*
unfolding *Q*
using *observable-io-targets*[*OF* ‹*observable M2*› ‹(β @ [*xy*]) $\in$ *L M2*›]
unfolding *after-io-targets*[*OF* ‹*observable M2*› ‹(β @ [*xy*]) $\in$ *L M2*›]
by (*metis UN-iff insertCI the-elem-eq*)
qed
ultimately have *after-initial M2* (β@[*xy*]) $\in$ *partition* (*after-initial M2* (α@[*xy*]))
unfolding *partition*
by *presburger*
moreover have *after-initial M2* (α@[*xy*]) = *after-initial M2* ((π@γ)@[*xy*])
using *converge-append*[*OF assms*(*2*) ‹*converge M2* α (π@γ)› ‹(α @ [*xy*]) $\in$ *L M2*› ‹(π@γ) $\in$ *L M2*›]
unfolding *convergence-minimal*[*OF assms*(*4*,*2*) ‹(α @ [*xy*]) $\in$ *L M2*› *converge-extend*[*OF assms*(*2*) ‹*converge M2* α (π@γ)› ‹(α @ [*xy*]) $\in$ *L M2*› ‹(π@γ) $\in$ *L M2*›]]
.
moreover have *after-initial M2* (β@[*xy*]) = *after-initial M2* ((τ@γ)@[*xy*])
using *converge-append*[*OF assms*(*2*) ‹*converge M2* β (τ@γ)› ‹(β @ [*xy*]) $\in$ *L M2*› ‹(τ@γ) $\in$ *L M2*›]
unfolding *convergence-minimal*[*OF assms*(*4*,*2*) ‹(β @ [*xy*]) $\in$ *L M2*› *converge-extend*[*OF assms*(*2*) ‹*converge M2* β (τ@γ)› ‹(β @ [*xy*]) $\in$ *L M2*› ‹(τ@γ) $\in$ *L M2*›]]
.
ultimately show *after M2 q2* (γ@[*xy*]) $\in$ *partition* (*after M2 q1* (γ@[*xy*]))
unfolding *q1 q2*
unfolding *after-split*[*OF assms*(*2*) *converge-extend*[*OF assms*(*2*) ‹*converge M2* α (π@γ)› ‹(α @ [*xy*]) $\in$ *L M2*› ‹(π@γ) $\in$ *L M2*›]]
unfolding *after-split*[*OF assms*(*2*) *converge-extend*[*OF assms*(*2*) ‹*converge M2* β (τ@γ)› ‹(β @ [*xy*]) $\in$ *L M2*› ‹(τ@γ) $\in$ *L M2*›]]
by (*metis* ‹γ @ [*xy*] $\in$ *LS M2 q1*› ‹π @ $\gamma \in$ *L M2*› ‹τ @ $\gamma \in$ *L M2*› ‹*after M2* (*after M2* (*FSM.initial M2*) (π @ γ)) [*xy*] = *after M2* (*FSM.initial M2*) ((π @ γ) @ [*xy*])› ‹*after M2* (*after M2* (*FSM.initial M2*) (τ @ γ)) [*xy*] = *after M2* (*FSM.initial M2*) ((τ @ γ) @ [*xy*])› *after-split assms*(*2*) *p1 q1 q2*)

```
          qed

          ultimately show ?thesis
            by blast
        qed
      qed
    qed
  qed

  ultimately show ?thesis
    using ofsm-table-set-observable[OF ‹observable M2› ‹q1 ∈ states M2› is-eq,
of m − size-r M1]
    unfolding q1 q2
    by blast
 qed

 ultimately have after M2 (FSM.initial M2) τ ∈ ofsm-table M2 partition (m −
n0) (after M2 (FSM.initial M2) π)
  using ofsm-table-subset[OF ‹size-r M1 ≤ n0›, of M2 partition initial M2]
  by (meson diff-le-mono2 in-mono ofsm-table-subset)

 moreover have after M2 (FSM.initial M2) π ∈ states M2
  by (metis IntD1 after-is-state assms(2) assms(7) assms(8))

 ultimately have after M2 (FSM.initial M2) τ ∈ ofsm-table-fix M2 partition 0
(after M2 (FSM.initial M2) π)
   using ofsm-table-fix-partition-fixpoint[OF ‹equivalence-relation-on-states M2
partition› ‹size M2 ≤ m›, of after M2 (FSM.initial M2) π]
  unfolding n0
  by blast

 then have LS M2 (after-initial M2 τ) = LS M2 (after-initial M2 π)
   unfolding ofsm-table-fix-set[OF ‹after M2 (FSM.initial M2) π ∈ states M2›
‹observable M2› ‹equivalence-relation-on-states M2 partition›]
  by blast
 then show ?thesis
  unfolding converge.simps
  by (metis assms(15) converge.elims(2))
qed


lemma preserves-divergence-minimally-distinguishing-prefixes-lower-bound :
 fixes M1 :: ('a,'b,'c) fsm
 fixes M2 :: ('d,'b,'c) fsm
 assumes observable M1
 and     observable M2
 and     minimal M1
 and     minimal M2
```

and *converge M1 u v*
and ¬*converge M2 u v*
and *u* ∈ *L M2*
and *v* ∈ *L M2*
and *minimally-distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) *w*
and *wp* ∈ *list.set* (*prefixes w*)
and *wp* ≠ *w*
and *wp* ∈ *LS M1* (*after-initial M1 u*) ∩ *LS M1* (*after-initial M1 v*)
and *preserves-divergence M1 M2* {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}
and *L M1* ∩ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)} = *L M2* ∩ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}
shows *card* (*after-initial M2* ‘ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes wp*)}) ≥ *length wp* + (*card* (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes wp*)))) + *1*
proof −

define *k* **where** *k* = *length wp*
then show *?thesis*
using *assms*(*10*,*11*,*12*,*13*,*14*)
proof (*induction k arbitrary*: *wp rule*: *less-induct*)
case (*less k*)

show *?case* **proof** (*cases k*)
case *0*
then have *wp* = []
using *less.prems* **by** *auto*

have {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes* []) } = {*u*,*v*}
by *auto*
moreover have (*after-initial M2 u*) ≠ (*after-initial M2 v*)
using *assms*(*9*) *assms*(*2*) *assms*(*4*) *assms*(*6*) *assms*(*7*) *assms*(*8*) *convergence-minimal* **by** *blast*
ultimately have *card* (*after-initial M2* ‘ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes* [])}) = *2*
by *auto*

have *FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes* [])) = {*after-initial M1 u*}
unfolding *prefixes-set* **by** *auto*
then have *length* [] + (*card* (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes* [])))) + *1* = *2*
by *auto*
then show *?thesis*
unfolding ‹*wp* = []›
using ‹*card* (*after-initial M2* ‘ {α@γ | α γ . α ∈ {*u*,*v*} ∧ γ ∈ *list.set* (*prefixes* [])}) = *2*›
by *simp*

next
case (*Suc k'*)

have $\bigwedge$ *w''*. *w''* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ *u@w''* $\in$ *L M1*
by (*metis after-language-iff assms*(*1*) *assms*(*5*) *converge.elims*(*2*) *inf-idem language-prefix less.prems*(*4*) *prefixes-set-ob*)
then have $\bigwedge$ *w''*. *w''* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ *v@w''* $\in$ *L M1*
by (*meson assms*(*1*) *assms*(*5*) *converge.elims*(*2*) *converge-extend*)

have $\bigwedge$ *w''* . *w''* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ *converge M1* (*u@w''*) (*v@w''*)
using ‹$\bigwedge$*w''*. *w''* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ *u @ w''* $\in$ *L M1*› *assms*(*1*) *assms*(*5*) *converge.simps converge-append* **by** *blast*

have $\bigwedge$ *w'* . *w'* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes w'*)} $\subseteq$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes wp*)}
using *prefixes-prefix-subset*[*of - wp*]
by *blast*
have $\bigwedge$ *w'* . {*u* @ γ | γ. γ $\in$ *set* (*prefixes w'*)} $\subseteq$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes w'*)}
using *prefixes-set-subset* **by** *blast*
have $\bigwedge$ *w'* . {*v* @ γ | γ. γ $\in$ *set* (*prefixes w'*)} $\subseteq$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes w'*)}
using *prefixes-set-subset* **by** *blast*

have *u@wp* $\in$ *L M1*
by (*metis Int-absorb after-language-iff assms*(*1*) *assms*(*5*) *converge.simps less.prems*(*4*))
moreover have *u@wp* $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes wp*)}
unfolding *prefixes-set* **by** *blast*
ultimately have *u@wp* $\in$ *L M2*
using *less.prems*(*6*) **by** *blast*
then have *wp* $\in$ *LS M2* (*after-initial M2 u*)
by (*meson after-language-iff assms*(*2*) *language-prefix*)

have *v@wp* $\in$ *L M1*
by (*meson* ‹*u @ wp* $\in$ *L M1*› *assms*(*1*) *assms*(*5*) *converge.simps converge-extend*)
moreover have *v@wp* $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes wp*)}
unfolding *prefixes-set* **by** *blast*
ultimately have *v@wp* $\in$ *L M2*
using *less.prems*(*6*) **by** *blast*
then have *wp* $\in$ *LS M2* (*after-initial M2 v*)
by (*meson after-language-iff assms*(*2*) *language-prefix*)

have *no-conv-2*: $\bigwedge$ *w''* . *w''* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ $\neg$*converge M2* (*u@w''*) (*v@w''*) $\wedge$ *u@w''* $\in$ *L M1* $\wedge$ *v@w''* $\in$ *L M1* $\wedge$ *u@w''* $\in$ *L M2* $\wedge$ *v@w''* $\in$ *L M2*
proof −

fix w'' **assume** $*$: $w'' \in$ *set* (*prefixes wp*)
then have $w'' \in$ *set* (*prefixes w*)
using *less.prems*
by (*metis* (*no-types*, *lifting*) *insert-subset mk-disjoint-insert prefixes-set-ob prefixes-set-subset*)

have $u@w'' \in L\ M1$
using ‹$\bigwedge w''$. $w'' \in$ *set* (*prefixes wp*) $\Longrightarrow u\ @\ w'' \in L\ M1$› $*$ **by** *auto*
then have $u@w'' \in L\ M2$
using *assms*(*14*) *less.prems* $*$
by (*metis* (*no-types*, *lifting*) ‹*wp* $\in$ *LS M2* (*after-initial M2 u*)› *after-language-iff assms*(*2*) *assms*(*7*) *language-prefix prefixes-set-ob*)
then have $w'' \in$ *LS M2* (*after-initial M2 u*)
by (*meson after-language-iff assms*(*2*) *language-prefix*)

have $v@w'' \in L\ M1$
using ‹$\bigwedge w''$. $w'' \in$ *set* (*prefixes wp*) $\Longrightarrow v\ @\ w'' \in L\ M1$› $*$ **by** *auto*
then have $v@w'' \in L\ M2$
using *assms*(*14*) *less.prems*(*1*) $*$
by (*metis* (*no-types*, *lifting*) ‹*wp* $\in$ *LS M2* (*after-initial M2 v*)› *after-language-iff assms*(*2*) *assms*(*8*) *language-prefix prefixes-set-ob*)
then have $w'' \in$ *LS M2* (*after-initial M2 v*)
by (*meson after-language-iff assms*(*2*) *language-prefix*)

have *distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) *w*
using *assms*(*9*) **unfolding** *minimally-distinguishes-def* **by** *auto*
show $\neg$*converge M2* ($u@w''$) ($v@w''$) $\wedge$ $u@w'' \in L\ M1$ $\wedge$ $v@w'' \in L\ M1$ $\wedge$ $u@w'' \in L\ M2$ $\wedge$ $v@w'' \in L\ M2$
using *distinguishes-diverge-prefix*[*OF assms*(*2*) ‹*distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) *w*› *assms*(*7*,*8*) ‹$w'' \in$ *set* (*prefixes w*)› ‹$w'' \in$ *LS M2* (*after-initial M2 u*)› ‹$w'' \in$ *LS M2* (*after-initial M2 v*)›]
‹$u@w'' \in L\ M1$› ‹$v@w'' \in L\ M1$› ‹$u@w'' \in L\ M2$› ‹$v@w'' \in L\ M2$›
by *blast*
qed

have *div-on-prefixes* : $\bigwedge$ w'' . $w'' \in$ *set* (*prefixes wp*) $\Longrightarrow$ *after-initial M2* ($u@w''$) $\neq$ *after-initial M2* ($v@w''$)
using *no-conv-2*
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*
then have *div-on-proper-prefixes* : $\bigwedge$ w' w'' . $w' \in$ *set* (*prefixes wp*) $\Longrightarrow$ $w'' \in$ *set* (*prefixes* w') $\Longrightarrow$ *after-initial M2* ($u@w''$) $\neq$ *after-initial M2* ($v@w''$)
using *prefixes-prefix-subset* **by** *blast*

have *wp* = (*butlast wp*)@[*last wp*]
using *Suc less.prems*(*1*)
by (*metis append-butlast-last-id length-greater-0-conv zero-less-Suc*)
then have (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes wp*))) = *Set.insert* (*FSM.after M1* (*after-initial M1 u*) *wp*) (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes* (*butlast wp*))))

using *prefixes-set-Cons-insert*
by (*metis image-insert*)

consider (*FSM.after M1* (*after-initial M1 u*) *wp*) $\notin$ (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes* (*butlast wp*)))) |
(*FSM.after M1* (*after-initial M1 u*) *wp*) $\in$ (*FSM.after M1* (*after-initial M1 u*) ‘ (*list.set* (*prefixes* (*butlast wp*))))
by *blast*

obtain *w-suffix* **where** *w* = (*wp*)@*w-suffix*
using *less.prems*(*2*)
using *prefixes-set-ob* **by** *blast*

define *wk* **where** *wk*: *wk* = (λ *i* . *take i wp*)
define *wk′* **where** *wk′*: *wk′* = (λ *i* . *drop i wp*)

have $\bigwedge$ *i* . (*wk i*)@(*wk′ i*) = *wp*
unfolding *wk wk′* **by** *auto*
then have $\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)
unfolding *prefixes-set*
by *auto*
then have $\bigwedge$ *i* . *set* (*prefixes* (*wk i*)) $\subseteq$ *set* (*prefixes wp*)
by (*simp add*: *prefixes-prefix-subset*)

have $\bigwedge$ *i* . *i* < *k* $\Longrightarrow$ *wk′ i* $\neq$ []
using *less.prems*(*1*)
by (*simp add*: *wk′*)

have *wk k* = *wp*
using *less.prems*(*1*)
by (*simp add*: *wk*)

have $\bigwedge$ *i* . $\neg$ *converge M2* (*u* @ *wk i*) (*v* @ *wk i*)
using *no-conv-2*[*OF* ‹$\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)›] **by** *blast*
have $\bigwedge$ *i* . *u*@*wk i* $\in$ *L M1*
using *no-conv-2*[*OF* ‹$\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)›] **by** *blast*
have $\bigwedge$ *i* . *u*@*wk i* $\in$ *L M2*
using *no-conv-2*[*OF* ‹$\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)›] **by** *blast*
have $\bigwedge$ *i* . *v*@*wk i* $\in$ *L M1*
using *no-conv-2*[*OF* ‹$\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)›] **by** *blast*
have $\bigwedge$ *i* . *v*@*wk i* $\in$ *L M2*
using *no-conv-2*[*OF* ‹$\bigwedge$ *i* . *wk i* $\in$ *set* (*prefixes wp*)›] **by** *blast*

have $\bigwedge$ *w″* . *w″* $\in$ *set* (*prefixes wp*) $\Longrightarrow$ *w″* = *wk* (*length w″*)

unfolding *prefixes-take-iff* **unfolding** *wk*
by *auto*
then have $\bigwedge$ *i w''* . *w''* $\in$ *set* (*prefixes* (*wk i*)) $\Longrightarrow \exists$ *j* . *w''* = *wk j* $\wedge$ *j* $\leq$ *i*
by (*metis min-def order-refl prefixes-take-iff take-take wk*)

have *prefixes-same-reaction*: $\bigwedge$ *j* . *j* < *k* $\Longrightarrow$ *w-suffix* $\in$ *LS M2* (*after-initial M2* (*u@wk j*)) = (*w-suffix* $\in$ *LS M2* (*after-initial M2* (*v@wk j*)))
proof −
fix *j* **assume** *j* < *k*

then have *wp* = (*wk j*)@(*wk' j*) **and** (*wk' j*) $\neq$ []
using ‹$\bigwedge$ *i* . (*wk i*)@(*wk' i*) = *wp*› ‹$\bigwedge$ *i* . *i* < *k* $\Longrightarrow$ *wk' i* $\neq$ []› **by** *auto*

have *distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) ((*wk j*)@(*wk' j*)@*w-suffix*)
using *assms*(*9*)
unfolding ‹*w* = (*wp*)@*w-suffix*› ‹*wp* = (*wk j*)@(*wk' j*)› *minimally-distinguishes-def*
by (*metis append.assoc*)

have *u@wk j* $\in$ *L M2*
using ‹$\bigwedge$*i*. *u* @ *wk i* $\in$ *L M2*› **by** *blast*
have *v@wk j* $\in$ *L M2*
using ‹$\bigwedge$*i*. *v* @ *wk i* $\in$ *L M2*› **by** *blast*

have $*$: *minimally-distinguishes M2* (*after-initial M2* (*u* @ *wk j*)) (*after-initial M2* (*v* @ *wk j*)) ((*wk' j*)@*w-suffix*)
using *assms*(*9*) *minimally-distinguishes-after-append-initial*[*OF assms*(*2*,*4*) ‹*u* $\in$ *L M2*› ‹*v* $\in$ *L M2*›, *of wk j*]
using ‹*w* = *wp* @ *w-suffix*› ‹*wk' j* $\neq$ []› ‹*wp* = *wk j* @ *wk' j*› **by** *auto*

then have $\neg$*distinguishes M2* (*after-initial M2* (*u@wk j*)) (*after-initial M2* (*v@wk j*)) *w-suffix*
unfolding *minimally-distinguishes-def*
by (*metis* $*$ ‹*u* @ *wk j* $\in$ *L M2*› ‹*v* @ *wk j* $\in$ *L M2*› ‹*wk' j* $\neq$ []› *append.left-neutral append.right-neutral assms*(*2*) *minimally-distinguishes-no-prefix*)
then show *w-suffix* $\in$ *LS M2* (*after-initial M2* (*u@wk j*)) = (*w-suffix* $\in$ *LS M2* (*after-initial M2* (*v@wk j*)))
unfolding *distinguishes-def* **by** *blast*
qed

have $\bigwedge$ *i* . *u@wk i* $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes* (*wp*))}
using ‹$\bigwedge$*i*. *wk i* $\in$ *set* (*prefixes wp*)› **by** *blast*
have $\bigwedge$ *i* . *v@wk i* $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes* (*wp*))}
using ‹$\bigwedge$*i*. *wk i* $\in$ *set* (*prefixes wp*)› **by** *blast*
have *u@*(*wp*) $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes* (*wp*))}
using *prefixes-set* **by** *blast*
have *v@*(*wp*) $\in$ {α @ γ |α γ. α $\in$ {*u*, *v*} $\wedge$ γ $\in$ *set* (*prefixes* (*wp*))}
using *prefixes-set* **by** *blast*

define q **where** q: $q = (\lambda\ i\ .\ \textit{after-initial M1}\ (u@(wk\ i)))$
define a **where** a: $a = (\lambda\ i\ .\ \textit{after-initial M2}\ (u@(wk\ i)))$
define b **where** b: $b = (\lambda\ i\ .\ \textit{after-initial M2}\ (v@(wk\ i)))$
define I' **where** I': $I' = (\lambda\ i\ .\ \{j\ .\ j \leq \textit{Suc}\ k' \wedge q\ i = q\ j\})$

define l **where** l: $l = \textit{card}\ (q\ `\ (\bigcup\ i \in \{..\textit{Suc}\ k'\}\ .\ I'\ i))$

have q-v: $\bigwedge\ i\ .\ q\ i = \textit{after-initial M1}\ (v@(wk\ i))$
unfolding q
by (*meson* ‹$\bigwedge i.\ wk\ i \in \textit{set}\ (\textit{prefixes}\ wp)$› ‹$\bigwedge w''.\ w'' \in \textit{set}\ (\textit{prefixes}\ wp) \Longrightarrow \textit{converge M1}\ (u\ @\ w'')\ (v\ @\ w'')$› *assms*(*1*) *assms*(*3*) *converge.simps convergence-minimal*)

have *q-divergence*: $\bigwedge\ i\ j\ .\ q\ i \neq q\ j \Longrightarrow a\ i \neq a\ j \wedge a\ i \neq b\ j \wedge b\ i \neq a\ j \wedge b\ i \neq b\ j$
proof −
fix $i\ j$ **assume** $q\ i \neq q\ j$
then have $\neg$ *converge M1* $(u@(wk\ i))\ (u@(wk\ j))$
unfolding q
using *assms*(*1*) *assms*(*3*) *converge.simps convergence-minimal* **by** *blast*
then have $\neg$ *converge M1* $(u@(wk\ i))\ (v@(wk\ j))$
$\neg$ *converge M1* $(v@(wk\ i))\ (u@(wk\ j))$
$\neg$ *converge M1* $(v@(wk\ i))\ (v@(wk\ j))$
using *assms*(*1*) *assms*(*3*) *assms*(*5*) *converge-trans-2* **by** *blast*+

have $\neg$ *converge M2* $(u@(wk\ i))\ (u@(wk\ j))$
using ‹$\neg$ *converge M1* $(u@(wk\ i))\ (u@(wk\ j))$›
using *less.prems*(*5*) **unfolding** *preserves-divergence.simps*
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M1$› [*of i*] ‹$\bigwedge\ i\ .\ u@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ (wp))\}$›[*of i*]
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M1$› [*of j*] ‹$\bigwedge\ i\ .\ u@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ (wp))\}$›[*of j*]
by *blast*
then have $a\ i \neq a\ j$
unfolding a
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M2$›
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*

have $\neg$ *converge M2* $(v@(wk\ i))\ (v@(wk\ j))$
using ‹$\neg$ *converge M1* $(v@(wk\ i))\ (v@(wk\ j))$›
using *less.prems*(*5*) **unfolding** *preserves-divergence.simps*
using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M1$› [*of i*] ‹$\bigwedge\ i\ .\ v@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ (wp))\}$›[*of i*]

using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M1$› [*of j*] ‹$\bigwedge\ i\ .\ v@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u, v\} \wedge \gamma \in set\ (prefixes\ (wp))\}$›[*of j*]
by *blast*
then have $b\ i \neq b\ j$
unfolding *b*
using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M2$›
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*

have $\neg$ *converge M2* ($u@(wk\ i)$) ($v@(wk\ j)$)
using ‹$\neg$ *converge M1* ($u@(wk\ i)$) ($v@(wk\ j)$)›
using *less.prems*(*5*) **unfolding** *preserves-divergence.simps*
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M1$› [*of i*] ‹$\bigwedge\ i\ .\ u@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u, v\} \wedge \gamma \in set\ (prefixes\ (wp))\}$›[*of i*]
using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M1$› [*of j*] ‹$\bigwedge\ i\ .\ v@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u, v\} \wedge \gamma \in set\ (prefixes\ (wp))\}$›[*of j*]
by *blast*
then have $a\ i \neq b\ j$
unfolding *a b*
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M2$› ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M2$›
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*

have $\neg$ *converge M2* ($v@(wk\ i)$) ($u@(wk\ j)$)
using ‹$\neg$ *converge M1* ($v@(wk\ i)$) ($u@(wk\ j)$)›
using *less.prems*(*5*) **unfolding** *preserves-divergence.simps*
using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M1$› [*of i*] ‹$\bigwedge\ i\ .\ v@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u, v\} \wedge \gamma \in set\ (prefixes\ (wp))\}$›[*of i*]
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M1$› [*of j*] ‹$\bigwedge\ i\ .\ u@wk\ i \in \{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u, v\} \wedge \gamma \in set\ (prefixes\ (wp))\}$›[*of j*]
by *blast*
then have $b\ i \neq a\ j$
unfolding *b a*
using ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M2$› ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M2$›
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*

show $a\ i \neq a\ j \wedge a\ i \neq b\ j \wedge b\ i \neq a\ j \wedge b\ i \neq b\ j$
using ‹$a\ i \neq a\ j$› ‹$a\ i \neq b\ j$› ‹$b\ i \neq a\ j$› ‹$b\ i \neq b\ j$› **by** *auto*
qed

have $\bigwedge\ i\ .\ a\ i \in states\ M2$
by (*metis* ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M2$› *a after-is-state assms*(*2*))
have $\bigwedge\ i\ .\ b\ i \in states\ M2$
by (*metis* ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M2$› *b after-is-state assms*(*2*))

have $\bigwedge\ i\ j\ .\ i \leq Suc\ k' \Longrightarrow j \leq Suc\ k' \Longrightarrow\ q\ i = q\ j \Longrightarrow I'\ i = I'\ j$
unfolding *q I′* **by** *force*

have $\bigwedge\ i\ .\ i \leq Suc\ k' \Longrightarrow i \in I'\ i$
unfolding *I′ q* **by** *force*
moreover have $\bigwedge\ i\ .\ I'\ i \subseteq \{..Suc\ k'\}$

unfolding I' **by** *force*
ultimately have $(\bigcup\ i \in \{..Suc\ k'\}\ .\ I'\ i) = \{..Suc\ k'\}$
by *blast*
then have *card* $(\bigcup\ i \in \{..Suc\ k'\}\ .\ I'\ i) = k'+2$
by *auto*

have *pt1*: *finite* $\{..Suc\ k'\}$
by *auto*
have *pt2*: $\{..Suc\ k'\} \neq \{\}$
by *auto*
have *pt3*: $(\bigwedge x.\ x \in \{..Suc\ k'\} \Longrightarrow I'\ x \subseteq \{..Suc\ k'\})$
using ‹$\bigwedge i.\ I'\ i \subseteq \{..Suc\ k'\}$› *atMost-atLeast0* **by** *blast*
have *pt4*: $(\bigwedge x.\ x \in \{..Suc\ k'\} \Longrightarrow I'\ x \neq \{\})$
using ‹$\bigwedge i.\ i \leq Suc\ k' \Longrightarrow i \in I'\ i$› **by** *auto*
have *pt5*: $(\bigwedge x\ y.\ x \in \{..Suc\ k'\} \Longrightarrow y \in \{..Suc\ k'\} \Longrightarrow I'\ x = I'\ y \vee I'\ x \cap I'\ y = \{\})$
using I' **by** *force*
have *pt6*: $\bigcup\ (I'\ `\ \{..Suc\ k'\}) = \{..Suc\ k'\}$
using ‹$\bigcup\ (I'\ `\ \{..Suc\ k'\}) = \{..Suc\ k'\}$› **by** *linarith*

obtain $l\ I$ **where** $I\ `\ \{..l\} = I'\ `\ \{..Suc\ k'\}$
and $\bigwedge i\ j.\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow I\ i \cap I\ j = \{\}$
and *card* $(I'\ `\ \{..Suc\ k'\}) = Suc\ l$
using *partition-helper*[*of* $\{..Suc\ k'\}\ I'$, *OF pt1 pt2 pt3 pt4 pt5 pt6*]
by *metis*

have $\bigwedge\ i\ .\ i \leq l \Longrightarrow \exists\ j\ .\ j \leq Suc\ k' \wedge I\ i = I'\ j$
using ‹$I\ `\ \{..l\} = I'\ `\ \{..Suc\ k'\}$›
by *blast*

define S **where** S: $S = (\lambda\ i\ .\ \bigcup\ j \in I\ i\ .\ \{a\ j,\ b\ j\})$

have $(\bigcup\ i \in \{..l\}\ .\ S\ i) = (\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\})$
unfolding S **using** ‹$I\ `\ \{..l\} = I'\ `\ \{..Suc\ k'\}$›
by (*metis* (*no-types*, *lifting*) *Sup.SUP-cong UN-UN-flatten* ‹$\bigcup\ (I'\ `\ \{..Suc\ k'\}) = \{..Suc\ k'\}$›)
then have *card* $(\bigcup\ i \in \{..l\}\ .\ S\ i) =$ *card* $(\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\})$
by *presburger*

moreover have $\bigwedge\ i\ j.\ i \leq l \Longrightarrow j \leq l \Longrightarrow i \neq j \Longrightarrow S\ i \cap S\ j = \{\}$
proof (*rule ccontr*)
fix $i\ j$ **assume** $i \leq l$ **and** $j \leq l$ **and** $i \neq j$ **and** $S\ i \cap S\ j \neq \{\}$
then obtain $ii\ jj$ **where** $ii \in I\ i$ **and** $jj \in I\ j$ **and** $\{a\ ii,\ b\ ii\} \cap \{a\ jj,\ b\ jj\} \neq \{\}$
unfolding S **by** *blast*

obtain i' j' **where** $i' \leq Suc\ k'$ **and** $j' \leq Suc\ k'$ **and** $I\ i = I'\ i'$ **and** $I\ j = I'\ j'$
using ‹$i \leq l$› ‹$j \leq l$› ‹$\bigwedge\ i\ .\ i \leq l \Longrightarrow \exists\ j\ .\ j \leq Suc\ k' \wedge I\ i = I'\ j$›
by *meson*

moreover have $I\ i \cap I\ j = \{\}$
by (*meson* ‹$\bigwedge j\ i.\ [\![i \leq l;\ j \leq l;\ i \neq j]\!] \Longrightarrow I\ i \cap I\ j = \{\}$› ‹$i \leq l$› ‹$i \neq j$› ‹$j \leq l$›)
ultimately have $I'\ i' \cap I'\ j' = \{\}$
by *blast*
then have $q\ i' \neq q\ j'$
unfolding I'
by (*metis* I' ‹$I\ i = I'\ i'$› ‹$\bigwedge i.\ i \leq Suc\ k' \Longrightarrow i \in I'\ i$› ‹$\bigwedge thesis.\ (\bigwedge i'\ j'.\ [\![i' \leq Suc\ k';\ j' \leq Suc\ k';\ I\ i = I'\ i';\ I\ j = I'\ j']\!] \Longrightarrow thesis) \Longrightarrow thesis$› *empty-iff inf.idem*)
then have $q\ ii \neq q\ jj$
using I' ‹$I\ i = I'\ i'$› ‹$I\ j = I'\ j'$› ‹$ii \in I\ i$› ‹$jj \in I\ j$› **by** *force*
then have $a\ ii \neq a\ jj\ a\ ii \neq b\ jj\ b\ ii \neq a\ jj\ b\ ii \neq b\ jj$
using *q-divergence*
by *blast+*
then show *False*
using ‹$\{a\ ii,\ b\ ii\} \cap \{a\ jj,\ b\ jj\} \neq \{\}$›
by *blast*
qed
moreover have $\forall\ i \in \{..l\}\ .\ finite\ (S\ i)$
unfolding S
by (*metis* (*no-types*, *lifting*) ‹$I\ `\ \{..l\} = I'\ `\ \{..Suc\ k'\}$› ‹$\bigcup\ (I'\ `\ \{..Suc\ k'\}) = \{..Suc\ k'\}$› *finite.emptyI finite.insertI finite-UN finite-atMost*)
ultimately have $card\ (\bigcup\ i \in \{..l\}\ .\ S\ i) = (\sum\ i \in \{..l\}\ .\ card\ (S\ i))$
using *atMost-iff*
using *card-UN-disjoint*[*OF finite-atMost*[*of l*], *of S*]
by *blast*

have *eq7*: $card\ (\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\}) = (\sum\ i \in \{..l\}\ .\ card\ (S\ i))$
unfolding ‹$card\ (\bigcup\ i \in \{..l\}\ .\ S\ i) = card\ (\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\})$›[*symmetric*]
unfolding ‹$card\ (\bigcup\ i \in \{..l\}\ .\ S\ i) = (\sum\ i \in \{..l\}\ .\ card\ (S\ i))$›
by *blast*

have *eq8*: $\bigwedge\ i\ .\ i \leq l \Longrightarrow card\ (S\ i) \geq Suc\ (card\ (I\ i))$
proof −
fix i **assume** $i \leq l$

have $S\ i \subseteq states\ M2$
unfolding S **using** ‹$\bigwedge\ i\ .\ a\ i \in states\ M2$› ‹$\bigwedge\ i\ .\ b\ i \in states\ M2$›
by *blast*

define W **where** W: $W = \{w' \in set\ (prefixes\ w).$
$w' \neq w \wedge$
$after\ M2\ (after\text{-}initial\ M2\ u)\ w' \in S\ i \wedge after\ M2\ (after\text{-}initial\ M2\ v)\ w' \in S\ i\}$

have $wk\ `\ I\ i \subseteq W$
proof
fix x **assume** $x \in wk\ `\ I\ i$
then obtain i' **where** $x = wk\ i'$ **and** $i' \in I\ i$
by *blast*
then have $a\ i' \in S\ i$ **and** $b\ i' \in S\ i$
unfolding S **by** *blast*+
then have $after\ M2\ (after\text{-}initial\ M2\ u)\ (wk\ i') \in S\ i$
$after\ M2\ (after\text{-}initial\ M2\ v)\ (wk\ i') \in S\ i$
unfolding $a\ b$
using ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M2$› ‹$\bigwedge i.\ v\ @\ wk\ i \in L\ M2$›
by (*metis after-split assms*(2))+
moreover have $wk\ i' \neq w$
by (*metis* (*no-types*) ‹$\bigwedge i.\ wk\ i \in set\ (prefixes\ wp)$› *less.prems*(2) *less.prems*(3) *nat-le-linear prefixes-take-iff take-all-iff*)
moreover have $wk\ i' \in set\ (prefixes\ w)$
using ‹$\bigwedge i.\ wk\ i \in set\ (prefixes\ wp)$› *less.prems*(2) *prefixes-prefix-subset* **by** *blast*
ultimately show $x \in W$
unfolding ‹$x = wk\ i'$› W
by *blast*
qed
moreover have *finite* W
proof −
have $W \subseteq (set\ (prefixes\ w))$
unfolding W **by** *blast*
then show *?thesis*
by (*meson List.finite-set rev-finite-subset*)
qed
ultimately have $card\ (wk\ `\ I\ i) \leq card\ W$
by (*meson card-mono*)
moreover have $card\ (wk\ `\ I\ i) = card\ (I\ i)$
proof −
have $\bigwedge\ x\ y\ .\ x \in I\ i \Longrightarrow y \in I\ i \Longrightarrow x \neq y \Longrightarrow wk\ x \neq wk\ y$
proof −
fix $x\ y$ **assume** $x \in I\ i\ y \in I\ i\ x \neq y$
then have $x \leq Suc\ k'\ y \leq Suc\ k'$
by (*metis UN-I* ‹$I\ `\ \{..l\} = I'\ `\ \{..Suc\ k'\}$› ‹$\bigcup\ (I'\ `\ \{..Suc\ k'\}) = \{..Suc\ k'\}$› ‹$i \leq l$› *atMost-iff*)+
then show $wk\ x \neq wk\ y$
using ‹$x \neq y$› ‹$k = length\ wp$›
unfolding *wk Suc*
using *take-diff* **by** *metis*
qed

moreover have *finite* (*I i*)
by (*metis* ‹*I* ‘ {..*l*} = *I′* ‘ {..*Suc k′*}› ‹*i* ≤ *l*› *atMost-iff finite-UN finite-atMost pt6*)
ultimately show *?thesis*
using *image-inj-card-helper* **by** *metis*
qed
ultimately have *card* (*I i*) ≤ *card W*
by *simp*
then have *card* (*I i*) ≤ *card* (*S i*) − *1*
using *minimally-distinguishes-proper-prefixes-card*[*OF assms*(*2*,*4*) *after-is-state*[*OF assms*(*2*) ‹*u* ∈ *L M2*›] *after-is-state*[*OF assms*(*2*) ‹*v* ∈ *L M2*›] ‹*minimally-distinguishes M2* (*after-initial M2 u*) (*after-initial M2 v*) *w*› ‹*S i* ⊆ *states M2*›]
unfolding *W*[*symmetric*]
by *simp*
moreover have *card* (*S i*) > *0*
proof −
have *card* (*I i*) > *0*
by (*metis* ‹⋀*i*. *i* ≤ *l* ⟹ ∃*j*≤*Suc k′*. *I i* = *I′ j*› ‹*card* (*wk* ‘ *I i*) = *card* (*I i*)› ‹*finite W*› ‹*i* ≤ *l*› ‹*wk* ‘ *I i* ⊆ *W*› *atMost-iff card-0-eq gr0I image-is-empty pt4 rev-finite-subset*)
then show *?thesis*
unfolding *S*
by (*metis S calculation diff-le-self le-0-eq not-gr-zero*)
qed
ultimately show *card* (*S i*) ≥ *Suc* (*card* (*I i*))
by *linarith*
qed

have (∑ *i* ∈ {..*l*} . *card* (*S i*)) ≥ (∑ *i* ∈ {..*l*} . (*Suc* (*card* (*I i*))))
using *eq8*
by (*meson atMost-iff sum-mono*)
moreover have (∑ *i* ∈ {..*l*} . (*Suc* (*card* (*I i*)))) = (*Suc l*) + *k′* + *2*
proof −
have (∑ *i* ∈ {..*l*} . (*Suc* (*card* (*I i*)))) = (*Suc l*) + (∑ *i* ∈ {..*l*} . (*card* (*I i*)))
by (*simp add*: *sum-Suc*)
moreover have (∑ *i* ∈ {..*l*} . (*card* (*I i*))) = *k′* + *2*
proof −
have *card* (⋃ *i* ∈ {..*l*} . *I i*) = *k′* + *2*
using ‹*card* (⋃ *i* ∈ {..*Suc k′*} . *I′ i*) = *k′*+*2*›
using ‹*I* ‘ {..*l*} = *I′* ‘ {..*Suc k′*}› **by** *presburger*
moreover have (∑ *i* ∈ {..*l*} . (*card* (*I i*))) = *card* (⋃ *i* ∈ {..*l*} . *I i*)
using *sum-image-inj-card-helper*[*of l I*]
by (*metis* ‹*I* ‘ {..*l*} = *I′* ‘ {..*Suc k′*}› ‹⋀*j i*. ⟦*i* ≤ *l*; *j* ≤ *l*; *i* ≠ *j*⟧ ⟹ *I i* ∩ *I j* = {}› ‹⋃ (*I′* ‘ {..*Suc k′*}) = {..*Suc k′*}› *atMost-iff finite-UN finite-atMost*)
ultimately show *?thesis*
by *auto*

qed
ultimately show *?thesis*
by *linarith*
qed
ultimately have $(\sum\ i \in \{..l\}\ .\ card\ (S\ i)) \geq k' + l + 3$
by *auto*
moreover have *card* (*after-initial M2* ‘ $\{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in set\ (prefixes\ wp)\}) = (\sum\ i \in \{..l\}\ .\ card\ (S\ i))$
proof −
have *after-initial M2* ‘ $\{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in set\ (prefixes\ wp)\} = (\bigcup\ i \in \{..l\}\ .\ S\ i)$
proof −
have $set\ (prefixes\ wp) = \{wk\ i \mid i\ .\ i \leq k\}$
using *less.prems*(*1*) **unfolding** *wk prefixes-set*
by (*metis* ‹$\bigwedge i.\ wk\ i \in set\ (prefixes\ wp)$› *append-eq-conv-conj le-cases prefixes-set-ob take-all wk*)
then have $*$:$\{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in set\ (prefixes\ wp)\} = (\bigcup\ i \in \{..Suc\ k'\}\ .\ \{u@wk\ i,\ v@wk\ i\})$
unfolding *Suc* **by** *auto*
have $**$: $(\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\}) = after\text{-}initial\ M2$ ‘ $(\bigcup\ i \in \{..Suc\ k'\}\ .\ \{u@wk\ i,\ v@wk\ i\})$
unfolding *a b* **by** *blast*
show *?thesis*
unfolding ‹$(\bigcup\ i \in \{..l\}\ .\ S\ i) = (\bigcup\ i \in \{..Suc\ k'\}\ .\ \{a\ i,\ b\ i\})$› $** *$
by *simp*
qed
then show *?thesis*
by (*simp add*: ‹$card\ (\bigcup\ (S\ ‘\ \{..l\})) = (\sum i{\leq}l.\ card\ (S\ i))$›)
qed
ultimately have *bound-l*: $card\ (after\text{-}initial\ M2$ ‘ $\{\alpha\ @\ \gamma\ |\alpha\ \gamma.\ \alpha \in \{u,\ v\} \wedge \gamma \in set\ (prefixes\ wp)\}) \geq k + l + 2$
unfolding *Suc* **by** *simp*

have *bound-r*: $length\ wp + card\ (after\ M1\ (after\text{-}initial\ M1\ u)$ ‘ $set\ (prefixes\ wp)) + 1 = k + l + 2$
proof −
have $set\ (prefixes\ wp) = \{wk\ i \mid i\ .\ i \leq k\}$
using *less.prems*(*1*) **unfolding** *wk prefixes-set*
by (*metis* ‹$\bigwedge i.\ wk\ i \in set\ (prefixes\ wp)$› *append-eq-conv-conj le-cases prefixes-set-ob take-all wk*)

let $?witness = \lambda i\ .\ SOME\ j\ .\ j \in i$
have $\bigwedge\ i\ .\ i \in (I'$ ‘ $\{..Suc\ k'\}) \Longrightarrow ?witness\ i \in i$
using ‹$\bigwedge i.\ i \leq Suc\ k' \Longrightarrow i \in I'\ i$› *some-in-eq* **by** *auto*
have $**$:$\bigwedge\ Ii\ Ij\ .\ Ii \in (I'$ ‘ $\{..Suc\ k'\}) \Longrightarrow Ij \in (I'$ ‘ $\{..Suc\ k'\}) \Longrightarrow Ii \neq Ij \Longrightarrow ?witness\ Ii \neq ?witness\ Ij$
proof −
fix *Ii Ij* **assume** $Ii \in (I'$ ‘ $\{..Suc\ k'\})$ **and** $Ij \in (I'$ ‘ $\{..Suc\ k'\})$ **and** $Ii \neq Ij$

then have $Ii \cap Ij = \{\}$
using *pt5* **by** *auto*
moreover have *?witness Ii* $\in$ *Ii*
using ‹$\bigwedge$ *i* . *i* $\in$ (*I'* ‘ {..*Suc k'*}) $\Longrightarrow$ *?witness i* $\in$ *i*› ‹*Ii* $\in$ (*I'* ‘ {..*Suc k'*})›
by *blast*
moreover have *?witness Ij* $\in$ *Ij*
using ‹$\bigwedge$ *i* . *i* $\in$ (*I'* ‘ {..*Suc k'*}) $\Longrightarrow$ *?witness i* $\in$ *i*› ‹*Ij* $\in$ (*I'* ‘ {..*Suc k'*})›
by *blast*
ultimately show *?witness Ii* $\neq$ *?witness Ij*
by *fastforce*
qed
have $*$: *finite* (*I'* ‘ {..*Suc k'*})
by *auto*

have *c1*: *card* (*I'* ‘ {..*Suc k'*}) = *card* (*?witness* ‘ (*I'* ‘ {..*Suc k'*}))
using *image-inj-card-helper*[*of I'* ‘ {..*Suc k'*} *?witness*, *OF* $*$ $**$]
by *auto*

have $*$: *finite* (*?witness* ‘ (*I'* ‘ {..*Suc k'*}))
by *auto*
have $**$:$\bigwedge$ *i j* . *i* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*})) $\Longrightarrow$ *j* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*})) $\Longrightarrow$ *i* $\neq$ *j* $\Longrightarrow$ *q i* $\neq$ *q j*
proof −
fix *i j* **assume** *i* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*})) **and** *j* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*})) **and** *i* $\neq$ *j*

obtain *i'* **where** *i* = *?witness* (*I' i'*) **and** *i* $\in$ *I' i'* **and** *i'* $\in$ {..*Suc k'*}
using ‹*i* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*}))›
using ‹$\bigwedge$*i*. *i* $\in$ *I'* ‘ {..*Suc k'*} $\Longrightarrow$ (*SOME j*. *j* $\in$ *i*) $\in$ *i*› **by** *blast*
obtain *j'* **where** *j* = *?witness* (*I' j'*) **and** *j* $\in$ *I' j'* **and** *j'* $\in$ {..*Suc k'*}
using ‹*j* $\in$ (*?witness* ‘ (*I'* ‘ {..*Suc k'*}))›
using ‹$\bigwedge$*i*. *i* $\in$ *I'* ‘ {..*Suc k'*} $\Longrightarrow$ (*SOME j*. *j* $\in$ *i*) $\in$ *i*› **by** *blast*

have *I' i'* $\neq$ *I' j'*
using ‹*i* $\neq$ *j*›
using ‹*i* = (*SOME j*. *j* $\in$ *I' i'*)› ‹*j* = (*SOME j*. *j* $\in$ *I' j'*)› **by** *fastforce*
then show *q i* $\neq$ *q j*
using ‹*i* $\in$ *I' i'*› ‹*j* $\in$ *I' j'*›
unfolding *q I'*
by *force*
qed

have *c2*: *card* (*I'* ‘ {..*Suc k'*}) = *card* (*q* ‘ (*?witness* ‘ (*I'* ‘ {..*Suc k'*})))
using *image-inj-card-helper*[*of* (*?witness* ‘ (*I'* ‘ {..*Suc k'*})) *q*, *OF* $*$ $**$] *c1*
by *force*

have q ‘ ($?witness$ ‘ (I' ‘ $\{..Suc\ k'\}$)) = q ‘ ($\bigcup$ $i \in \{..Suc\ k'\}$. $I'\ i$)
proof
show q ‘ $?witness$ ‘ I' ‘ $\{..Suc\ k'\} \subseteq q$ ‘ $\bigcup$ (I' ‘ $\{..Suc\ k'\}$)
proof
fix s **assume** $s \in q$ ‘ $?witness$ ‘ I' ‘ $\{..Suc\ k'\}$
then obtain Ii **where** $Ii \in I'$ ‘ $\{..Suc\ k'\}$ **and** $s = q$ ($?witness\ Ii$)
by *blast*
then have $s \in q$ ‘ Ii
using ‹$\bigwedge i.\ i \in I'$ ‘ $\{..Suc\ k'\} \Longrightarrow (SOME\ j.\ j \in i) \in i$› **by** *blast*
then show $s \in q$ ‘ $\bigcup$ (I' ‘ $\{..Suc\ k'\}$)
using ‹$Ii \in I'$ ‘ $\{..Suc\ k'\}$› **by** *blast*
qed
show q ‘ $\bigcup$ (I' ‘ $\{..Suc\ k'\}$) $\subseteq q$ ‘ $?witness$ ‘ I' ‘ $\{..Suc\ k'\}$
proof
fix s **assume** $s \in q$ ‘ $\bigcup$ (I' ‘ $\{..Suc\ k'\}$)
then obtain i **where** $s \in q$ ‘ ($I'\ i$) **and** $i \in \{..Suc\ k'\}$
by *blast*

have $?witness\ (I'\ i) \in I'\ i$
using ‹$\bigwedge i.\ i \in I'$ ‘ $\{..Suc\ k'\} \Longrightarrow (SOME\ j.\ j \in i) \in i$› ‹$i \in \{..Suc\ k'\}$› **by** *blast*
then have q ‘ ($I'\ i$) = $\{q\ (?witness\ (I'\ i))\}$
unfolding $q\ I'$
by *fastforce*
then have $s = q\ (?witness\ (I'\ i))$
using ‹$s \in q$ ‘ ($I'\ i$)› **by** *blast*
then show $s \in q$ ‘ $?witness$ ‘ I' ‘ $\{..Suc\ k'\}$
using ‹$i \in \{..Suc\ k'\}$› **by** *blast*
qed
qed
then have *c3*: $card\ (I'$ ‘ $\{..Suc\ k'\}) = card\ (q$ ‘ ($\bigcup$ $i \in \{..Suc\ k'\}$. $I'\ i$))
using *c2* **by** *auto*

have q ‘ ($\bigcup$ $i \in \{..Suc\ k'\}$. $I'\ i$) = $after\ M1$ ($after\text{-}initial\ M1\ u$) ‘ set ($prefixes\ wp$)
proof −
have $set\ (prefixes\ wp) = \{wk\ i \mid i\ .\ i \leq k\}$
using *less.prems*(*1*) **unfolding** *wk prefixes-set*
by (*metis* ‹$\bigwedge i.\ wk\ i \in set\ (prefixes\ wp)$› *append-eq-conv-conj le-cases prefixes-set-ob take-all wk*)
also have ... = wk ‘ $\{..Suc\ k'\}$
unfolding *Suc*
by (*simp add*: *atMost-def setcompr-eq-image*)
finally have $*$:$set\ (prefixes\ wp) = wk$ ‘ $\{..Suc\ k'\}$ **.**

have $\bigwedge$ i . $after\text{-}initial\ M1\ (u\ @\ wk\ i) = after\ M1\ (after\text{-}initial\ M1\ u)\ (wk\ i)$

by (*metis* ‹$\bigwedge i.\ u\ @\ wk\ i \in L\ M1$› *after-split assms*(*1*))
then have $**$: $\bigwedge\ X\ .\ q\ `\ X = after\ M1\ (after\text{-}initial\ M1\ u)\ `\ wk\ `\ X$
unfolding *q*
by *fastforce*

show *?thesis*
unfolding $*\ **$
unfolding ‹$(\bigcup\ i \in \{..Suc\ k'\}\ .\ I'\ i) = \{..Suc\ k'\}$›
by *simp*
qed

then have $card\ (I'\ `\ \{..Suc\ k'\}) = card\ (after\ M1\ (after\text{-}initial\ M1\ u)\ `\ set\ (prefixes\ wp))$
using *c3* **by** *auto*
then have $card\ (after\ M1\ (after\text{-}initial\ M1\ u)\ `\ set\ (prefixes\ wp)) = Suc\ l$
using ‹$card\ (I'\ `\ \{..Suc\ k'\}) = Suc\ l$›
by *auto*
then show *?thesis*
unfolding ‹$k = length\ wp$›[*symmetric*] **by** *auto*
qed

show *?thesis*
using *bound-l*
unfolding *bound-r* **.**
qed
qed
qed

lemma *sufficient-condition-for-convergence* :
fixes $M1 :: ('a,'b,'c)\ fsm$
fixes $M2 :: ('d,'b,'c)\ fsm$
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and $size\text{-}r\ M1 \le m$
and $size\ M2 \le m$
and $inputs\ M2 = inputs\ M1$
and $outputs\ M2 = outputs\ M1$
and $converge\ M1\ \pi\ \tau$
and $L\ M1 \cap T = L\ M2 \cap T$
and $\bigwedge\ \gamma\ x\ y\ .\ length\ (\gamma@[(x,y)]) \le m - size\text{-}r\ M1 \Longrightarrow$
$\gamma \in LS\ M1\ (after\text{-}initial\ M1\ \pi) \Longrightarrow$
$x \in inputs\ M1 \Longrightarrow y \in outputs\ M1 \Longrightarrow$
$\exists\ SC\ \alpha\ \beta\ .\ SC \subseteq T$
$\wedge\ is\text{-}state\text{-}cover\ M1\ SC$
$\wedge\ \{\omega@\omega' \mid \omega\ \omega'\ .\ \omega \in \{\alpha,\beta\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\} \subseteq SC$

$\wedge$ *converge M1* π α
$\wedge$ *converge M2* π α
$\wedge$ *converge M1* τ β
$\wedge$ *converge M2* τ β
$\wedge$ *preserves-divergence M1 M2 SC*
and $\exists$ *SC* α β . *SC* $\subseteq$ *T*
$\wedge$ *is-state-cover M1 SC*
$\wedge$ $\alpha \in$ *SC* $\wedge$ $\beta \in$ *SC*
$\wedge$ *converge M1* π α
$\wedge$ *converge M2* π α
$\wedge$ *converge M1* τ β
$\wedge$ *converge M2* τ β
$\wedge$ *preserves-divergence M1 M2 SC*
shows *converge M2* π τ
proof (*cases inputs M1* = {} $\vee$ *outputs M1* = {})
case *True*
then have *L M1* = {[]}
using *language-empty-IO* **by** *blast*
then have π = [] **and** τ = []
using *assms*(*9*) **by** *auto*
then show *?thesis*
by *auto*
next
case *False*

define *n* **where** *n*: *n* = *size-r M1*
have $n \leq m$
using *assms*(*5*) *n* **by** *auto*

show *?thesis* **proof** (*rule ccontr*)
assume $\neg$ *converge M2* π τ
moreover have $\pi \in$ *L M2* **and** $\tau \in$ *L M2*
using *assms*(*12*) **by** *auto*
ultimately have *after-initial M2* $\pi \neq$ *after-initial M2* τ
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*
then obtain *v* **where** *minimally-distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) *v*
using *minimally-distinguishes-ex*
by (*metis* ‹$\neg$ *converge M2* π τ› ‹$\pi \in$ *L M2*› ‹$\tau \in$ *L M2*› *after-is-state assms*(*2*) *converge.simps*)
then have *distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) *v*
unfolding *minimally-distinguishes-def* **by** *auto*
then have $v \neq$ []
by (*meson* ‹$\pi \in$ *L M2*› ‹$\tau \in$ *L M2*› *after-is-state assms*(*2*) *distinguishes-not-Nil*)

have *length v* > *m* − *n*
proof (*rule ccontr*)
assume $\neg$ *m* − *n* < *length v*

have ‹$v \in$ *set* (*prefixes v*)›
unfolding *prefixes-set* **by** *auto*

show *False* **proof** (*cases* $v \in$ *LS M1* (*after-initial M1* π))
case *True*

have v = (*butlast v*)@[*last v*]
using ‹$v \neq []$› **by** *fastforce*
then obtain *x y* **where** v = (*butlast v*)@[(*x*,*y*)]
using *prod.exhaust* **by** *metis*
then have (*x*,*y*) $\in$ *set v*
using *in-set-conv-decomp* **by** *force*
then have $x \in$ *inputs M1* **and** $y \in$ *outputs M1*
using *language-io*[*OF True*, *of x y*] **by** *auto*
moreover have *length* (*butlast v* @ [(*x*, *y*)]) $\leq m -$ *size-r M1*
using ‹$\neg$ $m - n <$ *length v*› ‹v = (*butlast v*)@[(*x*,*y*)]›
unfolding *n* **by** *auto*
moreover have *butlast v* $\in$ *LS M1* (*after-initial M1* π)
using *True language-prefix*[*of butlast v* [(*x*,*y*)]]
unfolding ‹v = (*butlast v*)@[(*x*,*y*)]›[*symmetric*]
by *metis*
ultimately obtain *SC* α β **where** *SC* $\subseteq$ *T*
and $\{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha,\beta\} \wedge \omega' \in$ *list.set* (*prefixes v*)$\}$
$\subseteq$ *SC*
and *converge M1* π α
and *converge M2* π α
and *converge M1* τ β
and *converge M2* τ β
using *assms*(*11*)[*of* (*butlast v*) *x y*]
unfolding ‹v = (*butlast v*)@[(*x*,*y*)]›[*symmetric*]
by *meson*

then have $\alpha@v \in T$ **and** $\beta@v \in T$
using ‹*SC* $\subseteq$ *T*› ‹$\{\omega@\gamma \mid \omega\ \gamma .\ \omega \in \{\alpha,\beta\} \wedge \gamma \in$ *list.set* (*prefixes v*)$\} \subseteq$
SC› ‹$v \in$ *set* (*prefixes v*)›
by *auto*

then have *L M1* $\cap \{\alpha@v,\beta@v\}$ = *L M2* $\cap \{\alpha@v,\beta@v\}$
using *assms*(*10*) **by** *blast*

have *after-initial M1* $\pi \neq$ *after-initial M1* τ
using *converge-distinguishable-helper*[*OF assms*(*1*−*4*) ‹*converge M1*
π α› ‹*converge M2* π α› ‹*converge M1* τ β› ‹*converge M2* τ β› ‹*distinguishes*
M2 (*after-initial M2* π) (*after-initial M2* τ) v› ‹*L M1* $\cap \{\alpha@v,\beta@v\}$ = *L M2* $\cap$
$\{\alpha@v,\beta@v\}$›] **.**
then show *False*

using ‹*converge M1 π τ*›
by (*meson assms*(*1*) *assms*(*3*) *converge.elims*(*2*) *convergence-minimal*)
next
case *False*

obtain *io′ x′ y′ io″* **where** $v = io'@[(x',y')]@io''$
and $io' \in LS\ M1\ (after\text{-}initial\ M1\ \pi)$
and $io'@[(x',y')] \notin LS\ M1\ (after\text{-}initial\ M1\ \pi)$
using *language-maximal-contained-prefix-ob*[*OF False - assms*(*1*)]
by (*metis after-is-state assms*(*1*) *assms*(*9*) *converge.simps*)

have $length\ io' < m - size\text{-}r\ M1$
using ‹$\neg\ m - n < length\ v$› **unfolding** ‹$v = io'@[(x',y')]@io''$› *n* **by** *auto*
then have $length\ (io'@[(x',y')]) \leq m - size\text{-}r\ M1$
by *auto*

have $x' \in inputs\ M1$ **and** $y' \in outputs\ M1$
proof −
have $x' \in inputs\ M1 \wedge y' \in outputs\ M1$
proof −
have $(x',y') \in set\ v$
unfolding ‹$v = io'@[(x',y')]@io''$› **by** *auto*
then have $(x',\ y') \in set\ (\pi\ @\ v)$ **and** $(x',\ y') \in set\ (\tau\ @\ v)$
by *auto*
have $\pi@v \in L\ M2 \vee \tau@v \in L\ M2$
using ‹*distinguishes M2* (*after-initial M2 π*) (*after-initial M2 τ*) *v*›
unfolding *distinguishes-def*
by (*metis Un-iff* ‹$\pi \in L\ M2$› ‹$\tau \in L\ M2$› *after-language-iff assms*(*2*))
then show *?thesis*
using *language-io*[*of π@v M2 initial M2*, *OF* - ‹$(x',\ y') \in set\ (\pi\ @\ v)$›]
language-io[*of τ@v M2 initial M2*, *OF* - ‹$(x',\ y') \in set\ (\tau\ @\ v)$›]
by (*metis assms*(*7*) *assms*(*8*))
qed
then show $x' \in inputs\ M1$ **and** $y' \in outputs\ M1$
by *auto*
qed

obtain *SC α β* **where** $SC \subseteq T$
and $\{\omega@\omega' \mid \omega\ \omega'\ .\ \omega \in \{\alpha,\beta\} \wedge \omega' \in list.set\ (prefixes\ (io'@[(x',y')]))\} \subseteq SC$
and $converge\ M1\ \pi\ \alpha$
and $converge\ M2\ \pi\ \alpha$
and $converge\ M1\ \tau\ \beta$
and $converge\ M2\ \tau\ \beta$
using *assms*(*11*)[*of io′ x′ y′*, *OF* ‹$length\ (io'@[(x',y')]) \leq m - size\text{-}r\ M1$› ‹$io' \in LS\ M1\ (after\text{-}initial\ M1\ \pi)$› ‹$x' \in inputs\ M1$› ‹$y' \in outputs\ M1$›]
by *meson*

show *False* **proof** (*cases* $v \in$ *set* (*prefixes* (io'@[(x',y')])))
case *True*
then have α@$v \in T$ **and** β@$v \in T$
using ‹$SC \subseteq T$› ‹{ω@ω' | ω ω' . $\omega \in \{\alpha,\beta\} \wedge \omega' \in$ *list.set* (*prefixes* (io'@[(x',y')]))} $\subseteq SC$›
by *auto*

then have *L M1* $\cap$ {α@v,β@v} = *L M2* $\cap$ {α@v,β@v}
using *assms*(*10*) **by** *blast*

have *after-initial M1* $\pi \neq$ *after-initial M1* τ
using *converge-distinguishable-helper*[*OF assms*(*1*−*4*) ‹*converge M1* π α› ‹*converge M2* π α› ‹*converge M1* τ β› ‹*converge M2* τ β› ‹*distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) v› ‹*L M1* $\cap$ {α@v,β@v} = *L M2* $\cap$ {α@v,β@v}›] **.**
then show *False*
using ‹*converge M1* π τ›
by (*meson assms*(*1*) *assms*(*3*) *converge.elims*(*2*) *convergence-minimal*)
next
case *False*
then obtain io''' io'''' **where** io'' = io'''@io''''
and v = io'@[(x',y')]@io'''
and $io''' \neq$ []
using *prefixes-prefix-suffix-ob*[*of* v io'@[(x',y')] io'']
using ‹$v \in$ *set* (*prefixes* v)›
unfolding ‹v = io'@[(x',y')]@io''›
by *auto*
then have io'@[(x',y')] $\in$ *set* (*prefixes* v) **and** io'@[(x',y')] $\neq v$
unfolding *prefixes-set* **by** *auto*
then have io'@[(x',y')] $\in$ *LS M2* (*after-initial M2* π)
using *minimally-distinguishes-proper-prefix-in-language*[*OF* ‹*minimally-distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) v›, *of* io'@[(x',y')]]
by *blast*
then have io'@[(x',y')] $\in$ *LS M2* (*after-initial M2* α)
using ‹*converge M2* π α› *converge.simps* **by** *blast*
then have α@(io'@[(x',y')]) $\in$ *L M2*
by (*meson* ‹*converge M2* π α› *after-language-iff assms*(*2*) *converge.elims*(*2*))
moreover have α@(io'@[(x',y')]) $\in T$
using ‹{ω@ω' | ω ω' . $\omega \in \{\alpha,\beta\} \wedge \omega' \in$ *list.set* (*prefixes* (io'@[(x',y')]))} $\subseteq SC$› ‹$SC \subseteq T$›
unfolding *prefixes-set* **by** *force*
moreover have α@(io'@[(x',y')]) $\notin$ *L M1*
by (*metis* ‹*converge M1* π α› ‹io' @ [(x', y')] $\notin$ *LS M1* (*after-initial M1* π)› *after-language-iff assms*(*1*) *converge.elims*(*2*))
ultimately show *False*
using *assms*(*10*) **by** *blast*
qed
qed
qed

define *vm* **where** *vm*: *vm* = *take* (*m*−*n*) *v*
define *v-suffix* **where** *v-suffix*: *v-suffix* = *drop* (*m*−*n*) *v*
have *length vm* = *m*−*n* **and** *vm* ≠ *v*
  using ‹*m* − *n* < *length v*› **unfolding** *vm* **by** *auto*
have *v* = *vm*@*v-suffix*
  unfolding *vm v-suffix* **by** *auto*
then have *vm* ∈ *set* (*prefixes v*)
  unfolding *prefixes-set* **by** *auto*

have *vm* ∈ *LS M2* (*after-initial M2* π) **and** *vm* ∈ *LS M2* (*after-initial M2* τ)
using *minimally-distinguishes-proper-prefix-in-language*[*OF* ‹*minimally-distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) *v*› ‹*vm* ∈ *set* (*prefixes v*)› ‹*vm* ≠ *v*›]
  by *auto*

have *vm* ∈ *LS M1* (*after-initial M1* π)
proof (*rule ccontr*)
  assume *False*: *vm* ∉ *LS M1* (*after-initial M1* π)

  obtain *io′ x′ y′ io″* **where** *vm* = *io′*@[(*x′*,*y′*)]@*io″*
                and *io′* ∈ *LS M1* (*after-initial M1* π)
                and *io′*@[(*x′*,*y′*)] ∉ *LS M1* (*after-initial M1* π)
    using *language-maximal-contained-prefix-ob*[*OF False - assms*(*1*)]
    by (*metis after-is-state assms*(*1*) *assms*(*9*) *converge.simps*)

  have *length io′* < *m* − *size-r M1*
   using ‹*length vm* = *m* − *n*› **unfolding** ‹*vm* = *io′*@[(*x′*,*y′*)]@*io″*› *n* **by** *auto*
  then have *length* (*io′*@[(*x′*,*y′*)]) ≤ *m* − *size-r M1*
    by *auto*

  have *x′* ∈ *inputs M1*
    using ‹*vm* ∈ *LS M2* (*after-initial M2* π)›
    unfolding ‹*vm* = *io′*@[(*x′*,*y′*)]@*io″*›
    using *language-io*[*of io′* @ [(*x′*, *y′*)] @ *io″ M2 initial M2 x′ y′*]
    by (*metis append-Cons assms*(*7*) *in-set-conv-decomp language-io*(*1*))

  have *y′* ∈ *outputs M1*
    using ‹*vm* ∈ *LS M2* (*after-initial M2* π)›
    unfolding ‹*vm* = *io′*@[(*x′*,*y′*)]@*io″*›
    using *language-io*[*of io′* @ [(*x′*, *y′*)] @ *io″ M2 initial M2 x′ y′*]
    by (*metis append-Cons assms*(*8*) *in-set-conv-decomp language-io*(*2*))

  obtain *SC* α β **where** *SC* ⊆ *T*
                and {ω@ω′ | ω ω′ . ω ∈ {α,β} ∧ ω′ ∈ *list.set* (*prefixes* (*io′*@[(*x′*,*y′*)]))} ⊆ *SC*

and *converge M1* π α
and *converge M2* π α
and *converge M1* τ β
and *converge M2* τ β
using *assms*(*11*)[*of io′ x′ y′, OF* ‹*length* (*io′*@[(*x′*,*y′*)]) ≤ *m* − *size-r M1*› ‹*io′* ∈ *LS M1* (*after-initial M1* π)› ‹*x′* ∈ *inputs M1*› ‹*y′* ∈ *outputs M1*›]
by *meson*

have *io′*@[(*x′*,*y′*)] ∈ *LS M2* (*after-initial M2* π)
using ‹*vm* ∈ *LS M2* (*after-initial M2* π)› *language-prefix* **unfolding** ‹*vm* = *io′*@[(*x′*,*y′*)]@*io″*›
by (*metis append-assoc*)
then have α@(*io′*@[(*x′*,*y′*)]) ∈ *L M2*
by (*metis* ‹*converge M2* π α› *after-language-iff assms*(*2*) *converge.simps*)
moreover have α@(*io′*@[(*x′*,*y′*)]) ∈ *T*
using ‹{ω@ω' | ω ω' . ω ∈ {α,β} ∧ ω' ∈ *list.set* (*prefixes* (*io′*@[(*x′*,*y′*)]))} ⊆ *SC*› ‹*SC* ⊆ *T*›
unfolding *prefixes-set* **by** *force*
moreover have α@(*io′*@[(*x′*,*y′*)]) ∉ *L M1*
by (*metis* ‹*converge M1* π α› ‹*io′* @ [(*x′*, *y′*)] ∉ *LS M1* (*after-initial M1* π)› *after-language-iff assms*(*1*) *converge.elims*(*2*))
ultimately show *False*
using *assms*(*10*) **by** *blast*
qed

obtain *SC* α β **where** *SC* ⊆ *T*
and *is-state-cover M1 SC*
and {ω@ω' | ω ω' . ω ∈ {α,β} ∧ ω' ∈ *list.set* (*prefixes vm*)} ⊆ *SC*
and *converge M1* π α
and *converge M2* π α
and *converge M1* τ β
and *converge M2* τ β
and *preserves-divergence M1 M2 SC*
proof (*cases vm rule*: *rev-cases*)
case *Nil*
then have *list.set* (*prefixes vm*) = {[]}
by *auto*
then have ⋀ α β . {ω@ω' | ω ω' . ω ∈ {α,β} ∧ ω' ∈ *list.set* (*prefixes vm*)} = {α,β}
by *blast*
then show *?thesis* **using** *assms*(*12*) *that*
by *force*
next
case (*snoc blvm lvm*)
then obtain *x y* **where** *vm* = *blvm*@[(*x*,*y*)]
using *prod.exhaust* **by** *metis*

have ∗:*length* (*blvm*@[(*x*,*y*)]) ≤ *m* − *size-r M1*

using ‹*length vm* = *m* − *n*› ‹*vm* = *blvm* @ [(*x*, *y*)]› *n* **by** *fastforce*
have ∗∗:*blvm* ∈ *LS M1* (*after-initial M1* π)
using ‹*vm* = *blvm* @ [(*x*, *y*)]› *language-prefix* ‹*vm* ∈ *LS M1* (*after-initial M1* π)›
by *metis*
have ∗∗∗:*x* ∈ *inputs M1* **and** ∗∗∗∗:*y* ∈ *outputs M1*
using *language-io*[*OF* ‹*vm* ∈ *LS M1* (*after-initial M1* π)›, *of x y*]
unfolding ‹*vm* = *blvm* @ [(*x*, *y*)]› **by** *auto*
show *?thesis*
using *assms*(*11*)[*OF* ∗ ∗∗ ∗∗∗ ∗∗∗∗] *that*
unfolding ‹*vm* = *blvm* @ [(*x*, *y*)]›[*symmetric*] **by** *force*
qed

have *vm* ∈ *LS M1* (*after-initial M1* α) ∩ *LS M1* (*after-initial M1* β)
using ‹*converge M1* π α› ‹*converge M1* π τ› ‹*converge M1* τ β› ‹*vm* ∈ *LS M1* (*after-initial M1* π)› **by** *auto*
then have *vm* ∈ *LS M1* (*after-initial M1* α) **by** *blast*
have α ∈ *L M2*
using ‹*converge M2* π α› **by** *auto*
have β ∈ *L M2*
using ‹*converge M2* τ β› **by** *auto*
have *minimally-distinguishes M2* (*after-initial M2* α) (*after-initial M2* β) *v*
using ‹*minimally-distinguishes M2* (*after-initial M2* π) (*after-initial M2* τ) *v*›
by (*metis* ‹α ∈ *L M2*› ‹β ∈ *L M2*› ‹π ∈ *L M2*› ‹τ ∈ *L M2*› ‹*converge M2* π α› ‹*converge M2* τ β› *assms*(*2*) *assms*(*4*) *convergence-minimal*)

have *converge M1* α β
using ‹*converge M1* π α› ‹*converge M1* τ β› *assms*(*9*) **by** *auto*
have ¬*converge M2* α β
using ‹*converge M2* π α› ‹*converge M2* τ β› ‹¬*converge M2* π τ› **by** *auto*

have *preserves-divergence M1 M2* {ω@ω′ | ω ω′ . ω ∈ {α,β} ∧ ω′ ∈ *list.set* (*prefixes vm*)}
using ‹{ω@ω′ | ω ω′ . ω ∈ {α,β} ∧ ω′ ∈ *list.set* (*prefixes vm*)} ⊆ *SC*› ‹*preserves-divergence M1 M2 SC*›
unfolding *preserves-divergence.simps* **by** *blast*

have *L M1* ∩ {α′ @ γ |α′ γ. α′ ∈ {α, β} ∧ γ ∈ *set* (*prefixes vm*)} = *L M2* ∩ {α′ @ γ |α′ γ. α′ ∈ {α, β} ∧ γ ∈ *set* (*prefixes vm*)}
using ‹{ω@ω′ | ω ω′ . ω ∈ {α,β} ∧ ω′ ∈ *list.set* (*prefixes vm*)} ⊆ *SC*› ‹*SC* ⊆ *T*› *assms*(*10*)
by *blast*

have *card-geq*: *card* (*after-initial M2* ‘ {α′ @ γ |α′ γ. α′ ∈ {α, β} ∧ γ ∈ *set* (*prefixes vm*)}) ≥ (*m*−*n*) + *card* (*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*))

+ 1
using *preserves-divergence-minimally-distinguishing-prefixes-lower-bound*[*OF assms*(*1*−*4*) ‹*converge M1* α β› ‹¬*converge M2* α β› ‹$\alpha \in$ *L M2*› ‹$\beta \in$ *L M2*› ‹*minimally-distinguishes M2* (*after-initial M2* α) (*after-initial M2* β) *v*› ‹*vm* $\in$ *set* (*prefixes v*)› ‹*vm* $\neq$ *v*› ‹*vm* $\in$ *LS M1* (*after-initial M1* α) $\cap$ *LS M1* (*after-initial M1* β)› ‹*preserves-divergence M1 M2* $\{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha,\beta\} \wedge \omega' \in$ *list.set* (*prefixes vm*)$\}$› ‹*L M1* $\cap \{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$ = *L M2* $\cap \{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$›]
unfolding ‹*length vm* = *m*−*n*› **.**

have *after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\} \subseteq$ *states M2*
proof
fix *q* **assume** *q* $\in$ *after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$
then obtain *w1 w2* **where** *q* = *after-initial M2* (*w1*@*w2*)
and *w1* $\in \{\alpha,\beta\}$
and *w2* $\in$ *set* (*prefixes vm*)
by *blast*

have *w2* $\in$ *LS M2* (*after-initial M2* α)
using ‹*w2* $\in$ *set* (*prefixes vm*)› **unfolding** *prefixes-set*
by (*metis* ‹*converge M2* π α› ‹*vm* $\in$ *LS M2* (*after-initial M2* π)› ‹*w2* $\in$ *set* (*prefixes vm*)› *converge.elims*(*2*) *language-prefix prefixes-set-ob*)
then have *after-initial M2* (α@*w2*) $\in$ *states M2*
by (*meson* ‹$\alpha \in$ *L M2*› *after-is-state after-language-iff assms*(*2*))

have *w2* $\in$ *LS M2* (*after-initial M2* β)
using ‹*w2* $\in$ *set* (*prefixes vm*)› **unfolding** *prefixes-set*
by (*metis* ‹*converge M2* τ β› ‹*vm* $\in$ *LS M2* (*after-initial M2* τ)› ‹*w2* $\in$ *set* (*prefixes vm*)› *converge.elims*(*2*) *language-prefix prefixes-set-ob*)
then have *after-initial M2* (β@*w2*) $\in$ *states M2*
by (*meson* ‹$\beta \in$ *L M2*› *after-is-state after-language-iff assms*(*2*))

show *q* $\in$ *states M2*
unfolding ‹*q* = *after-initial M2* (*w1*@*w2*)›
using ‹*w1* $\in \{\alpha,\beta\}$› ‹*after-initial M2* (α@*w2*) $\in$ *states M2*› ‹*after-initial M2* (β@*w2*) $\in$ *states M2*›
by *blast*
qed
have *upper-bound*: *card* (*after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$) $\leq$ *m*
proof −
have *card* (*after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$) $\leq$ *size M2*
using ‹*after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\} \subseteq$ *states M2*›
using *fsm-states-finite*[*of M2*] **unfolding** *FSM.size-def*
by (*simp add*: *card-mono*)

then show *?thesis*
using ‹*size M2* $\leq$ *m*› **by** *linarith*
qed

have *after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*) $\subseteq$ *reachable-states M1*
proof
fix *q* **assume** *q* $\in$ *after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*)
then obtain *vm′* **where** *q* = *after M1* (*after-initial M1* α) *vm′* **and** *vm′* $\in$ *set* (*prefixes vm*)
by *auto*

have *vm′* $\in$ *LS M1* (*after-initial M1* α)
using ‹*vm′* $\in$ *set* (*prefixes vm*)› **unfolding** *prefixes-set*
by (*metis* ‹*vm* $\in$ *LS M1* (*after-initial M1* α)› ‹*vm′* $\in$ *set* (*prefixes vm*)› *language-prefix prefixes-set-ob*)
then have α@*vm′* $\in$ *L M1*
by (*meson* ‹*converge M1* π α› *after-language-iff assms*(*1*) *converge.simps*)
moreover have *q* = *after-initial M1* (α@*vm′*)
unfolding ‹*q* = *after M1* (*after-initial M1* α) *vm′*›
by (*meson after-split assms*(*1*) *calculation*)
ultimately show *q* $\in$ *reachable-states M1*
using *after-reachable-initial*[*OF assms*(*1*)] **by** *auto*
qed
moreover have *finite* (*reachable-states M1*)
using *fsm-states-finite*[*of M1*] *reachable-state-is-state*[*of - M1*]
by (*metis fsm-states-finite restrict-to-reachable-states-simps*(*2*))
ultimately have *card* (*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*)) $\leq$ *n*
unfolding *n*
by (*metis card-mono*)

have $\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ $\exists$ *io* $\in$ *SC*. *q* $\in$ *io-targets M1 io* (*FSM.initial M1*)
using ‹*is-state-cover M1 SC*›
by *auto*

obtain *V* **where** *is-state-cover-assignment M1 V*
and $\bigwedge$*q*. *q* $\in$ *reachable-states M1* $\Longrightarrow$ *V q* $\in$ *SC*
using *state-cover-assignment-from-state-cover*[*OF* ‹*is-state-cover M1 SC*›]
by *blast*

define *unreached-states* **where** *unreached-states*: *unreached-states* = *reachable-states M1* − (*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*))

have *size-r M1* = *card* (*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*)) + *card unreached-states*
by (*metis* ‹*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*) $\subseteq$ *reachable-states M1*› ‹*card* (*after M1* (*after-initial M1* α) ‘ *set* (*prefixes vm*)) $\leq$ *n*› ‹*finite* (*reachable-states M1*)› *card-Diff-subset le-add-diff-inverse n rev-finite-subset*

unreached-states)

have *unreached-V*: $\bigwedge$ *q* . *q* ∈ *unreached-states* ⟹ *V q* ∈ *L M1* ∧ *V q* ∈ *L M2* ∧ *V q* ∈ *SC*
proof −
fix *q* **assume** *q* ∈ *unreached-states*
then have *q* ∈ *reachable-states M1*
unfolding *unreached-states* **by** *auto*
then have *after-initial M1* (*V q*) = *q*
using *is-state-cover-assignment-observable-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *auto*

have *V q* ∈ *L M1*
using *is-state-cover-assignment-language*[*OF* ‹*is-state-cover-assignment M1 V*›] ‹*q* ∈ *reachable-states M1*›
by *auto*
moreover have *V q* ∈ *T* **and** *V q* ∈ *SC*
using ‹$\bigwedge$*q*. *q* ∈ *reachable-states M1* ⟹ *V q* ∈ *SC*› ‹*q* ∈ *reachable-states M1*› ‹*SC* ⊆ *T*›
by *auto*
ultimately have *V q* ∈ *L M2*
by (*metis Int-iff assms*(*10*))

show *V q* ∈ *L M1* ∧ *V q* ∈ *L M2* ∧ *V q* ∈ *SC*
using ‹*V q* ∈ *L M1*› ‹*V q* ∈ *L M2*› ‹*V q* ∈ *SC*› **by** *auto*
qed

have $\bigwedge$ *q1 q2* . *q1* ∈ *unreached-states* ⟹ *q2* ∈ *unreached-states* ⟹ *q1* ≠ *q2* ⟹ *after-initial M2* (*V q1*) ≠ *after-initial M2* (*V q2*)
proof −
fix *q1 q2* **assume** *q1* ∈ *unreached-states* **and** *q2* ∈ *unreached-states* **and** *q1* ≠ *q2*

then have *q1* ∈ *reachable-states M1* **and** *q2* ∈ *reachable-states M1*
unfolding *unreached-states* **by** *auto*
then have *after-initial M1* (*V q1*) = *q1* **and** *after-initial M1* (*V q2*) = *q2*
using *is-state-cover-assignment-observable-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *auto*
then have *V q1* ≠ *V q2*
using ‹*q1* ≠ *q2*›
by *metis*

have *V q1* ∈ *L M1* **and** *V q2* ∈ *L M1*
using *is-state-cover-assignment-language*[*OF* ‹*is-state-cover-assignment M1 V*›] ‹*q1* ∈ *reachable-states M1*› ‹*q2* ∈ *reachable-states M1*›
by *auto*
moreover have *V q1* ∈ *T* **and** *V q2* ∈ *T* **and** *V q1* ∈ *SC* **and** *V q2* ∈ *SC*

using ‹$\bigwedge q.\ q \in$ *reachable-states M1* $\Longrightarrow$ *V q* $\in$ *SC*› ‹*q1* $\in$ *reachable-states M1*› ‹*q2* $\in$ *reachable-states M1*› ‹*SC* $\subseteq$ *T*›
by *auto*
ultimately have *V q1* $\in$ *L M2* **and** *V q2* $\in$ *L M2*
by (*metis Int-iff assms*(*10*))+

have $\neg$*converge M1* (*V q1*) (*V q2*)
by (*meson* ‹*is-state-cover-assignment M1 V*› ‹*q1* $\in$ *reachable-states M1*› ‹*q1* $\neq$ *q2*› ‹*q2* $\in$ *reachable-states M1*› *assms*(*1*) *assms*(*3*) *state-cover-assignment-diverges*)

then have $\neg$*converge M2* (*V q1*) (*V q2*)
using ‹*V q1* $\in$ *L M1*› ‹*V q2* $\in$ *L M1*› ‹*V q1* $\in$ *SC*› ‹*V q2* $\in$ *SC*› ‹*preserves-divergence M1 M2 SC*›
unfolding *preserves-divergence.simps* **by** *blast*
then have *after-initial M2* (*V q1*) $\neq$ *after-initial M2* (*V q2*)
using ‹*V q1* $\in$ *L M2*› ‹*V q2* $\in$ *L M2*›
using *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*
then show *after-initial M2* (*V q1*) $\neq$ *after-initial M2* (*V q2*)
by *auto*
qed

have *lower-bound*: *size M2* $\geq$ *card* (*after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$) + *card unreached-states*
proof −
have *finite unreached-states*
by (*simp add*: ‹*finite* (*reachable-states M1*)› *unreached-states*)
then have *finite* ((λ *q* . *after-initial M2* (*V q*)) ‘ *unreached-states*)
by *simp*

have *card unreached-states* = *card* ((λ *q* . *after-initial M2* (*V q*)) ‘ *unreached-states*)
using *image-inj-card-helper*[*of unreached-states* (λ *q* . *after-initial M2* (*V q*)), *OF* ‹*finite unreached-states*› ‹$\bigwedge$ *q1 q2* . *q1* $\in$ *unreached-states* $\Longrightarrow$ *q2* $\in$ *unreached-states* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *after-initial M2* (*V q1*) $\neq$ *after-initial M2* (*V q2*)›]
by *auto*

have *card-helper*: $\bigwedge$ *A B C* . *A* $\cap$ *B* = $\{\}$ $\Longrightarrow$ *A* $\subseteq$ *C* $\Longrightarrow$ *B* $\subseteq$ *C* $\Longrightarrow$ *finite C* $\Longrightarrow$ *card C* $\geq$ *card A* + *card B*
by (*metis Int-Un-distrib card-Un-disjoint card-mono finite-subset inf.absorb-iff2*)

have $\bigwedge$ *q* . *q* $\in$ *unreached-states* $\Longrightarrow$ *after-initial M2* (*V q*) $\notin$ (*after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$)
proof
fix *q* **assume** *q* $\in$ *unreached-states*
and *after-initial M2* (*V q*) $\in$ *after-initial M2* ‘ $\{\alpha' @ \gamma \mid \alpha'\ \gamma.\ \alpha' \in \{\alpha, \beta\} \wedge \gamma \in$ *set* (*prefixes vm*)$\}$

then obtain *w1 w2* **where** *after-initial M2* (*V q*) = *after-initial M2* (*w1@w2*)
and *w1* ∈ {α,β}
and *w2* ∈ *set* (*prefixes vm*)
by *blast*
then have (*w1@w2*) ∈ *SC*
using ‹{ω @ ω′ |ω ω′. ω ∈ {α, β} ∧ ω′ ∈ *set* (*prefixes vm*)} ⊆ *SC*› **by** *blast*

have *w2* ∈ *LS M2* (*after-initial M2* α)
using ‹*w2* ∈ *set* (*prefixes vm*)› **unfolding** *prefixes-set*
by (*metis* ‹*converge M2* π α› ‹*vm* ∈ *LS M2* (*after-initial M2* π)› ‹*w2* ∈ *set* (*prefixes vm*)› *converge.elims*(*2*) *language-prefix prefixes-set-ob*)
moreover have *w2* ∈ *LS M2* (*after-initial M2* β)
using ‹*w2* ∈ *set* (*prefixes vm*)› **unfolding** *prefixes-set*
by (*metis* ‹*converge M2* τ β› ‹*vm* ∈ *LS M2* (*after-initial M2* τ)› ‹*w2* ∈ *set* (*prefixes vm*)› *converge.elims*(*2*) *language-prefix prefixes-set-ob*)
ultimately have *w1@w2* ∈ *L M2*
using ‹*w1* ∈ {α,β}›
by (*metis* ‹*converge M2* π α› ‹*converge M2* τ β› *after-language-iff assms*(*2*) *converge.simps empty-iff insert-iff*)
then have *converge M2* (*V q*) (*w1@w2*)
using *unreached-V*[*OF* ‹*q* ∈ *unreached-states*›]
using ‹*after-initial M2* (*V q*) = *after-initial M2* (*w1* @ *w2*)› *assms*(*2*) *assms*(*4*) *convergence-minimal* **by** *blast*
moreover have ¬*converge M1* (*V q*) (*w1@w2*)
proof −
have *after M1* (*after-initial M1* α) *w2* = *after M1* (*after-initial M1* β) *w2*
by (*metis* ‹*converge M1* α β› *assms*(*1*) *assms*(*3*) *converge.simps convergence-minimal*)
then have *q* ≠ (*after M1* (*after-initial M1 w1*) *w2*)
using ‹*q* ∈ *unreached-states*› ‹*w1* ∈ {α,β}›
unfolding *unreached-states*
by (*metis DiffD2* ‹*w2* ∈ *set* (*prefixes vm*)› *image-eqI insert-iff singletonD*)
moreover have (*after M1* (*after-initial M1 w1*) *w2*) = (*after-initial M1* (*w1@w2*))
by (*metis* (*no-types*, *lifting*) *Int-iff* ‹*SC* ⊆ *T*› ‹*w1* @ *w2* ∈ *L M2*› ‹*w1* @ *w2* ∈ *SC*› *after-split assms*(*1*) *assms*(*10*) *in-mono*)
moreover have *q* = *after-initial M1* (*V q*)
using *is-state-cover-assignment-observable-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›] ‹*q* ∈ *unreached-states*›
unfolding *unreached-states*
by (*metis Diff-iff*)
ultimately show *?thesis*
by (*metis assms*(*1*) *assms*(*3*) *converge.elims*(*2*) *convergence-minimal*)
qed
moreover have *V q* ∈ *L M1* ∩ *SC*
using *unreached-V*[*OF* ‹*q* ∈ *unreached-states*›] **by** *auto*
moreover have *w1@w2* ∈ *L M1* ∩ *SC*

using ‹$SC \subseteq T$› ‹$w1$ @ $w2 \in L\ M2$› ‹$w1$ @ $w2 \in SC$› *assms*(*10*) **by** *auto*
ultimately show *False*
using ‹*preserves-divergence M1 M2 SC*›
unfolding *preserves-divergence.simps*
by *blast*
qed
then have $*$: $((\lambda\ q\ .\ \textit{after-initial}\ M2\ (V\ q))\ `\ \textit{unreached-states}) \cap (\textit{after-initial}\ M2\ `\ \{\alpha'\ @\ \gamma\ |\alpha'\ \gamma.\ \alpha' \in \{\alpha,\ \beta\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ vm)\}) = \{\}$
by *blast*

have $**$: $((\lambda\ q\ .\ \textit{after-initial}\ M2\ (V\ q))\ `\ \textit{unreached-states}) \subseteq \textit{states}\ M2$
using *unreached-V*
by (*meson after-is-state assms*(*2*) *image-subset-iff*)
moreover note ‹$(\textit{after-initial}\ M2\ `\ \{\alpha'\ @\ \gamma\ |\alpha'\ \gamma.\ \alpha' \in \{\alpha,\ \beta\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ vm)\}) \subseteq \textit{states}\ M2$›

show *?thesis*
unfolding ‹$\textit{card unreached-states} = \textit{card}\ ((\lambda\ q\ .\ \textit{after-initial}\ M2\ (V\ q))\ `\ \textit{unreached-states})$› *FSM.size-def*
using *card-helper*[*OF* $*$ $**$ ‹$(\textit{after-initial}\ M2\ `\ \{\alpha'\ @\ \gamma\ |\alpha'\ \gamma.\ \alpha' \in \{\alpha,\ \beta\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ vm)\}) \subseteq \textit{states}\ M2$› *fsm-states-finite*[*of M2*]]
by *linarith*
qed

moreover have *card-geq-unreached*: $\textit{card}\ (\textit{after-initial}\ M2\ `\ \{\alpha'\ @\ \gamma\ |\alpha'\ \gamma.\ \alpha' \in \{\alpha,\ \beta\} \wedge \gamma \in \textit{set}\ (\textit{prefixes}\ vm)\}) + \textit{card unreached-states} \geq m + 1$
using *card-geq*
using ‹$\textit{size-r}\ M1 \leq m$›
unfolding *n*
unfolding ‹$\textit{size-r}\ M1 = \textit{card}\ (\textit{after}\ M1\ (\textit{after-initial}\ M1\ \alpha)\ `\ \textit{set}\ (\textit{prefixes}\ vm)) + \textit{card unreached-states}$›
by *linarith*

ultimately have $\textit{size}\ M2 \geq m + 1$
by *linarith*
then show *False*
using ‹$\textit{size}\ M2 \leq m$›
by *linarith*
qed
qed

lemma *establish-convergence-from-pass* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and $\textit{size-r}\ M1 \leq m$

and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1 V*
and *L M1* $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1*
and *converge M1 u v*
and *u* $\in$ *L M2*
and *v* $\in$ *L M2*
and *prop1*: $\bigwedge\gamma$ *x y*.
length (γ @ [(*x*, *y*)]) $\leq$ (*m* − *size-r M1*) $\Longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
x $\in$ *FSM.inputs M1* $\Longrightarrow$
y $\in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) $\wedge$
preserves-divergence M1 M2 ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
and *prop2*: *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {*u*, *v*})
shows *converge M2 u v*
proof −

define *language-up-to-depth* **where** *language-up-to-depth*: *language-up-to-depth* = {γ . $\gamma \in$ *LS M1* (*after-initial M1 u*) $\wedge$ *length* γ < (*m* − *size-r M1*)}

define *T1* **where** *T1*: *T1* = $\bigcup$ {((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) | γ *x y* . *length* (γ @ [(*x*, *y*)]) $\leq$ (*m* − *size-r M1*) $\wedge$ $\gamma \in$ *LS M1* (*after-initial M1 u*) $\wedge$ *x* $\in$ *inputs M1* $\wedge$ *y* $\in$ *outputs M1*}
define *T2* **where** *T2*: *T2* = ((*V* ‘ *reachable-states M1*) $\cup$ {*u*, *v*})

define *T* **where** *T*: *T* = *T1* $\cup$ *T2*

have *union-intersection-helper*: $\bigwedge$ *A B C* . (*A* $\cap \bigcup C$ = *B* $\cap \bigcup C$) = ($\forall$ *C*′ $\in$ *C* . *A* $\cap$ *C*′ = *B* $\cap$ *C*′)
by *blast*

have *L M1* $\cap$ *T* = *L M2* $\cap$ *T*
proof −
have (*L M1* $\cap$ *T1* = *L M2* $\cap$ *T1*)
unfolding *T1 union-intersection-helper*
using *prop1* **by** *blast*
moreover have *L M1* $\cap$ *T2* = *L M2* $\cap$ *T2*
proof−
have *u* $\in$ *L M1* **and** *v* $\in$ *L M1*
using ‹*converge M1 u v*› **by** *auto*
moreover note ‹*L M1* $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1*› ‹*u* $\in$ *L M2*› ‹*v* $\in$ *L M2*›

ultimately show *?thesis*
unfolding *T2* **by** *blast*
qed
ultimately show *?thesis*
unfolding *T* **by** *blast*
qed

have *prop1'*: ($\bigwedge\gamma$ *x y*.
length (γ @ [(*x*, *y*)]) $\leq$ *m* − *size-r M1* $\Longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
x $\in$ *FSM.inputs M1* $\Longrightarrow$
y $\in$ *FSM.outputs M1* $\Longrightarrow$
$\exists$ *SC* α β.
SC $\subseteq$ *T* $\wedge$
is-state-cover M1 SC $\wedge$
{ω @ ω' |ω ω'. $\omega \in \{\alpha, \beta\}$ $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))} $\subseteq$ *SC* $\wedge$
converge M1 u α $\wedge$ *converge M2 u* α $\wedge$ *converge M1 v* β $\wedge$ *converge M2 v* β
$\wedge$ *preserves-divergence M1 M2 SC*)
proof −
fix γ *x y*

define *SC* **where** *SC*: *SC* = ((*V* ‘ *reachable-states M1*) $\cup$ {ω @ ω' |ω ω'. $\omega \in$
{*u*, *v*} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})

assume *length* (γ @ [(*x*, *y*)]) $\leq$ *m* − *size-r M1*
$\gamma \in$ *LS M1* (*after-initial M1 u*)
x $\in$ *FSM.inputs M1*
y $\in$ *FSM.outputs M1*

then have *L M1* $\cap$ *SC* = *L M2* $\cap$ *SC*
preserves-divergence M1 M2 SC
using *prop1*[*of* γ *x y*]
unfolding *SC*
by *blast+*

have *SC* $\subseteq$ *T*
unfolding *T T1 SC*
using ‹*length* (γ @ [(*x*, *y*)]) $\leq$ *m* − *size-r M1*› ‹$\gamma \in$ *LS M1* (*after-initial M1*
u)› ‹*x* $\in$ *FSM.inputs M1*› ‹*y* $\in$ *FSM.outputs M1*›
by *blast*
moreover have *is-state-cover M1 SC*
proof −
have *is-state-cover M1* (*V* ‘ *reachable-states M1*)
using ‹*is-state-cover-assignment M1 V*›
by (*metis is-minimal-state-cover.simps minimal-state-cover-is-state-cover*)
moreover have (*V* ‘ *reachable-states M1*) $\subseteq$ *SC*
unfolding *SC*
by *blast*
ultimately show *?thesis*

unfolding *is-state-cover.simps* **by** *blast*
qed
moreover have $\{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))$\} \subseteq SC$
unfolding *SC* **by** *auto*
moreover have *converge M1 u u*
using ‹*converge M1 u v*› **by** *auto*
moreover have *converge M1 v v*
using ‹*converge M1 u v*› **by** *auto*
moreover have *converge M2 u u*
using ‹$u \in L$ *M2*› **by** *auto*
moreover have *converge M2 v v*
using ‹$v \in L$ *M2*› **by** *auto*
moreover note ‹*preserves-divergence M1 M2 SC*›
ultimately show $\exists$ *SC* α β.
SC $\subseteq T \wedge$
is-state-cover M1 SC $\wedge$
$\{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{\alpha, \beta\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))$\} \subseteq SC \wedge$
converge M1 u $\alpha \wedge$ *converge M2 u* $\alpha \wedge$ *converge M1 v* $\beta \wedge$ *converge M2 v* β $\wedge$ *preserves-divergence M1 M2 SC*
by *blast*
qed

have *prop2'*: $\exists$ *SC* α β.
SC $\subseteq T \wedge$
is-state-cover M1 SC $\wedge$
$\alpha \in SC \wedge \beta \in SC \wedge$ *converge M1 u* $\alpha \wedge$ *converge M2 u* $\alpha \wedge$ *converge M1 v* β $\wedge$ *converge M2 v* $\beta \wedge$ *preserves-divergence M1 M2 SC*
proof −
define *SC* **where** *SC*: *SC* = ((*V* ‘ *reachable-states M1*) $\cup$ {u, v})
have *SC* $\subseteq T$
unfolding *T T2 SC* **by** *auto*
moreover have *is-state-cover M1 SC*
proof −
have *is-state-cover M1* (*V* ‘ *reachable-states M1*)
using ‹*is-state-cover-assignment M1 V*›
by (*metis is-minimal-state-cover.simps minimal-state-cover-is-state-cover*)
moreover have (*V* ‘ *reachable-states M1*) $\subseteq SC$
unfolding *SC*
by *blast*
ultimately show *?thesis*
unfolding *is-state-cover.simps* **by** *blast*
qed
moreover have $u \in SC$ **and** $v \in SC$
unfolding *SC* **by** *auto*
moreover have *converge M1 u u*
using ‹*converge M1 u v*› **by** *auto*
moreover have *converge M1 v v*
using ‹*converge M1 u v*› **by** *auto*

moreover have *converge M2 u u*
using ‹$u \in L\ M2$› **by** *auto*
moreover have *converge M2 v v*
using ‹$v \in L\ M2$› **by** *auto*
moreover have ‹*preserves-divergence M1 M2 SC*›
using *prop2* **unfolding** *SC* **.**
ultimately show *?thesis*
by *blast*
qed

show *converge M2 u v*
using *sufficient-condition-for-convergence*[*OF assms*(*1−8*,*11*) ‹$L\ M1 \cap T = L\ M2 \cap T$› *prop1' prop2'*]
by *blast*
qed

15.3 Proving Language Equivalence by Establishing a Convergence Preserving Initialised Transition Cover

definition *transition-cover* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *bool* **where**
transition-cover M A = ($\forall$ $q \in$ *reachable-states M* . $\forall$ $x \in$ *inputs M* . $\forall$ $y \in$ *outputs M* . $\exists$ α. $\alpha \in A \wedge \alpha@[(x,y)] \in A \wedge \alpha \in L\ M \wedge$ *after-initial M* $\alpha = q$)

lemma *initialised-convergence-preserving-transition-cover-is-complete* :
fixes *M1* :: $('a,'b,'c)$ *fsm*
fixes *M2* :: $('d,'b,'c)$ *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $L\ M1 \cap T = L\ M2 \cap T$
and $A \subseteq T$
and *transition-cover M1 A*
and $[] \in A$
and *preserves-convergence M1 M2 A*
shows $L\ M1 = L\ M2$
proof −

have *convergence-right*: $\bigwedge$ α β . $\alpha \in A \Longrightarrow$ *converge M1* α $\beta \Longrightarrow$ *converge M2* α β
proof −
fix α β
assume $\alpha \in A$ **and** *converge M1* α β

then show *converge M2* α β
proof (*induction* β *arbitrary*: α *rule*: *rev-induct*)

```
  case Nil
  then have α ∈ L M1 ∩ A
    by auto
  moreover have [] ∈ L M1 ∩ A
    using ‹[] ∈ A›
    by auto
  ultimately show ?case
    using ‹preserves-convergence M1 M2 A› ‹converge M1 α []›
    unfolding preserves-convergence.simps
    by blast
next
  case (snoc xy β)

  obtain x y where xy = (x,y)
    by force


  have α ∈ L M1
  and  β @ [(x,y)] ∈ L M1
  and  LS M1 (after-initial M1 α) = LS M1 (after-initial M1 (β @ [(x,y)]))
    using snoc unfolding ‹xy = (x,y)›
    by auto
  then have β ∈ L M1
    using language-prefix by metis
  then have after-initial M1 β ∈ reachable-states M1
    using after-reachable[OF ‹observable M1› - reachable-states-initial]
    by metis
  moreover have x ∈ inputs M1 and y ∈ outputs M1
    using language-io[OF ‹β @ [(x,y)] ∈ L M1›] by auto
  ultimately obtain γ where γ ∈ A
                    and γ @ [(x, y)] ∈ A
                    and γ ∈ L M1
                    and after-initial M1 γ = after-initial M1 β
    using ‹transition-cover M1 A›
    unfolding transition-cover-def
    by blast
  then have converge M1 γ β
    using ‹β ∈ L M1›
    by auto
  then have converge M2 γ β
    using snoc.IH[OF ‹γ ∈ A›]
    by simp
  then have β ∈ L M2
    by auto

  have converge M1 β γ
    using ‹converge M1 γ β› by auto
  then have converge M1 (β @ [(x, y)]) (γ @ [(x, y)])
    using converge-append[OF assms(1) - ‹β @ [(x,y)] ∈ L M1› ‹γ ∈ L M1›]
```

```
      by auto
    then have γ @ [(x, y)] ∈ L M1
      by auto
    then have γ @ [(x, y)] ∈ L M2
      using ‹γ @ [(x, y)] ∈ A› assms(7,8)
      by blast
    then have converge M2 (β @ [(x, y)]) (γ @ [(x, y)])
      using converge-append[OF assms(2) ‹converge M2 γ β› - ‹β ∈ L M2›]
      by auto

    have converge M1 α (γ @ [(x, y)])
      using ‹converge M1 (β @ [(x, y)]) (γ @ [(x, y)])›
      using ‹converge M1 α (β @ [xy])›
      unfolding ‹xy = (x,y)›
      by auto
    then have converge M2 α (γ @ [(x, y)])
      using ‹α ∈ A› ‹γ @ [(x, y)] ∈ A› ‹preserves-convergence M1 M2 A›
      unfolding preserves-convergence.simps
      by auto
    then show ?case
      using ‹converge M2 (β @ [(x, y)]) (γ @ [(x, y)])›
      unfolding ‹xy = (x,y)›
      by auto
  qed
qed

have reaching-sequence-ex : ⋀ q . q ∈ reachable-states M1 ⟹ ∃ α . α ∈ A ∧ α
∈ L M1 ∧ after-initial M1 α = q
proof −
  fix q assume q ∈ reachable-states M1
  then show ∃ α . α ∈ A ∧ α ∈ L M1 ∧ after-initial M1 α = q
  proof (induction rule: reachable-states-cases)
    case init
    then show ?case
      using ‹[] ∈ A›
      using language-contains-empty-sequence
      by (metis after.simps(1))
  next
    case (transition t)

    obtain γ where γ ∈ A and γ@[(t-input t,t-output t)] ∈ A and γ ∈ L M1
and after-initial M1 γ = t-source t
      using ‹transition-cover M1 A›
            ‹t-source t ∈ reachable-states M1›
            fsm-transition-input[OF ‹t ∈ transitions M1›]
            fsm-transition-output[OF ‹t ∈ transitions M1›]
      unfolding transition-cover-def
      by blast
```

have γ@[(*t-input t*,*t-output t*)] $\in$ *L M1*
using *after-language-iff*[*OF assms*(*1*) ‹$\gamma \in$ *L M1*›, *of* [(*t-input t*,*t-output t*)]] ‹*t* $\in$ *transitions M1*›
unfolding ‹*after-initial M1* γ = *t-source t*› *LS-single-transition*
by *auto*
moreover have *after M1* (*after-initial M1* γ) [(*t-input t*,*t-output t*)] = *t-target t*
using *after-transition*[*OF assms*(*1*)] ‹*t* $\in$ *transitions M1*›
unfolding ‹*after-initial M1* γ = *t-source t*›
by *auto*
ultimately have *after-initial M1* (γ@[(*t-input t*,*t-output t*)]) = *t-target t*
using *after-split*[*OF assms*(*1*)]
by *metis*
then show *?case*
using ‹γ@[(*t-input t*,*t-output t*)] $\in$ *A*› ‹γ@[(*t-input t*,*t-output t*)] $\in$ *L M1*›
by *blast*
qed
qed

have *arbitrary-convergence*: $\bigwedge$ α β . *converge M1* α β $\Longrightarrow$ *converge M2* α β
proof −
fix α β
assume *converge M1* α β

then have $\alpha \in$ *L M1* **and** $\beta \in$ *L M1*
by *auto*
then have *after-initial M1* $\alpha \in$ *reachable-states M1*
using *after-reachable*[*OF assms*(*1*) - *reachable-states-initial*]
by *auto*
then obtain γ **where** $\gamma \in$ *A* **and** $\gamma \in$ *L M1* **and** *after-initial M1* γ = *after-initial M1* α
using *reaching-sequence-ex* **by** *blast*
moreover have *after-initial M1* α = *after-initial M1* β
using *convergence-minimal*[*OF assms*(*3*,*1*) ‹$\alpha \in$ *L M1*› ‹$\beta \in$ *L M1*›] ‹*converge M1* α β›
by *blast*
ultimately have *converge M1* γ α **and** *converge M1* γ β
using ‹$\alpha \in$ *L M1*› ‹$\beta \in$ *L M1*›
by *auto*
then have *converge M2* γ α **and** *converge M2* γ β
using *convergence-right*[*OF* ‹$\gamma \in$ *A*›]
by *auto*
then show *converge M2* α β
by *auto*
qed

have *L M1* $\subseteq$ *L M2*
proof

fix α **assume** $\alpha \in L\ M1$
then have *converge M1* α α
by *auto*
then have *converge M2* α α
using *arbitrary-convergence*
by *blast*
then show $\alpha \in L\ M2$
by *auto*
qed
moreover have $L\ M2 \subseteq L\ M1$
proof (*rule ccontr*)
assume $\neg\ L\ M2 \subseteq L\ M1$
then obtain α' **where** $\alpha' \in L\ M2 - L\ M1$
by *auto*

obtain α *xy* β **where** $\alpha' = \alpha$ @ [*xy*] @ β **and** $\alpha \in L\ M2 \cap L\ M1$ **and** α @ [*xy*] $\in L\ M2 - L\ M1$
using *minimal-failure-prefix-ob*[*OF assms*(*1*,*2*) *fsm-initial fsm-initial* ‹$\alpha' \in L\ M2 - L\ M1$›]
by *blast*
moreover obtain *x y* **where** *xy* = (*x*,*y*)
by *force*
ultimately have $\alpha \in L\ M2$ **and** $\alpha \in L\ M1$ **and** α @ [(*x*,*y*)] $\in L\ M2$ **and** α @ [(*x*,*y*)] $\notin L\ M1$
by *auto*

have $x \in$ *inputs M1* **and** $y \in$ *outputs M1*
using *language-io*[*OF* ‹α @ [(*x*,*y*)] $\in L\ M2$›]
unfolding ‹*inputs M2* = *inputs M1*› ‹*outputs M2* = *outputs M1*›
by *auto*
moreover have *after-initial M1* $\alpha \in$ *reachable-states M1*
using *after-reachable*[*OF assms*(*1*) ‹$\alpha \in L\ M1$› *reachable-states-initial*]
by *auto*
ultimately obtain γ **where** $\gamma \in A$ **and** γ@[(*x*,*y*)] $\in A$ **and** $\gamma \in L\ M1$ **and** *after-initial M1* γ = *after-initial M1* α
using ‹*transition-cover M1 A*›
unfolding *transition-cover-def*
by *blast*
then have *converge M1* α γ
using ‹$\alpha \in L\ M1$›
by *auto*
then have *converge M2* α γ
using *arbitrary-convergence*
by *blast*

have $\gamma \in L\ M2$
using ‹$\gamma \in A$› ‹$\gamma \in L\ M1$› *assms*(*7*,*8*)
by *blast*

```
    have γ @ [(x,y)] ∈ L M2
      using ‹α @ [(x,y)] ∈ L M2› ‹converge M2 α γ›
      using after-language-iff[OF assms(2) ‹α ∈ L M2› ]
      using after-language-iff[OF assms(2) ‹γ ∈ L M2› ]
      unfolding convergence-minimal[OF assms(4,2) ‹α ∈ L M2› ‹γ ∈ L M2›]
      by auto

    have γ @ [(x,y)] ∉ L M1
      using ‹α @ [(x,y)] ∉ L M1›
      using after-language-iff[OF assms(1) ‹γ ∈ L M1› ]
      using after-language-iff[OF assms(1) ‹α ∈ L M1› ]
      unfolding ‹after-initial M1 γ = after-initial M1 α›
      by auto
    then have γ @ [(x,y)] ∉ L M2
      using ‹γ@[(x,y)] ∈ A› assms(7,8)
      by auto
    then show False
      using ‹γ @ [(x,y)] ∈ L M2›
      by auto
  qed
  ultimately show ?thesis
    by blast
qed

end
```

16 Convergence Graphs

This theory introduces the invariants required for the initialisation, insertion, lookup, and merge operations on convergence graphs.

```
theory Convergence-Graph
imports Convergence ../Prefix-Tree
begin


lemma after-distinguishes-diverge :
  assumes observable M1
  and     observable M2
  and     minimal M1
  and     minimal M2
  and     α ∈ L M1
  and     β ∈ L M1
  and     γ ∈ set (after T1 α) ∩ set (after T1 β)
  and     distinguishes M1 (after-initial M1 α) (after-initial M1 β) γ
  and     L M1 ∩ set T1 = L M2 ∩ set T1
shows ¬converge M2 α β
proof
  have γ ≠ []
```

using *assms*(*5*,*6*,*8*)
by (*metis after-distinguishes-language append-Nil2 assms*(*1*))
then have $\alpha \in$ *set T1* **and** $\beta \in$ *set T1*
using *assms*(*7*) *after-set-Cons*[*of* γ]
by *auto*

assume *converge M2* α β
moreover have $\alpha \in$ *L M2*
using *assms*(*5*,*9*) ‹$\alpha \in$ *set T1*› **by** *blast*
moreover have $\beta \in$ *L M2*
using *assms*(*6*,*9*) ‹$\beta \in$ *set T1*› **by** *blast*
ultimately have (*after-initial M2* α) = (*after-initial M2* β)
using *convergence-minimal*[*OF assms*(*4*,*2*)]
by *blast*
then have $\alpha@\gamma \in$ *L M2* = ($\beta@\gamma \in$ *L M2*)
using ‹*converge M2* α β› *assms*(*2*) *converge-append-language-iff* **by** *blast*
moreover have ($\alpha@\gamma \in$ *L M1*) $\neq$ ($\beta@\gamma \in$ *L M1*)
using *after-distinguishes-language*[*OF assms*(*1*,*5*,*6*,*8*)] **.**
moreover have $\alpha@\gamma \in$ *set T1* **and** $\beta@\gamma \in$ *set T1*
using *assms*(*7*) **unfolding** *after-set*
by (*metis IntE append-Nil2 assms*(*5*) *assms*(*6*) *calculation*(*2*) *insertE mem-Collect-eq*)+
ultimately show *False*
using *assms*(*9*)
by *blast*
qed

16.1 Required Invariants on Convergence Graphs

definition *convergence-graph-lookup-invar* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*e*,'*b*,'*c*) *fsm* $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$
'*d* $\Rightarrow$
bool
where
convergence-graph-lookup-invar M1 M2 cg-lookup G = ($\forall$ α . $\alpha \in$ *L M1* $\longrightarrow$ $\alpha \in$ *L M2* $\longrightarrow$ $\alpha \in$ *list.set* (*cg-lookup G* α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*cg-lookup G* α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β))

lemma *convergence-graph-lookup-invar-simp*:
assumes *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and $\alpha \in$ *L M1* **and** $\alpha \in$ *L M2*
and $\beta \in$ *list.set* (*cg-lookup G* α)
shows *converge M1* α β **and** *converge M2* α β
using *assms* **unfolding** *convergence-graph-lookup-invar-def* **by** *blast*+

definition *convergence-graph-initial-invar* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*e*,'*b*,'*c*) *fsm* $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$
(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ '*d*) $\Rightarrow$
bool

where
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial = (∀ *T* . (*L M1* ∩ *set T* = (*L M2* ∩ *set T*)) ⟶ *finite-tree T* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-initial M1 T*))

definition *convergence-graph-insert-invar* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*e*,'*b*,'*c*) *fsm* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
bool
where
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert = (∀ *G* γ . γ ∈ *L M1* ⟶ γ ∈ *L M2* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-insert G* γ))

definition *convergence-graph-merge-invar* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*e*,'*b*,'*c*) *fsm* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
bool
where
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge = (∀ *G* γ γ'. *converge M1* γ γ' ⟶ *converge M2* γ γ' ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟶ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-merge G* γ γ'))

end

17 An Always-Empty Convergence Graph

This theory implements a convergence graph that always returns an empty list for any lookup. By using this graph it is possible to represent methods via the SPY and H-Frameworks that do not distribute distinguishing traces over converging traces.

theory *Empty-Convergence-Graph*
imports *Convergence-Graph*
begin

type-synonym *empty-cg* = *unit*

definition *empty-cg-empty* :: *empty-cg* **where**
empty-cg-empty = ()

definition *empty-cg-initial* :: (('*a*,'*b*,'*c*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ *empty-cg*) **where**
empty-cg-initial M T = *empty-cg-empty*

definition *empty-cg-insert* :: (*empty-cg* ⇒ ('*b*×'*c*) *list* ⇒ *empty-cg*) **where**
empty-cg-insert G v = *empty-cg-empty*

definition *empty-cg-lookup* :: (*empty-cg* ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) **where**
empty-cg-lookup G v = [*v*]

definition *empty-cg-merge* :: (*empty-cg* ⇒ ('b×'c) *list* ⇒ ('b×'c) *list* ⇒ *empty-cg*) **where**
empty-cg-merge G u v = *empty-cg-empty*

lemma *empty-graph-initial-invar*: *convergence-graph-initial-invar M1 M2 empty-cg-lookup empty-cg-initial*
unfolding *convergence-graph-initial-invar-def convergence-graph-lookup-invar-def empty-cg-lookup-def empty-cg-initial-def*
by *auto*

lemma *empty-graph-insert-invar*: *convergence-graph-insert-invar M1 M2 empty-cg-lookup empty-cg-insert*
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def empty-cg-lookup-def empty-cg-insert-def*
by *auto*

lemma *empty-graph-merge-invar*: *convergence-graph-merge-invar M1 M2 empty-cg-lookup empty-cg-merge*
unfolding *convergence-graph-merge-invar-def convergence-graph-lookup-invar-def empty-cg-lookup-def empty-cg-merge-def*
by *auto*

end

18 H-Framework

This theory defines the H-Framework and provides completeness properties.

theory *H-Framework*
imports *Convergence-Graph ../Prefix-Tree ../State-Cover*
begin

18.1 Abstract H-Condition

definition *satisfies-abstract-h-condition* :: ('a,'b,'c) *fsm* ⇒ ('e,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment* ⇒ *nat* ⇒ *bool* **where**
satisfies-abstract-h-condition M1 M2 V m = (∀ *q γ* .
q ∈ *reachable-states M1* ⟶
length γ ≤ *Suc* (*m*−*size-r M1*) ⟶
list.set γ ⊆ *inputs M1* × *outputs M1* ⟶
butlast γ ∈ *LS M1 q* ⟶
(*let traces* = (*V* ‘ *reachable-states M1*)
∪ {*V q* @ *ω'* | *ω'*. *ω'* ∈ *list.set* (*prefixes γ*)}
in (*L M1* ∩ *traces* = *L M2* ∩ *traces*)
∧ *preserves-divergence M1 M2 traces*))

lemma *abstract-h-condition-exhaustiveness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
and *satisfies-abstract-h-condition M I V m*
shows *L M* = *L I*
proof (*rule ccontr*)
assume *L M* $\neq$ *L I*

define Π **where** Π: Π = (*V* ‘ *reachable-states M*)
define *n* **where** *n*: *n* = *size-r M*
define $\mathcal{X}$ **where** $\mathcal{X}$: $\mathcal{X}$ = (λ *q* . {*io*@[(*x*,*y*)] | *io x y* . *io* $\in$ *LS M q* $\wedge$ *length io* $\leq$ *m*−*n* $\wedge$ *x* $\in$ *inputs M* $\wedge$ *y* $\in$ *outputs M*})

have *pass-prop*: $\bigwedge$ *q* γ . *q* $\in$ *reachable-states M* $\Longrightarrow$ *length* γ $\leq$ *Suc* (*m*−*n*) $\Longrightarrow$ *list.set* γ $\subseteq$ *inputs M* $\times$ *outputs M* $\Longrightarrow$ *butlast* γ $\in$ *LS M q* $\Longrightarrow$
(*L M* $\cap$ (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L I* $\cap$ (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
and *dist-prop*: $\bigwedge$ *q* γ . *q* $\in$ *reachable-states M* $\Longrightarrow$ *length* γ $\leq$ *Suc* (*m*−*n*) $\Longrightarrow$ *list.set* γ $\subseteq$ *inputs M* $\times$ *outputs M* $\Longrightarrow$ *butlast* γ $\in$ *LS M q* $\Longrightarrow$
preserves-divergence M I (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
using ‹*satisfies-abstract-h-condition M I V m*›
unfolding *satisfies-abstract-h-condition-def Let-def* Π *n* **by** *blast*+

have *pass-prop*-$\mathcal{X}$: $\bigwedge$ *q* γ . *q* $\in$ *reachable-states M* $\Longrightarrow$ γ $\in$ $\mathcal{X}$ *q* $\Longrightarrow$
(*L M* $\cap$ (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L I* $\cap$ (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
and *dist-prop*-$\mathcal{X}$: $\bigwedge$ *q* γ . *q* $\in$ *reachable-states M* $\Longrightarrow$ γ $\in$ $\mathcal{X}$ *q* $\Longrightarrow$
preserves-divergence M I (Π $\cup$ {*V q* @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
proof −
fix *q* γ **assume** $*$: *q* $\in$ *reachable-states M* **and** γ $\in$ $\mathcal{X}$ *q*
then obtain *io x y* **where** γ = *io*@[(*x*,*y*)] **and** *io* $\in$ *LS M q* **and** *length io* $\leq$ *m*−*n* **and** *x* $\in$ *inputs M* **and** *y* $\in$ *outputs M*
unfolding $\mathcal{X}$ **by** *blast*

have $**$:*length* γ $\leq$ *Suc* (*m*−*n*)
using ‹γ = *io*@[(*x*,*y*)]› ‹*length io* $\leq$ *m*−*n*› **by** *auto*

have $***$:*list.set* $\gamma \subseteq$ *inputs* $M \times$ *outputs* M
using *language-io*[*OF* ‹$io \in LS\ M\ q$›] ‹$x \in$ *inputs* M› ‹$y \in$ *outputs* M›
unfolding ‹$\gamma = io@[(x,y)]$› **by** *auto*
have $****$:*butlast* $\gamma \in LS\ M\ q$
unfolding ‹$\gamma = io@[(x,y)]$› **using** ‹$io \in LS\ M\ q$› **by** *auto*

show $(L\ M \cap (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* $\gamma)\})$
$= L\ I \cap (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* $\gamma)\}))$
using *pass-prop*[*OF* $*$ $**$ $***$ $****$] **.**
show *preserves-divergence* $M\ I$ $(\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* $\gamma)\})$
using *dist-prop*[*OF* $*$ $**$ $***$ $****$] **.**
qed

have $(L\ M \cap \Pi) = (L\ I \cap \Pi)$
using *pass-prop*[*OF reachable-states-initial*, *of* []] *language-contains-empty-sequence*[*of* M] **by** *auto*
moreover have $\Pi \subseteq L\ M$
unfolding Π **using** *state-cover-assignment-after*(*1*)[*OF assms*(*1*) ‹*is-state-cover-assignment* $M\ V$›]
by *blast*
ultimately have $\Pi \subseteq L\ I$
using ‹$\Pi = (V$ ‘ *reachable-states* $M)$› **by** *blast*

obtain $\pi\ \tau'$ **where** $\pi \in \Pi$
and $\pi\ @\ \tau' \in (L\ M - L\ I) \cup (L\ I - L\ M)$
and $\bigwedge io\ q\ .\ q \in$ *reachable-states* $M \Longrightarrow (V\ q)@io \in (L\ M - L\ I) \cup (L\ I - L\ M) \Longrightarrow$ *length* $\tau' \leq$ *length* io
using ‹$(L\ M \cap \Pi) = (L\ I \cap \Pi)$›
using *minimal-sequence-to-failure-from-state-cover-assignment-ob*[*OF* ‹$L\ M \neq L\ I$› ‹*is-state-cover-assignment* $M\ V$›]
unfolding Π
by *blast*

obtain q **where** $q \in$ *reachable-states* M **and** $\pi = V\ q$
using ‹$\pi \in \Pi$› **unfolding** Π **by** *blast*
then have $\pi \in L\ M$ **and** *after-initial* $M\ \pi = q$
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment* $M\ V$›]
by *blast+*

have τ'-*min*: $\bigwedge \pi'\ io\ .\ \pi' \in \Pi \Longrightarrow \pi'@io \in (L\ M - L\ I) \cup (L\ I - L\ M) \Longrightarrow$ *length* $\tau' \leq$ *length* io
proof −
fix $\pi'\ io$
assume $\pi' \in \Pi$ **and** $\pi'@io \in (L\ M - L\ I) \cup (L\ I - L\ M)$

then obtain q **where** $q \in$ *reachable-states* M **and** $\pi' = V\ q$
unfolding Π **by** *blast*
then show *length* $\tau' \leq$ *length io*
using ‹$\bigwedge$ *io q* . $q \in$ *reachable-states* $M \Longrightarrow (V\ q)@io \in (L\ M - L\ I) \cup (L\ I - L\ M) \Longrightarrow$ *length* $\tau' \leq$ *length io*›
‹$\pi'@io \in (L\ M - L\ I) \cup (L\ I - L\ M)$› **by** *auto*
qed

obtain $\pi\tau$ *xy* τ'' **where** $\pi\ @\ \tau' = \pi\tau\ @\ [xy]\ @\ \tau''$
and $\pi\tau \in L\ M \cap L\ I$
and $\pi\tau@[xy] \in (L\ I - L\ M) \cup (L\ M - L\ I)$
using *minimal-failure-prefix-ob*[*OF* ‹*observable M*› ‹*observable I*› *fsm-initial fsm-initial*, *of* $\pi\ @\ \tau'$]
using *minimal-failure-prefix-ob*[*OF* ‹*observable I*› ‹*observable M*› *fsm-initial fsm-initial*, *of* $\pi\ @\ \tau'$]
using ‹$\pi\ @\ \tau' \in (L\ M - L\ I) \cup (L\ I - L\ M)$›
by (*metis Int-commute Un-iff*)

moreover obtain *x y* **where** $xy = (x,y)$
using *surjective-pairing* **by** *blast*

moreover have $\pi\tau = \pi\ @$ *butlast* τ'
proof −
have *length* $\pi\tau \geq$ *length* π
proof (*rule ccontr*)
assume $\neg$ *length* $\pi \leq$ *length* $\pi\tau$
then have *length* $(\pi\tau@[xy]) \leq$ *length* π
by *auto*
then have *take* (*length* $(\pi\tau@[xy]))\ \pi = \pi\tau@[xy]$
using ‹$\pi\ @\ \tau' = \pi\tau\ @\ [xy]\ @\ \tau''$›
by (*metis append-assoc append-eq-append-conv-if*)
then have $\pi = (\pi\tau@[xy])\ @$ (*drop* (*length* $(\pi\tau@[xy]))\ \pi$)
by (*metis append-take-drop-id*)
then have $\pi\tau@[xy] \in L\ M \cap L\ I$
using ‹$\pi \in \Pi$› ‹$\Pi \subseteq L\ I$› ‹$\Pi \subseteq L\ M$›
using *language-prefix*[*of* $(\pi\tau@[xy])$ *drop* (*length* $(\pi\tau@[xy]))\ \pi$, *of M initial M*]
using *language-prefix*[*of* $(\pi\tau@[xy])$ *drop* (*length* $(\pi\tau@[xy]))\ \pi$, *of I initial I*]
by *auto*
then show *False*
using ‹$\pi\tau@[xy] \in (L\ I - L\ M) \cup (L\ M - L\ I)$› **by** *blast*
qed

then have $\pi\tau = \pi\ @$ (*take* (*length* $\pi\tau$ − *length* π) τ')
using ‹$\pi\ @\ \tau' = \pi\tau\ @\ [xy]\ @\ \tau''$›
by (*metis dual-order.refl take-all take-append take-le*)

then have π @ ((*take* (*length* $\pi\tau$ − *length* π) τ')@[*xy*]) ∈ (*L I* − *L M*) ∪ (*L M* − *L I*)
using ‹$\pi\tau$@[*xy*] ∈ (*L I* − *L M*) ∪ (*L M* − *L I*)›
by (*metis append-assoc*)
then have *length* τ' ≤ *Suc* (*length* (*take* (*length* $\pi\tau$ − *length* π) τ'))
using τ'*-min*[*OF* ‹π ∈ Π›]
by (*metis Un-commute length-append-singleton*)
moreover have *length* τ' ≥ *Suc* (*length* (*take* (*length* $\pi\tau$ − *length* π) τ'))
using ‹π @ τ' = $\pi\tau$ @ [*xy*] @ τ''› ‹$\pi\tau$ = π @ *take* (*length* $\pi\tau$ − *length* π) τ'› *not-less-eq-eq* **by** *fastforce*
ultimately have *length* τ' = *Suc* (*length* (*take* (*length* $\pi\tau$ − *length* π) τ'))
by *simp*
then show *?thesis*
proof −
have π @ τ' = (π @ *take* (*length* $\pi\tau$ − *length* π) τ') @ [*xy*] @ τ''
using ‹π @ τ' = $\pi\tau$ @ [*xy*] @ τ''› ‹$\pi\tau$ = π @ *take* (*length* $\pi\tau$ − *length* π) τ'› **by** *presburger*
then have *take* (*length* $\pi\tau$ − *length* π) τ' = *butlast* τ'
by (*metis* (*no-types*) ‹*length* τ' = *Suc* (*length* (*take* (*length* $\pi\tau$ − *length* π) τ'))› *append-assoc append-butlast-last-id append-eq-append-conv diff-Suc-1 length-butlast length-greater-0-conv zero-less-Suc*)
then show *?thesis*
using ‹$\pi\tau$ = π @ *take* (*length* $\pi\tau$ − *length* π) τ'› **by** *fastforce*
qed
qed

ultimately have π @ (*butlast* τ') ∈ *L M* ∩ *L I*
and (π @ (*butlast* τ'))@[(*x*,*y*)] ∈ (*L I* − *L M*) ∪ (*L M* − *L I*)
by *auto*

have τ' = (*butlast* τ')@[(*x*,*y*)]
using ‹π @ τ' = $\pi\tau$ @ [*xy*] @ τ''› ‹*xy* = (*x*,*y*)›
unfolding ‹$\pi\tau$ = π @ *butlast* τ'›
by (*metis* (*no-types*, *opaque-lifting*) *append-Cons append-butlast-last-id butlast.simps*(*1*) *butlast-append last-appendR list.distinct*(*1*) *self-append-conv*)

have *x* ∈ *inputs M* **and** *y* ∈ *outputs M*
proof −
have ∗: (*x*,*y*) ∈ *list.set* ((π @ (*butlast* τ'))@[(*x*,*y*)])
by *auto*

show *x* ∈ *inputs M*
using ‹(π @ (*butlast* τ'))@[(*x*,*y*)] ∈ (*L I* − *L M*) ∪ (*L M* − *L I*)›
language-io(*1*)[*OF* - ∗, *of I*]
language-io(*1*)[*OF* - ∗, *of M*]
‹*inputs I* = *inputs M*›
by *blast*

```
    show y ∈ outputs M
      using ‹(π @ (butlast τ'))@[(x,y)] ∈ (L I − L M) ∪ (L M − L I)›
            language-io(2)[OF - ∗, of I]
            language-io(2)[OF - ∗, of M]
            ‹outputs I = outputs M›
      by blast
  qed

  have π @ (butlast τ') ∈ L M
    using ‹π @ (butlast τ') ∈ L M ∩ L I› by auto
  have list.set (π @ τ') ⊆ inputs M × outputs M
    using ‹π @ τ' ∈ (L M − L I) ∪ (L I − L M)›
    using language-io[of ‹π @ τ'› M initial M]
    using language-io[of ‹π @ τ'› I initial I]
    unfolding assms(6,7) by fast
  then have list.set τ' ⊆ inputs M × outputs M
    by auto
  have list.set (butlast τ') ⊆ inputs M × outputs M
    using language-io[OF ‹π @ (butlast τ') ∈ L M›] by force
  have butlast τ' ∈ LS M q
    using after-language-iff[OF assms(1) ‹π ∈ L M›] ‹π @ (butlast τ') ∈ L M›
    unfolding ‹after-initial M π = q›
    by blast

  have length τ' > m − n + 1
  proof (rule ccontr)
    assume ¬ m − n + 1 < length τ'
    then have length τ' ≤ Suc (m − n)
      by auto

    have π @ τ' ∈ (Π ∪ { V q @ ω' |ω'. ω' ∈ list.set (prefixes τ')})
      unfolding ‹π = V q› using ‹q ∈ reachable-states M›  unfolding prefixes-set
by auto
    then have L M ∩ {π @ τ'} = L I ∩ {π @ τ'}
       using pass-prop[OF ‹q ∈ reachable-states M› ‹length τ' ≤ Suc (m − n)›
‹list.set τ' ⊆ inputs M × outputs M› ‹butlast τ' ∈ LS M q›]
      by blast
    then show False
      using ‹π @ τ' ∈ (L M − L I) ∪ (L I − L M)› by blast
  qed



  define τ where τ-def: τ = (λi . take i (butlast τ'))

  have ⋀ i . i > 0 ⟹ i ≤ m − n + 1 ⟹ (τ i) ∈ 𝒳 q
  proof −
    fix i assume i > 0 and i ≤ m − n + 1
```

then have $\tau\ i = (butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]$
using τ-*def* ‹*length* $\tau' > m - n + 1$›
by (*metis add-less-same-cancel2 append-butlast-last-id length-butlast less-diff-conv list.size*(*3*) *not-add-less2 take-eq-Nil*)

have *length* $(butlast\ (\tau\ i)) \leq m - n$
using τ-*def* ‹*length* $\tau' > m - n + 1$› ‹$i \leq m - n + 1$› **by** *auto*

have $q \in$ *io-targets* $M\ \pi$ (*initial* M)
using ‹*is-state-cover-assignment* $M\ V$› ‹$q \in$ *reachable-states* M› ‹$\pi = V\ q$›
by *simp*
then have $(butlast\ \tau') \in LS\ M\ q$
using ‹$\pi\ @\ (butlast\ \tau') \in L\ M \cap L\ I$›
using *observable-io-targets-language*[*OF* - ‹*observable* M›]
by *force*
then have $\tau\ i\ @\ (drop\ i\ (butlast\ \tau')) \in LS\ M\ q$
using τ-*def* **by** *auto*
then have $\tau\ i \in LS\ M\ q$
using *language-prefix*
by *fastforce*
then have $butlast\ (\tau\ i) \in LS\ M\ q$
using *language-prefix* ‹$\tau\ i = (butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]$›
by *metis*

have $(fst\ (last\ (\tau\ i)), snd\ (last\ (\tau\ i))) \in list.set\ ((butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)])$
using ‹$\tau\ i = (butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]$›
using *in-set-conv-decomp* **by** *fastforce*
then have $fst\ (last\ (\tau\ i)) \in inputs\ M$
and $snd\ (last\ (\tau\ i)) \in outputs\ M$
using ‹$\tau\ i \in LS\ M\ q$› ‹$\tau\ i = (butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]$› *language-io*[*of* $(butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]\ M\ q\ fst\ (last\ (\tau\ i))\ snd\ (last\ (\tau\ i))$]
by *auto*

then show $\tau\ i \in \mathcal{X}\ q$
unfolding $\mathcal{X}$
using ‹$length\ (butlast\ (\tau\ i)) \leq m - n$› ‹$\tau\ i = (butlast\ (\tau\ i))\ @\ [last\ (\tau\ i)]$›
‹$butlast\ (\tau\ i) \in LS\ M\ q$›
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq surjective-pairing*)
qed

have $\bigwedge i\ .\ i \leq m{-}n{+}1 \Longrightarrow \pi\ @\ (\tau\ i) \in L\ M \cap L\ I$
proof −
fix i **assume** $i \leq m{-}n{+}1$

have $butlast\ \tau' = \tau\ i\ @\ (drop\ i\ (butlast\ \tau'))$
unfolding τ-*def* **by** *auto*
then have ‹$(\pi\ @\ \tau\ i)\ @\ (drop\ i\ (butlast\ \tau')) \in L\ M \cap L\ I$›
using ‹$\pi\ @\ (butlast\ \tau') \in L\ M \cap L\ I$›

by *auto*
then show π @ $(\tau\ i) \in L\ M \cap L\ I$
using *language-prefix*[*of* $(\pi$ @ $\tau\ i)$ $(drop\ i\ (butlast\ \tau'))$, *of M initial M*]
using *language-prefix*[*of* $(\pi$ @ $\tau\ i)$ $(drop\ i\ (butlast\ \tau'))$, *of I initial I*]
by *blast*
qed

have *B-diff*: $\Pi \cap (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\} = \{\}$
proof −
have $\bigwedge$ *io1 io2* . $io1 \in \Pi \Longrightarrow io2 \in (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\} \Longrightarrow io1 \neq io2$
proof (*rule ccontr*)
fix *io1 io2* **assume** $io1 \in \Pi$ $io2 \in (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}$ $\neg$ $io1 \neq io2$

then obtain i **where** $io2 = \pi$ @ $(\tau\ i)$ **and** $i \geq 1$ **and** $i \leq m - n + 1$ **and** π @ $(\tau\ i) \in \Pi$
by *auto*
then have π @ $(\tau\ i) \in L\ M$
using ‹$\bigwedge\ i\ .\ i \leq m-n+1 \Longrightarrow \pi$ @ $(\tau\ i) \in L\ M \cap L\ I$› **by** *auto*

obtain q **where** $q \in$ *reachable-states M* **and** $V\ q = \pi$ @ $(\tau\ i)$
using ‹π @ $(\tau\ i) \in \Pi$› Π
by *auto*
moreover have $(\pi$ @ $(\tau\ i))$@$(drop\ i\ \tau') \in (L\ M - L\ I) \cup (L\ I - L\ M)$
using τ-*def* ‹π @ $\tau' \in (L\ M - L\ I) \cup (L\ I - L\ M)$› ‹$length\ \tau' > m - n + 1$› ‹$i \leq m - n + 1$›
by (*metis append-assoc append-take-drop-id le-iff-sup sup.strict-boundedE take-butlast*)
ultimately have $length\ \tau' \leq length\ (drop\ i\ \tau')$
using ‹$\bigwedge$ *io q* . $q \in$ *reachable-states M* $\Longrightarrow (V\ q)$@$io \in (L\ M - L\ I) \cup (L\ I - L\ M) \Longrightarrow length\ \tau' \leq length\ io$›
by *presburger*
then show *False*
using ‹$length\ \tau' > m - n + 1$› ‹$i \geq 1$›
by (*metis One-nat-def* ‹$i \leq m - n + 1$› *diff-diff-cancel diff-is-0-eq′ le-trans length-drop less-Suc-eq nat-less-le*)
qed
then show *?thesis*
by *blast*
qed

have *same-targets-in-I*: $\exists\ \alpha\ \beta$.
$\alpha \in \Pi \cup (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}$
$\wedge\ \beta \in \Pi \cup (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}$
$\wedge\ \alpha \neq \beta \wedge (after\text{-}initial\ I\ \alpha = after\text{-}initial\ I\ \beta)$
proof −
have *after-initial I* ‘ $(\Pi \cup (\lambda i\ .\ \pi$ @ $(\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}) \subseteq$ *states I*

proof
fix q **assume** $q \in$ *after-initial I* ‘ $(\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\})$
then obtain *io* **where** $io \in (\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\})$ **and** q $=$ *after-initial I io*
by *blast*
then have $io \in L\ I$
using ‹$\bigwedge i\ .\ i \leq m-n+1 \Longrightarrow \pi\ @\ (\tau\ i) \in L\ M \cap L\ I$› ‹$\Pi \subseteq L\ I$› **by** *auto*
then show $q \in$ *states I*
unfolding ‹$q =$ *after-initial I io*›
using *observable-after-path*[*OF* ‹*observable I*›, *of io initial I*] *path-target-is-state*[*of I initial I*]
by *metis*
qed
then have *card* (*after-initial I* ‘ $(\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\})) \leq m$
using ‹*size I* $\leq m$› *fsm-states-finite*[*of I*] **unfolding** *FSM.size-def*
by (*meson card-mono le-trans*)
moreover have *card* $(\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}) = m+1$
proof −
have $*$: *card* $\Pi = n$
using *state-cover-assignment-card*[*OF* ‹*is-state-cover-assignment M V*› ‹*observable M*›] **unfolding** $\Pi\ n$ **.**
have $**$: *card* $((\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\}) = m-n+1$
proof −
have *finite* $((\lambda i\ .\ \pi\ @\ (\tau\ i))$ ‘ $\{1\ ..\ m-n+1\})$
by *auto*
moreover have *inj-on* $(\lambda i\ .\ \pi\ @\ (\tau\ i))\ \{1\ ..\ m-n+1\}$
proof
fix $x\ y$ **assume** $x \in \{1..m - n + 1\}$ $y \in \{1..m - n + 1\}$ $\pi\ @\ \tau\ x = \pi\ @\ \tau\ y$

then have *take* $x\ \tau' =$ *take* $y\ \tau'$
unfolding τ*-def* ‹*length* $\tau' > m - n + 1$›
by (*metis* (*no-types*, *lifting*) ‹$m - n + 1 <$ *length* τ'› *atLeastAtMost-iff diff-is-0-eq le-trans nat-less-le same-append-eq take-butlast zero-less-diff*)
moreover have $x \leq$ *length* τ'
using ‹$x \in \{1..m - n + 1\}$› ‹*length* $\tau' > m - n + 1$› **by** *auto*
moreover have $y \leq$ *length* τ'
using ‹$y \in \{1..m - n + 1\}$› ‹*length* $\tau' > m - n + 1$› **by** *auto*
ultimately show $x{=}y$
by (*metis length-take min.absorb2*)
qed
moreover have *card* $\{1..m - n + 1\} = m - n + 1$
by *auto*
ultimately show *?thesis*
by (*simp add*: *card-image*)
qed

have $***$: $n + (m - n + 1) = m+1$
unfolding n **using** ‹$m \geq$ *size-r M*› **by** *auto*

have *finite* Π
unfolding Π **using** *fsm-states-finite restrict-to-reachable-states-simps*(*2*)
by (*metis finite-imageI*)
have *finite* $((\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\})$
by *auto*

show *?thesis*
using *card-Un-disjoint*[*OF* ‹*finite* Π› ‹*finite* $((\lambda i.\ \pi\ @\ \tau\ i)\ `\ \{1..m - n + 1\})$› ‹$\Pi \cap (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} = \{\}$›]
unfolding $*$ $**$ $***$ **.**
qed
ultimately have $*$: *card* (*after-initial I* ‘ $(\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}))$ $<$ *card* $(\Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\})$
by *simp*

show *?thesis*
using *pigeonhole*[*OF* $*$] **unfolding** *inj-on-def* **by** *blast*
qed

have *same-targets-in-M*: $\bigwedge\ \alpha\ \beta\ .$
$\alpha \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \Longrightarrow$
$\beta \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \Longrightarrow$
$\alpha \neq \beta \Longrightarrow$
(*after-initial I* α = *after-initial I* β) $\Longrightarrow$
(*after-initial M* α = *after-initial M* β)
proof (*rule ccontr*)
fix $\alpha\ \beta$ **assume** $\alpha \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}$
and $\beta \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}$
and $\alpha \neq \beta$
and (*after-initial I* α = *after-initial I* β)
and (*after-initial M* $\alpha \neq$ *after-initial M* β)

have $*$: $(\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \subseteq \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\}$
using ‹$\bigwedge\ i\ .\ i > 0 \Longrightarrow i \leq m - n + 1 \Longrightarrow (\tau\ i) \in \mathcal{X}\ q$› ‹$q \in$ *reachable-states M*› ‹$\pi = V\ q$›
by *force*

have $\alpha \in L\ M$ **and** $\beta \in L\ M$ **and** $\alpha \in L\ I$ **and** $\beta \in L\ I$
using ‹$\alpha \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}$›
‹$\beta \in \Pi \cup (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}$›
‹$\bigwedge\ i\ .\ i \leq m-n+1 \Longrightarrow \pi\ @\ (\tau\ i) \in L\ M \cap L\ I$›
‹$\Pi \subseteq L\ M$› ‹$\Pi \subseteq L\ I$›
by *auto*
then have $\neg$ *converge M* $\alpha\ \beta$ **and** *converge I* $\alpha\ \beta$
using ‹*after-initial M* $\alpha \neq$ *after-initial M* β›

using ‹*minimal M*›
using *after-is-state*[*OF assms*(*1*) ‹$\alpha \in L\ M$›]
using *after-is-state*[*OF assms*(*1*) ‹$\beta \in L\ M$›]
unfolding *converge.simps minimal.simps* ‹*after-initial I* α = *after-initial I* β› **by** *auto*
then have $\neg$ *converge M* β α **and** *converge I* β α
using *converge-sym* **by** *blast+*

have *split-helper*: $\bigwedge$ (*P* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*) . ($\exists$ *i j* . *P i j* $\wedge$ *i* $\neq$ *j*) = (($\exists$ *i j* . *P i j* $\wedge$ *i* < *j*) $\vee$ ($\exists$ *i j* . *P i j* $\wedge$ *i* > *j*))
proof
show $\bigwedge$ (*P* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*) . $\exists$ *i j*. *P i j* $\wedge$ *i* $\neq$ *j* $\Longrightarrow$ ($\exists$ *i j*. *P i j* $\wedge$ *i* < *j*) $\vee$ ($\exists$ *i j*. *P i j* $\wedge$ *j* < *i*)
proof −
fix *P* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*
assume $\exists$ *i j*. *P i j* $\wedge$ *i* $\neq$ *j*
then obtain *i j* **where** *P i j* **and** *i* $\neq$ *j* **by** *auto*
then have *i* < *j* $\vee$ *i* > *j* **by** *auto*
then show ($\exists$ *i j*. *P i j* $\wedge$ *i* < *j*) $\vee$ ($\exists$ *i j*. *P i j* $\wedge$ *j* < *i*) **using** ‹*P i j*› **by** *auto*
qed
show $\bigwedge$ (*P* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*) . ($\exists$ *i j*. *P i j* $\wedge$ *i* < *j*) $\vee$ ($\exists$ *i j*. *P i j* $\wedge$ *j* < *i*) $\Longrightarrow$ $\exists$ *i j*. *P i j* $\wedge$ *i* $\neq$ *j* **by** *auto*
qed
have *split-scheme*:($\exists$ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*))
= (($\exists$ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *i* < *j* $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*))
$\vee$ ($\exists$ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *i* > *j* $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*)))
using ‹$\alpha \neq \beta$›
using *split-helper*[*of* λ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*)]
by *blast*

consider ($\alpha \in \Pi \wedge \beta \in \Pi$) |
($\exists$ *i* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ $\alpha \in \Pi \wedge \beta$ = π @ (τ *i*)) |
($\exists$ *i* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ $\beta \in \Pi \wedge \alpha$ = π @ (τ *i*)) |
($\exists$ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *i* < *j* $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*)) |
($\exists$ *i j* . *i* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *j* $\in$ {*1* .. *m*−*n*+*1*} $\wedge$ *i* > *j* $\wedge$ α = π @ (τ *i*) $\wedge$ β = π @ (τ *j*))
using ‹$\alpha \in \Pi \cup (\lambda i\ .\ \pi$ @ (τ *i*)) ‘ {*1* .. *m*−*n*+*1*}›
‹$\beta \in \Pi \cup (\lambda i\ .\ \pi$ @ (τ *i*)) ‘ {*1* .. *m*−*n*+*1*}›
using *split-scheme*
by *blast*

then have $\exists$ α' β' . $\alpha' \in L\ M \wedge \beta' \in L\ M \wedge \neg$*converge M* α' β' $\wedge$ *converge I*

α' β' $\wedge$
($(\alpha' \in \Pi \wedge \beta' \in \Pi)$
$\vee$ $(\exists\ i\ .\ i \in \{1\ ..\ m-n+1\} \wedge \alpha' \in \Pi \wedge \beta' = \pi\ @\ (\tau\ i))$
$\vee$ $(\exists\ i\ j\ .\ i < j \wedge i \in \{1\ ..\ m-n+1\} \wedge j \in \{1\ ..\ m-n+1\} \wedge \alpha' = \pi\ @\ (\tau\ i) \wedge \beta' = \pi\ @\ (\tau\ j)))
using ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$› ‹$\neg$ *converge* $M\ \alpha\ \beta$› ‹*converge* $I\ \alpha\ \beta$› ‹$\neg$ *converge* $M\ \beta\ \alpha$› ‹*converge* $I\ \beta\ \alpha$›
by *metis*

then obtain $\alpha'\ \beta'$ **where** $\alpha' \in L\ M$ **and** $\beta' \in L\ M$ **and** $\neg$*converge* $M\ \alpha'\ \beta'$ **and** *converge* $I\ \alpha'\ \beta'$
and $(\alpha' \in \Pi \wedge \beta' \in \Pi)$
$\vee$ $(\exists\ i\ .\ i \in \{1\ ..\ m-n+1\} \wedge \alpha' \in \Pi \wedge \beta' = \pi\ @\ (\tau\ i))$
$\vee$ $(\exists\ i\ j\ .\ i < j \wedge i \in \{1\ ..\ m-n+1\} \wedge j \in \{1\ ..\ m-n+1\} \wedge \alpha' = \pi\ @\ (\tau\ i) \wedge \beta' = \pi\ @\ (\tau\ j))$
by *blast*
then consider $\alpha' \in \Pi \wedge \beta' \in \Pi$
| $\exists\ i\ .\ i \in \{1\ ..\ m-n+1\} \wedge \alpha' \in \Pi \wedge \beta' = \pi\ @\ (\tau\ i)$
| $\exists\ i\ j\ .\ i < j \wedge i \in \{1\ ..\ m-n+1\} \wedge j \in \{1\ ..\ m-n+1\} \wedge \alpha' = \pi\ @\ (\tau\ i) \wedge \beta' = \pi\ @\ (\tau\ j)$
by *blast*

then show *False* **proof** *cases*
case *1*
moreover have *preserves-divergence* $M\ I\ \Pi$
using *dist-prop*[*OF reachable-states-initial*, *of* []] *language-contains-empty-sequence*[*of* M] **by** *auto*
ultimately show *?thesis*
using ‹$\neg$*converge* $M\ \alpha'\ \beta'$› ‹*converge* $I\ \alpha'\ \beta'$› ‹$\alpha' \in L\ M$› ‹$\beta' \in L\ M$›
unfolding *preserves-divergence.simps*
by *blast*
next
case *2*
then obtain i **where** $i \in \{1\ ..\ m-n+1\}$ **and** $\alpha' \in \Pi$ **and** $\beta' = \pi\ @\ (\tau\ i)$
by *blast*
then have $\tau\ i \in \mathcal{X}\ q$
using ‹$\bigwedge\ i\ .\ i > 0 \Longrightarrow i \leq m - n + 1 \Longrightarrow (\tau\ i) \in \mathcal{X}\ q$›
by *force*

have $\beta' \in \{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in$ *list.set* (*prefixes* $(\tau\ i))\}$
unfolding ‹$\beta' = \pi\ @\ (\tau\ i)$› ‹$\pi = V\ q$› *prefixes-set* **by** *auto*
then have $\neg$*converge* $I\ \alpha'\ \beta'$
using ‹$\alpha' \in \Pi$› ‹$\neg$*converge* $M\ \alpha'\ \beta'$› ‹$\alpha' \in L\ M$› ‹$\beta' \in L\ M$›
using *dist-prop-*$\mathcal{X}$[*OF* ‹$q \in$ *reachable-states* M› ‹$\tau\ i \in \mathcal{X}\ q$›]
unfolding *preserves-divergence.simps* **by** *blast*
then show *False*
using ‹*converge* $I\ \alpha'\ \beta'$› **by** *blast*
next
case *3*

then obtain $i\ j$ **where** $i \in \{1\ ..\ m-n+1\}$ **and** $j \in \{1\ ..\ m-n+1\}$ **and** $\alpha' = \pi\ @\ (\tau\ i)$ **and** $\beta' = \pi\ @\ (\tau\ j)$ **and** $i < j$ **by** *blast*
then have $\tau\ j \in \mathcal{X}\ q$
using ‹$\bigwedge\ i\ .\ i > 0 \Longrightarrow i \leq m - n + 1 \Longrightarrow (\tau\ i) \in \mathcal{X}\ q$›
by *force*

have $(\tau\ i) = take\ i\ (\tau\ j)$
using ‹$i < j$› **unfolding** *τ-def*
by *simp*
then have $(\tau\ i) \in list.set\ (prefixes\ (\tau\ j))$
unfolding *prefixes-set*
by (*metis* (*mono-tags*) *append-take-drop-id mem-Collect-eq*)
then have $\alpha' \in \{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ (\tau\ j))\}$
unfolding ‹$\alpha' = \pi\ @\ (\tau\ i)$› ‹$\pi = V\ q$›
by *simp*
moreover have $\beta' \in \{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ (\tau\ j))\}$
unfolding ‹$\beta' = \pi\ @\ (\tau\ j)$› ‹$\pi = V\ q$› *prefixes-set* **by** *auto*
ultimately have $\neg converge\ I\ \alpha'\ \beta'$
using ‹$\neg converge\ M\ \alpha'\ \beta'$› ‹$\alpha' \in L\ M$› ‹$\beta' \in L\ M$›
using *dist-prop-*$\mathcal{X}$[*OF* ‹$q \in reachable\text{-}states\ M$› ‹$\tau\ j \in \mathcal{X}\ q$›]
unfolding *preserves-divergence.simps* **by** *blast*
then show *False*
using ‹$converge\ I\ \alpha'\ \beta'$› **by** *blast*
qed
qed

have *case-helper*: $\bigwedge\ A\ B\ P\ .\ (\bigwedge\ x\ y\ .\ P\ x\ y = P\ y\ x) \Longrightarrow$
$(\exists\ x\ y\ .\ x \in A \cup B \wedge y \in A \cup B \wedge P\ x\ y) =$
$((\exists\ x\ y\ .\ x \in A \wedge y \in A \wedge P\ x\ y)$
$\vee\ (\exists\ x\ y\ .\ x \in A \wedge y \in B \wedge P\ x\ y)$
$\vee\ (\exists\ x\ y\ .\ x \in B \wedge y \in B \wedge P\ x\ y))$
by *auto*
have $*$: $(\bigwedge x\ y.\ (x \neq y \wedge FSM.after\ I\ (FSM.initial\ I)\ x = FSM.after\ I\ (FSM.initial\ I)\ y) =$
$(y \neq x \wedge FSM.after\ I\ (FSM.initial\ I)\ y = FSM.after\ I\ (FSM.initial\ I)\ x))$
by *auto*

consider $(a)\ \exists\ \alpha\ \beta\ .\ \alpha \in \Pi \wedge \beta \in \Pi \wedge \alpha \neq \beta \wedge (after\text{-}initial\ I\ \alpha = after\text{-}initial\ I\ \beta)$ |
$(b)\ \exists\ \alpha\ \beta\ .\ \alpha \in \Pi \wedge \beta \in (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \wedge \alpha \neq \beta \wedge (after\text{-}initial\ I\ \alpha = after\text{-}initial\ I\ \beta)$ |
$(c)\ \exists\ \alpha\ \beta\ .\ \alpha \in (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \wedge \beta \in (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\} \wedge \alpha \neq \beta \wedge (after\text{-}initial\ I\ \alpha = after\text{-}initial\ I\ \beta)$
using *same-targets-in-I*
unfolding *case-helper*[*of* $\lambda\ x\ y\ .\ x \neq y \wedge (after\text{-}initial\ I\ x = after\text{-}initial\ I\ y)\ \Pi\ (\lambda i\ .\ \pi\ @\ (\tau\ i))\ `\ \{1\ ..\ m-n+1\}$, *OF* $*$]
by *blast*

```
  then show False proof cases
    case a
    then obtain α β  where α ∈ Π and β ∈ Π and α ≠ β and (after-initial I α
= after-initial I β) by blast
    then have (after-initial M α = after-initial M β)
      using same-targets-in-M by blast

    obtain q1 q2 where q1 ∈ reachable-states M and α = V q1
                  and q2 ∈ reachable-states M and β = V q2
      using ‹α ∈ Π› ‹β ∈ Π› ‹α ≠ β›
      unfolding Π by blast
    then have q1 ≠ q2
      using ‹α ≠ β› by auto

    have α ∈ L M
      using ‹Π ⊆ L M› ‹α ∈ Π› by blast
    have q1 = after-initial M α
       using ‹is-state-cover-assignment M V› ‹q1 ∈ reachable-states M› observ-
able-io-targets[OF ‹observable M› ‹α ∈ L M›]
      unfolding is-state-cover-assignment.simps ‹α = V q1›
     by (metis ‹is-state-cover-assignment M V› assms(1) is-state-cover-assignment-observable-after)

    have β ∈ L M
      using ‹Π ⊆ L M› ‹β ∈ Π› by blast
    have q2 = after-initial M β
       using ‹is-state-cover-assignment M V› ‹q2 ∈ reachable-states M› observ-
able-io-targets[OF ‹observable M› ‹β ∈ L M›]
      unfolding is-state-cover-assignment.simps ‹β = V q2›
     by (metis ‹is-state-cover-assignment M V› assms(1) is-state-cover-assignment-observable-after)

    show False
      using ‹q1 ≠ q2› ‹(after-initial M α = after-initial M β)›
      unfolding ‹q1 = after-initial M α› ‹q2 = after-initial M β›
      by simp
  next
    case b
    then obtain α β where α ∈ Π
                    and β ∈ (λi. π @ τ i) ` {1..m − n + 1}
                    and α ≠ β
                   and FSM.after I (FSM.initial I) α = FSM.after I (FSM.initial
I) β
      by blast
    then have FSM.after M (FSM.initial M) α = FSM.after M (FSM.initial M)
β
      using same-targets-in-M by blast

    obtain i where β = π@(τ i) and i ∈ {1..m − n + 1}
      using ‹β ∈ (λi. π @ τ i) ` {1..m − n + 1}› by auto
```

have $\alpha \in L\ M$ **and** $\alpha \in L\ I$
using ‹$\Pi \subseteq L\ M$› ‹$\Pi \subseteq L\ I$› ‹$\alpha \in \Pi$› **by** *blast+*
have $\beta \in L\ M$ **and** $\beta \in L\ I$
using ‹$\beta \in (\lambda i.\ \pi\ @\ \tau\ i)\ `\ \{1..m - n + 1\}$› ‹$\bigwedge\ i\ .\ i \leq m{-}n{+}1 \Longrightarrow \pi\ @\ (\tau\ i) \in L\ M \cap L\ I$›
by *auto*

let *?io* = *drop* $i\ \tau'$

have $\tau' = (\tau\ i)\ @\ ?io$
using ‹$i \in \{1..m - n + 1\}$› ‹*length* $\tau' > m{-}n{+}1$›
unfolding τ*-def*
by (*metis* (*no-types*, *lifting*) *antisym-conv append-take-drop-id atLeastAtMost-iff less-or-eq-imp-le linorder-neqE-nat order.trans take-butlast*)
then have $\beta@?io \in (L\ M - L\ I) \cup (L\ I - L\ M)$
using ‹$\beta = \pi@(\tau\ i)$› ‹$\pi\ @\ \tau' \in (L\ M - L\ I) \cup (L\ I - L\ M)$›
by *auto*
then have $\alpha@?io \in (L\ M - L\ I) \cup (L\ I - L\ M)$
using *observable-after-eq*[*OF* ‹*observable M*› ‹*FSM.after M* (*FSM.initial M*) α = *FSM.after M* (*FSM.initial M*) β› ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$›]
observable-after-eq[*OF* ‹*observable I*› ‹*FSM.after I* (*FSM.initial I*) α = *FSM.after I* (*FSM.initial I*) β› ‹$\alpha \in L\ I$› ‹$\beta \in L\ I$›]
by *blast*
then show *False*
using τ'*-min*[*OF* ‹$\alpha \in \Pi$›, *of ?io*] ‹*length* $\tau' > m - n + 1$› ‹$i \in \{1..m - n + 1\}$›
by (*metis One-nat-def add-diff-cancel-left′ atLeastAtMost-iff diff-diff-cancel diff-is-0-eq′ length-drop less-Suc-eq nat-le-linear not-add-less2*)
next
case *c*
then have $\exists\ i\ j\ .\ i \neq j \wedge i \in \{1..m - n + 1\} \wedge j \in \{1..m - n + 1\} \wedge$ (*after-initial I* $(\pi@(\tau\ i))$ = *after-initial I* $(\pi@(\tau\ j))$)
by *force*
then have $\exists\ i\ j\ .\ i < j \wedge i \in \{1..m - n + 1\} \wedge j \in \{1..m - n + 1\} \wedge$ (*after-initial I* $(\pi@(\tau\ i))$ = *after-initial I* $(\pi@(\tau\ j))$)
by (*metis linorder-neqE-nat*)
then obtain $i\ j$ **where** $i \in \{1..m - n + 1\}$
and $j \in \{1..m - n + 1\}$
and $i < j$
and *FSM.after I* (*FSM.initial I*) $(\pi@(\tau\ i))$ = *FSM.after I* (*FSM.initial I*) $(\pi@(\tau\ j))$
by *force*

have $(\pi@(\tau\ i)) \in (\lambda i.\ \pi\ @\ \tau\ i)\ `\ \{1..m - n + 1\}$ **and** $(\pi@(\tau\ j)) \in (\lambda i.\ \pi\ @\ \tau\ i)\ `\ \{1..m - n + 1\}$
using ‹$i \in \{1..m - n + 1\}$› ‹$j \in \{1..m - n + 1\}$›
by *auto*
moreover have $(\pi@(\tau\ i)) \neq (\pi@(\tau\ j))$
proof −

have $j \leq$ *length* (*butlast* τ')
using ‹$j \in \{1..m - n + 1\}$› ‹*length* $\tau' > m - n + 1$› **by** *auto*
moreover have $\bigwedge$ *xs* . $j \leq$ *length xs* $\Longrightarrow i < j \Longrightarrow$ *take j xs* $\neq$ *take i xs*
by (*metis dual-order.strict-implies-not-eq length-take min.absorb2 min-less-iff-conj*)
ultimately show *?thesis*
using ‹$i < j$› **unfolding** τ*-def*
by *fastforce*
qed
ultimately have *FSM.after M* (*FSM.initial M*) (π@(τ *i*)) = *FSM.after M* (*FSM.initial M*) (π@(τ *j*))
using ‹*FSM.after I* (*FSM.initial I*) (π@(τ *i*)) = *FSM.after I* (*FSM.initial I*) (π@(τ *j*))›
same-targets-in-M
by *blast*

have (π@(τ *i*)) $\in$ *L M* **and** (π@(τ *i*)) $\in$ *L I* **and** (π@(τ *j*)) $\in$ *L M* **and** (π@(τ *j*)) $\in$ *L I*
using ‹$\bigwedge$ *i* . $i \leq m{-}n{+}1 \Longrightarrow \pi$ @ (τ *i*) $\in$ *L M* $\cap$ *L I*› ‹$i \in \{1..m - n + 1\}$› ‹$j \in \{1..m - n + 1\}$›
by *auto*

let *?io* = *drop j* τ'
have τ' = (τ *j*) @ *?io*
using ‹$j \in \{1..m - n + 1\}$› ‹*length* $\tau' > m{-}n{+}1$›
unfolding τ*-def*
by (*metis* (*no-types*, *lifting*) *antisym-conv append-take-drop-id atLeastAtMost-iff less-or-eq-imp-le linorder-neqE-nat order.trans take-butlast*)
then have (π@(τ *j*))@*?io* $\in$ (*L M* − *L I*) $\cup$ (*L I* − *L M*)
using ‹π @ τ' $\in$ (*L M* − *L I*) $\cup$ (*L I* − *L M*)›
by (*simp add*: τ*-def*)
then have (π@(τ *i*))@*?io* $\in$ (*L M* − *L I*) $\cup$ (*L I* − *L M*)
using *observable-after-eq*[*OF* ‹*observable M*› ‹*FSM.after M* (*FSM.initial M*) (π@(τ *i*)) = *FSM.after M* (*FSM.initial M*) (π@(τ *j*))› ‹(π@(τ *i*)) $\in$ *L M*› ‹(π@(τ *j*)) $\in$ *L M*›]
observable-after-eq[*OF* ‹*observable I*› ‹*FSM.after I* (*FSM.initial I*) (π@(τ *i*)) = *FSM.after I* (*FSM.initial I*) (π@(τ *j*))› ‹(π@(τ *i*)) $\in$ *L I*› ‹(π@(τ *j*)) $\in$ *L I*›]
by *blast*
then have π@(τ *i*)@*?io* $\in$ (*L M* − *L I*) $\cup$ (*L I* − *L M*)
by *auto*
moreover have *length* τ' > *length* (τ *i* @ *drop j* τ')
using ‹*length* $\tau' > m - n + 1$› ‹$j \in \{1..m - n + 1\}$› ‹$i < j$› **unfolding** τ*-def* **by** *force*
ultimately show *False*
using τ'*-min*[*OF* ‹$\pi \in \Pi$›, *of* (τ *i*) @ *?io*]
by *simp*
qed
qed

lemma *abstract-h-condition-soundness* :
assumes *observable M*
and *observable I*
and *is-state-cover-assignment M V*
and $L\ M = L\ I$
shows *satisfies-abstract-h-condition M I V m*
using *assms*(*3*,*4*) *equivalence-preserves-divergence*[*OF assms*(*1*,*2*,*4*)]
unfolding *satisfies-abstract-h-condition-def Let-def* **by** *blast*

lemma *abstract-h-condition-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
shows *satisfies-abstract-h-condition M I V m* $\longleftrightarrow$ ($L\ M = L\ I$)
using *abstract-h-condition-soundness*[*OF assms*(*1*,*2*,*8*)]
using *abstract-h-condition-exhaustiveness*[*OF assms*]
by *blast*

18.2 Definition of the Framework

definition *h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$
(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment*) $\Rightarrow$

(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$ (('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ '*d*) $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$ (('*b*$\times$'*c*) *prefix-tree* $\times$ '*d*)) $\Rightarrow$
(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition list* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition list*) $\Rightarrow$
(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ '*d* $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$ *nat* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition list* $\Rightarrow$ (('*a*,'*b*,'*c*) *transition list* $\times$ ('*b*$\times$'*c*) *prefix-tree* $\times$ '*d*)) $\Rightarrow$

(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ '*d* $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$ ('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ '*c* $\Rightarrow$ (('*b*$\times$'*c*) *prefix-tree*) $\times$ '*d*) $\Rightarrow$
(('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ '*d*) $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$

nat ⇒
(′b×′c) *prefix-tree*

where
h-framework M
get-state-cover
handle-state-cover
sort-transitions
handle-unverified-transition
handle-unverified-io-pair
cg-initial
cg-insert
cg-lookup
cg-merge
m
= (*let*
rstates-set = *reachable-states M*;
rstates = *reachable-states-as-list M*;
rstates-io = *List.product rstates* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*));
undefined-io-pairs = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M q x y* = *None*) *rstates-io*;
V = *get-state-cover M*;
TG1 = *handle-state-cover M V cg-initial cg-insert cg-lookup*;
sc-covered-transitions = (⋃ *q* ∈ *rstates-set* . *covered-transitions M V* (*V q*));
unverified-transitions = *sort-transitions M V* (*filter* (λ*t* . *t-source t* ∈ *rstates-set* ∧ *t* ∉ *sc-covered-transitions*) (*transitions-as-list M*));
verify-transition = (λ (*X*,*T*,*G*) *t* . *handle-unverified-transition M V T G cg-insert cg-lookup cg-merge m t X*);
TG2 = *snd* (*foldl verify-transition* (*unverified-transitions*, *TG1*) *unverified-transitions*);
verify-undefined-io-pair = (λ *T* (*q*,(*x*,*y*)) . *fst* (*handle-unverified-io-pair M V T* (*snd TG2*) *cg-insert cg-lookup q x y*))
in
foldl verify-undefined-io-pair (*fst TG2*) *undefined-io-pairs*)

18.3 Required Conditions on Procedural Parameters

definition *separates-state-cover* :: ((′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ (′*a*,′*b*,′*c*) *state-cover-assignment* ⇒ ((′*a*,′*b*,′*c*) *fsm* ⇒ (′*b*×′*c*) *prefix-tree* ⇒ ′*d*) ⇒ (′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*) ⇒ (′*d* ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list list*) ⇒ ((′*b*×′*c*) *prefix-tree* × ′*d*)) ⇒
(′*a*,′*b*,′*c*) *fsm* ⇒
(′*e*,′*b*,′*c*) *fsm* ⇒
((′*a*,′*b*,′*c*) *fsm* ⇒ (′*b*×′*c*) *prefix-tree* ⇒ ′*d*) ⇒
(′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*) ⇒
(′*d* ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list list*) ⇒
bool

where
separates-state-cover f M1 M2 cg-initial cg-insert cg-lookup =
(∀ *V* .

(*V* ‘ *reachable-states M1* ⊆ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)))
∧ *finite-tree* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial ⟶
L M1 ∩ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*f M1 V cg-initial cg-insert cg-lookup*)) ⟶
(*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V cg-initial cg-insert cg-lookup*)))))

definition *handles-transition* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
nat ⇒
('*a*,'*b*,'*c*) *transition* ⇒
('*a*,'*b*,'*c*) *transition list* ⇒
(('*a*,'*b*,'*c*) *transition list* × ('*b*×'*c*) *prefix-tree* × '*d*)) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*e*,'*b*,'*c*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
bool

where
handles-transition f M1 M2 V T0 cg-insert cg-lookup cg-merge =
(∀ *T G m t X* .
(*set T* ⊆ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))))
∧ (*finite-tree T* ⟶ *finite-tree* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
size-r M1 ≤ *m* ⟶

size M2 $\leq$ *m* $\longrightarrow$
inputs M2 = *inputs M1* $\longrightarrow$
outputs M2 = *outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\longrightarrow$
V ‘ *reachable-states M1* $\subseteq$ *set T* $\longrightarrow$
t $\in$ *transitions M1* $\longrightarrow$
t-source t $\in$ *reachable-states M1* $\longrightarrow$
((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\neq$ (*V* (*t-target t*)) $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge $\longrightarrow$
L M1 $\cap$ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*)))
= *L M2* $\cap$ *set* (*fst* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*))) $\longrightarrow$
(*set T0* $\subseteq$ *set T*) $\longrightarrow$
($\forall$ γ . (*length* γ $\leq$ (*m*−*size-r M1*) $\wedge$ *list.set* γ $\subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* γ $\in$ *LS M1* (*t-target t*))
$\longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))))
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*f M1 V T G cg-insert cg-lookup cg-merge m t X*)))))

definition *handles-io-pair* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$
('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$
('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$
'*d* $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$
'*a* $\Rightarrow$ '*b* $\Rightarrow$ '*c* $\Rightarrow$
(('*b*$\times$'*c*) *prefix-tree* $\times$ '*d*)) $\Rightarrow$
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$
('*e*,'*b*,'*c*) *fsm* $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ '*d*) $\Rightarrow$
('*d* $\Rightarrow$ ('*b*$\times$'*c*) *list* $\Rightarrow$ ('*b*$\times$'*c*) *list list*) $\Rightarrow$

bool

where
handles-io-pair f M1 M2 cg-insert cg-lookup =
($\forall$ *V T G q x y* .
(*set T* $\subseteq$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ (*finite-tree T* $\longrightarrow$ *finite-tree* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ (*observable M1* $\longrightarrow$
observable M2 $\longrightarrow$
minimal M1 $\longrightarrow$

minimal M2 $\longrightarrow$
inputs M2 = *inputs M1* $\longrightarrow$
outputs M2 = *outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1* $\longrightarrow$
q $\in$ *reachable-states M1* $\longrightarrow$
x $\in$ *inputs M1* $\longrightarrow$
y $\in$ *outputs M1* $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
L M1 $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\longrightarrow$
(*L M1* $\cap$ {(*V q*)@[(*x*,*y*)]} = *L M2* $\cap$ {(*V q*)@[(*x*,*y*)]})
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))))

18.4 Completeness and Finiteness of the Scheme

lemma *unverified-transitions-handle-all-transitions* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: $\bigwedge$ *t* γ . *t* $\in$ *transitions M1* $\Longrightarrow$
t-source t $\in$ *reachable-states M1* $\Longrightarrow$
length $\gamma \leq k \Longrightarrow$
list.set $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\Longrightarrow$
butlast $\gamma \in$ *LS M1* (*t-target t*) $\Longrightarrow$
(*V* (*t-target t*) $\neq$ (*V* (*t-source t*))@[(*t-input t*, *t-output t*)]) $\Longrightarrow$
((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: $\bigwedge$ *q x y* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *x* $\in$ *inputs M1* $\Longrightarrow$ *y* $\in$ *outputs M1* $\Longrightarrow$ *h-obs M1 q x y* = *None* $\Longrightarrow$ *L M1* $\cap$ {*V q* @ [(*x*,*y*)]} = *L M2* $\cap$ {*V q* @ [(*x*,*y*)]}
and *t* $\in$ *transitions M1*
and *t-source t* $\in$ *reachable-states M1*
and *length* $\gamma \leq k$
and *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1*
and *butlast* $\gamma \in$ *LS M1* (*t-target t*)
shows (*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output*

$t)]) @ \omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (V ‘ *reachable-states M1* $\cup$ {((V (*t-source* t))@[(*t-input* t,*t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)})
proof (*cases* V (*t-target* t) $\neq$ V (*t-source* t) @ [(*t-input* t, *t-output* t)])
case *True*
then show *?thesis*
using *handles-unverified-transitions*[*OF assms*(*7*−*11*)]
by *blast*
next
case *False*
then have V (*t-source* t) @ [(*t-input* t, *t-output* t)] = V (*t-target* t)
by *simp*
have $\bigwedge \gamma$. *length* $\gamma \leq k \Longrightarrow$
list.set $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\Longrightarrow$
butlast $\gamma \in$ *LS M1* (*t-target* t) $\Longrightarrow$
L M1 $\cap$ (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)}) =
L M2 $\cap$ (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)}) $\wedge$
preserves-divergence M1 M2 (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)})
proof −
fix γ **assume** *length* $\gamma \leq k$ **and** *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* **and** *butlast* $\gamma \in$ *LS M1* (*t-target* t)
then show *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)}) =
L M2 $\cap$ (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)}) $\wedge$
preserves-divergence M1 M2 (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* γ)})
using ‹$t \in$ *transitions M1*› ‹*t-source* $t \in$ *reachable-states M1*› ‹V (*t-source* t) @ [(*t-input* t, *t-output* t)] = V (*t-target* t)›
proof (*induction* γ *arbitrary*: t)
case *Nil*
have {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* []) } = {V (*t-target* t)}
unfolding *Nil* **by** *auto*
then have $*$: (V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ $\omega' \mid \omega'. \omega' \in$ *list.set* (*prefixes* [])}) = V ‘ *reachable-states M1*
using *reachable-states-next*[*OF Nil.prems*(*5*,*4*)] **by** *blast*
show *?case*
unfolding $*$
using *assms*(*3*,*4*)
by *blast*
next
case (*Cons xy* γ)
then obtain x y **where** $xy = (x,y)$ **by** *auto*
then have $x \in$ *inputs M1* **and** $y \in$ *outputs M1*
using *Cons.prems*(*2*) **by** *auto*

have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*OF Cons.prems*(*5*,*4*)] **by** *blast*
then have *after-initial M1* (*V* (*t-target t*)) = *t-target t*
using ‹*is-state-cover-assignment M1 V*›
by (*metis assms*(*1*) *is-state-cover-assignment-observable-after*)

show *?case* **proof** (*cases* [*xy*] ∈ *LS M1* (*t-target t*))
case *False*
then have *h-obs M1* (*t-target t*) *x y* = *None*
using *Cons.prems*(*4*,*5*) ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*› **unfolding** ‹*xy* = (*x*,*y*)›
by (*meson assms*(*1*) *h-obs-language-single-transition-iff*)
then have *L M1* ∩ {*V* (*t-target t*) @ [(*x*, *y*)]} = *L M2* ∩ {*V* (*t-target t*) @ [(*x*, *y*)]}
using *handles-undefined-io-pairs*[*OF* ‹*t-target t* ∈ *reachable-states M1*› ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*›] **by** *blast*

have *V* (*t-target t*) @ [(*x*, *y*)] ∉ *L M1*
using *False* ‹*after-initial M1* (*V* (*t-target t*)) = *t-target t*›
unfolding ‹*xy* = (*x*,*y*)›
by (*metis assms*(*1*) *language-prefix observable-after-language-none*)
then have *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {*V* (*t-target t*) @ [(*x*, *y*)]})
using *assms*(*4*)
unfolding *preserves-divergence.simps*
by *blast*

have γ = []
using *False Cons.prems*(*3*)
by (*metis* (*no-types*, *lifting*) *LS-single-transition* ‹*xy* = (*x*, *y*)› *but-last.simps*(*2*) *language-next-transition-ob*)
then have *list.set* (*prefixes* (*xy*#γ)) = {[], [(*x*,*y*)]}
unfolding ‹*xy* = (*x*,*y*)›
by *force*
then have {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ ω' |ω'. ω' ∈ *list.set* (*prefixes* (*xy* # γ))} = {*V* (*t-target t*), *V* (*t-target t*) @ [(*x*, *y*)]}
unfolding *Cons* **by** *auto*
then have ∗:(*V* ‘ *reachable-states M1* ∪ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ ω' |ω'. ω' ∈ *list.set* (*prefixes* (*xy* # γ))}) = (*V* ‘ *reachable-states M1* ∪ {*V* (*t-target t*) @ [(*x*, *y*)]})
using *reachable-states-next*[*OF Cons.prems*(*5*,*4*)] **by** *blast*

show *?thesis*
unfolding ∗
using *assms*(*3*)
‹*L M1* ∩ {*V* (*t-target t*) @ [(*x*, *y*)]} = *L M2* ∩ {*V* (*t-target t*) @ [(*x*,

$y)]\}$›
‹*preserves-divergence M1 M2* (V ‘ *reachable-states M1* $\cup$ {V (*t-target* t) @ [(x, y)]})›
by *blast*
next
case *True*

then obtain t' **where** *t-source* t' = *t-target* t **and** *t-input* t' = x **and** *t-output* t' = y **and** $t' \in$ *transitions M1*
unfolding ‹$xy = (x,y)$›
by *auto*
then have *t-target* $t' \in$ *reachable-states M1* **and** *t-source* $t' \in$ *reachable-states M1*
using *reachable-states-next*[*OF* ‹*t-target* $t \in$ *reachable-states M1*›, *of* t'] ‹*t-target* $t \in$ *reachable-states M1*› **by** *auto*

have $*$: *length* $\gamma \leq k$
using *Cons.prems*(*1*) **by** *auto*
have $**$: *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1*
using *Cons.prems*(*2*) **by** *auto*
have $***$: *butlast* $\gamma \in$ *LS M1* (*t-target* t')
using *Cons.prems*(*3*)
by (*metis True* ‹$t' \in$ *FSM.transitions M1*› ‹*t-input* $t' = x$› ‹*t-output* $t' = y$› ‹*t-source* t' = *t-target* t› ‹$xy = (x, y)$› *assms*(*1*) *butlast.simps*(*1*) *butlast.simps*(*2*) *observable-language-transition-target*)

have {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* (xy # γ))} = {((V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ [xy]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)} $\cup$ {V (*t-source* t) @ [(*t-input* t, *t-output* t)]}
by (*induction* γ; *auto*)
moreover have {((V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ [xy]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)} = {(V (*t-source* t') @ [(*t-input* t', *t-output* t')]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)}
unfolding ‹*t-source* t' = *t-target* t› ‹*t-input* $t' = x$› ‹*t-output* $t' = y$› ‹$xy = (x,y)$›[*symmetric*] *Cons.prems*(*6*)[*symmetric*] **by** *simp*
ultimately have {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* (xy # γ))} = {(V (*t-source* t') @ [(*t-input* t', *t-output* t')]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)} $\cup$ {V (*t-target* t)}
unfolding *Cons* **by** *force*
then have $****$: V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t') @ [(*t-input* t', *t-output* t')]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)}
= V ‘ *reachable-states M1* $\cup$ {(V (*t-source* t) @ [(*t-input* t, *t-output* t)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* (xy # γ))}
using ‹*t-source* t' = *t-target* t› ‹*t-source* $t' \in$ *reachable-states M1*› **by** *force*

show *?thesis* **proof** (*cases* V (*t-source* t') @ [(*t-input* t', *t-output* t')] = V (*t-target* t'))

case *True*
show *?thesis*
using *Cons.IH*[*OF* ∗ ∗∗ ∗∗∗ ‹$t' \in$ *transitions M1*› ‹*t-source* $t' \in$ *reachable-states M1*› *True*]
unfolding ∗∗∗∗ **.**
next
case *False*
then show *?thesis*
using *handles-unverified-transitions*[*OF* ‹$t' \in$ *transitions M1*› ‹*t-source* $t' \in$ *reachable-states M1*› ∗ ∗∗ ∗∗∗]
unfolding ∗∗∗∗
by *presburger*
qed
qed
qed
qed
then show *?thesis*
using *assms*(*9*−*11*)
by *blast*
qed

lemma *abstract-h-condition-by-transition-and-io-pair-coverage* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: ⋀ t γ . $t \in$ *transitions M1* ⟹
t-source $t \in$ *reachable-states M1* ⟹
length $\gamma \leq k$ ⟹
list.set $\gamma \subseteq$ *inputs M1* × *outputs M1* ⟹
butlast $\gamma \in$ *LS M1* (*t-target t*) ⟹
((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: ⋀ q x y . $q \in$ *reachable-states M1* ⟹ $x \in$ *inputs M1* ⟹ $y \in$ *outputs M1* ⟹ *h-obs M1 q x y* = *None* ⟹ *L M1* ∩ {*V q* @ [(x,y)]} = *L M2* ∩ {*V q* @ [(x,y)]}
and $q \in$ *reachable-states M1*
and *length* $\gamma \leq$ *Suc k*
and *list.set* $\gamma \subseteq$ *inputs M1* × *outputs M1*
and *butlast* $\gamma \in$ *LS M1 q*
shows (*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))

$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
proof (*cases* γ)
  case *Nil*
  show *?thesis*
    using *assms*(*3*,*4*,*7*) **unfolding** *Nil* **by** *auto*
next
  case (*Cons xy* γ')
  then obtain *x y* **where** *xy* = (*x*,*y*) **using** *prod.exhaust* **by** *metis*
  then have *x* $\in$ *inputs M1* **and** *y* $\in$ *outputs M1*
    using *assms*(*9*) *Cons* **by** *auto*

  show *?thesis* **proof** (*cases* [*xy*] $\in$ *LS M1 q*)
    case *False*
    then have *h-obs M1 q x y* = *None*
      using *assms*(*7*) ‹*x* $\in$ *inputs M1*› ‹*y* $\in$ *outputs M1*› **unfolding** ‹*xy* = (*x*,*y*)›
      by (*meson assms*(*1*) *h-obs-language-single-transition-iff*)
    then have *L M1* $\cap$ {*V q* @ [(*x*,*y*)]} = *L M2* $\cap$ {*V q* @ [(*x*,*y*)]}
      using *handles-undefined-io-pairs*[*OF assms*(*7*) ‹*x* $\in$ *inputs M1*› ‹*y* $\in$ *outputs M1*›] **by** *blast*

    have *V q* @ [(*x*, *y*)] $\notin$ *L M1*
      using *observable-after-language-none*[*OF assms*(*1*), *of V q initial M1* [(*x*,*y*)]]
      using *state-cover-assignment-after*[*OF assms*(*1*,*2*,*7*)]
      by (*metis False* ‹*xy* = (*x*, *y*)›)
    then have *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ [(*x*, *y*)]})
      using *assms*(*4*)
      unfolding *preserves-divergence.simps*
      by *blast*

    have $\gamma' = []$
      using *False assms*(*10*) *language-prefix*[*of* [*xy*] γ' *M1 q*]
      unfolding *Cons*
    by (*metis* (*no-types*, *lifting*) *LS-single-transition* ‹*xy* = (*x*, *y*)› *butlast.simps*(*2*) *language-next-transition-ob*)
    then have γ = [(*x*,*y*)]
      unfolding *Cons* ‹*xy* = (*x*,*y*)› **by** *auto*
    then have $*$: (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}) = *V* ‘ *reachable-states M1* $\cup$ {*V q* @ [(*x*,*y*)]}
      using *assms*(*7*) **by** *auto*

    show *?thesis*
      unfolding $*$
        using *assms*(*3*) ‹*L M1* $\cap$ {*V q* @ [(*x*,*y*)]} = *L M2* $\cap$ {*V q* @ [(*x*,*y*)]}› ‹*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ [(*x*, *y*)]})›
      by *blast*
  next

case *True*
moreover have *butlast* $((x,y)\#\gamma') \in$ *LS M1 q*
using *assms*(*10*) **unfolding** *Cons* ‹$xy = (x,y)$› **.**
ultimately have (x,y) # (*butlast* γ') $\in$ *LS M1 q*
unfolding ‹$xy = (x,y)$› **by** (*cases* γ'; *auto*)
then obtain q' **where** *h-obs M1 q x y = Some* q' **and** *butlast* $\gamma' \in$ *LS M1* q'
using *h-obs-language-iff*[*OF assms*(*1*), *of x y butlast* γ' *q*]
by *blast*
then have $(q,x,y,q') \in$ *transitions M1*
unfolding *h-obs-Some*[*OF assms*(*1*)] **by** *blast*

have *length* $\gamma' \leq k$
using *assms*(*8*) **unfolding** *Cons* **by** *auto*
have *list.set* $\gamma' \subseteq$ *inputs M1* $\times$ *outputs M1*
using *assms*(*9*) **unfolding** *Cons* **by** *auto*

have $*$:(*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {(*V q* @ [(*x*,*y*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ')})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {(*V q* @ [(*x*,*y*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ')}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {(*V q* @ [(*x*,*y*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ')})
using *handles-unverified-transitions*[*OF* ‹$(q,x,y,q') \in$ *transitions M1*› - ‹*length* $\gamma' \leq k$› ‹*list.set* $\gamma' \subseteq$ *inputs M1* $\times$ *outputs M1*›]
assms(*7*) ‹*butlast* $\gamma' \in$ *LS M1* q'›
unfolding *fst-conv snd-conv*
by *blast*

have {*V q* @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)} = {(*V q* @ [(*x*, *y*)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ')} $\cup$ {*V q*}
unfolding *Cons* ‹$xy = (x,y)$› **by** *auto*
then have $**$: *V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)}
= *V* ‘ *reachable-states M1* $\cup$ {(*V q* @ [(*x*, *y*)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ')}
using *assms*(*7*) **by** *blast*

show *?thesis*
using $*$ **unfolding** $**$ **.**
qed
qed

lemma *abstract-h-condition-by-unverified-transition-and-io-pair-coverage* :
assumes *observable M1*
and *is-state-cover-assignment M1 V*
and *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*

and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *handles-unverified-transitions*: $\bigwedge$ *t* γ . *t* $\in$ *transitions M1* $\Longrightarrow$
t-source t $\in$ *reachable-states M1* $\Longrightarrow$
length $\gamma \leq k \Longrightarrow$
list.set $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\Longrightarrow$
butlast $\gamma \in$ *LS M1* (*t-target t*) $\Longrightarrow$
(*V* (*t-target t*) $\neq$ (*V* (*t-source t*))@[(*t-input t*, *t-output t*)]) $\Longrightarrow$
((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
and *handles-undefined-io-pairs*: $\bigwedge$ *q x y* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *x* $\in$ *inputs M1* $\Longrightarrow$ *y* $\in$ *outputs M1* $\Longrightarrow$ *h-obs M1 q x y* = *None* $\Longrightarrow$ *L M1* $\cap$ {*V q* @ [(*x*,*y*)]} = *L M2* $\cap$ {*V q* @ [(*x*,*y*)]}
and *q* $\in$ *reachable-states M1*
and *length* $\gamma \leq$ *Suc k*
and *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1*
and *butlast* $\gamma \in$ *LS M1 q*
shows (*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*V q* @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
using *unverified-transitions-handle-all-transitions*[*OF assms*(*1*−*6*), *of k*]
using *abstract-h-condition-by-transition-and-io-pair-coverage*[*OF assms*(*1*−*4*) - *assms*(*6*−*10*)]
by *presburger*

lemma *h-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'e*,*'b*,*'c*) *fsm*
fixes *cg-insert* :: (*'d* $\Rightarrow$ (*'b*$\times$*'c*) *list* $\Rightarrow$ *'d*)
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1* (*get-state-cover M1*)
and $\bigwedge$ *xs* . *List.set xs* = *List.set* (*sort-transitions M1* (*get-state-cover M1*) *xs*)

and *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*
and *separates-state-cover handle-state-cover M1 M2 cg-initial cg-insert cg-lookup*
and *handles-transition handle-unverified-transition M1 M2* (*get-state-cover M1*) (*fst* (*handle-state-cover M1* (*get-state-cover M1*) *cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup cg-merge*
and *handles-io-pair handle-unverified-io-pair M1 M2 cg-insert cg-lookup*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*))
= (*L M2* $\cap$ *set* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*)))
(**is** (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*)))
and *finite-tree* (*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m*)
proof
show (*L M1* = *L M2*) $\Longrightarrow$ ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))
by *blast*

define *rstates* **where** *rstates*: *rstates* = *reachable-states-as-list M1*
define *rstates-io* **where** *rstates-io*: *rstates-io* = *List.product rstates* (*List.product* (*inputs-as-list M1*) (*outputs-as-list M1*))
define *undefined-io-pairs* **where** *undefined-io-pairs*: *undefined-io-pairs* = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M1 q x y* = *None*) *rstates-io*
define *V* **where** *V*: *V* = *get-state-cover M1*
define *n* **where** *n*: *n* = *size-r M1*
define *TG1* **where** *TG1*: *TG1* = *handle-state-cover M1 V cg-initial cg-insert cg-lookup*

define *sc-covered-transitions* **where** *sc-covered-transitions*: *sc-covered-transitions* = ($\bigcup$ *q* $\in$ *reachable-states M1* . *covered-transitions M1 V* (*V q*))
define *unverified-transitions* **where** *unverified-transitions*: *unverified-transitions* = *sort-transitions M1 V* (*filter* (λ*t* . *t-source t* $\in$ *reachable-states M1* $\wedge$ *t* $\notin$ *sc-covered-transitions*) (*transitions-as-list M1*))
define *verify-transition* **where** *verify-transition*: *verify-transition* = (λ (*X*,*T*,*G*) *t* . *handle-unverified-transition M1 V T G cg-insert cg-lookup cg-merge m t X*)
define *TG2* **where** *TG2*: *TG2* = *snd* (*foldl verify-transition* (*unverified-transitions*, *TG1*) *unverified-transitions*)
define *verify-undefined-io-pair* **where** *verify-undefined-io-pair*: *verify-undefined-io-pair* = (λ *T* (*q*,(*x*,*y*)) . *fst* (*handle-unverified-io-pair M1 V T* (*snd TG2*) *cg-insert cg-lookup q x y*))
define *T3* **where** *T3*: *T3* = *foldl verify-undefined-io-pair* (*fst TG2*) *undefined-io-pairs*

have *?TS* = *T3*

unfolding *rstates rstates-io undefined-io-pairs V TG1 sc-covered-transitions unverified-transitions verify-transition TG2 verify-undefined-io-pair T3*
unfolding *h-framework-def Let-def*
by *force*
then have $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow L\ M1 \cap set\ T3 = L\ M2 \cap set\ T3$
by *simp*

have *is-state-cover-assignment M1 V*
unfolding *V* **using** *assms*(*9*) **.**

define *T1* **where** *T1*: *T1* = *fst TG1*
moreover define *G1* **where** *G1*: *G1* = *snd TG1*
ultimately have *TG1* = (*T1*,*G1*)
by *auto*

have *T1-state-cover*: *V* ‘ *reachable-states M1* ⊆ *set T1*
and *T1-finite*: *finite-tree T1*
using ‹*separates-state-cover handle-state-cover M1 M2 cg-initial cg-insert cg-lookup*›
unfolding *T1 TG1 separates-state-cover-def*
by *blast+*

have *T1-V-div*: $(L\ M1 \cap set\ T1 = (L\ M2 \cap set\ T1)) \Longrightarrow$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *G1-invar*: $(L\ M1 \cap set\ T1 = (L\ M2 \cap set\ T1)) \Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G1*
using ‹*separates-state-cover handle-state-cover M1 M2 cg-initial cg-insert cg-lookup*›
unfolding *T1 G1 TG1 separates-state-cover-def*
using *assms*(*1*−*4*,*7*,*8*) ‹*is-state-cover-assignment M1 V*› *assms*(*12*,*11*)
by *blast+*

have *sc-covered-transitions-alt-def*: *sc-covered-transitions* = {*t* . *t* ∈ *transitions M1* ∧ *t-source t* ∈ *reachable-states M1* ∧ (*V* (*t-target t*) = (*V* (*t-source t*))@[(*t-input t*, *t-output t*)])}
(**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix *t* **assume** *t* ∈ *?A*
then obtain *q* **where** *t* ∈ *covered-transitions M1 V* (*V q*) **and** *q* ∈ *reachable-states M1*
unfolding *sc-covered-transitions*

by *blast*
then have $V\ q \in L\ M1$ **and** *after-initial M1* $(V\ q) = q$
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast+*

then obtain p **where** *path M1* (*initial M1*) p **and** *p-io* $p = V\ q$
by *auto*
then have $*$: *the-elem* (*paths-for-io M1* (*initial M1*) $(V\ q)$) $= p$
using *observable-paths-for-io*[*OF assms*(*1*) ‹$V\ q \in L\ M1$›]
unfolding *paths-for-io-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *mem-Collect-eq observable-path-unique singletonI the-elem-eq*)

have $t \in$ *list.set* p **and** V (*t-source* t) @ [(*t-input* t, *t-output* t)] $= V$ (*t-target* t)
using ‹$t \in$ *covered-transitions M1* V $(V\ q)$›
unfolding *covered-transitions-def Let-def* $*$
by *auto*

have $t \in$ *transitions M1*
using ‹$t \in$ *list.set* p› ‹*path M1* (*initial M1*) p›
by (*meson path-transitions subsetD*)
moreover have *t-source* $t \in$ *reachable-states M1*
using *reachable-states-path*[*OF reachable-states-initial* ‹*path M1* (*initial M1*) p› ‹$t \in$ *list.set* p›] **.**
ultimately show $t \in$ *?B*
using ‹V (*t-source* t) @ [(*t-input* t, *t-output* t)] $= V$ (*t-target* t)›
by *auto*
qed

show *?B* $\subseteq$ *?A*
proof
fix t **assume** $t \in$ *?B*
then have $t \in$ *transitions M1*
t-source $t \in$ *reachable-states M1*
$(V$ (*t-source* t))@[(*t-input* t, *t-output* t)] $= V$ (*t-target* t)
by *auto*
then have *t-target* $t \in$ *reachable-states M1*
using *reachable-states-next*[*of t-source t M1 t*]
by *blast*
then have V (*t-target* t) $\in L\ M1$ **and** *after-initial M1* $(V$ (*t-target* t)) $=$ (*t-target* t)
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast+*
then obtain p **where** *path M1* (*initial M1*) p **and** *p-io* $p = V$ (*t-target* t)
by *auto*
then have $*$: *the-elem* (*paths-for-io M1* (*initial M1*) $(V$ (*t-target* t))) $= p$

using *observable-paths-for-io*[*OF assms*(*1*) ‹*V* (*t-target t*) ∈ *L M1*›]
unfolding *paths-for-io-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *mem-Collect-eq observable-path-unique singletonI the-elem-eq*)

have *V* (*t-source t*) ∈ *L M1* **and** *after-initial M1* (*V* (*t-source t*)) = (*t-source t*)
using ‹*t-source t* ∈ *reachable-states M1*›
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast*+
then obtain *p′* **where** *path M1* (*initial M1*) *p′* **and** *p-io p′* = *V* (*t-source t*)
by *auto*

have *path M1* (*initial M1*) (*p′*@[*t*])
using *after-path*[*OF assms*(*1*) ‹*path M1* (*initial M1*) *p′*›] ‹*path M1* (*initial M1*) *p′*› ‹*t*∈*transitions M1*›
unfolding ‹*p-io p′* = *V* (*t-source t*)›
unfolding ‹*after-initial M1* (*V* (*t-source t*)) = (*t-source t*)›
by (*metis path-append single-transition-path*)
moreover have *p-io* (*p′*@[*t*]) = *p-io p*
using ‹*p-io p′* = *V* (*t-source t*)›
unfolding ‹*p-io p* = *V* (*t-target t*)› ‹(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)›[*symmetric*]
by *auto*
ultimately have *p′*@[*t*] = *p*
using *observable-path-unique*[*OF assms*(*1*) - ‹*path M1* (*initial M1*) *p*›]
by *force*
then have *t* ∈ *list.set p*
by *auto*
then have *t* ∈ *covered-transitions M1 V* (*V* (*t-target t*))
using ‹(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)›
unfolding *covered-transitions-def Let-def* ∗
by *auto*
then show *t* ∈ *?A*
using ‹*t-target t* ∈ *reachable-states M1*›
unfolding *sc-covered-transitions*
by *blast*
qed
qed

have *T1-covered-transitions-conv*: ⋀ *t* . (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*)) ⟹ *t* ∈ *sc-covered-transitions* ⟹ *converge M2* (*V* (*t-target t*)) ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])
proof −
fix *t* **assume** (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*))
t ∈ *sc-covered-transitions*

then have $t \in$ *transitions M1*
t-source t $\in$ *reachable-states M1*
$(V$ (*t-source t*))@[(*t-input t*, *t-output t*)] = V (*t-target t*)
unfolding *sc-covered-transitions-alt-def*
by *auto*
then have *t-target t* $\in$ *reachable-states M1*
using *reachable-states-next*[*of t-source t M1 t*]
by *blast*
then have V (*t-target t*) $\in$ *L M1*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast*
moreover have V (*t-target t*) $\in$ *set T1*
using *T1-state-cover* ‹*t-target t* $\in$ *reachable-states M1*›
by *blast*
ultimately have V (*t-target t*) $\in$ *L M2*
using ‹(*L M1* $\cap$ *set T1* = (*L M2* $\cap$ *set T1*))›
by *blast*
then show *converge M2* (V (*t-target t*)) ((V (*t-source t*))@[(*t-input t*, *t-output t*)])
unfolding ‹(V (*t-source t*))@[(*t-input t*, *t-output t*)] = V (*t-target t*)›
by *auto*
qed

have *unverified-transitions-alt-def* : *list.set unverified-transitions* = {*t* . *t* $\in$ *transitions M1* $\wedge$ *t-source t* $\in$ *reachable-states M1* $\wedge$ (V (*t-target t*) $\neq$ (V (*t-source t*))@[(*t-input t*, *t-output t*)])}
unfolding *unverified-transitions sc-covered-transitions-alt-def V*
unfolding *assms*(*10*)[*symmetric*]
using *transitions-as-list-set*[*of M1*]
by *auto*

have *cg-insert-invar* : $\bigwedge$ *G* γ . $\gamma \in$ *L M1* $\Longrightarrow$ $\gamma \in$ *L M2* $\Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G* $\Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-insert G* γ)
using *assms*(*12*)
unfolding *convergence-graph-insert-invar-def*
by *blast*

have *cg-merge-invar* : $\bigwedge$ *G* γ γ'. *convergence-graph-lookup-invar M1 M2 cg-lookup G* $\Longrightarrow$ *converge M1* γ γ' $\Longrightarrow$ *converge M2* γ γ' $\Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-merge G* γ γ')
using *assms*(*13*)
unfolding *convergence-graph-merge-invar-def*
by *blast*

define *T2* **where** *T2*: *T2* = *fst TG2*
define *G2* **where** *G2*: *G2* = *snd TG2*

have *handles-transition handle-unverified-transition M1 M2 V T1 cg-insert cg-lookup cg-merge*
using *assms*(*15*)
unfolding *T1 TG1 V* **.**
then have *verify-transition-retains-testsuite*: $\bigwedge$ *t T G X* . *set T* $\subseteq$ *set* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))
and *verify-transition-retains-finiteness*: $\bigwedge$ *t T G X* . *finite-tree T* $\Longrightarrow$ *finite-tree* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))
unfolding *verify-transition case-prod-conv handles-transition-def*
by *presburger+*

define *handles-unverified-transition*
where *handles-unverified-transition*: *handles-unverified-transition* = (λ*t* .
($\forall$ γ . (*length* γ $\leq$ (*m*−*size-r M1*) $\wedge$ *list.set* γ $\subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* γ $\in$ *LS M1* (*t-target t*))
$\longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))))

have *verify-transition-cover-prop*: $\bigwedge$ *t T G X* . (*L M1* $\cap$ (*set* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))) = *L M2* $\cap$ (*set* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))))
$\Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G*
$\Longrightarrow$ *t* $\in$ *transitions M1*
$\Longrightarrow$ *t-source t* $\in$ *reachable-states M1*
$\Longrightarrow$ *set T1* $\subseteq$ *set T*
$\Longrightarrow$ ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\neq$ (*V* (*t-target t*))
$\Longrightarrow$ *handles-unverified-transition t* $\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))
proof −
fix *t T G X*

assume *a1*: (*L M1* ∩ (*set* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))) = *L M2* ∩ (*set* (*fst* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))))
assume *a2*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a3*: *t* ∈ *transitions M1*
assume *a4*: *t-source t* ∈ *reachable-states M1*
assume *a5*: *set T1* ⊆ *set T*
assume *a6*: ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*))

obtain *X′ T′ G′* **where** *TG′*: (*X′*,*T′*,*G′*) = *handle-unverified-transition M1 V T G cg-insert cg-lookup cg-merge m t X*
using *prod.exhaust* **by** *metis*
have *T′*: *T′* = *fst* (*snd* (*handle-unverified-transition M1 V T G cg-insert cg-lookup cg-merge m t X*))
and *G′*: *G′* = *snd* (*snd* (*handle-unverified-transition M1 V T G cg-insert cg-lookup cg-merge m t X*))
unfolding *TG′*[*symmetric*] **by** *auto*

have *verify-transition* (*X*,*T*,*G*) *t* = (*X′*,*T′*,*G′*)
using *TG′*[*symmetric*]
unfolding *verify-transition G′ Let-def case-prod-conv*
by *force*
then have *set T* ⊆ *set T′*
using *verify-transition-retains-testsuite*[*of T X G t*] **unfolding** *T′*
by *auto*
then have *set T1* ⊆ *set T′*
using *a5* **by** *blast*
then have (*L M1* ∩ (*set T1*) = *L M2* ∩ (*set T1*))
using *a1* **unfolding** ‹*verify-transition* (*X*,*T*,*G*) *t* = (*X′*,*T′*,*G′*)› *fst-conv snd-conv*
by *blast*
then have ∗: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
using *T1-V-div*
by *auto*

have *L M1* ∩ *set T′* = *L M2* ∩ *set T′*
using *a1* ‹*set T* ⊆ *set T′*› **unfolding** *T′* ‹*verify-transition* (*X*,*T*,*G*) *t* = (*X′*,*T′*,*G′*)› *fst-conv snd-conv*
by *blast*

have ∗∗: *V* ‘ *reachable-states M1* ⊆ *set T*
using *a5 T1-state-cover* **by** *blast*

show *handles-unverified-transition t* ∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*verify-transition* (*X*,*T*,*G*) *t*)))
unfolding ‹*verify-transition* (*X*,*T*,*G*) *t* = (*X′*,*T′*,*G′*)› *snd-conv*
unfolding *G′*
using ‹*handles-transition handle-unverified-transition M1 M2 V T1 cg-insert*

cg-lookup cg-merge›
unfolding *handles-transition-def*

using *assms*(*1*−*8*) ‹*is-state-cover-assignment M1 V*› ∗ ∗∗ *a3 a4 a2 a6* ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*› ‹*convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*› ‹*L M1* ∩ *set T′* = *L M2* ∩ *set T′*› *a5*
unfolding *T′*
unfolding *handles-unverified-transition*
by *blast*
qed

have *verify-transition-foldl-invar-1*: ⋀ *X ts* . *list.set ts* ⊆ *list.set unverified-transitions* ⟹
set T1 ⊆ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ∧ *finite-tree* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
proof −
fix *X ts* **assume** *list.set ts* ⊆ *list.set unverified-transitions*
then show *set T1* ⊆ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ∧ *finite-tree* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case*
using *T1-finite* **by** *auto*
next
case (*snoc t ts*)
then have *t* ∈ *transitions M1* **and** *t-source t* ∈ *reachable-states M1*
unfolding *unverified-transitions-alt-def*
by *force*+

have *p1*: *list.set ts* ⊆ *list.set unverified-transitions*
using *snoc.prems*(*1*) **by** *auto*

have *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ⊆ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
using *verify-transition-retains-testsuite*
unfolding *foldl-append*
unfolding *foldl.simps*
by (*metis prod.collapse*)

have ∗∗: *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
and ∗∗∗: *finite-tree* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *snoc.IH*[*OF p1*]
by *auto*

have *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
using ∗∗ *verify-transition-retains-testsuite* ‹*set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ⊆ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))›

by *auto*
moreover have *finite-tree* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
using *verify-transition-retains-finiteness*[*OF* ∗∗∗, *of fst* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*) *snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))]
by *auto*
ultimately show *?case*
by *simp*
qed
qed
then have *T2-invar-1*: *set T1* ⊆ *set T2*
and *T2-finite* : *finite-tree T2*
unfolding *TG2 G2 T2* ‹*TG1* = (*T1*,*G1*)›
by *auto*

have *verify-transition-foldl-invar-2*: ⋀ *X ts* . *list.set ts* ⊆ *list.set unverified-transitions* ⟹
L M1 ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ⟹
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
proof −
fix *X ts* **assume** *list.set ts* ⊆ *list.set unverified-transitions*
and *L M1* ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
then show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case*
using *G1-invar* **by** *auto*
next
case (*snoc t ts*)
then have *t* ∈ *transitions M1* **and** *t-source t* ∈ *reachable-states M1*
unfolding *unverified-transitions-alt-def*
by *force*+

have *p1*: *list.set ts* ⊆ *list.set unverified-transitions*
using *snoc.prems*(*1*) **by** *auto*

have *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) ⊆ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
using *verify-transition-retains-testsuite* **unfolding** *foldl-append foldl.simps*
by (*metis fst-conv prod-eq-iff snd-conv*)
then have *p2*: *L M1* ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* ∩ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *snoc.prems*(*2*)
by *blast*

have $*$:*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *snoc.IH*[*OF p1 p2*]
by *auto*

have $**$: *Prefix-Tree.set T1* $\subseteq$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *verify-transition-foldl-invar-1*[*OF p1*] **by** *blast*

have $***$: ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\neq$ (*V* (*t-target t*))
using *snoc.prems*(*1*) **unfolding** *unverified-transitions-alt-def* **by** *force*

have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*verify-transition* ((*fst* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)), *fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)), *snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) *t*)))
using *verify-transition-cover-prop*[*OF* - $*$ ‹*t* $\in$ *transitions M1*› ‹*t-source t* $\in$ *reachable-states M1*› $**$ $***$, *of* (*fst* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))] *snoc.prems*(*2*)
unfolding *prod.collapse*
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
by *auto*
moreover have *Prefix-Tree.set T1* $\subseteq$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t*]))))
using $**$ *verify-transition-retains-testsuite*
using *snoc.prems*(*1*) *verify-transition-foldl-invar-1* **by** *blast*
ultimately show *?case*
by *simp*
qed
qed
then have *T2-invar-2*: *L M1* $\cap$ *set T2* = *L M2* $\cap$ *set T2* $\Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
unfolding *TG2 G2 T2* ‹*TG1* = (*T1*,*G1*)› **by** *auto*

have *T2-cover*: $\bigwedge$ *t* . *L M1* $\cap$ *set T2* = *L M2* $\cap$ *set T2* $\Longrightarrow$ *t* $\in$ *list.set unverified-transitions* $\Longrightarrow$ *handles-unverified-transition t*
proof −
have $\bigwedge$ *X t ts* . *t* $\in$ *list.set ts* $\Longrightarrow$ *list.set ts* $\subseteq$ *list.set unverified-transitions* $\Longrightarrow$ *L M1* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) $\Longrightarrow$ *handles-unverified-transition t*
proof −
fix *X t ts*
assume *t* $\in$ *list.set ts* **and** *list.set ts* $\subseteq$ *list.set unverified-transitions* **and** *L M1* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))

then show *handles-unverified-transition t*

proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc t′ ts*)

then have $t \in$ *transitions M1* **and** *t-source t* $\in$ *reachable-states M1*
unfolding *unverified-transitions-alt-def*
by *blast+*

have $t' \in$ *transitions M1* **and** *t-source t′* $\in$ *reachable-states M1*
using *snoc.prems*(*2*)
unfolding *unverified-transitions-alt-def*
by *auto*

have *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) $\subseteq$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t′*]))))
using *verify-transition-retains-testsuite* **unfolding** *foldl-append foldl.simps*
by (*metis fst-conv prod-eq-iff snd-conv*)
then have *L M1* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* $\cap$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *snoc.prems*(*3*)
by *blast*

have $*$: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*verify-transition* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*) *t′*))) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*verify-transition* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*) *t′*)))
using *snoc.prems*(*3*) **by** *auto*

have $**$: *V* (*t-source t′*) @ [(*t-input t′*, *t-output t′*)] $\neq$ *V* (*t-target t′*)
using *snoc.prems*(*2*) **unfolding** *unverified-transitions-alt-def* **by** *force*

have *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using ‹*set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) $\subseteq$ *set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) (*ts*@[*t′*]))))› *snoc.prems*(*3*)
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) $\wedge$ *Prefix-Tree.set T1* $\subseteq$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))
using *snoc.prems*(*2*) *verify-transition-foldl-invar-1*[*of ts*] *verify-transition-foldl-invar-2*[*of ts*]
by *auto*
then have *covers-t′*: *handles-unverified-transition t′*
by (*metis* $*$ $**$ ‹*t′* $\in$ *FSM.transitions M1*› ‹*t-source t′* $\in$ *reachable-states M1*› *prod.collapse verify-transition-cover-prop*)

show *?case* **proof** (*cases t* = *t′*)
case *True*

then show *?thesis*
using *covers-t'* **by** *auto*
next
case *False*
then have $t \in$ *list.set ts*
using *snoc.prems*(*1*) **by** *auto*

show *handles-unverified-transition t*
using *snoc.IH*[*OF* ‹$t \in$ *list.set ts*›] *snoc.prems*(*2*) ‹*L M1* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*))) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*foldl verify-transition* (*X*, *T1*, *G1*) *ts*)))›
by *auto*
qed
qed
qed

then show $\bigwedge$ *t* . *L M1* $\cap$ *set T2* = *L M2* $\cap$ *set T2* $\Longrightarrow$ *t* $\in$ *list.set unverified-transitions* $\Longrightarrow$ *handles-unverified-transition t*
unfolding *TG2 T2 G2* ‹*TG1* = (*T1*,*G1*)›
by *simp*
qed

have *verify-undefined-io-pair-retains-testsuite*: $\bigwedge$ *qxy T* . *set T* $\subseteq$ *set* (*verify-undefined-io-pair T qxy*)
proof −
fix *qxy* :: ('*a* × '*b* × '*c*)
fix *T*
obtain *q x y* **where** *qxy* = (*q*,*x*,*y*)
using *prod.exhaust* **by** *metis*
show ‹*set T* $\subseteq$ *set* (*verify-undefined-io-pair T qxy*)›
unfolding ‹*qxy* = (*q*,*x*,*y*)›
using ‹*handles-io-pair handle-unverified-io-pair M1 M2 cg-insert cg-lookup*›
unfolding *handles-io-pair-def verify-undefined-io-pair case-prod-conv*
by *blast*
qed
have *verify-undefined-io-pair-folding-retains-testsuite*: $\bigwedge$ *qxys T* . *set T* $\subseteq$ *set* (*foldl verify-undefined-io-pair T qxys*)
proof −
fix *qxys T*
show *set T* $\subseteq$ *set* (*foldl verify-undefined-io-pair T qxys*)
using *verify-undefined-io-pair-retains-testsuite*
by (*induction qxys rule*: *rev-induct*; *force*)
qed

have *verify-undefined-io-pair-retains-finiteness*: $\bigwedge$ *qxy T* . *finite-tree T* $\Longrightarrow$ *finite-tree* (*verify-undefined-io-pair T qxy*)

proof −
fix *qxy* :: ('*a* × '*b* × '*c*)
fix *T* :: ('*b*×'*c*) *prefix-tree*
assume *finite-tree T*
obtain *q x y* **where** *qxy* = (*q*,*x*,*y*)
using *prod.exhaust* **by** *metis*
show ‹*finite-tree* (*verify-undefined-io-pair T qxy*)›
unfolding ‹*qxy* = (*q*,*x*,*y*)›
using ‹*handles-io-pair handle-unverified-io-pair M1 M2 cg-insert cg-lookup*› ‹*finite-tree T*›
unfolding *handles-io-pair-def verify-undefined-io-pair case-prod-conv*
by *blast*
qed
have *verify-undefined-io-pair-folding-retains-finiteness*: ⋀ *qxys T* . *finite-tree T* ⟹ *finite-tree* (*foldl verify-undefined-io-pair T qxys*)
proof −
fix *qxys*
fix *T* :: ('*b*×'*c*) *prefix-tree*
assume *finite-tree T*
then show *finite-tree* (*foldl verify-undefined-io-pair T qxys*)
using *verify-undefined-io-pair-retains-finiteness*
by (*induction qxys rule*: *rev-induct*; *force*)
qed

show *finite-tree ?TS*
using *T2 T2-finite T3* ‹*h-framework M1 get-state-cover handle-state-cover sort-transitions handle-unverified-transition handle-unverified-io-pair cg-initial cg-insert cg-lookup cg-merge m = T3*› *verify-undefined-io-pair-folding-retains-finiteness*
by *auto*

assume ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))

have *set T2* ⊆ *set T3*
unfolding *T3 T2*
proof (*induction undefined-io-pairs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case*
using *verify-undefined-io-pair-retains-testsuite*[*of* (*foldl verify-undefined-io-pair* (*fst TG2*) *xs*) *x*]
by *force*
qed
then have *passes-T2*: *L M1* ∩ *set T2* = *L M2* ∩ *set T2*
using ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ (*L M1* ∩ *set T3* = *L M2* ∩ *set T3*)› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›

by *blast*

have *set T1* ⊆ *set T3*
and *G2-invar*: ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
using *T2-invar-1 T2-invar-2*[*OF passes-T2*] ‹*set T2* ⊆ *set T3*›
by *auto*
then have *passes-T1*: *L M1* ∩ *set T1* = *L M2* ∩ *set T1*
using ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *L M1* ∩ *set T3* = *L M2* ∩ *set T3*› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›
by *blast*

have *T3-preserves-divergence* : *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
using *T1-V-div*[*OF passes-T1*] **.**

have *T3-state-cover* : *V* ‘ *reachable-states M1* ⊆ *set T3*
using *T1-state-cover* ‹*set T1* ⊆ *set T3*›
by *blast*
then have *T3-passes-state-cover* : *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
using *T1-state-cover passes-T1* **by** *blast*

have *rstates-io-set* : *list.set rstates-io* = {(*q*,(*x*,*y*)) . *q* ∈ *reachable-states M1* ∧ *x* ∈ *inputs M1* ∧ *y* ∈ *outputs M1*}
unfolding *rstates-io rstates*
using *reachable-states-as-list-set*[*of M1*] *inputs-as-list-set*[*of M1*] *outputs-as-list-set*[*of M1*]
by *force*
then have *undefined-io-pairs-set* : *list.set undefined-io-pairs* = {(*q*,(*x*,*y*)) . *q* ∈ *reachable-states M1* ∧ *x* ∈ *inputs M1* ∧ *y* ∈ *outputs M1* ∧ *h-obs M1 q x y* = *None*}
unfolding *undefined-io-pairs*
by *auto*

have *verify-undefined-io-pair-prop* : ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ (⋀ *q x y T* . *L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) ⟹
q ∈ *reachable-states M1* ⟹ *x* ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹
V ‘ *reachable-states M1* ⊆ *set T* ⟹
(*L M1* ∩ {(*V q*)@[(*x*,*y*)]} = *L M2* ∩ {(*V q*)@[(*x*,*y*)]}))
proof −
fix *q x y T*
assume *L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set*

(*verify-undefined-io-pair T* (*q*,(*x*,*y*)))
and *q* ∈ *reachable-states M1* **and** *x* ∈ *inputs M1* **and** *y* ∈ *outputs M1*
and *V* ‘ *reachable-states M1* ⊆ *set T*
and ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))

have *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
using *T3-state-cover* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *L M1* ∩ *Prefix-Tree.set T3* = *L M2* ∩ *Prefix-Tree.set T3*› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›
by *blast*

have *L M1* ∩ *set* (*fst* (*handle-unverified-io-pair M1 V T G2 cg-insert cg-lookup q x y*)) = *L M2* ∩ *set* (*fst* (*handle-unverified-io-pair M1 V T G2 cg-insert cg-lookup q x y*))
using ‹*L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*)))›
unfolding *verify-undefined-io-pair case-prod-conv combine-set G2*
by *blast*

show (*L M1* ∩ {(*V q*)@[(*x*,*y*)]} = *L M2* ∩ {(*V q*)@[(*x*,*y*)]})
using *assms*(*16*)
unfolding *handles-io-pair-def*
using *assms*(*1*−*4*,*7*,*8*) ‹*is-state-cover-assignment M1 V*› ‹*L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*›
‹*q* ∈ *reachable-states M1*› ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*›
G2-invar[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›] ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*›
‹*L M1* ∩ *set* (*fst* (*handle-unverified-io-pair M1 V T G2 cg-insert cg-lookup q x y*)) = *L M2* ∩ *set* (*fst* (*handle-unverified-io-pair M1 V T G2 cg-insert cg-lookup q x y*))›
by *blast*
qed

have *T3-covers-undefined-io-pairs* : (⋀ *q x y* . *q* ∈ *reachable-states M1* ⟹ *x* ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹ *h-obs M1 q x y* = *None* ⟹
(*L M1* ∩ {(*V q*)@[(*x*,*y*)]} = *L M2* ∩ {(*V q*)@[(*x*,*y*)]}))
proof −
fix *q x y* **assume** *q* ∈ *reachable-states M1* **and** *x* ∈ *inputs M1* **and** *y* ∈ *outputs M1* **and** *h-obs M1 q x y* = *None*

have ⋀ *q x y qxys T* . *L M1* ∩ *set* (*foldl verify-undefined-io-pair T qxys*) = *L M2* ∩ *set* (*foldl verify-undefined-io-pair T qxys*) ⟹ (*V* ‘ *reachable-states M1*) ⊆ *set T* ⟹ (*q*,(*x*,*y*)) ∈ *list.set qxys* ⟹ *list.set qxys* ⊆ *list.set undefined-io-pairs* ⟹
(*L M1* ∩ {(*V q*)@[(*x*,*y*)]} = *L M2* ∩ {(*V q*)@[(*x*,*y*)]})
(**is** ⋀ *q x y qxys T* . *?P1 qxys T* ⟹ (*V* ‘ *reachable-states M1*) ⊆ *set T* ⟹ (*q*,(*x*,*y*)) ∈ *list.set qxys* ⟹ *list.set qxys* ⊆ *list.set undefined-io-pairs* ⟹ *?P2 q x y qxys T*)

proof −
fix *q x y qxys T*
assume *?P1 qxys T* **and** (*q*,(*x*,*y*)) ∈ *list.set qxys* **and** *list.set qxys* ⊆ *list.set undefined-io-pairs* **and** (*V* ‘ *reachable-states M1*) ⊆ *set T*
then show *?P2 q x y qxys T*
proof (*induction qxys rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a qxys*)

have *set* (*foldl verify-undefined-io-pair T qxys*) ⊆ *set* (*foldl verify-undefined-io-pair T* (*qxys*@[*a*]))
using *verify-undefined-io-pair-retains-testsuite*
by *auto*
then have ∗:*L M1* ∩ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*) = *L M2* ∩ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*)
using *snoc.prems*(*1*)
by *blast*

have ∗∗: *V* ‘ *reachable-states M1* ⊆ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*)
using *snoc.prems*(*4*) *verify-undefined-io-pair-folding-retains-testsuite*
by *blast*

show *?case* **proof** (*cases a* = (*q*,(*x*,*y*)))
case *True*
then have ∗∗∗: *q* ∈ *reachable-states M1*
using *snoc.prems*(*3*)
unfolding *undefined-io-pairs-set*
by *auto*

have *x* ∈ *inputs M1* **and** *y* ∈ *outputs M1*
using *snoc.prems*(*2*,*3*) **unfolding** *undefined-io-pairs-set* **by** *auto*

have ∗∗∗∗: *L M1* ∩ *set* (*verify-undefined-io-pair* (*foldl verify-undefined-io-pair T qxys*) (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair* (*foldl verify-undefined-io-pair T qxys*) (*q*,(*x*,*y*)))
using *snoc.prems*(*1*) **unfolding** *True* **by** *auto*

show *?thesis*
using *verify-undefined-io-pair-prop*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))› ∗∗∗∗ ∗∗∗ ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*› ∗∗]
unfolding *True*
by *auto*
next
case *False*
then have (*q*, *x*, *y*) ∈ *list.set qxys* **and** *list.set qxys* ⊆ *list.set undefined-io-pairs*

using *snoc.prems*(*2*,*3*) **by** *auto*
then show *?thesis*
using *snoc.IH*[*OF* ∗ - - *snoc.prems*(*4*)]
using ‹*set* (*foldl verify-undefined-io-pair T qxys*) ⊆ *set* (*foldl verify-undefined-io-pair T* (*qxys*@[*a*]))›
by *blast*
qed
qed
qed
moreover have *L M1* ∩ *set* (*foldl verify-undefined-io-pair T2 undefined-io-pairs*) = *L M2* ∩ *set* (*foldl verify-undefined-io-pair T2 undefined-io-pairs*)
using ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *L M1* ∩ *set T3* = *L M2* ∩ *set T3*› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›
unfolding *T3 T2* **.**
moreover have (*V* ‘ *reachable-states M1*) ⊆ *set T2*
using *T1-state-cover T2 T2-invar-1 passes-T2* **by** *fastforce*
moreover have (*q*,(*x*,*y*)) ∈ *list.set undefined-io-pairs*
unfolding *undefined-io-pairs-set*
using ‹*q* ∈ *reachable-states M1*› ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*› ‹*h-obs M1 q x y* = *None*›
by *blast*
ultimately show (*L M1* ∩ {(*V q*)@[(*x*,*y*)]} = *L M2* ∩ {(*V q*)@[(*x*,*y*)]})
unfolding *T3 T2*
by *blast*
qed

have *handles-unverified-transitions*:
(⋀*t* *γ*. *t* ∈ *FSM.transitions M1* ⟹
t-source t ∈ *reachable-states M1* ⟹
length γ ≤ *m*−*n* ⟹
list.set γ ⊆ *FSM.inputs M1* × *FSM.outputs M1* ⟹
butlast γ ∈ *LS M1* (*t-target t*) ⟹
V (*t-target t*) ≠ *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] ⟹
L M1 ∩ (*V* ‘ *reachable-states M1* ∪ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ *ω′* |*ω′*. *ω′* ∈ *list.set* (*prefixes γ*)}) =
L M2 ∩ (*V* ‘ *reachable-states M1* ∪ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ *ω′* |*ω′*. *ω′* ∈ *list.set* (*prefixes γ*)}) ∧
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1* ∪ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ *ω′* |*ω′*. *ω′* ∈ *list.set* (*prefixes γ*)}))
using *T2-cover*[*OF passes-T2*]
unfolding *unverified-transitions-alt-def*
unfolding *handles-unverified-transition*
unfolding ‹*?TS* = *T3*› *n* **by** *blast*

have *satisfies-abstract-h-condition M1 M2 V m*
unfolding *satisfies-abstract-h-condition-def Let-def*

using *abstract-h-condition-by-unverified-transition-and-io-pair-coverage*[**where** *k=m−n,OF assms(1) ‹is-state-cover-assignment M1 V› T3-passes-state-cover T3-preserves-divergence handles-unverified-transitions T3-covers-undefined-io-pairs*]
unfolding *‹?TS = T3› n* **by** *blast*

then show *L M1 = L M2*
using *abstract-h-condition-completeness*[*OF assms(1,2,3,6,5,7,8) ‹is-state-cover-assignment M1 V›*]
by *blast*
qed

end

19 SPY-Framework

This theory defines the SPY-Framework and provides completeness properties.

theory *SPY-Framework*
imports *H-Framework*
begin

19.1 Definition of the Framework

definition *spy-framework* :: (*'a::linorder,'b::linorder,'c::linorder*) *fsm* ⇒
(('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment*) ⇒

(('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment* ⇒ (('a,'b,'c) *fsm* ⇒ ('b×'c) *prefix-tree* ⇒ 'd) ⇒ ('d ⇒ ('b×'c) *list* ⇒ 'd) ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ (('b×'c) *prefix-tree* × 'd)) ⇒
(('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment* ⇒ ('a,'b,'c) *transition list* ⇒ ('a,'b,'c) *transition list*) ⇒
(('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment* ⇒ ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ 'd) ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ *nat* ⇒ ('a,'b,'c) *transition* ⇒ (('b×'c) *prefix-tree* × 'd)) ⇒
(('a,'b,'c) *fsm* ⇒ ('a,'b,'c) *state-cover-assignment* ⇒ ('b×'c) *prefix-tree* ⇒ 'd ⇒ ('d ⇒ ('b×'c) *list* ⇒ 'd) ⇒ ('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒ 'a ⇒ 'b ⇒ 'c ⇒ (('b×'c) *prefix-tree*) × 'd) ⇒
(('a,'b,'c) *fsm* ⇒ ('b×'c) *prefix-tree* ⇒ 'd) ⇒
('d ⇒ ('b×'c) *list* ⇒ 'd) ⇒
('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list list*) ⇒
('d ⇒ ('b×'c) *list* ⇒ ('b×'c) *list* ⇒ 'd) ⇒
nat ⇒
('b×'c) *prefix-tree*
where
spy-framework M
get-state-cover
separate-state-cover
sort-unverified-transitions

establish-convergence
append-io-pair
cg-initial
cg-insert
cg-lookup
cg-merge
m
= (*let*
rstates-set = *reachable-states M*;
rstates = *reachable-states-as-list M*;
rstates-io = *List.product rstates* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*));
undefined-io-pairs = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M q x y* = *None*) *rstates-io*;
V = *get-state-cover M*;
n = *size-r M*;
TG1 = *separate-state-cover M V cg-initial cg-insert cg-lookup*;
sc-covered-transitions = (⋃ *q* ∈ *rstates-set* . *covered-transitions M V* (*V q*));
unverified-transitions = *sort-unverified-transitions M V* (*filter* (λ*t* . *t-source t* ∈ *rstates-set* ∧ *t* ∉ *sc-covered-transitions*) (*transitions-as-list M*));
verify-transition = (λ (*T*,*G*) *t* . *let TGxy* = *append-io-pair M V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*);
(*T*′,*G*′) = *establish-convergence M V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*;
G′′ = *cg-merge G*′ ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
in (*T*′,*G*′′));
TG2 = *foldl verify-transition TG1 unverified-transitions*;
verify-undefined-io-pair = (λ *T* (*q*,(*x*,*y*)) . *fst* (*append-io-pair M V T* (*snd TG2*) *cg-insert cg-lookup q x y*))
in
foldl verify-undefined-io-pair (*fst TG2*) *undefined-io-pairs*)

19.2 Required Conditions on Procedural Parameters

definition *verifies-transition* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
nat ⇒
('*a*,'*b*,'*c*) *transition* ⇒
(('*b*×'*c*) *prefix-tree* × '*d*)) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*e*,'*b*,'*c*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒

bool

where

verifies-transition f M1 M2 V T0 cg-insert cg-lookup =
(∀ *T G m t* .
(*set T* ⊆ *set* (*fst* (*f M1 V T G cg-insert cg-lookup m t*)))
∧ (*finite-tree T* ⟶ *finite-tree* (*fst* (*f M1 V T G cg-insert cg-lookup m t*)))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
size-r M1 ≤ *m* ⟶
size M2 ≤ *m* ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) ⟶
V ‘ *reachable-states M1* ⊆ *set T* ⟶
t ∈ *transitions M1* ⟶
t-source t ∈ *reachable-states M1* ⟶
((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*)) ⟶
((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ∈ *L M2* ⟶
convergence-graph-lookup-invar M1 M2 cg-lookup G ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
L M1 ∩ *set* (*fst* (*f M1 V T G cg-insert cg-lookup m t*)) = *L M2* ∩ *set* (*fst* (*f M1 V T G cg-insert cg-lookup m t*)) ⟶
(*set T0* ⊆ *set T*) ⟶
(*converge M2* ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) (*V* (*t-target t*)))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup m t*))))

definition *verifies-io-pair* :: (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
'*a* ⇒ '*b* ⇒ '*c* ⇒
(('*b*×'*c*) *prefix-tree* × '*d*)) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*e*,'*b*,'*c*) *fsm* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒

bool

where

verifies-io-pair f M1 M2 cg-insert cg-lookup =
(∀ *V T G q x y* .
(*set T* ⊆ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))

$\wedge$ (*finite-tree T* $\longrightarrow$ *finite-tree* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ (*observable M1* $\longrightarrow$
observable M2 $\longrightarrow$
minimal M1 $\longrightarrow$
minimal M2 $\longrightarrow$
inputs M2 = *inputs M1* $\longrightarrow$
outputs M2 = *outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1*) = *L M2* $\cap$ *V* ‘ *reachable-states M1* $\longrightarrow$
q $\in$ *reachable-states M1* $\longrightarrow$
x $\in$ *inputs M1* $\longrightarrow$
y $\in$ *outputs M1* $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
L M1 $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\longrightarrow$
($\exists$ α .
converge M1 α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
α $\in$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\wedge$
α@[(*x*,*y*)] $\in$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))))

lemma *verifies-io-pair-handled*:
assumes *verifies-io-pair f M1 M2 cg-insert cg-lookup*
shows *handles-io-pair f M1 M2 cg-insert cg-lookup*
proof –

have $*$:$\bigwedge$ *V T G q x y* . *set T* $\subseteq$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*))
using *assms* **unfolding** *verifies-io-pair-def*
by *metis*

have $***$:$\bigwedge$ *V T G q x y* . *finite-tree T* $\longrightarrow$ *finite-tree* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*))
using *assms* **unfolding** *verifies-io-pair-def*
by *metis*

have $**$:$\bigwedge$ *V T G q x y*.
observable M1 $\Longrightarrow$
observable M2 $\Longrightarrow$
minimal M1 $\Longrightarrow$
minimal M2 $\Longrightarrow$
FSM.inputs M2 = *FSM.inputs M1* $\Longrightarrow$
FSM.outputs M2 = *FSM.outputs M1* $\Longrightarrow$
is-state-cover-assignment M1 V $\Longrightarrow$
L M1 $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1* $\Longrightarrow$

$q \in$ *reachable-states M1* $\Longrightarrow$
$x \in$ *inputs M1* $\Longrightarrow$
$y \in$ *outputs M1* $\Longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
L M1 $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\Longrightarrow$
(*L M1* $\cap$ {(*V q*)@[(*x*,*y*)]} = *L M2* $\cap$ {(*V q*)@[(*x*,*y*)]})
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))
proof −
fix *V T G q x y*
assume *a01*: *observable M1*
moreover assume *a02*: *observable M2*
moreover assume *a03*: *minimal M1*
moreover assume *a04*: *minimal M2*
moreover assume *a05*: *FSM.inputs M2* = *FSM.inputs M1*
moreover assume *a06*: *FSM.outputs M2* = *FSM.outputs M1*
moreover assume *a07*: *is-state-cover-assignment M1 V*
moreover assume *a09*: *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
moreover assume *a10*: $q \in$ *reachable-states M1*
moreover assume *a11*: $x \in$ *inputs M1*
moreover assume *a12*: $y \in$ *outputs M1*
moreover assume *a13*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*
moreover assume *a14*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
moreover assume *a15*: *L M1* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*))

ultimately have $*$:($\exists\alpha$. *converge M1* α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
$\alpha \in$ *Prefix-Tree.set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)) $\wedge$ α @ [(*x*, *y*)] $\in$ *Prefix-Tree.set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*)))
and $**$: *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))
using *assms* **unfolding** *verifies-io-pair-def*
by *presburger*+

have (*L M1* $\cap$ {(*V q*)@[(*x*,*y*)]} = *L M2* $\cap$ {(*V q*)@[(*x*,*y*)]})
proof −
obtain α **where** *converge M1* α (*V q*) **and** *converge M2* α (*V q*) **and** α @ [(*x*, *y*)] $\in$ *Prefix-Tree.set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*))
using $*$ **by** *blast*

have (*V q*)@[(*x*,*y*)] $\in$ *L M1* = (α@[(*x*,*y*)] $\in$ *L M1*)
using ‹*converge M1* α (*V q*)› **using** *a01 a07*
by (*meson converge-append-language-iff*)
moreover have (*V q*)@[(*x*,*y*)] $\in$ *L M2* = (α@[(*x*,*y*)] $\in$ *L M2*)

using ‹*converge M2* α (*V q*)› **using** *a02 a07*
by (*meson converge-append-language-iff*)
moreover have α @ [(*x*, *y*)] $\in$ *L M1* = (α @ [(*x*, *y*)] $\in$ *L M2*)
using ‹α @ [(*x*, *y*)] $\in$ *Prefix-Tree.set* (*fst* (*f M1 V T G cg-insert cg-lookup q x y*))› *a15*
by *blast*
ultimately show *?thesis*
by *blast*
qed
then show (*L M1* $\cap$ {(*V q*)@[(*x*,*y*)]} = *L M2* $\cap$ {(*V q*)@[(*x*,*y*)]}) $\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*f M1 V T G cg-insert cg-lookup q x y*))
using ** **by** *blast*
qed

show *?thesis*
unfolding *handles-io-pair-def*
using * *** ** **by** *presburger*
qed

19.3 Completeness and Finiteness of the Framework

lemma *spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1* (*get-state-cover M1*)
and $\bigwedge$ *xs* . *List.set xs* = *List.set* (*sort-unverified-transitions M1* (*get-state-cover M1*) *xs*)
and *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*
and *separates-state-cover separate-state-cover M1 M2 cg-initial cg-insert cg-lookup*
and *verifies-transition establish-convergence M1 M2* (*get-state-cover M1*) (*fst* (*separate-state-cover M1* (*get-state-cover M1*) *cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup*
and *verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*spy-framework M1 get-state-cover separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m*))
= (*L M2* $\cap$ *set* (*spy-framework M1 get-state-cover*

separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m)))
(**is** (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*)))
and *finite-tree* (*spy-framework M1 get-state-cover separate-state-cover sort-unverified-transitions establish-convergence append-io-pair cg-initial cg-insert cg-lookup cg-merge m*)
proof
show (*L M1* = *L M2*) $\Longrightarrow$ ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))
by *blast*

define *rstates* **where** *rstates*: *rstates* = *reachable-states-as-list M1*
define *rstates-io* **where** *rstates-io*: *rstates-io* = *List.product rstates* (*List.product* (*inputs-as-list M1*) (*outputs-as-list M1*))
define *undefined-io-pairs* **where** *undefined-io-pairs*: *undefined-io-pairs* = *List.filter* (λ (*q*,(*x*,*y*)) . *h-obs M1 q x y* = *None*) *rstates-io*
define *V* **where** *V*: *V* = *get-state-cover M1*
define *n* **where** *n*: *n* = *size-r M1*
define *TG1* **where** *TG1*: *TG1* = *separate-state-cover M1 V cg-initial cg-insert cg-lookup*

define *sc-covered-transitions* **where** *sc-covered-transitions*: *sc-covered-transitions* = ($\bigcup$ *q* $\in$ *reachable-states M1* . *covered-transitions M1 V* (*V q*))
define *unverified-transitions* **where** *unverified-transitions*: *unverified-transitions* = *sort-unverified-transitions M1 V* (*filter* (λ*t* . *t-source t* $\in$ *reachable-states M1* $\wedge$ *t* $\notin$ *sc-covered-transitions*) (*transitions-as-list M1*))
define *verify-transition* **where** *verify-transition*: *verify-transition* = (λ (*T*,*G*) *t* . *let TGxy* = *append-io-pair M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*);
(*T*′,*G*′) = *establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*;
G″ = *cg-merge G*′ ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
in (*T*′,*G*″))
define *TG2* **where** *TG2*: *TG2* = (*foldl verify-transition TG1 unverified-transitions*)
define *verify-undefined-io-pair* **where** *verify-undefined-io-pair*: *verify-undefined-io-pair* = (λ *T* (*q*,(*x*,*y*)) . *fst* (*append-io-pair M1 V T* (*snd TG2*) *cg-insert cg-lookup q x y*))
define *T3* **where** *T3*: *T3* = *foldl verify-undefined-io-pair* (*fst TG2*) *undefined-io-pairs*

have *?TS* = *T3*
unfolding *rstates rstates-io undefined-io-pairs V n TG1 sc-covered-transitions unverified-transitions verify-transition TG2 verify-undefined-io-pair T3*
unfolding *spy-framework-def Let-def*
by *force*
then have ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*)) $\Longrightarrow$ *L M1* $\cap$ *set T3* = *L M2* $\cap$ *set T3*
by *simp*

have *is-state-cover-assignment M1 V*
unfolding *V* **using** *assms*(*9*) **.**

define *T1* **where** *T1*: *T1* = *fst TG1*
moreover define *G1* **where** *G1*: *G1* = *snd TG1*
ultimately have *TG1* = (*T1*,*G1*)
by *auto*

have *T1-state-cover*: *V* ‘ *reachable-states M1* ⊆ *set T1*
and *T1-finite*: *finite-tree T1*
using ‹*separates-state-cover separate-state-cover M1 M2 cg-initial cg-insert cg-lookup*›
unfolding *T1 TG1 separates-state-cover-def*
by *blast*+

have *T1-V-div*: (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*)) ⟹ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *G1-invar*: (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*)) ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G1*
using ‹*separates-state-cover separate-state-cover M1 M2 cg-initial cg-insert cg-lookup*›
unfolding *T1 G1 TG1 separates-state-cover-def*
using *assms*(*1*−*4*,*7*,*8*) ‹*is-state-cover-assignment M1 V*› *assms*(*12*,*11*)
by *blast*+

have *verifies-transition establish-convergence M1 M2 V T1 cg-insert cg-lookup*
using *assms*(*15*)
unfolding *T1 TG1 V* **.**

have *sc-covered-transitions-alt-def*: *sc-covered-transitions* = {*t* . *t* ∈ *transitions M1* ∧ *t-source t* ∈ *reachable-states M1* ∧ (*V* (*t-target t*) = (*V* (*t-source t*))@[(*t-input t*, *t-output t*)])}
(**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix *t* **assume** *t* ∈ *?A*
then obtain *q* **where** *t* ∈ *covered-transitions M1 V* (*V q*) **and** *q* ∈ *reachable-states M1*
unfolding *sc-covered-transitions*
by *blast*
then have *V q* ∈ *L M1* **and** *after-initial M1* (*V q*) = *q*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment*

M1 V›]
by *blast+*

then obtain *p* **where** *path M1* (*initial M1*) *p* **and** *p-io p* = *V q*
by *auto*
then have ∗: *the-elem* (*paths-for-io M1* (*initial M1*) (*V q*)) = *p*
using *observable-paths-for-io*[*OF assms*(*1*) ‹*V q* ∈ *L M1*›]
unfolding *paths-for-io-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *mem-Collect-eq observable-path-unique singletonI the-elem-eq*)

have *t* ∈ *list.set p* **and** *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] = *V* (*t-target t*)
using ‹*t* ∈ *covered-transitions M1 V* (*V q*)›
unfolding *covered-transitions-def Let-def* ∗
by *auto*

have *t* ∈ *transitions M1*
using ‹*t* ∈ *list.set p*› ‹*path M1* (*initial M1*) *p*›
by (*meson path-transitions subsetD*)
moreover have *t-source t* ∈ *reachable-states M1*
using *reachable-states-path*[*OF reachable-states-initial* ‹*path M1* (*initial M1*) *p*› ‹*t* ∈ *list.set p*›] **.**
ultimately show *t* ∈ *?B*
using ‹*V* (*t-source t*) @ [(*t-input t*, *t-output t*)] = *V* (*t-target t*)›
by *auto*
qed

show *?B* ⊆ *?A*
proof
fix *t* **assume** *t* ∈ *?B*
then have *t* ∈ *transitions M1*
t-source t ∈ *reachable-states M1*
(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)
by *auto*
then have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*of t-source t M1 t*]
by *blast*
then have *V* (*t-target t*) ∈ *L M1* **and** *after-initial M1* (*V* (*t-target t*)) = (*t-target t*)
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast+*
then obtain *p* **where** *path M1* (*initial M1*) *p* **and** *p-io p* = *V* (*t-target t*)
by *auto*
then have ∗: *the-elem* (*paths-for-io M1* (*initial M1*) (*V* (*t-target t*))) = *p*
using *observable-paths-for-io*[*OF assms*(*1*) ‹*V* (*t-target t*) ∈ *L M1*›]
unfolding *paths-for-io-def*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *mem-Collect-eq observable-path-unique*

singletonI the-elem-eq)

have *V* (*t-source t*) ∈ *L M1* **and** *after-initial M1* (*V* (*t-source t*)) = (*t-source t*)
using ‹*t-source t* ∈ *reachable-states M1*›
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast*+
then obtain *p′* **where** *path M1* (*initial M1*) *p′* **and** *p-io p′* = *V* (*t-source t*)
by *auto*

have *path M1* (*initial M1*) (*p′*@[*t*])
using *after-path*[*OF assms*(*1*) ‹*path M1* (*initial M1*) *p′*›] ‹*path M1* (*initial M1*) *p′*› ‹*t*∈*transitions M1*›
unfolding ‹*p-io p′* = *V* (*t-source t*)›
unfolding ‹*after-initial M1* (*V* (*t-source t*)) = (*t-source t*)›
by (*metis path-append single-transition-path*)
moreover have *p-io* (*p′*@[*t*]) = *p-io p*
using ‹*p-io p′* = *V* (*t-source t*)›
unfolding ‹*p-io p* = *V* (*t-target t*)› ‹(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)›[*symmetric*]
by *auto*
ultimately have *p′*@[*t*] = *p*
using *observable-path-unique*[*OF assms*(*1*) - ‹*path M1* (*initial M1*) *p*›]
by *force*
then have *t* ∈ *list.set p*
by *auto*
then have *t* ∈ *covered-transitions M1 V* (*V* (*t-target t*))
using ‹(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)›
unfolding *covered-transitions-def Let-def* ∗
by *auto*
then show *t* ∈ *?A*
using ‹*t-target t* ∈ *reachable-states M1*›
unfolding *sc-covered-transitions*
by *blast*
qed
qed

have *T1-covered-transitions-conv*: ⋀ *t* . (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*)) ⟹ *t* ∈ *sc-covered-transitions* ⟹ *converge M2* (*V* (*t-target t*)) ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])
proof −
fix *t* **assume** (*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*))
t ∈ *sc-covered-transitions*

then have *t* ∈ *transitions M1*
t-source t ∈ *reachable-states M1*
(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)

unfolding *sc-covered-transitions-alt-def*
by *auto*
then have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*of t-source t M1 t*]
by *blast*
then have *V* (*t-target t*) ∈ *L M1*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *blast*
moreover have *V* (*t-target t*) ∈ *set T1*
using *T1-state-cover* ‹*t-target t* ∈ *reachable-states M1*›
by *blast*
ultimately have *V* (*t-target t*) ∈ *L M2*
using ‹(*L M1* ∩ *set T1* = (*L M2* ∩ *set T1*))›
by *blast*
then show *converge M2* (*V* (*t-target t*)) ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])
unfolding ‹(*V* (*t-source t*))@[(*t-input t*, *t-output t*)] = *V* (*t-target t*)›
by *auto*
qed

have *unverified-transitions-alt-def* : *list.set unverified-transitions* = {*t* . *t* ∈ *transitions M1* ∧ *t-source t* ∈ *reachable-states M1* ∧ (*V* (*t-target t*) ≠ (*V* (*t-source t*))@[(*t-input t*, *t-output t*)])}
unfolding *unverified-transitions sc-covered-transitions-alt-def V*
unfolding *assms*(*10*)[*symmetric*]
using *transitions-as-list-set*[*of M1*]
by *auto*

have *cg-insert-invar* : ⋀ *G* γ . γ ∈ *L M1* ⟹ γ ∈ *L M2* ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-insert G* γ)
using *assms*(*12*)
unfolding *convergence-graph-insert-invar-def*
by *blast*

have *cg-merge-invar* : ⋀ *G* γ γ′. *convergence-graph-lookup-invar M1 M2 cg-lookup G* ⟹ *converge M1* γ γ′ ⟹ *converge M2* γ γ′ ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*cg-merge G* γ γ′)
using *assms*(*13*)
unfolding *convergence-graph-merge-invar-def*
by *blast*

define *T2* **where** *T2*: *T2* = *fst TG2*
define *G2* **where** *G2*: *G2* = *snd TG2*

have *verify-transition-retains-testsuite*: $\bigwedge$ *t T G* . *set T* $\subseteq$ *set* (*fst* (*verify-transition* (*T*,*G*) *t*))
proof −
fix *t T G*
define *TGxy* **where** *TGxy*: *TGxy* = *append-io-pair M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)
obtain *T′ G′* **where** *TG′*: (*T′*,*G′*) = *establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*
using *prod.exhaust* **by** *metis*
define *G″* **where** *G″*: *G″* = *cg-merge G′* ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))

have ∗:*verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)
using *TG′*[*symmetric*]
unfolding *verify-transition G″ TGxy Let-def case-prod-conv*
by *force*

have *set T* $\subseteq$ *set* (*fst TGxy*)
using ‹*verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*›
unfolding *verifies-io-pair-def TGxy*
by *blast*
also have *set* (*fst TGxy*) $\subseteq$ *set* (*fst* (*T′*,*G′*))
using ‹*verifies-transition establish-convergence M1 M2 V T1 cg-insert cg-lookup*›
unfolding *TG′ verifies-transition-def*
by *blast*
finally show *set T* $\subseteq$ *set* (*fst* (*verify-transition* (*T*,*G*) *t*))
unfolding ∗ *fst-conv* **.**
qed

have *verify-transition-retains-finiteness*: $\bigwedge$ *t T G* . *finite-tree T* $\Longrightarrow$ *finite-tree* (*fst* (*verify-transition* (*T*,*G*) *t*))
proof −
fix *T* :: (*′b*×*′c*) *prefix-tree*
fix *t G* **assume** *finite-tree T*

define *TGxy* **where** *TGxy*: *TGxy* = *append-io-pair M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)
obtain *T′ G′* **where** *TG′*: (*T′*,*G′*) = *establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*
using *prod.exhaust* **by** *metis*
define *G″* **where** *G″*: *G″* = *cg-merge G′* ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))

have $*$:*verify-transition* (T,G) t = (T',G'')
using *TG'*[*symmetric*]
unfolding *verify-transition G'' TGxy Let-def case-prod-conv*
by *force*

have *finite-tree* (*fst TGxy*)
using ‹*verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*› ‹*finite-tree T*›
unfolding *verifies-io-pair-def TGxy*
by *blast*
then have *finite-tree* (*fst* (T',G'))
using ‹*verifies-transition establish-convergence M1 M2 V T1 cg-insert cg-lookup*›
unfolding *TG' verifies-transition-def*
by *blast*
then show *finite-tree* (*fst* (*verify-transition* (T,G) t))
unfolding $*$ *fst-conv* **.**
qed

define *covers-unverified-transition*
where *covers-unverified-transition*: *covers-unverified-transition* = (λt (T',G')
.
((∃ α β . *converge M1* α (*V* (*t-source t*)) ∧
converge M2 α (*V* (*t-source t*)) ∧
converge M1 β (*V* (*t-target t*)) ∧
converge M2 β (*V* (*t-target t*)) ∧
α@[(*t-input t*,*t-output t*)] ∈ (*set T'*) ∧
β ∈ (*set T'*))
∧ (*converge M2* ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) (*V* (*t-target t*)))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup G'*))

have *verify-transition-cover-prop*: ⋀ *t T G* . (*L M1* ∩ (*set* (*fst* (*verify-transition* (T,G) t))) = *L M2* ∩ (*set* (*fst* (*verify-transition* (T,G) t))))
⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G*
⟹ t ∈ *transitions M1*
⟹ *t-source t* ∈ *reachable-states M1*
⟹ ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*))
⟹ *set T1* ⊆ *set T*
⟹ *covers-unverified-transition t* (*verify-transition* (T,G) t)
proof −
fix *t T G*
assume *a1*: (*L M1* ∩ (*set* (*fst* (*verify-transition* (T,G) t))) = *L M2* ∩ (*set* (*fst* (*verify-transition* (T,G) t))))
assume *a2*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*

assume *a3*: $t \in$ *transitions M1*
assume *a4*: *t-source t* $\in$ *reachable-states M1*
assume *a5*: *set T1* $\subseteq$ *set T*
assume *a6*: ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\neq$ (*V* (*t-target t*))

define *TGxy* **where** *TGxy*: *TGxy* = *append-io-pair M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)
obtain *T′ G′* **where** *TG′*: (*T′*,*G′*) = *establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*
using *prod.exhaust* **by** *metis*
have *T′*: *T′* = *fst* (*establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*)
and *G′*: *G′* = *snd* (*establish-convergence M1 V* (*fst TGxy*) (*snd TGxy*) *cg-insert cg-lookup m t*)
unfolding *TG′*[*symmetric*] **by** *auto*
define *G″* **where** *G″*: *G″* = *cg-merge G′* ((*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))

have *verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)
using *TG′*[*symmetric*]
unfolding *verify-transition G″ TGxy Let-def case-prod-conv*
by *force*
then have *set T* $\subseteq$ *set T′*
using *verify-transition-retains-testsuite*[*of T G t*] **unfolding** *T′*
by *auto*
then have (*L M1* $\cap$ (*set T1*) = *L M2* $\cap$ (*set T1*))
using *a1 a5* **unfolding** ‹*verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)› *fst-conv*
by *blast*
then have $*$: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
using *T1-V-div*
by *auto*

have *set* (*fst TGxy*) $\subseteq$ *set* (*fst* (*T′*,*G′*))
using ‹*verifies-transition establish-convergence M1 M2 V T1 cg-insert cg-lookup*›
unfolding *TG′ verifies-transition-def*
by *blast*
then have *set* (*fst TGxy*) $\subseteq$ *set* (*fst* (*verify-transition* (*T*,*G*) *t*))
unfolding ‹*verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)› *fst-conv* **.**
then have *L M1* $\cap$ *set* (*fst TGxy*) = *L M2* $\cap$ *set* (*fst TGxy*)
using *a1* **by** *blast*

have *L M1* $\cap$ *set T′* = *L M2* $\cap$ *set T′* **and** *L M1* $\cap$ *set T* = *L M2* $\cap$ *set T*
using *a1* ‹*set T* $\subseteq$ *set T′*› **unfolding** *T′* ‹*verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)› *fst-conv*
by *blast+*

have $**$: *V* ‘ *reachable-states M1* $\subseteq$ *set T*
using *a5 T1-state-cover* **by** *blast*

have *L M1 ∩ V ‘ reachable-states M1 = L M2 ∩ V ‘ reachable-states M1*
using *T1-state-cover ‹L M1 ∩ Prefix-Tree.set T1 = L M2 ∩ Prefix-Tree.set T1›* **by** *blast*

have (∃ *α. converge M1 α (V (t-source t)) ∧*
converge M2 α (V (t-source t)) ∧
α ∈ Prefix-Tree.set (fst TGxy) ∧
α @ [((t-input t), (t-output t))] ∈ Prefix-Tree.set (fst TGxy))
and *convergence-graph-lookup-invar M1 M2 cg-lookup (snd TGxy)*
using *‹verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup›*
unfolding *verifies-io-pair-def*
using *assms(1−4,7,8) ‹is-state-cover-assignment M1 V› ‹L M1 ∩ V ‘ reachable-states M1 = L M2 ∩ V ‘ reachable-states M1› a4 fsm-transition-input[OF a3] fsm-transition-output[OF a3] a2 ‹convergence-graph-insert-invar M1 M2 cg-lookup cg-insert› ‹L M1 ∩ set (fst TGxy) = L M2 ∩ set (fst TGxy)›*
unfolding *TGxy*
by *blast+*

then obtain *w* **where** *converge M1 w (V (t-source t))*
converge M2 w (V (t-source t))
w ∈ Prefix-Tree.set (fst TGxy)
w @ [((t-input t), (t-output t))] ∈ set (fst TGxy)
by *blast*
then have *w @ [((t-input t), (t-output t))] ∈ L M1 ⟷ w @ [((t-input t), (t-output t))] ∈ L M2*
using *‹L M1 ∩ set (fst TGxy) = L M2 ∩ set (fst TGxy)›*
by *blast*
moreover have *w @ [((t-input t), (t-output t))] ∈ L M1 ⟷ V (t-source t) @ [(t-input t, t-output t)] ∈ L M1*
using *‹converge M1 w (V (t-source t))›*
by *(meson assms(1) converge-append-language-iff)*
moreover have *V (t-source t) @ [(t-input t, t-output t)] ∈ L M1*
using *state-cover-transition-converges[OF assms(1) ‹is-state-cover-assignment M1 V› ‹t ∈ transitions M1› ‹t-source t ∈ reachable-states M1›]*
by *auto*
ultimately have *w @ [(t-input t, t-output t)] ∈ L M2*
by *blast*
then have *V (t-source t) @ [(t-input t, t-output t)] ∈ L M2*
using *‹converge M2 w (V (t-source t))›*
by *(meson assms(2) converge-append-language-iff)*
have *V ‘ reachable-states M1 ⊆ set T*
using *a5 T1-state-cover* **by** *blast*

have *set T ⊆ set (fst TGxy)*
using *‹verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup›*
unfolding *verifies-io-pair-def TGxy* **by** *blast*
then have *set T1 ⊆ set (fst TGxy)*

using *a5* **by** *blast*
then have $\bigwedge$ *io* . *set* (*after T1 io*) $\subseteq$ *set* (*after* (*fst TGxy*) *io*)
unfolding *after-set* **by** *auto*

have *V* ‘ *reachable-states M1* $\subseteq$ *set* (*fst TGxy*)
using ** ‹*Prefix-Tree.set T* $\subseteq$ *Prefix-Tree.set* (*fst TGxy*)› **by** *auto*

have *p2*: *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup G′*
using ‹*verifies-transition establish-convergence M1 M2 V T1 cg-insert cg-lookup*›
unfolding *verifies-transition-def*
using *assms*(*1*−*8*) ‹*is-state-cover-assignment M1 V*› ‹*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)› ‹*V* ‘ *reachable-states M1* $\subseteq$ *set* (*fst TGxy*)› *a3 a4 a6* ‹*V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TGxy*)› ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*› ‹*L M1* $\cap$ *set T′* = *L M2* $\cap$ *set T′*›
using ‹*set T1* $\subseteq$ *set* (*fst TGxy*)›
unfolding *T′ G′*
by *blast*+

have *w* @ [((*t-input t*), (*t-output t*))] $\in$ *set T′*
using ‹*w* @ [((*t-input t*), (*t-output t*))] $\in$ *set* (*fst TGxy*)›
using *T′* ‹*Prefix-Tree.set* (*fst TGxy*) $\subseteq$ *Prefix-Tree.set* (*fst* (*T′*, *G′*))› **by** *auto*

have *p1*: ($\exists\,\alpha\ \beta$.
converge M1 α (*V* (*t-source t*)) $\wedge$
converge M2 α (*V* (*t-source t*)) $\wedge$
converge M1 β (*V* (*t-target t*)) $\wedge$
converge M2 β (*V* (*t-target t*)) $\wedge$
α @ [(*t-input t*, *t-output t*)] $\in$ *set T′* $\wedge$
$\beta \in$ *set T′*)
proof −
have *V* (*t-source t*) $\in$ *L M1*
using *state-cover-assignment-after*(*1*)[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› ‹*t-source t* $\in$ *reachable-states M1*›] **.**
have *V* (*t-target t*) $\in$ *L M1*
using *state-cover-assignment-after*(*1*)[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› *reachable-states-next*[*OF* ‹*t-source t* $\in$ *reachable-states M1*› ‹*t* $\in$ *transitions M1*›]]
by *auto*

note ‹*converge M1 w* (*V* (*t-source t*))› **and** ‹*converge M2 w* (*V* (*t-source t*))›
moreover have *converge M1* (*V* (*t-target t*)) (*V* (*t-target t*))
using ‹*V* (*t-target t*) $\in$ *L M1*› **by** *auto*
moreover have *converge M2* (*V* (*t-target t*)) (*V* (*t-target t*))
using *reachable-states-next*[*OF* ‹*t-source t* $\in$ *reachable-states M1*› ‹*t* $\in$

transitions M1›] ‹*V* (*t-target t*) ∈ *L M1*› ‹*L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*›
by *auto*
moreover note ‹*w* @ [(*t-input t*, *t-output t*)] ∈ *set T′*›
moreover have *V* (*t-target t*) ∈ *set T′*
using ‹*V* ‘ *reachable-states M1* ⊆ *set T*› ‹*set T* ⊆ *set T′*› *reachable-states-next*[*OF* ‹*t-source t* ∈ *reachable-states M1*› ‹*t* ∈ *transitions M1*›]
by *auto*
ultimately show *?thesis*
by *blast*
qed

have *p3*: *convergence-graph-lookup-invar M1 M2 cg-lookup G″*
unfolding *G″*
using *cg-merge-invar*[*OF* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*›]
state-cover-transition-converges[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› *a3 a4*]
‹*converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))›
by *blast*
show *covers-unverified-transition t* (*verify-transition* (*T*,*G*) *t*)
using *p1 p2 p3*
unfolding ‹*verify-transition* (*T*,*G*) *t* = (*T′*,*G″*)› *fst-conv snd-conv covers-unverified-transition*
by *blast*
qed

have *verify-transition-foldl-invar-1*: ⋀ *ts* . *list.set ts* ⊆ *list.set unverified-transitions* ⟹
set T1 ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ∧ *finite-tree* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
proof −
fix *ts* **assume** *list.set ts* ⊆ *list.set unverified-transitions*
then show *set T1* ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ∧ *finite-tree* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case*
using *T1-finite* **by** *auto*
next
case (*snoc t ts*)
then have *t* ∈ *transitions M1* **and** *t-source t* ∈ *reachable-states M1*
unfolding *unverified-transitions-alt-def*
by *force*+

have *p1*: *list.set ts* ⊆ *list.set unverified-transitions*
using *snoc.prems*(*1*) **by** *auto*

have *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
using *verify-transition-retains-testsuite* **unfolding** *foldl-append foldl.simps*
by (*metis eq-fst-iff*)

have ∗∗: *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
and ∗∗∗: *finite-tree* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *snoc.IH*[*OF p1*]
by *auto*

have *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
using ∗∗ *verify-transition-retains-testsuite* ‹*set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))›
by *auto*
moreover have *finite-tree* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
using *verify-transition-retains-finiteness*[*OF* ∗∗∗, *of snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)]
by *auto*
ultimately show *?case*
by *simp*
qed
qed
then have *T2-invar-1*: *set T1* ⊆ *set T2*
and *T2-finite* : *finite-tree T2*
unfolding *TG2 G2 T2* ‹*TG1* = (*T1*,*G1*)›
by *auto*

have *verify-transition-foldl-invar-2*: ⋀ *ts* . *list.set ts* ⊆ *list.set unverified-transitions* ⟹
L M1 ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⟹
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))
proof −
fix *ts* **assume** *list.set ts* ⊆ *list.set unverified-transitions*
and *L M1* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
then show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))
proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case*
using *G1-invar* **by** *auto*
next
case (*snoc t ts*)
then have *t* ∈ *transitions M1* **and** *t-source t* ∈ *reachable-states M1* **and** ((*V*

(*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*))
unfolding *unverified-transitions-alt-def*
by *force+*

have *p1*: *list.set ts* ⊆ *list.set unverified-transitions*
using *snoc.prems*(*1*) **by** *auto*

have *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
using *verify-transition-retains-testsuite* **unfolding** *foldl-append foldl.simps*
by (*metis eq-fst-iff*)
then have *p2*: *L M1* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *snoc.prems*(*2*)
by *blast*

have ∗:*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *snoc.IH*[*OF p1 p2*]
by *auto*

have ∗∗: *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *verify-transition-foldl-invar-1*[*OF p1*] **by** *blast*

have *covers-unverified-transition t* (*verify-transition* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*), *snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)) *t*)
using *verify-transition-cover-prop*[*OF* - ∗ ‹*t* ∈ *transitions M1*› ‹*t-source t* ∈ *reachable-states M1*› ‹((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*))› ∗∗] *snoc.prems*(*2*)
unfolding *prod.collapse*
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*verify-transition* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*), *snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)) *t*))
unfolding *covers-unverified-transition*
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
by *auto*
moreover have *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))
using ∗∗ *verify-transition-retains-testsuite* ‹*set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t*])))›
by *auto*
ultimately show *?case*
by *simp*
qed
qed

then have *T2-invar-2*: *L M1* ∩ *set T2* = *L M2* ∩ *set T2* ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
unfolding *TG2 G2 T2* ‹*TG1* = (*T1*,*G1*)› **by** *auto*

have *T2-cover*: ⋀ *t* . *L M1* ∩ *set T2* = *L M2* ∩ *set T2* ⟹ *t* ∈ *list.set unverified-transitions* ⟹ *covers-unverified-transition t* (*T2*,*G2*)
proof −
have ⋀ *t ts* . *t* ∈ *list.set ts* ⟹ *list.set ts* ⊆ *list.set unverified-transitions* ⟹ *L M1* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⟹ *covers-unverified-transition t* (*foldl verify-transition* (*T1*, *G1*) *ts*)
proof −
fix *t ts*
assume *t* ∈ *list.set ts* **and** *list.set ts* ⊆ *list.set unverified-transitions* **and** *L M1* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))

then show *covers-unverified-transition t* (*foldl verify-transition* (*T1*, *G1*) *ts*)
proof (*induction ts rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc t′ ts*)

then have *t* ∈ *transitions M1* **and** *t-source t* ∈ *reachable-states M1* **and** ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) ≠ (*V* (*t-target t*))
unfolding *unverified-transitions-alt-def* **by** *force+*

have *t′* ∈ *transitions M1* **and** *t-source t′* ∈ *reachable-states M1* **and** ((*V* (*t-source t′*)) @ [(*t-input t′*,*t-output t′*)]) ≠ (*V* (*t-target t′*))
using *snoc.prems*(*2*)
unfolding *unverified-transitions-alt-def*
by *auto*

have *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t′*]))))
using *verify-transition-retains-testsuite* **unfolding** *foldl-append foldl.simps*
by (*metis eq-fst-iff*)
then have *L M1* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *snoc.prems*(*3*)
by *blast*

have ∗: *L M1* ∩ *Prefix-Tree.set* (*fst* (*verify-transition* (*foldl verify-transition* (*T1*, *G1*) *ts*) *t′*)) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*verify-transition* (*foldl verify-transition* (*T1*, *G1*) *ts*) *t′*))
using *snoc.prems*(*3*) **by** *auto*

have *L M1* ∩ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L*

M2 ∩ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using ‹*set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ⊆ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))› *snoc.prems*(*3*)
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)) ∧ *Prefix-Tree.set T1* ⊆ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))
using *snoc.prems*(*2*) *verify-transition-foldl-invar-1*[*of ts*] *verify-transition-foldl-invar-2*[*of ts*]
by *auto*
then have *covers-t'*: *covers-unverified-transition t'* (*verify-transition* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*), *snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)) *t'*)

using *verify-transition-cover-prop*[*OF* - - ‹*t'* ∈ *transitions M1*› ‹*t-source t'* ∈ *reachable-states M1*› ‹((*V* (*t-source t'*)) @ [(*t-input t'*,*t-output t'*)]) ≠ (*V* (*t-target t'*))›, *of* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))]
unfolding *prod.collapse*
using ∗
by *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*verify-transition* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*), *snd* (*foldl verify-transition* (*T1*, *G1*) *ts*)) *t'*))
unfolding *covers-unverified-transition*
by *force*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))
by *auto*

show *?case* **proof** (*cases t* = *t'*)
case *True*
then show *?thesis*
using *covers-t'* **by** *auto*
next
case *False*
then have *t* ∈ *list.set ts*
using *snoc.prems*(*1*) **by** *auto*

have *list.set ts* ⊆ *list.set* (*unverified-transitions*)
using *snoc.prems*(*2*) **by** *auto*

have *covers-unverified-transition t* (*foldl verify-transition* (*T1*, *G1*) *ts*)
using *snoc.IH*[*OF* ‹*t* ∈ *list.set ts*›] *snoc.prems*(*2*) ‹*L M1* ∩ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))›
by *auto*
then have *covers-unverified-transition t* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*), *snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))
by *auto*

then have ($\exists \alpha\ \beta$. *converge M1* α (*V* (*t-source t*)) $\wedge$
converge M2 α (*V* (*t-source t*)) $\wedge$
converge M1 β (*V* (*t-target t*)) $\wedge$
converge M2 β (*V* (*t-target t*)) $\wedge$ α @ [(*t-input t*, *t-output t*)] $\in$ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) $\wedge$ $\beta \in$ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*))) $\wedge$
converge M2 (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
unfolding *covers-unverified-transition*
by *blast*
moreover have *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) $\subseteq$ *set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))
using *verify-transition-retains-testsuite*[*of* (*fst* (*foldl verify-transition* (*T1*, *G1*) *ts*)) (*snd* (*foldl verify-transition* (*T1*, *G1*) *ts*))]
unfolding *prod.collapse*
by *auto*
ultimately have ($\exists \alpha\ \beta$.
converge M1 α (*V* (*t-source t*)) $\wedge$
converge M2 α (*V* (*t-source t*)) $\wedge$
converge M1 β (*V* (*t-target t*)) $\wedge$
converge M2 β (*V* (*t-target t*)) $\wedge$ α @ [(*t-input t*, *t-output t*)] $\in$ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*]))) $\wedge$ $\beta \in$ *Prefix-Tree.set* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))) $\wedge$
converge M2 (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
by *blast*
then have *covers-unverified-transition t* (*fst* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])), *snd* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))
unfolding *covers-unverified-transition*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl verify-transition* (*T1*, *G1*) (*ts*@[*t'*])))›
by *blast*
then show *?thesis*
by *auto*
qed
qed
qed

then show $\bigwedge$ *t* . *L M1* $\cap$ *set T2* = *L M2* $\cap$ *set T2* $\Longrightarrow$ *t* $\in$ *list.set unverified-transitions* $\Longrightarrow$ *covers-unverified-transition t* (*T2*,*G2*)
unfolding *TG2 T2 G2* ‹*TG1* = (*T1*,*G1*)›
by *simp*
qed

have *verify-undefined-io-pair-retains-testsuite*: $\bigwedge$ *qxy T* . *set T* $\subseteq$ *set* (*verify-undefined-io-pair T qxy*)

proof −
fix *qxy* :: ('*a* × '*b* × '*c*)
fix *T*
obtain *q x y* **where** *qxy* = (*q*,*x*,*y*)
using *prod.exhaust* **by** *metis*
show ‹*set T* ⊆ *set* (*verify-undefined-io-pair T qxy*)›
unfolding ‹*qxy* = (*q*,*x*,*y*)›
using ‹*verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*›
unfolding *verifies-io-pair-def verify-undefined-io-pair case-prod-conv*
by *blast*
qed
have *verify-undefined-io-pair-folding-retains-testsuite*: ⋀ *qxys T* . *set T* ⊆ *set* (*foldl verify-undefined-io-pair T qxys*)
proof −
fix *qxys T*
show *set T* ⊆ *set* (*foldl verify-undefined-io-pair T qxys*)
using *verify-undefined-io-pair-retains-testsuite*
by (*induction qxys rule*: *rev-induct*; *force*)
qed

have *verify-undefined-io-pair-retains-finiteness*: ⋀ *qxy T* . *finite-tree T* ⟹ *finite-tree* (*verify-undefined-io-pair T qxy*)
proof −
fix *qxy* :: ('*a* × '*b* × '*c*)
fix *T* :: ('*b*×'*c*) *prefix-tree*
assume *finite-tree T*
obtain *q x y* **where** *qxy* = (*q*,*x*,*y*)
using *prod.exhaust* **by** *metis*
show ‹*finite-tree* (*verify-undefined-io-pair T qxy*)›
unfolding ‹*qxy* = (*q*,*x*,*y*)›
using ‹*verifies-io-pair append-io-pair M1 M2 cg-insert cg-lookup*› ‹*finite-tree T*›
unfolding *verifies-io-pair-def verify-undefined-io-pair case-prod-conv*
by *blast*
qed
have *verify-undefined-io-pair-folding-retains-finiteness*: ⋀ *qxys T* . *finite-tree T* ⟹ *finite-tree* (*foldl verify-undefined-io-pair T qxys*)
proof −
fix *qxys*
fix *T* :: ('*b*×'*c*) *prefix-tree*
assume *finite-tree T*
then show *finite-tree* (*foldl verify-undefined-io-pair T qxys*)
using *verify-undefined-io-pair-retains-finiteness*
by (*induction qxys rule*: *rev-induct*; *force*)
qed

have *set T2* ⊆ *set T3*
unfolding *T3 T2*
proof (*induction undefined-io-pairs rule*: *rev-induct*)

case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case*
using *verify-undefined-io-pair-retains-testsuite*[*of* (*foldl verify-undefined-io-pair* (*fst TG2*) *xs*) *x*]
by *force*
qed
then have *passes-T2*: $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow L\ M1 \cap set\ T2 = L\ M2 \cap set\ T2$
using ‹$((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow (L\ M1 \cap set\ T3 = L\ M2 \cap set\ T3)$›
by *blast*

have $set\ T1 \subseteq set\ T3$
and *G2-invar*: $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow$ *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
using *T2-invar-1 T2-invar-2*[*OF passes-T2*] ‹$set\ T2 \subseteq set\ T3$›
by *auto*
then have *passes-T1*: $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow L\ M1 \cap set\ T1 = L\ M2 \cap set\ T1$
using ‹$((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow L\ M1 \cap set\ T3 = L\ M2 \cap set\ T3$›
by *blast*

have *T3-preserves-divergence* : $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
using *T1-V-div*[*OF passes-T1*] **.**

have *T3-state-cover* : *V* ‘ *reachable-states M1* $\subseteq$ *set T3*
using *T1-state-cover* ‹$set\ T1 \subseteq set\ T3$›
by *blast*

have *T3-covers-transitions* : $((L\ M1 \cap set\ ?TS) = (L\ M2 \cap set\ ?TS)) \Longrightarrow (\bigwedge t\ .\ t \in transitions\ M1 \Longrightarrow$ *t-source t* $\in$ *reachable-states M1* $\Longrightarrow$
$(\exists \alpha\ \beta.$
converge M1 α (*V* (*t-source t*)) $\wedge$
converge M2 α (*V* (*t-source t*)) $\wedge$
converge M1 β (*V* (*t-target t*)) $\wedge$
converge M2 β (*V* (*t-target t*)) $\wedge$
α @ [(*t-input t*, *t-output t*)] $\in$ *set T3* $\wedge$
$\beta \in$ *set T3*)
$\wedge$ *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)))
(**is** $((L\ M1 \cap set\ ?TS') = (L\ M2 \cap set\ ?TS')) \Longrightarrow (\bigwedge t\ .\ t \in transitions\ M1 \Longrightarrow t\text{-}source\ t \in reachable\text{-}states\ M1 \Longrightarrow ?P1\ t\ T3 \wedge ?P2\ t))$
proof −
fix *t* **assume** $t \in transitions\ M1$ **and** *t-source t* $\in$ *reachable-states M1* **and** ((*L*

M1 ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))
then consider *t* ∈ *sc-covered-transitions* | *t* ∈ *list.set unverified-transitions*
unfolding *sc-covered-transitions-alt-def unverified-transitions-alt-def*
by *blast*
then show *?P1 t T3* ∧ *?P2 t*
proof *cases*
case *1*

have (*V* (*t-source t*)) ∈ *L M1*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› ‹*t-source t* ∈ *reachable-states M1*›]
by *auto*
then have *p3*: *converge M1* (*V* (*t-source t*)) (*V* (*t-source t*))
by *auto*

have (*V* (*t-source t*)) ∈ *L M2*
using *passes-T1*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›] *T1-state-cover* ‹*t-source t* ∈ *reachable-states M1*› ‹(*V* (*t-source t*)) ∈ *L M1*›
by (*metis IntI image-subset-iff inf.cobounded1 subsetD*)
then have *p4*: *converge M2* (*V* (*t-source t*)) (*V* (*t-source t*))
by *auto*

have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*OF* ‹*t-source t* ∈ *reachable-states M1*› ‹*t* ∈ *transitions M1*›]
by *auto*
then have (*V* (*t-target t*)) ∈ *L M1*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*›]
by *auto*
then have *p5*: *converge M1* (*V* (*t-target t*)) (*V* (*t-target t*))
by *auto*

have (*V* (*t-target t*)) ∈ *L M2*
using *passes-T1*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›] *T1-state-cover* ‹*t-target t* ∈ *reachable-states M1*› ‹(*V* (*t-target t*)) ∈ *L M1*›
by *blast*
then have *p6*: *converge M2* (*V* (*t-target t*)) (*V* (*t-target t*))
by *auto*

have *p8*: (*V* (*t-target t*)) ∈ *set T3*
using *T3-state-cover* ‹*t-target t* ∈ *reachable-states M1*›
by *auto*
then have *p7*: (*V* (*t-source t*)) @ [(*t-input t*, *t-output t*)] ∈ *set T3*
using *1*
unfolding *sc-covered-transitions-alt-def*
by *auto*

have *?P2 t*

using *T1-covered-transitions-conv*[*OF passes-T1*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›] *1*]
by *auto*
then show *?thesis*
using *p3 p4 p5 p6 p7 p8*
by *blast*
next
case *2*

show *?thesis*
using *T2-cover*[*OF passes-T2*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›] *2*] ‹*set T2* ⊆ *set T3*›
unfolding *covers-unverified-transition*
by *blast*
qed
qed

have *T3-covers-defined-io-pairs* : ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ (⋀ *q x y q′* . *q* ∈ *reachable-states M1* ⟹ *h-obs M1 q x y* = *Some q′* ⟹
(∃ *α β*.
converge M1 α (*V q*) ∧
converge M2 α (*V q*) ∧
converge M1 β (*V q′*) ∧
converge M2 β (*V q′*) ∧
α @ [(*x*,*y*)] ∈ *set T3* ∧
β ∈ *set T3*)
∧ *converge M1* (*V q* @ [(*x*,*y*)]) (*V q′*) ∧ *converge M2* (*V q* @ [(*x*,*y*)]) (*V q′*))
(**is** ((*L M1* ∩ *set ?TS′*) = (*L M2* ∩ *set ?TS′*)) ⟹ (⋀ *q x y q′* . *q* ∈ *reachable-states M1* ⟹ *h-obs M1 q x y* = *Some q′* ⟹ *?P q x y q′*))
proof −
fix *q x y q′* **assume** *q* ∈ *reachable-states M1* **and** *h-obs M1 q x y* = *Some q′* **and** ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))
then have (*q*,*x*,*y*,*q′*) ∈ *transitions M1* **and** *t-source* (*q*,*x*,*y*,*q′*) ∈ *reachable-states M1*
using *h-obs-Some*[*OF assms*(*1*)] **by** *auto*
moreover have *converge M1* (*V q* @ [(*x*,*y*)]) (*V q′*)
using *state-cover-transition-converges*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› *calculation*]
by *auto*
ultimately show *?P q x y q′*
using *T3-covers-transitions*[*of* (*q*,*x*,*y*,*q′*), *OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›]
unfolding *fst-conv snd-conv*
by *blast*
qed

have *rstates-io-set* : *list.set rstates-io* = {(*q*,(*x*,*y*)) . *q* ∈ *reachable-states M1* ∧ *x* ∈ *inputs M1* ∧ *y* ∈ *outputs M1*}
unfolding *rstates-io rstates*
using *reachable-states-as-list-set*[*of M1*] *inputs-as-list-set*[*of M1*] *outputs-as-list-set*[*of M1*]
by *force*
then have *undefined-io-pairs-set* : *list.set undefined-io-pairs* = {(*q*,(*x*,*y*)) . *q* ∈ *reachable-states M1* ∧ *x* ∈ *inputs M1* ∧ *y* ∈ *outputs M1* ∧ *h-obs M1 q x y* = *None*}
unfolding *undefined-io-pairs*
by *auto*

have *verify-undefined-io-pair-prop* : ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ (⋀ *q x y T* . *L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) ⟹
q ∈ *reachable-states M1* ⟹ *x* ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹
V ‘ *reachable-states M1* ⊆ *set T* ⟹
∃ *α*. *converge M1 α* (*V q*) ∧
converge M2 α (*V q*) ∧
α ∈ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) ∧
α@[(*x*,*y*)] ∈ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))))
proof −
fix *q x y T*
assume *L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*)))
and *q* ∈ *reachable-states M1* **and** *x* ∈ *inputs M1* **and** *y* ∈ *outputs M1*
and *V* ‘ *reachable-states M1* ⊆ *set T*
and ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))

have *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
using *T3-state-cover* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *L M1* ∩ *Prefix-Tree.set T3* = *L M2* ∩ *Prefix-Tree.set T3*› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›
by *blast*

have *L M1* ∩ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*)) = *L M2* ∩ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*))
using ‹*L M1* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*)))›
unfolding *verify-undefined-io-pair case-prod-conv combine-set G2*
by *blast*

have (∃*α*. *converge M1 α* (*V q*) ∧

converge M2 α (*V q*) $\wedge$
$\alpha \in$ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*)) $\wedge$
α @ [(*x*, *y*)] $\in$ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*)))
using *assms*(*16*)
unfolding *verifies-io-pair-def*
using *assms*(*1*−*4*,*7*,*8*) ‹*is-state-cover-assignment M1 V*› ‹*L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*›
‹*q* $\in$ *reachable-states M1*› ‹*x* $\in$ *inputs M1*› ‹*y* $\in$ *outputs M1*›
G2-invar[*OF* ‹((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))›] ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*›
‹*L M1* $\cap$ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*append-io-pair M1 V T G2 cg-insert cg-lookup q x y*))›
by *blast*
then show $\exists$ α. *converge M1* α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
$\alpha \in$ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*))) $\wedge$
α@[(*x*,*y*)] $\in$ *set* (*verify-undefined-io-pair T* (*q*,(*x*,*y*)))
unfolding *verify-undefined-io-pair G2 case-prod-conv combine-set*
by *blast*
qed

have *T3-covers-undefined-io-pairs* : ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*)) $\Longrightarrow$ ($\bigwedge$ *q x y* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *x* $\in$ *inputs M1* $\Longrightarrow$ *y* $\in$ *outputs M1* $\Longrightarrow$ *h-obs M1 q x y* = *None* $\Longrightarrow$
($\exists$ α .
converge M1 α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
$\alpha \in$ *set T3*$\wedge$
α@[(*x*,*y*)] $\in$ *set T3*))
proof −
fix *q x y* **assume** *q* $\in$ *reachable-states M1* **and** *x* $\in$ *inputs M1* **and** *y* $\in$ *outputs M1* **and** *h-obs M1 q x y* = *None* **and** ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))

have $\bigwedge$ *q x y qxys T* . *L M1* $\cap$ *set* (*foldl verify-undefined-io-pair T qxys*) = *L M2* $\cap$ *set* (*foldl verify-undefined-io-pair T qxys*) $\Longrightarrow$ (*V* ‘ *reachable-states M1*) $\subseteq$ *set T* $\Longrightarrow$ (*q*,(*x*,*y*)) $\in$ *list.set qxys* $\Longrightarrow$ *list.set qxys* $\subseteq$ *list.set undefined-io-pairs* $\Longrightarrow$
($\exists$ α .
converge M1 α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
$\alpha \in$ *set* (*foldl verify-undefined-io-pair T qxys*)$\wedge$
α@[(*x*,*y*)] $\in$ *set* (*foldl verify-undefined-io-pair T qxys*))
(**is** $\bigwedge$ *q x y qxys T* . *?P1 qxys T* $\Longrightarrow$ (*V* ‘ *reachable-states M1*) $\subseteq$ *set T* $\Longrightarrow$ (*q*,(*x*,*y*)) $\in$ *list.set qxys* $\Longrightarrow$ *list.set qxys* $\subseteq$ *list.set undefined-io-pairs* $\Longrightarrow$ *?P2 q x y qxys T*)
proof −
fix *q x y qxys T*
assume *?P1 qxys T* **and** (*q*,(*x*,*y*)) $\in$ *list.set qxys* **and** *list.set qxys* $\subseteq$ *list.set*

undefined-io-pairs **and** (*V* ‘ *reachable-states M1*) ⊆ *set T*
then show *?P2 q x y qxys T*
proof (*induction qxys rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a qxys*)

have *set* (*foldl verify-undefined-io-pair T qxys*) ⊆ *set* (*foldl verify-undefined-io-pair T* (*qxys*@[*a*]))
using *verify-undefined-io-pair-retains-testsuite*
by *auto*
then have ∗:*L M1* ∩ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*) = *L M2* ∩ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*)
using *snoc.prems*(*1*)
by *blast*

have ∗∗: *V* ‘ *reachable-states M1* ⊆ *Prefix-Tree.set* (*foldl verify-undefined-io-pair T qxys*)
using *snoc.prems*(*4*) *verify-undefined-io-pair-folding-retains-testsuite*
by *blast*

show *?case* **proof** (*cases a* = (*q*,(*x*,*y*)))
case *True*
then have ∗∗∗: *q* ∈ *reachable-states M1*
using *snoc.prems*(*3*)
unfolding *undefined-io-pairs-set*
by *auto*

have *x* ∈ *inputs M1* **and** *y* ∈ *outputs M1*
using *snoc.prems*(*2*,*3*) **unfolding** *undefined-io-pairs-set* **by** *auto*

have ∗∗∗∗: *L M1* ∩ *set* (*verify-undefined-io-pair* (*foldl verify-undefined-io-pair T qxys*) (*q*,(*x*,*y*))) = *L M2* ∩ *set* (*verify-undefined-io-pair* (*foldl verify-undefined-io-pair T qxys*) (*q*,(*x*,*y*)))
using *snoc.prems*(*1*) **unfolding** *True* **by** *auto*

show *?thesis*
using *verify-undefined-io-pair-prop*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))› ∗∗∗∗ ∗∗∗ ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*› ∗∗]
unfolding *True*
by *auto*
next
case *False*
then have (*q*, *x*, *y*) ∈ *list.set qxys* **and** *list.set qxys* ⊆ *list.set undefined-io-pairs*
using *snoc.prems*(*2*,*3*) **by** *auto*
then show *?thesis*
using *snoc.IH*[*OF* ∗ - - *snoc.prems*(*4*)]

using ‹*set* (*foldl verify-undefined-io-pair T qxys*) ⊆ *set* (*foldl verify-undefined-io-pair T* (*qxys*@[*a*]))›
by *blast*
qed
qed
qed
moreover have *L M1* ∩ *set* (*foldl verify-undefined-io-pair T2 undefined-io-pairs*) = *L M2* ∩ *set* (*foldl verify-undefined-io-pair T2 undefined-io-pairs*)
using ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ *L M1* ∩ *set T3* = *L M2* ∩ *set T3*› ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))›
unfolding *T3 T2* **.**
moreover have (*V* ‘ *reachable-states M1*) ⊆ *set T2*
using *T1-state-cover T2 T2-invar-1 passes-T2* **by** *fastforce*
moreover have (*q*,(*x*,*y*)) ∈ *list.set undefined-io-pairs*
unfolding *undefined-io-pairs-set*
using ‹*q* ∈ *reachable-states M1*› ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*› ‹*h-obs M1 q x y* = *None*›
by *blast*
ultimately show (∃ α .
converge M1 α (*V q*) ∧
converge M2 α (*V q*) ∧
α ∈ *set T3*∧
α@[(*x*,*y*)] ∈ *set T3*)
unfolding *T3 T2*
by *blast*
qed

define *TCfun* **where** *TCfun*: *TCfun* = (λ (*q*,(*x*,*y*)) . *case h-obs M1 q x y of*
None ⇒ {{α, α@[(*x*,*y*)]} | α . *converge M1* α (*V q*) ∧ *converge M2* α (*V q*) ∧ α ∈ *set T3* ∧ α@[(*x*,*y*)] ∈ *set T3*} |
Some q′ ⇒ {{α,α@[(*x*,*y*)], β} | α β . *converge M1* α (*V q*) ∧ *converge M2* α (*V q*) ∧ *converge M1* β (*V q′*) ∧ *converge M2* β (*V q′*) ∧ α @ [(*x*,*y*)] ∈ *set T3* ∧ β ∈ *set T3* ∧ *converge M1* (*V q* @ [(*x*,*y*)]) (*V q′*) ∧ *converge M2* (*V q* @ [(*x*,*y*)]) (*V q′*)})
define *TC* **where** *TC*: *TC* = *Set.insert* [] (⋃(⋃ (*TCfun* ‘ (*reachable-states M1* × (*inputs M1* × *outputs M1*)))))

have *TCfun-nonempty*: ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*)) ⟹ (⋀ *q x y* . *q* ∈ *reachable-states M1* ⟹ *x* ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹ *TCfun* (*q*,(*x*,*y*)) ≠ {})
proof −
fix *q x y* **assume** ∗:*q* ∈ *reachable-states M1* **and** ∗∗:*x* ∈ *inputs M1* **and** ∗∗∗:*y* ∈ *outputs M1* **and** ((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))

show *TCfun* $(q,(x,y)) \neq \{\}$
proof (*cases h-obs M1 q x y*)
case *None*
then have *TCfun* $(q,(x,y)) = \{\{\alpha,\ \alpha\ @\ [(x,\ y)]\}\ |\alpha.$ *converge M1* α $(V\ q) \wedge$ *converge M2* α $(V\ q) \wedge \alpha \in$ *set T3* $\wedge\ \alpha\ @\ [(x,\ y)] \in$ *set T3*$\}$
unfolding *TCfun* **by** *auto*
moreover have $\{\{\alpha,\ \alpha\ @\ [(x,\ y)]\}\ |\alpha.$ *converge M1* α $(V\ q) \wedge$ *converge M2* α $(V\ q) \wedge \alpha \in$ *set T3* $\wedge\ \alpha\ @\ [(x,\ y)] \in$ *set T3*$\} \neq \{\}$
using *T3-covers-undefined-io-pairs*[*OF* ‹$((L\ M1 \cap$ *set ?TS*$) = (L\ M2 \cap$ *set ?TS*$))$› $*$ $**$ $***$ *None*]
by *blast*
ultimately show *?thesis*
by *blast*
next
case (*Some* q')
then have *TCfun* $(q,(x,y)) = \{\{\alpha,\alpha@[(x,y)],\ \beta\}\ |\ \alpha\ \beta\ .$ *converge M1* α $(V\ q) \wedge$ *converge M2* α $(V\ q) \wedge$ *converge M1* β $(V\ q') \wedge$ *converge M2* β $(V\ q') \wedge \alpha\ @\ [(x,y)] \in$ *set T3* $\wedge\ \beta \in$ *set T3* $\wedge$ *converge M1* $(V\ q\ @\ [(x,y)])$ $(V\ q') \wedge$ *converge M2* $(V\ q\ @\ [(x,y)])$ $(V\ q')\}$
using *TCfun* **by** *auto*
moreover have $\{\{\alpha,\alpha@[(x,y)],\ \beta\}\ |\ \alpha\ \beta\ .$ *converge M1* α $(V\ q) \wedge$ *converge M2* α $(V\ q) \wedge$ *converge M1* β $(V\ q') \wedge$ *converge M2* β $(V\ q') \wedge \alpha\ @\ [(x,y)] \in$ *set T3* $\wedge\ \beta \in$ *set T3* $\wedge$ *converge M1* $(V\ q\ @\ [(x,y)])$ $(V\ q') \wedge$ *converge M2* $(V\ q\ @\ [(x,y)])$ $(V\ q')\} \neq \{\}$
using *T3-covers-defined-io-pairs*[*OF* ‹$((L\ M1 \cap$ *set ?TS*$) = (L\ M2 \cap$ *set ?TS*$))$› $*$ *Some*]
by *blast*
ultimately show *?thesis*
by *blast*
qed
qed

have *TC-in-T3*: $TC \subseteq$ *set T3*
proof
fix α **assume** $\alpha \in TC$
show $\alpha \in$ *set T3*
proof (*cases* $\alpha = []$)
case *True*
then show *?thesis*
using *T3-state-cover* ‹*is-state-cover-assignment M1 V*› *reachable-states-initial*[*of M1*]
by *auto*
next
case *False*
then obtain $q\ x\ y$ **where** $q \in$ *reachable-states M1*
and $x \in$ *inputs M1*
and $y \in$ *outputs M1*
and $\alpha \in \bigcup$ (*TCfun* $(q,(x,y))$)

using ‹$\alpha \in TC$› **unfolding** *TC*
by *auto*

show $\alpha \in$ *set T3*
using ‹$\alpha \in \bigcup$ (*TCfun* (*q*,(*x*,*y*)))› *set-prefix*[*of* α [(*x*,*y*)] *T3*] **unfolding** *TCfun*
by (*cases h-obs M1 q x y*;*auto*)
qed
qed

have *TC-is-transition-cover* : ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*)) $\Longrightarrow$ *transition-cover M1 TC*
proof −
assume ((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))

have $\bigwedge$ *q x y* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *x* $\in$ *inputs M1* $\Longrightarrow$ *y* $\in$ *outputs M1* $\Longrightarrow \exists \alpha.\ \alpha \in TC \wedge \alpha$ @ [(*x*, *y*)] $\in TC \wedge \alpha \in L\ M1 \wedge$ *after-initial M1* $\alpha = q$
proof −
fix *q x y* **assume** *q* $\in$ *reachable-states M1*
and *x* $\in$ *inputs M1*
and *y* $\in$ *outputs M1*
then have (*q*,(*x*,*y*)) $\in$ (*reachable-states M1* $\times$ *FSM.inputs M1* $\times$ *FSM.outputs M1*)
by *blast*

show $\exists \alpha.\ \alpha \in TC \wedge \alpha$ @ [(*x*, *y*)] $\in TC \wedge \alpha \in L\ M1 \wedge$ *after-initial M1* $\alpha = q$
proof (*cases h-obs M1 q x y*)
case *None*
then have *TCfun* (*q*,(*x*,*y*)) = {{α, α @ [(*x*, *y*)]} |α. *converge M1* α (*V q*) $\wedge$ *converge M2* α (*V q*) $\wedge \alpha \in$ *set T3* $\wedge \alpha$ @ [(*x*, *y*)] $\in$ *set T3*}
unfolding *TCfun* **by** *auto*
then obtain α **where** *converge M1* α (*V q*) **and** $\alpha \in \bigcup$ (*TCfun* (*q*,(*x*,*y*))) $\wedge \alpha$ @ [(*x*, *y*)] $\in \bigcup$ (*TCfun* (*q*,(*x*,*y*)))
using *TCfun-nonempty*[*OF* ‹((*L M1* $\cap$ *set ?TS*) = (*L M2* $\cap$ *set ?TS*))› ‹*q* $\in$ *reachable-states M1*› ‹*x* $\in$ *inputs M1*› ‹*y* $\in$ *outputs M1*›]
by *auto*
then have *after-initial M1* $\alpha = q$
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› ‹*q* $\in$ *reachable-states M1*›]
using *convergence-minimal*[*OF assms*(*3*,*1*)]
by (*metis converge.elims*(*2*))
then have $\exists \alpha.\ \alpha \in \bigcup$ (*TCfun* (*q*,(*x*,*y*))) $\wedge \alpha$ @ [(*x*, *y*)] $\in \bigcup$ (*TCfun* (*q*,(*x*,*y*))) $\wedge \alpha \in L\ M1 \wedge$ *after-initial M1* $\alpha = q$
using ‹$\alpha \in \bigcup$ (*TCfun* (*q*,(*x*,*y*))) $\wedge \alpha$ @ [(*x*, *y*)] $\in \bigcup$ (*TCfun* (*q*,(*x*,*y*)))›
using ‹*converge M1* α (*V q*)› *converge.elims*(*2*) **by** *blast*
moreover have $\bigcup$ (*TCfun* (*q*,(*x*,*y*))) $\subseteq TC$

unfolding *TC* **using** ‹(*q*,(*x*,*y*)) ∈ (*reachable-states M1* × *FSM.inputs M1* × *FSM.outputs M1*)›
by *blast*
ultimately show *?thesis*
by *blast*
next
case (*Some q′*)
then have *TCfun* (*q*,(*x*,*y*)) = {{α,α@[(*x*,*y*)], β} | α β . *converge M1* α (*V q*) ∧ *converge M2* α (*V q*) ∧ *converge M1* β (*V q′*) ∧ *converge M2* β (*V q′*) ∧ α @ [(*x*,*y*)] ∈ *set T3* ∧ β ∈ *set T3* ∧ *converge M1* (*V q* @ [(*x*,*y*)]) (*V q′*) ∧ *converge M2* (*V q* @ [(*x*,*y*)]) (*V q′*)}
using *TCfun*
by *auto*
moreover obtain *S* **where** *S* ∈ *TCfun* (*q*,(*x*,*y*))
using *TCfun-nonempty*[*OF* ‹((*L M1* ∩ *set ?TS*) = (*L M2* ∩ *set ?TS*))› ‹*q* ∈ *reachable-states M1*› ‹*x* ∈ *inputs M1*› ‹*y* ∈ *outputs M1*›]
by *blast*
ultimately obtain α **where** *converge M1* α (*V q*) **and** α ∈ *S* ∧ α @ [(*x*, *y*)] ∈ *S*
by *auto*
then have *after-initial M1* α = *q*
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› ‹*q* ∈ *reachable-states M1*›]
using *convergence-minimal*[*OF assms*(*3*,*1*)]
by (*metis converge.elims*(*2*))
moreover have α ∈ ⋃ (*TCfun* (*q*,(*x*,*y*))) ∧ α @ [(*x*, *y*)] ∈ ⋃ (*TCfun* (*q*,(*x*,*y*)))
using ‹α ∈ *S* ∧ α @ [(*x*, *y*)] ∈ *S*› ‹*S* ∈ *TCfun* (*q*,(*x*,*y*))›
by *auto*
ultimately have ∃α. α ∈ ⋃ (*TCfun* (*q*,(*x*,*y*))) ∧ α @ [(*x*, *y*)] ∈ ⋃ (*TCfun* (*q*,(*x*,*y*))) ∧ α ∈ *L M1* ∧ *after-initial M1* α = *q*
using ‹α ∈ ⋃ (*TCfun* (*q*,(*x*,*y*))) ∧ α @ [(*x*, *y*)] ∈ ⋃ (*TCfun* (*q*,(*x*,*y*)))›
using ‹*converge M1* α (*V q*)› *converge.elims*(*2*) **by** *blast*
moreover have ⋃ (*TCfun* (*q*,(*x*,*y*))) ⊆ *TC*
unfolding *TC* **using** ‹(*q*,(*x*,*y*)) ∈ (*reachable-states M1* × *FSM.inputs M1* × *FSM.outputs M1*)›
by *blast*
ultimately show *?thesis*
by *blast*
qed
qed
then show *?thesis*
unfolding *transition-cover-def*
by *blast*
qed

have *TC-preserves-convergence*: *preserves-convergence M1 M2 TC*
proof −
have ⋀ α β . α ∈ *L M1* ∩ *TC* ⟹ β ∈ *L M1* ∩ *TC* ⟹ *converge M1* α β

$\Longrightarrow$ *converge M2* α β
proof −
fix α β **assume** $\alpha \in L$ *M1* $\cap$ *TC*
$\beta \in L$ *M1* $\cap$ *TC*
converge M1 α β

have $*$: $\bigwedge \alpha$. $\alpha \in L$ *M1* $\Longrightarrow \alpha \in$ *TC* $\Longrightarrow \exists$ q . $q \in$ *reachable-states M1* $\wedge$ *converge M1* α (V q) $\wedge$ *converge M2* α (V q)
proof −
fix α **assume** $\alpha \in L$ *M1* **and** $\alpha \in$ *TC*

show $\exists$ q . $q \in$ *reachable-states M1* $\wedge$ *converge M1* α (V q) $\wedge$ *converge M2* α (V q)
proof (*cases* $\alpha = []$)
case *True*

then have V (*initial M1*) = α
using ‹*is-state-cover-assignment M1 V*› *reachable-states-initial*[*of M1*]
by *auto*
then have *converge M1* α (V (*initial M1*)) **and** *converge M2* α (V (*initial M1*))
unfolding *True* **by** *auto*
then show *?thesis*
using *reachable-states-initial*[*of M1*]
by *auto*
next
case *False*
then have $\alpha \in (\bigcup(\bigcup$ (*TCfun* ‘ (*reachable-states M1* $\times$ (*inputs M1* $\times$ *outputs M1*)))))
using ‹$\alpha \in$ *TC*›
unfolding *TC*
by *blast*
then obtain q x y **where** $q \in$ *reachable-states M1*
and $x \in$ *inputs M1*
and $y \in$ *outputs M1*
and $\alpha \in \bigcup$ (*TCfun* (q,(x,y)))
unfolding *TC* **by** *auto*

show $\exists$ q . $q \in$ *reachable-states M1* $\wedge$ *converge M1* α (V q) $\wedge$ *converge M2* α (V q)
proof (*cases h-obs M1 q x y*)
case *None*
then have *TCfun* (q,(x,y)) = {{α, α @ [(x, y)]} |α. *converge M1* α (V q) $\wedge$ *converge M2* α (V q) $\wedge$ $\alpha \in$ *set T3* $\wedge$ α @ [(x, y)] $\in$ *set T3*}
unfolding *TCfun* **by** *auto*
then obtain α' **where** $\alpha \in \{\alpha', \alpha'$ @ [(x, y)]}
and *converge M1* α' (V q)
and *converge M2* α' (V q)
using ‹$\alpha \in \bigcup$ (*TCfun* (q,(x,y)))›

by *auto*

have $[(x,y)] \notin$ *LS M1 q*
using *None* **unfolding** *h-obs-None*[*OF assms*(*1*)] *LS-single-transition*
by *auto*
moreover have *after-initial M1* $\alpha' = q$
using ‹*converge M1* α' (*V q*)›
using *state-cover-assignment-after*[*OF assms*(*1*) ‹*is-state-cover-assignment M1 V*› ‹*q* ∈ *reachable-states M1*›]
using *convergence-minimal*[*OF assms*(*3*,*1*) - -]
by (*metis converge.elims*(*2*))
ultimately have α' @ [(*x*, *y*)] $\notin$ *L M1*
using *after-language-iff*[*OF assms*(*1*), *of* α' *initial M1* [(*x*,*y*)]] ‹*converge M1* α' (*V q*)›
by (*meson converge.elims*(*2*))
then have $\alpha' = \alpha$
using ‹$\alpha \in \{\alpha', \alpha'$ @ [(*x*, *y*)]}› ‹$\alpha \in$ *L M1*›
by *blast*
then show *?thesis*
using ‹*q* ∈ *reachable-states M1*› ‹*converge M1* α' (*V q*)› ‹*converge M2* α' (*V q*)›
by *blast*
next
case (*Some* q')
then have $q' \in$ *reachable-states M1*
unfolding *h-obs-Some*[*OF assms*(*1*)]
using *reachable-states-next*[*OF* ‹*q* ∈ *reachable-states M1*›, *of* (*q*,*x*,*y*,q')]
by *auto*

have *TCfun* (*q*,(*x*,*y*)) = {{α,α@[(*x*,*y*)], β} | α β . *converge M1* α (*V q*) ∧ *converge M2* α (*V q*) ∧ *converge M1* β (*V* q') ∧ *converge M2* β (*V* q') ∧ α @ [(*x*,*y*)] ∈ *set T3* ∧ β ∈ *set T3* ∧ *converge M1* (*V q* @ [(*x*,*y*)]) (*V* q') ∧ *converge M2* (*V q* @ [(*x*,*y*)]) (*V* q')}
using *Some TCfun*
by *auto*
then obtain α' β **where** $\alpha \in \{\alpha', \alpha'$@[(*x*,*y*)], $\beta\}$
and *converge M1* α' (*V q*)
and *converge M2* α' (*V q*)
and *converge M1* β (*V* q')
and *converge M2* β (*V* q')
and *converge M1* (*V q* @ [(*x*,*y*)]) (*V* q')
and *converge M2* (*V q* @ [(*x*,*y*)]) (*V* q')
using ‹$\alpha \in \bigcup$ (*TCfun* (*q*,(*x*,*y*)))›
by *auto*

then consider $\alpha = \alpha'$ | $\alpha = \alpha'$@[(*x*,*y*)] | $\alpha = \beta$
by *blast*
then show *?thesis* **proof** *cases*
case *1*

then show *?thesis*
using ‹*q* ∈ *reachable-states M1*› ‹*converge M1* α' (*V q*)› ‹*converge M2* α' (*V q*)›
by *blast*
next
case *2*

have *converge M1* (α'@[(*x*,*y*)]) (*V q* @ [(*x*,*y*)])
using ‹*converge M1* α' (*V q*)› ‹*converge M1* (*V q* @ [(*x*,*y*)]) (*V q'*)›
using *converge-append*[*OF assms*(*1*), *of V q* α' [(*x*,*y*)]]
by *auto*
then have *converge M1* (α'@[(*x*,*y*)]) (*V q'*)
using ‹*converge M1* β (*V q'*)› ‹*converge M1* (*V q* @ [(*x*,*y*)]) (*V q'*)›
by *auto*

have *converge M2* (α'@[(*x*,*y*)]) (*V q* @ [(*x*,*y*)])
using ‹*converge M2* α' (*V q*)› ‹*converge M2* (*V q* @ [(*x*,*y*)]) (*V q'*)›
using *converge-append*[*OF assms*(*2*), *of V q* α' [(*x*,*y*)]]
by *auto*
then have *converge M2* (α'@[(*x*,*y*)]) (*V q'*)
using ‹*converge M2* β (*V q'*)› ‹*converge M2* (*V q* @ [(*x*,*y*)]) (*V q'*)›
by *auto*

show *?thesis*
using *2* ‹*q'* ∈ *reachable-states M1*› ‹*converge M1* (α'@[(*x*,*y*)]) (*V q'*)› ‹*converge M2* (α'@[(*x*,*y*)]) (*V q'*)›
by *auto*
next
case *3*
then show *?thesis*
using ‹*converge M1* β (*V q'*)› ‹*converge M2* β (*V q'*)› ‹*q'* ∈ *reachable-states M1*›
by *blast*
qed
qed
qed
qed

obtain *q* **where** *q* ∈ *reachable-states M1* **and** *converge M1* α (*V q*) **and** *converge M2* α (*V q*)
using ∗ ‹α ∈ *L M1* ∩ *TC*›
by *blast*

obtain *q'* **where** *q'* ∈ *reachable-states M1* **and** *converge M1* β (*V q'*) **and** *converge M2* β (*V q'*)
using ∗ ‹β ∈ *L M1* ∩ *TC*›
by *blast*

have *converge M1* (*V q*) (*V q'*)

```
      using ‹converge M1 α (V q)› ‹converge M1 β (V q')› ‹converge M1 α β›
      by auto
    then have q = q'
      using convergence-minimal[OF assms(3,1), of V q V q']
    unfolding state-cover-assignment-after[OF assms(1) ‹is-state-cover-assignment
M1 V› ‹q ∈ reachable-states M1›]
              state-cover-assignment-after[OF assms(1) ‹is-state-cover-assignment
M1 V› ‹q' ∈ reachable-states M1›]
      by auto
    then have V q = V q'
      by auto
    then show converge M2 α β
      using ‹converge M2 α (V q)› ‹converge M2 β (V q')›
      by auto
  qed

  then show ?thesis
    unfolding preserves-convergence.simps
    by blast
 qed

 have [] ∈ TC
  unfolding TC by blast

 show ((L M1 ∩ set ?TS) = (L M2 ∩ set ?TS)) ⟹ L M1 = L M2
  using initialised-convergence-preserving-transition-cover-is-complete[OF assms(1−4,7,8)

                                                                ‹((L M1 ∩ set ?TS)
= (L M2 ∩ set ?TS)) ⟹ L M1 ∩ set T3 = L M2 ∩ set T3›
                                                                  TC-in-T3
                                                              TC-is-transition-cover
                                                                 ‹[] ∈ TC›
                                                            TC-preserves-convergence]
  by assumption


 show finite-tree ?TS
  using T2 T2-finite T3 verify-undefined-io-pair-folding-retains-finiteness
  by (simp add: ‹?TS = T3›)
qed

end
```

20 Pair-Framework

This theory defines the Pair-Framework and provides completeness properties.

theory *Pair-Framework*

imports *H-Framework*
begin

20.1 Classical H-Condition

definition *satisfies-h-condition* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *state-cover-assignment* $\Rightarrow$ $('b \times 'c)$ *list set* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
satisfies-h-condition M V T m = (*let*
Π = (V ‘ *reachable-states* M);
n = *card* (*reachable-states* M);
$\mathcal{X} = \lambda\ q\ .\ \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m-n \wedge x \in inputs\ M \wedge y \in outputs\ M\}$;
$A = \Pi \times \Pi$;
$B = \Pi \times \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \mathcal{X}\ q\}$;
$C = (\bigcup\ q \in reachable\text{-}states\ M\ .\ \bigcup\ \tau \in \mathcal{X}\ q\ .\ \{\ (V\ q)\ @\ \tau' \mid \tau'\ .\ \tau' \in list.set\ (prefixes\ \tau)\} \times \{(V\ q)@\tau\})$
in
is-state-cover-assignment M V
$\wedge\ \Pi \subseteq T$
$\wedge\ \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \mathcal{X}\ q\} \subseteq T$
$\wedge\ (\forall\ (\alpha,\beta) \in A \cup B \cup C\ .\ \alpha \in L\ M \longrightarrow$
$\beta \in L\ M \longrightarrow$
$after\text{-}initial\ M\ \alpha \neq after\text{-}initial\ M\ \beta \longrightarrow$
$(\exists\ \omega\ .\ \alpha@\omega \in T \wedge$
$\beta@\omega \in T \wedge$
$distinguishes\ M\ (after\text{-}initial\ M\ \alpha)\ (after\text{-}initial\ M\ \beta)\ \omega)))$

lemma *h-condition-satisfies-abstract-h-condition* :
assumes *observable* M
and *observable* I
and *minimal* M
and $size\ I \leq m$
and $m \geq size\text{-}r\ M$
and $inputs\ I = inputs\ M$
and $outputs\ I = outputs\ M$
and *satisfies-h-condition* M V T m
and $(L\ M \cap T = L\ I \cap T)$
shows *satisfies-abstract-h-condition* M I V m
proof −

define Π **where** Π: Π = (V ‘ *reachable-states* M)
define n **where** n: $n = size\text{-}r\ M$
define $\mathcal{X}$ **where** $\mathcal{X}$: $\mathcal{X} = (\lambda\ q\ .\ \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m-n \wedge x \in inputs\ M \wedge y \in outputs\ M\})$
define A **where** A: $A = \Pi \times \Pi$
define B **where** B: $B = \Pi \times \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \mathcal{X}\ q\}$
define C **where** C: $C = (\bigcup\ q \in reachable\text{-}states\ M\ .\ \bigcup\ \tau \in \mathcal{X}\ q\ .\ \{\ (V\ q)\ @$

$\tau' \mid \tau' . \tau' \in$ *list.set* (*prefixes* τ)} $\times$ {(V q)@τ})

have *satisfies-h-condition* M V T m = (*is-state-cover-assignment* M V
$\wedge$ $\Pi \subseteq T$
$\wedge$ { (V q) @ τ | q τ . $q \in$ *reachable-states* M $\wedge$ $\tau \in \mathcal{X}$ q} $\subseteq T$
$\wedge$ ($\forall$ (α,β) $\in A \cup B \cup C$. $\alpha \in L$ M $\longrightarrow$
$\beta \in L$ M $\longrightarrow$
after-initial M $\alpha \neq$ *after-initial* M β $\longrightarrow$
($\exists$ ω . α@$\omega \in T$ $\wedge$
β@$\omega \in T$ $\wedge$
distinguishes M (*after-initial* M α) (*after-initial* M β) ω)))
unfolding *satisfies-h-condition-def Let-def* Π n $\mathcal{X}$ A B C
by *auto*

then have *is-state-cover-assignment* M V
and $\Pi \subseteq T$
and { (V q) @ τ | q τ . $q \in$ *reachable-states* M $\wedge$ $\tau \in \mathcal{X}$ q} $\subseteq T$
and *distinguishing-tests*: $\bigwedge$ α β . (α,β) $\in A \cup B \cup C \Longrightarrow$
$\alpha \in L$ $M \Longrightarrow$
$\beta \in L$ $M \Longrightarrow$
after-initial M $\alpha \neq$ *after-initial* M $\beta \Longrightarrow$
($\exists$ ω . α@$\omega \in T$ $\wedge$
β@$\omega \in T$ $\wedge$
distinguishes M (*after-initial* M α) (*after-initial* M β) ω)
using ‹*satisfies-h-condition* M V T m› **by** *blast+*

have $\Pi \subseteq L$ I **and** $\Pi \subseteq L$ M
using ‹$\Pi \subseteq T$› ‹Π = (V ‘ *reachable-states* M)› ‹L M $\cap$ T = L I $\cap$ T›
state-cover-assignment-language[*OF* ‹*is-state-cover-assignment* M V›] **by** *blast+*

have ($\bigwedge$ q γ . $q \in$ *reachable-states* $M \Longrightarrow$ *length* $\gamma \leq$ *Suc* ($m-$*size-r* M) $\Longrightarrow$ *list.set* $\gamma \subseteq$ *inputs* M $\times$ *outputs* M $\Longrightarrow$ *butlast* $\gamma \in$ *LS* M q $\Longrightarrow$ (L M $\cap$ (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}) = L I $\cap$ (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})) $\wedge$ (*preserves-divergence* M I (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))) $\Longrightarrow$
satisfies-abstract-h-condition M I V m
unfolding *satisfies-abstract-h-condition-def Let-def*
by *blast*
moreover have ($\bigwedge$ q γ . $q \in$ *reachable-states* $M \Longrightarrow$ *length* $\gamma \leq$ *Suc* ($m-$*size-r* M) $\Longrightarrow$ *list.set* $\gamma \subseteq$ *inputs* M $\times$ *outputs* M $\Longrightarrow$ *butlast* $\gamma \in$ *LS* M q $\Longrightarrow$ (L M $\cap$ (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}) = L I $\cap$ (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})) $\wedge$ (*preserves-divergence* M I (V ‘ *reachable-states* M $\cup$ {V q @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})))
proof −

fix q γ
assume *a1*: $q \in$ *reachable-states M*
and *a2*: *length* $\gamma \leq$ *Suc* ($m-$*size-r M*)
and *a3*: *list.set* $\gamma \subseteq$ *inputs M* $\times$ *outputs M*
and *a4*: *butlast* $\gamma \in$ *LS M q*

have $\{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\} \subseteq \{V\ q\} \cup \{V\ q\ @\ \tau \mid \tau.\ \tau \in \mathcal{X}\ q\}$
proof
fix v **assume** $v \in \{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\}$
then obtain w **where** $v = V\ q\ @\ w$ **and** $w \in$ *list.set* (*prefixes* γ)
by *blast*

show $v \in \{V\ q\} \cup \{V\ q\ @\ \tau \mid \tau.\ \tau \in \mathcal{X}\ q\}$
proof (*cases w rule*: *rev-cases*)
case *Nil*
show *?thesis* **unfolding** ‹$v = V\ q\ @\ w$› *Nil* Π **using** *a1* **by** *auto*
next
case (*snoc* w' *xy*)

obtain w'' **where** $\gamma = w'@[xy]@w''$
using ‹$w \in$ *list.set* (*prefixes* γ)›
unfolding *prefixes-set snoc* **by** *auto*

obtain w''' x y **where** $\gamma = (w'@w''')@[(x,y)]$
proof (*cases* w'' *rule*: *rev-cases*)
case *Nil*
show *?thesis*
using *that*[*of* [] *fst xy snd xy*]
unfolding ‹$\gamma = w'@[xy]@w''$› *Nil* **by** *auto*
next
case (*snoc* w''' xy')
show *?thesis*
using *that*[*of* $[xy]@w'''$ *fst* xy' *snd* xy']
unfolding ‹$\gamma = w'@[xy]@w''$› *snoc* **by** *auto*
qed
then have *butlast* $\gamma = w'@w'''$
using *butlast-snoc* **by** *metis*

have $w' \in$ *LS M q*
using *a4* **unfolding** ‹$v = V\ q\ @\ w$› ‹*butlast* $\gamma = w'@w'''$›
using *language-prefix* **by** *metis*
moreover have *length* $w' \leq m -$ *size-r M*
using *a2* **unfolding** ‹$v = V\ q\ @\ w$› ‹$\gamma = (w'@w''')@[(x,y)]$› **by** *auto*
moreover have *fst xy* $\in$ *FSM.inputs M* $\wedge$ *snd xy* $\in$ *FSM.outputs M*
using *a3* **unfolding** ‹$v = V\ q\ @\ w$› ‹$\gamma = w'@[xy]@w''$› **by** *auto*
ultimately have $w'@[(\mathit{fst}\ xy,\ \mathit{snd}\ xy)] \in \mathcal{X}\ q$
unfolding *snoc* $\mathcal{X}$ *n* **by** *blast*

then have $w \in \mathcal{X}\ q$
unfolding *snoc* **by** *auto*
then show *?thesis*
unfolding ‹$v = V\ q\ @\ w$› **using** *a1* **by** *blast*
qed
qed

have *preserves-divergence* $M\ I\ (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
proof −
have $\bigwedge\ \alpha\ \beta\ .\ \alpha \in L\ M \Longrightarrow \alpha \in (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\}) \Longrightarrow \beta \in L\ M \Longrightarrow \beta \in\ (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\}) \Longrightarrow$
$\neg$ *converge* $M\ \alpha\ \beta \Longrightarrow \neg$ *converge* $I\ \alpha\ \beta$
proof −
fix $\alpha\ \beta$
assume $\alpha \in L\ M$
and $\alpha \in (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
and $\beta \in L\ M$
and $\beta \in\ (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
and $\neg$ *converge* $M\ \alpha\ \beta$
then have *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta$
by *auto*
then have $\alpha \neq \beta$
by *auto*

obtain $v\ w$ **where** $\{v,w\} = \{\alpha,\beta\}$ **and** $*$:$(v \in \Pi \wedge w \in \Pi)$
$\vee\ (v \in \Pi \wedge w \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
$\vee\ (v \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\} \wedge w \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
using ‹$\alpha \in (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$›
‹$\beta \in (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$›
by *blast*

from $*$ **consider** $(v \in \Pi \wedge w \in \Pi)$ |
$(v \in \Pi \wedge w \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$ |
$(v \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\} \wedge w \in \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\})$
by *blast*
then have $(v,w) \in A \cup B \cup C \vee (w,v) \in A \cup B \cup C$
proof *cases*
case *1*
then show *?thesis* **unfolding** A **by** *blast*
next
case *2*
then show *?thesis*
using ‹$\{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* $(\textit{prefixes}\ \gamma)\} \subseteq \{V\ q\} \cup \{V\ q\ @\ \tau \mid \tau.\ \tau \in \mathcal{X}\ q\}$› *a1*
unfolding $A\ B\ \Pi$

by *blast*
next
case *3*

then obtain $io\ io'$ **where** $v = V\ q\ @\ io$ **and** $io \in list.set\ (prefixes\ \gamma)$
and $w = V\ q\ @\ io'$ **and** $io' \in list.set\ (prefixes\ \gamma)$
by *auto*

have $v \neq w$
using ‹$\{v,w\} = \{\alpha,\beta\}$› ‹$\alpha \neq \beta$› **by** *force*
then have $length\ io \neq length\ io'$
using ‹$io \in list.set\ (prefixes\ \gamma)$› ‹$io' \in list.set\ (prefixes\ \gamma)$›
unfolding ‹$v = V\ q\ @\ io$› ‹$w = V\ q\ @\ io'$› *prefixes-set*
by *force*

have $io \in list.set\ (prefixes\ io') \vee io' \in list.set\ (prefixes\ io)$
using *prefixes-prefixes*[*OF* ‹$io \in list.set\ (prefixes\ \gamma)$› ‹$io' \in list.set\ (prefixes\ \gamma)$›] **.**
then obtain $u\ u'$ **where** $\{u,u@u'\} = \{io,io'\}$
and $u \in list.set\ (prefixes\ (u@u'))$
unfolding *prefixes-set* **by** *auto*

have $(u,u@u') = (io,io') \vee (u,u@u') = (io',io)$
using ‹$\{u,u@u'\} = \{io,io'\}$›
by (*metis empty-iff insert-iff*)

have $u \neq u@u'$
using ‹$length\ io \neq length\ io'$› ‹$\{u,u@u'\} = \{io,io'\}$› **by** *force*
then have $u@u' \neq []$
by *auto*
moreover have $\bigwedge\ w\ .\ w \neq [] \Longrightarrow w \in list.set\ (prefixes\ \gamma) \Longrightarrow w \in \mathcal{X}\ q$
using ‹$\{V\ q\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\} \subseteq \{V\ q\} \cup \{V\ q\ @\ \tau\ |\ \tau.\ \tau \in \mathcal{X}\ q\}$›
by *auto*
moreover have $u@u' \in list.set\ (prefixes\ \gamma)$
using ‹$(u,u@u') = (io,io') \vee (u,u@u') = (io',io)$› ‹$io \in list.set\ (prefixes\ \gamma)$› ‹$io' \in list.set\ (prefixes\ \gamma)$› **by** *auto*
ultimately have $u@u' \in \mathcal{X}\ q$
by *blast*
then have $(V\ q\ @\ u,\ V\ q\ @\ (u@u')) \in C$
unfolding C
using *a1* ‹$u \in list.set\ (prefixes\ (u@u'))$› **by** *blast*
moreover have $(V\ q\ @\ u,\ V\ q\ @\ (u@u')) \in \{(v,w),\ (w,v)\}$
unfolding ‹$v = V\ q\ @\ io$› ‹$w = V\ q\ @\ io'$›
using ‹$(u,u@u') = (io,io') \vee (u,u@u') = (io',io)$› **by** *auto*
ultimately show *?thesis*
by *blast*
qed
moreover have $(\alpha,\beta) = (v,w) \vee (\alpha,\beta) = (w,v)$

using ‹$\{v,w\} = \{\alpha,\beta\}$›
by (*metis empty-iff insert-iff*)
ultimately consider $(\alpha,\beta) \in A \cup B \cup C \mid (\beta,\alpha) \in A \cup B \cup C$
by *blast*

then obtain ω **where** $\alpha@\omega \in T$ **and** $\beta@\omega \in T$ **and** *distinguishes M* (*after-initial M* α) (*after-initial M* β) ω
using *distinguishing-tests*[*OF* - ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$› ‹*after-initial M* α $\neq$ *after-initial M* β›]
using *distinguishing-tests*[*OF* - ‹$\beta \in L\ M$› ‹$\alpha \in L\ M$›] ‹*after-initial M* α $\neq$ *after-initial M* β›
by (*metis distinguishes-sym*)

show $\neg$ *converge I* α β
using *distinguish-diverge*[*OF assms*(*1*,*2*) ‹*distinguishes M* (*after-initial M* α) (*after-initial M* β) ω› ‹$\alpha@\omega \in T$› ‹$\beta@\omega \in T$› ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$› *assms*(*9*)]
.
qed
then show *?thesis*
unfolding *preserves-divergence.simps* **by** *blast*
qed

moreover have $(L\ M \cap (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\}) = L\ I \cap (\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\}))$
proof −
have $L\ M \cap \Pi = L\ I \cap \Pi$
using ‹$\Pi \subseteq L\ I$› ‹$\Pi \subseteq L\ M$›
by *blast*
moreover have $L\ M \cap \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\} = L\ I \cap \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\}$
using ‹$\{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\} \subseteq T$›
using *assms*(*9*)
by *blast*
ultimately have $*$:$L\ M \cap (\Pi \cup \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\}) = L\ I \cap (\Pi \cup \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\})$
by *blast*
have $**$:$(\Pi \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\}) \subseteq \Pi \cup \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in$ *reachable-states M* $\wedge\ \tau \in \mathcal{X}\ q\}$
using ‹$\{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in$ *list.set* (*prefixes* γ)$\} \subseteq \{V\ q\} \cup \{V\ q\ @\ \tau \mid \tau.\ \tau \in \mathcal{X}\ q\}$›
using *a1* **unfolding** Π **by** *blast*

have *scheme*: $\bigwedge A\ B\ C\ D\ .\ A \cap C = B \cap C \Longrightarrow D \subseteq C \Longrightarrow A \cap D = B \cap D$
by (*metis* (*no-types*, *opaque-lifting*) *Int-absorb1 inf-assoc*)
show *?thesis*
using *scheme*[*OF* $*$ $**$] **.**
qed

ultimately show $(L\ M \cap (V\ `\ reachable\text{-}states\ M \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}) = L\ I \cap (V\ `\ reachable\text{-}states\ M \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\})) \wedge (preserves\text{-}divergence\ M\ I\ (V\ `\ reachable\text{-}states\ M \cup \{V\ q\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}))$
unfolding Π **by** *blast*
qed
ultimately show *?thesis*
by *blast*
qed

lemma *h-condition-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and $size\ I \leq m$
and $m \geq size\text{-}r\ M$
and $inputs\ I = inputs\ M$
and $outputs\ I = outputs\ M$
and *satisfies-h-condition M V T m*
shows $(L\ M = L\ I) \longleftrightarrow (L\ M \cap T = L\ I \cap T)$
proof −
have *is-state-cover-assignment M V* **using** *assms*(*8*) **unfolding** *satisfies-h-condition-def Let-def* **by** *blast*
then show *?thesis*
using *h-condition-satisfies-abstract-h-condition*[*OF assms*]
using *abstract-h-condition-completeness*[*OF assms*(*1*−*7*)]
by *blast*
qed

20.2 Helper Functions

fun *language-up-to-length-with-extensions* :: $'a \Rightarrow ('a \Rightarrow 'b \Rightarrow (('c \times 'a)\ list)) \Rightarrow 'b\ list \Rightarrow ('b \times 'c)\ list\ list \Rightarrow nat \Rightarrow ('b \times 'c)\ list\ list$
where
language-up-to-length-with-extensions q hM iM ex 0 = ex |
language-up-to-length-with-extensions q hM iM ex (*Suc k*) =
ex @ *concat* (*map* (λx .*concat* (*map* ($\lambda(y,q')$. (*map* (λp . (x,y) # p)
(*language-up-to-length-with-extensions q′ hM iM ex k*)))
(*hM q x*)))
iM)

lemma *language-up-to-length-with-extensions-set* :
assumes $q \in states\ M$
shows *List.set* (*language-up-to-length-with-extensions q* (λ *q x* . *sorted-list-of-set* (*h M* (q,x))) (*inputs-as-list M*) *ex k*)
$= \{io@xy \mid io\ xy\ .\ io \in LS\ M\ q \wedge length\ io \leq k \wedge xy \in List.set\ ex\}$
(**is** *?S1 q k* = *?S2 q k*)
proof

```
  let ?hM = (λ q x . sorted-list-of-set (h M (q,x)))
  let ?iM = inputs-as-list M

  show ?S1 q k ⊆ ?S2 q k
  proof
    fix io assume io ∈ ?S1 q k
    then show io ∈ ?S2 q k
      using assms proof (induction k arbitrary: q io)
      case 0
      then obtain xy where io = []@xy
                    and xy ∈ List.set ex
                    and [] ∈ LS M q
        by auto
      then show ?case by force
    next
      case (Suc k)

      show ?case proof (cases io ∈ List.set ex)
        case True
        then have io = []@io
              and io ∈ List.set ex
              and [] ∈ LS M q
          using Suc.prems(2) by auto
        then show ?thesis by force
      next
        case False
        then obtain x where x ∈ List.set ?iM
                    and ∗: io ∈ List.set (concat (map (λ(y,q') . map (λp . (x,y) #
p)
                                                    (language-up-to-length-with-extensions
q' ?hM ?iM ex k))
                                                    (?hM q x)))
          using Suc.prems(1)
          unfolding language-up-to-length-with-extensions.simps
          by fastforce

        have x ∈ inputs M
          using ‹x ∈ List.set ?iM› inputs-as-list-set by auto

        obtain yq' where (yq') ∈ List.set (?hM q x)
                  and io ∈ List.set ((λ(y,q') . (map (λp . (x,y) # p)
                                                    (language-up-to-length-with-extensions
q' ?hM ?iM ex k))) yq')
          using concat-map-elem[OF ∗] by blast
        moreover obtain y q' where yq' = (y,q')
          using prod.exhaust-sel by blast
        ultimately have (y,q') ∈ List.set (?hM q x)
              and io ∈ List.set ((map (λp . (x,y) # p) (language-up-to-length-with-extensions
q' ?hM ?iM ex k)))
```

```
        by auto


      have (y,q') ∈ h M (q,x)
        using ‹(y,q') ∈ List.set (?hM q x)›
            by (metis empty-iff empty-set sorted-list-of-set.fold-insort-key.infinite
sorted-list-of-set.set-sorted-key-list-of-set)
      then have q' ∈ states M
            and y ∈ outputs M
            and (q,x,y,q') ∈ transitions M
          unfolding h-simps using fsm-transition-target fsm-transition-output by
auto

      obtain p where io = (x,y) # p
                and p ∈ List.set (language-up-to-length-with-extensions q' ?hM ?iM
ex k)
     using ‹io ∈ List.set ((map (λp . (x,y) # p) (language-up-to-length-with-extensions
q' ?hM ?iM ex k)))›
        by force
     then have p ∈ {io @ xy |io xy. io ∈ LS M q' ∧ length io ≤ k ∧ xy ∈ list.set
ex}
        using Suc.IH[OF - ‹q' ∈ states M›]
        by auto
      then obtain ioP xy where p = ioP@xy
                          and ioP ∈ LS M q'
                          and length ioP ≤ k
                          and xy ∈ list.set ex
        by blast

      have io = ((x,y)#ioP)@xy
        using ‹io = (x,y) # p› ‹p = ioP@xy› by auto
      moreover have ((x,y)#ioP) ∈ LS M q
         using LS-prepend-transition[OF ‹(q,x,y,q') ∈ transitions M›] ‹ioP ∈ LS
M q'›
        by auto
      moreover have length ((x,y)#ioP) ≤ Suc k
        using ‹length ioP ≤ k›
        by simp
      ultimately show ?thesis
        using ‹xy ∈ list.set ex› by blast
    qed
  qed
 qed

 show ?S2 q k ⊆ ?S1 q k
 proof
  fix io' assume io' ∈ ?S2 q k
  then show io' ∈ ?S1 q k
    using assms proof (induction k arbitrary: q io')
```

```
    case 0
    then show ?case by auto
  next
    case (Suc k)

    then obtain io xy where io' = io@xy
                        and io ∈ LS M q
                        and length io ≤ Suc k
                        and xy ∈ list.set ex
      by blast

    show ?case proof (cases io)
      case Nil
      then show ?thesis
        using ‹io ∈ LS M q› ‹xy ∈ list.set ex›
        unfolding ‹io' = io@xy›
        by auto
    next
      case (Cons a io'')

      obtain p where path M q p and p-io p = io
        using ‹io ∈ LS M q› by auto
      then obtain t p' where p = t#p'
        using Cons
        by blast

      then have t ∈ transitions M
            and t-source t = q
            and path M (t-target t) p'
        using ‹path M q p› by auto

      have a = (t-input t, t-output t)
       and p-io p' = io''
        using ‹p-io p = io› Cons ‹p = t#p'›
        by auto

      have io'' ∈ LS M (t-target t)
        using ‹p-io p' = io''› ‹path M (t-target t) p'› by auto
      moreover have length io'' ≤ k
        using ‹length io ≤ Suc k› Cons by auto
     ultimately have io''@xy ∈ {io @ xy |io xy. io ∈ LS M (t-target t) ∧ length
io ≤ k ∧ xy ∈ list.set ex}
        using ‹xy ∈ list.set ex› by blast

    moreover define f where f-def: f = (λ q . (language-up-to-length-with-extensions
q ?hM ?iM ex k))
      ultimately have io''@xy ∈ list.set (f (t-target t))
        using Suc.IH[OF - fsm-transition-target[OF ‹t ∈ transitions M›]]
        by auto
```

```
      moreover have (t-output t, t-target t) ∈ list.set (?hM q (t-input t))
      proof −
        have (h M (q,t-input t)) ⊆ image (snd ∘ snd) (transitions M)
          unfolding h-simps by force
        then have finite (h M (q,t-input t))
          using fsm-transitions-finite
          using finite-surj by blast
        moreover have (t-output t, t-target t) ∈ h M (q,t-input t)
          using ‹t ∈ transitions M› ‹t-source t = q›
          by auto
        ultimately show ?thesis
          by simp
      qed
      ultimately have a#(io''@xy) ∈ list.set (concat (map (λ(y,q') . (map (λp .
((t-input t),y) # p)
                                                    (f q')))
                                  (?hM q (t-input t))))
        unfolding ‹a = (t-input t, t-output t)›
        by force
      moreover have t-input t ∈ list.set ?iM
         using fsm-transition-input[OF ‹t ∈ transitions M›] inputs-as-list-set by
auto
        ultimately have a#(io''@xy) ∈ list.set (concat (map (λx .concat (map
(λ(y,q') . (map (λp . (x,y) # p)
                                                    (f q')))
                                  (?hM q x)))
            ?iM))
        by force
      then have a#(io''@xy) ∈ ?S1 q (Suc k)
        unfolding language-up-to-length-with-extensions.simps
        unfolding f-def by force
      then show ?thesis
        unfolding ‹io' = io@xy› Cons by simp
    qed
  qed
 qed
qed
```

```
fun h-extensions :: ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ nat ⇒ ('b
×'c) list list where
  h-extensions M q k = (let
    iM = inputs-as-list M;
    ex = map (λxy . [xy]) (List.product iM (outputs-as-list M));
    hM = (λ q x . sorted-list-of-set (h M (q,x)))
  in
    language-up-to-length-with-extensions q hM iM ex k)
```

lemma *h-extensions-set* :
assumes $q \in$ *states M*
shows *List.set* (*h-extensions M q k*) = {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *k* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
proof −

define *ex* **where** *ex*: *ex* = *map* (λ*xy* . [*xy*]) (*List.product* (*inputs-as-list M*) (*outputs-as-list M*))
then have *List.set ex* = {[*xy*] | *xy* . *xy* ∈ *list.set* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*))}
by *auto*
then have ∗: *List.set ex* = {[(*x*,*y*)] | *x y* . *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
using *inputs-as-list-set*[*of M*] *outputs-as-list-set*[*of M*]
by *auto*

have *h-extensions M q k* = *language-up-to-length-with-extensions q* (λ *q x* . *sorted-list-of-set* (*h M* (*q*,*x*))) (*inputs-as-list M*) *ex k*
unfolding *ex h-extensions.simps Let-def*
by *auto*
then have *List.set* (*h-extensions M q k*) = {*io* @ *xy* |*io xy*. *io* ∈ *LS M q* ∧ *length io* ≤ *k* ∧ *xy* ∈ *list.set ex*}
using *language-up-to-length-with-extensions-set*[*OF assms*]
by *auto*
then show *?thesis*
unfolding ∗ **by** *blast*
qed

fun *paths-up-to-length-with-targets* :: *'a* ⇒ (*'a* ⇒ *'b* ⇒ ((*'a*,*'b*,*'c*) *transition list*)) ⇒ *'b list* ⇒ *nat* ⇒ ((*'a*,*'b*,*'c*) *path* × *'a*) *list*
where
paths-up-to-length-with-targets q hM iM 0 = [([],*q*)] |
paths-up-to-length-with-targets q hM iM (*Suc k*) =
([],*q*) # (*concat* (*map* (λ*x* .*concat* (*map* (λ*t* . (*map* (λ(*p*,*q*). (*t* # *p*,*q*))
(*paths-up-to-length-with-targets* (*t-target t*) *hM iM k*)))
(*hM q x*)))
iM))

lemma *paths-up-to-length-with-targets-set* :
assumes $q \in$ *states M*
shows *List.set* (*paths-up-to-length-with-targets q* (λ *q x* . *map* (λ(*y*,*q'*) . (*q*,*x*,*y*,*q'*)) (*sorted-list-of-set* (*h M* (*q*,*x*)))) (*inputs-as-list M*) *k*)
= {(*p*, *target q p*) | *p* . *path M q p* ∧ *length p* ≤ *k*}
(**is** *?S1 q k* = *?S2 q k*)
proof
let *?hM* = (λ *q x* . *map* (λ(*y*,*q'*) . (*q*,*x*,*y*,*q'*)) (*sorted-list-of-set* (*h M* (*q*,*x*))))

```
  let ?iM = inputs-as-list M

  have hM: ⋀ q x . list.set (?hM q x) = {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions
M}
  proof −
    fix q x show list.set (?hM q x) = {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions M}
    proof
      show list.set (?hM q x) ⊆ {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions M}
      proof
        fix t assume t ∈ list.set (?hM q x)
       then obtain y q' where t = (q,x,y,q') and (y,q') ∈ list.set (sorted-list-of-set
(h M (q,x)))
          by auto
        then have (y,q') ∈ h M (q,x)
              by (metis empty-iff empty-set sorted-list-of-set.fold-insort-key.infinite
sorted-list-of-set.set-sorted-key-list-of-set)
        then show t ∈ {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions M}
          unfolding h-simps ‹t = (q,x,y,q')› by blast
      qed

      show {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions M} ⊆ list.set (?hM q x)
      proof
        fix t assume t ∈ {(q,x,y,q') | y q' . (q,x,y,q') ∈ transitions M}
        then obtain y q' where t = (q,x,y,q') and (q,x,y,q') ∈ {(q,x,y,q') | y q' .
(q,x,y,q') ∈ transitions M}
          by auto
        then have (y,q') ∈ h M (q,x)
          by auto

        have (h M (q,x)) ⊆ image (snd ∘ snd) (transitions M)
          unfolding h-simps by force
        then have finite (h M (q,x))
          using fsm-transitions-finite
          using finite-surj by blast
        then have (y,q') ∈ list.set (sorted-list-of-set (h M (q,x)))
          using ‹(y,q') ∈ h M (q,x)› by auto
        then show t ∈ list.set (?hM q x)
          unfolding ‹t = (q,x,y,q')› by auto
      qed
    qed
  qed

  show ?S1 q k ⊆ ?S2 q k
  proof
    fix pq assume pq ∈ ?S1 q k
    then show pq ∈ ?S2 q k
    using assms proof (induction k arbitrary: q pq)
      case 0
      then show ?case by force
```

```
next
  case (Suc k)

  obtain p q' where pq = (p,q')
    by fastforce

  show ?case proof (cases p)
    case Nil
    have q' = q
      using Suc.prems(1)
      unfolding ‹pq = (p,q')› Nil paths-up-to-length-with-targets.simps
      by force
    then show ?thesis
      unfolding ‹pq = (p,q')› Nil using Suc.prems(2) by auto
  next
    case (Cons t p')

    obtain x where x ∈ list.set ?iM
                and *:(t#p',q') ∈ list.set (concat (map (λt . (map (λ(p,q). (t #
p,q))
                                                (paths-up-to-length-with-targets (t-target
t) ?hM ?iM k)))
                                              (?hM q x)))
     using Suc.prems(1) unfolding ‹pq = (p,q')› Cons paths-up-to-length-with-targets.simps

      by fastforce

    have x ∈ inputs M
      using ‹x ∈ List.set ?iM› inputs-as-list-set by auto

    have t ∈ list.set (?hM q x)
       and **:(p',q') ∈ list.set (paths-up-to-length-with-targets (t-target t) ?hM
?iM k)
      using * by auto

    have t ∈ transitions M and t-source t = q
      using ‹t ∈ list.set (?hM q x)› hM by auto

    have q' = target (t-target t) p'
     and path M (t-target t) p'
     and length p' ≤ k
      using Suc.IH[OF ** fsm-transition-target[OF ‹t ∈ transitions M›]]
      by auto

    have q' = target q p
      unfolding Cons using ‹q' = target (t-target t) p'› by auto
    moreover have path M q p
    unfolding Cons using ‹path M (t-target t) p'› ‹t ∈ transitions M› ‹t-source
t = q› by auto
```

```
      moreover have length p ≤ Suc k
        unfolding Cons using ‹length p′ ≤ k› by auto
      ultimately show ?thesis
        unfolding ‹pq = (p,q′)› by blast
    qed
  qed
qed

show ?S2 q k ⊆ ?S1 q k
proof
  fix pq assume pq ∈ ?S2 q k

  obtain p q′ where pq = (p,q′)
    by fastforce

  show pq ∈ ?S1 q k
   using assms ‹pq ∈ ?S2 q k› unfolding ‹pq = (p,q′)› proof (induction k
arbitrary: q p q′)
    case 0
    then show ?case by force
  next
    case (Suc k)
    then have q′ = target q p
          and path M q p
          and length p ≤ Suc k
      by auto

    show ?case proof (cases p)
      case Nil
      then have q′ = q
        using Suc.prems(2) by auto
      then show ?thesis unfolding Nil by auto
    next
      case (Cons t p′)

      then have q′ = target q (t#p′)
          and path M q (t#p′)
          and length (t#p′) ≤ Suc k
        using Suc.prems(2)
        by auto

      have t ∈ transitions M and t-source t = q
        using ‹path M q (t#p′)› by auto
      then have t ∈ list.set (?hM q (t-input t))
        unfolding hM
        by (metis (mono-tags, lifting) mem-Collect-eq prod.exhaust-sel)

      have t-input t ∈ list.set ?iM
         using fsm-transition-input[OF ‹t ∈ transitions M›] inputs-as-list-set by
```

auto

have $q' = target\ (t\text{-}target\ t)\ p'$
using ‹$q' = target\ q\ (t\#p')$› **by** *auto*
moreover have $path\ M\ (t\text{-}target\ t)\ p'$
using ‹$path\ M\ q\ (t\#p')$› **by** *auto*
moreover have $length\ p' \leq k$
using ‹$length\ (t\#p') \leq Suc\ k$› **by** *auto*
ultimately have $(p',q') \in ?S2\ (t\text{-}target\ t)\ k$
by *blast*
moreover define f **where** *f-def*: $f = (\lambda q\ .\ (paths\text{-}up\text{-}to\text{-}length\text{-}with\text{-}targets\ q\ ?hM\ ?iM\ k))$
ultimately have $(p',q') \in list.set\ (f\ (t\text{-}target\ t))$
using *Suc.IH*[*OF fsm-transition-target*[*OF* ‹$t \in transitions\ M$›]]
by *blast*
then have $**$: $(t\#p',q') \in list.set\ ((map\ (\lambda(p,q).\ (t\ \#\ p,q))\ (f\ (t\text{-}target\ t))))$
by *auto*

have *scheme*: $\bigwedge\ x\ y\ ys\ f\ .\ x \in list.set\ (f\ y) \Longrightarrow y \in list.set\ ys \Longrightarrow x \in list.set\ (concat\ (map\ f\ ys))$
by *auto*

have $(t\#p',q') \in list.set\ (concat\ (map\ (\lambda t\ .\ (map\ (\lambda(p,q).\ (t\ \#\ p,q))$
$(f\ (t\text{-}target\ t))))$
$(?hM\ q\ (t\text{-}input\ t))))$
using *scheme*[*of* $(t\#p',q')\ \lambda\ t.\ (map\ (\lambda(p,q).\ (t\ \#\ p,q))\ (f\ (t\text{-}target\ t)))$, *OF* $**$ ‹$t \in list.set\ (?hM\ q\ (t\text{-}input\ t))$›]
.

then have $(t\#p',q') \in list.set\ (concat$
$(map\ (\lambda x.\ concat$
$(map\ (\lambda t.\ map\ (\lambda(p,\ y).\ (t\ \#\ p,\ y))$
$(f\ (t\text{-}target\ t)))$
$(map\ (\lambda(y,\ q').\ (q,\ x,\ y,\ q'))\ (sorted\text{-}list\text{-}of\text{-}set\ (h\ M\ (q,\ x))))))$
$(inputs\text{-}as\text{-}list\ M)))$
using ‹$t\text{-}input\ t \in list.set\ ?iM$› **by** *force*

then show *?thesis*
unfolding *paths-up-to-length-with-targets.simps f-def Cons* **by** *auto*
qed
qed
qed
qed

fun *pairs-to-distinguish* :: $('a::linorder,'b::linorder,'c::linorder)\ fsm \Rightarrow ('a,'b,'c)\ state\text{-}cover\text{-}assignment \Rightarrow ('a \Rightarrow (('a,'b,'c)\ path \times 'a)\ list) \Rightarrow 'a\ list \Rightarrow (((('b \times$

$'c)$ *list* $\times$ $'a)$ $\times$ $(('b \times 'c)$ *list* $\times$ $'a))$ *list* **where**
pairs-to-distinguish M V $\mathcal{X}'$ *rstates* = (*let*
Π = *map* (λq . (V q,q)) *rstates*;
A = *List.product* Π Π;
B = *List.product* Π (*concat* (*map* (λq . *map* (λ (τ,q') . ((V q)@ *p-io* τ,q')) ($\mathcal{X}'$ q)) *rstates*));
C = *concat* (*map* (λq . *concat* (*map* (λ (τ',q'). *map* ($\lambda\tau''$. (((V q)@ *p-io* τ'', *target* q τ''),((V q)@ *p-io* τ',q'))) (*prefixes* τ')) ($\mathcal{X}'$ q))) *rstates*)
in
filter (λ((α,q'),(β,q'')) . $q' \neq q''$) (A@B@C))

lemma *pairs-to-distinguish-elems* :
assumes *observable M*
and *is-state-cover-assignment M V*
and *list.set rstates* = *reachable-states M*
and $\bigwedge$ q p q' . $q \in$ *reachable-states M* $\Longrightarrow$ (p,q') $\in$ *list.set* ($\mathcal{X}'$ q) $\longleftrightarrow$ *path M q p* $\wedge$ *target q p* = q' $\wedge$ *length p* $\leq$ *m*$-$*n*+*1*
and ((α,*q1*),(β,*q2*)) $\in$ *list.set* (*pairs-to-distinguish M V* $\mathcal{X}'$ *rstates*)

shows *q1* $\in$ *states M* **and** *q2* $\in$ *states M* **and** *q1* $\neq$ *q2*
and $\alpha \in L$ M **and** $\beta \in L$ M **and** *q1* = *after-initial M* α **and** *q2* = *after-initial M* β
proof $-$

define Π **where** Π: Π = *map* (λq . (V q,q)) *rstates*
moreover define A **where** A: A = *List.product* Π Π
moreover define B **where** B: B = *List.product* Π (*concat* (*map* (λq . *map* (λ (τ,q') . ((V q)@ *p-io* τ,q')) ($\mathcal{X}'$ q)) *rstates*))
moreover define C **where** C: C = *concat* (*map* (λq . *concat* (*map* (λ (τ',q'). *map* ($\lambda\tau''$. (((V q)@ *p-io* τ'', *target* q τ''),((V q)@ *p-io* τ',q'))) (*prefixes* τ')) ($\mathcal{X}'$ q))) *rstates*)
ultimately have *pairs-def*: *pairs-to-distinguish M V* $\mathcal{X}'$ *rstates* = *filter* (λ((α,q'),(β,q'')) . $q' \neq q''$) (A@B@C)
unfolding *pairs-to-distinguish.simps Let-def* **by** *force*

show *q1* $\neq$ *q2*
using *assms*(*5*) **unfolding** *pairs-def* **by** *auto*

consider ((α,*q1*),(β,*q2*)) $\in$ *list.set A* | ((α,*q1*),(β,*q2*)) $\in$ *list.set B* | ((α,*q1*),(β,*q2*)) $\in$ *list.set C*
using *assms*(*5*) **unfolding** *pairs-def* **by** *auto*
then have *q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M* $\wedge$ $\alpha \in L$ M $\wedge$ $\beta \in L$ M $\wedge$ *q1* = *after-initial M* α $\wedge$ *q2* = *after-initial M* β
proof *cases*
case *1*
then have (α,*q1*) $\in$ *list.set* Π **and** (β,*q2*) $\in$ *list.set* Π
unfolding A **by** *auto*
then show *?thesis* **unfolding** Π **using** *assms*(*3*)
using *reachable-state-is-state*

using *state-cover-assignment-after*[*OF assms*(*1*,*2*)]
by *force*
next
case *2*
then have (α,*q1*) $\in$ *list.set* Π **and** (β,*q2*) $\in$ *list.set* (*concat* (*map* (λ*q* . *map* (λ (τ,q') . ((*V q*)@ *p-io* τ,q')) ($\mathcal{X}'$ *q*)) *rstates*))
unfolding *B* **by** *auto*

then obtain *q* **where** *q* $\in$ *reachable-states M*
and (β,*q2*) $\in$ *list.set* (*map* (λ (τ,q') . ((*V q*)@ *p-io* τ,q')) ($\mathcal{X}'$ *q*))
unfolding *assms*(*3*)[*symmetric*] **by** (*meson concat-map-elem*)
then obtain τ **where** (τ,*q2*) $\in$ *list.set* ($\mathcal{X}'$ *q*) **and** β = (*V q*)@ *p-io* τ
by *force*
then have *path M q* τ **and** *target q* τ = *q2*
unfolding *assms*(*4*)[*OF* ‹*q* $\in$ *reachable-states M*›] **by** *auto*
moreover obtain *p* **where** *path M* (*initial M*) *p* **and** *p-io p* = *V q* **and** *target* (*initial M*) *p* = *q*
using *state-cover-assignment-after*[*OF assms*(*1*,*2*) ‹*q* $\in$ *reachable-states M*›]
after-path[*OF assms*(*1*)]
by *auto*
ultimately have *path M* (*initial M*) (*p*@τ) **and** *target* (*initial M*) (*p*@τ) = *q2* **and** *p-io* (*p*@τ) = β
unfolding ‹β = (*V q*)@ *p-io* τ› **by** *auto*
then have *q2* = *after-initial M* β
by (*metis* (*mono-tags*, *lifting*) *after-path assms*(*1*))
moreover have β $\in$ *L M*
using ‹*path M* (*initial M*) (*p*@τ)› ‹*p-io* (*p*@τ) = β›
by (*metis* (*mono-tags*, *lifting*) *language-state-containment*)
moreover have *q2* $\in$ *states M*
by (*metis* ‹*path M q* τ› ‹*target q* τ = *q2*› *path-target-is-state*)
moreover have *q1* $\in$ *states M*
using ‹(α,*q1*) $\in$ *list.set* Π› *assms*(*3*) *reachable-state-is-state* **unfolding** Π **by** *fastforce*
moreover have α $\in$ *L M* **and** *q1* = *after-initial M* α
using ‹(α,*q1*) $\in$ *list.set* Π› *assms*(*3*) *state-cover-assignment-after*[*OF assms*(*1*,*2*)] **unfolding** Π **by** *auto*
ultimately show *?thesis*
by *blast*
next
case *3*
then obtain *q* **where** *q* $\in$ *reachable-states M*
and ((α,*q1*),(β,*q2*)) $\in$ *list.set* (*concat* (*map* (λ (τ',q'). *map* ($\lambda\tau''$. (((*V q*)@ *p-io* τ'', *target q* τ''),((*V q*)@ *p-io* τ',q'))) (*prefixes* τ')) ($\mathcal{X}'$ *q*)))
unfolding *assms*(*3*)[*symmetric*] *C* **by** *force*
then obtain τ' **where** (τ',*q2*) $\in$ *list.set* ($\mathcal{X}'$ *q*) **and** β = *V q* @ *p-io* τ'
and ((α,*q1*),(β,*q2*)) $\in$ *list.set* (*map* ($\lambda\tau''$. (((*V q*)@ *p-io* τ'', *target q* τ''),((*V q*)@ *p-io* τ',*q2*))) (*prefixes* τ'))
by *force*
then obtain τ'' **where** τ'' $\in$ *list.set* (*prefixes* τ') **and** α = *V q* @ *p-io* τ''

and $((\alpha,q1),(\beta,q2)) = (((V\ q)@\ p\text{-}io\ \tau'',\ target\ q\ \tau''),((V\ q)@\ p\text{-}io\ \tau',q2))$
by *auto*
then have $q1 = target\ q\ \tau''$
by *auto*

have $path\ M\ q\ \tau'$ **and** $target\ q\ \tau' = q2$
using ‹$(\tau',q2) \in list.set\ (\mathcal{X}'\ q)$› **unfolding** *assms*(*4*)[*OF* ‹$q \in reachable\text{-}states\ M$›] **by** *simp+*
then have $path\ M\ q\ \tau''$
using ‹$\tau'' \in list.set\ (prefixes\ \tau')$›
using *prefixes-set-ob* **by** *force*
then have $q1 \in states\ M$
using *path-target-is-state* **unfolding** ‹$q1 = target\ q\ \tau''$› **by** *force*
moreover have $\alpha \in L\ M$
unfolding ‹$\alpha = V\ q\ @\ p\text{-}io\ \tau''$›
using *state-cover-assignment-after*[*OF assms*(*1*,*2*)]
by (*metis* (*mono-tags*, *lifting*) ‹$path\ M\ q\ \tau''$› ‹$q \in reachable\text{-}states\ M$› *assms*(*1*) *language-state-containment observable-after-language-append*)
moreover have $q1 = after\text{-}initial\ M\ \alpha$
unfolding ‹$\alpha = V\ q\ @\ p\text{-}io\ \tau''$›
using *state-cover-assignment-after*[*OF assms*(*1*,*2*) ‹$q \in reachable\text{-}states\ M$›]
by (*metis* (*mono-tags*, *lifting*) ‹$\alpha = V\ q\ @\ p\text{-}io\ \tau''$› ‹$path\ M\ q\ \tau''$› ‹$q1 = target\ q\ \tau''$› *after-path after-split assms*(*1*) *calculation*(*2*))
moreover have $q2 \in states\ M$
using ‹$path\ M\ q\ \tau'$› ‹$target\ q\ \tau' = q2$› *path-target-is-state* **by** *force*
moreover have $\beta \in L\ M$
by (*metis* (*mono-tags*, *lifting*) ‹$\alpha = V\ q\ @\ p\text{-}io\ \tau''$› ‹$\beta = V\ q\ @\ p\text{-}io\ \tau'$› ‹$path\ M\ q\ \tau'$› ‹$q \in reachable\text{-}states\ M$› *assms*(*1*) *assms*(*2*) *calculation*(*2*) *is-state-cover-assignment-observable-after language-prefix language-state-containment observable-after-language-append*)
moreover have $q2 = after\text{-}initial\ M\ \beta$
unfolding ‹$\beta = V\ q\ @\ p\text{-}io\ \tau'$›
using *state-cover-assignment-after*[*OF assms*(*1*,*2*) ‹$q \in reachable\text{-}states\ M$›]
by (*metis* (*mono-tags*, *lifting*) ‹$\beta = V\ q\ @\ p\text{-}io\ \tau'$› ‹$path\ M\ q\ \tau'$› ‹$target\ q\ \tau' = q2$› *after-path after-split assms*(*1*) *calculation*(*5*))
ultimately show *?thesis*
by *blast*
qed
then show $q1 \in states\ M$ **and** $q2 \in states\ M$ **and** $\alpha \in L\ M$ **and** $\beta \in L\ M$ **and** $q1 = after\text{-}initial\ M\ \alpha$ **and** $q2 = after\text{-}initial\ M\ \beta$
by *auto*
qed

lemma *pairs-to-distinguish-containment* :
assumes *observable M*
and *is-state-cover-assignment M V*
and $list.set\ rstates = reachable\text{-}states\ M$

and $\bigwedge q\ p\ q'$. $q \in$ *reachable-states M* $\Longrightarrow (p,q') \in$ *list.set* $(\mathcal{X}'\ q) \longleftrightarrow$ *path M q p* $\wedge$ *target q p* $= q' \wedge$ *length p* $\leq m-n+1$
and $(\alpha,\beta) \in (V$ ‘ *reachable-states M*$) \times (V$ ‘ *reachable-states M*$)$
$\cup (V$ ‘ *reachable-states M*$) \times \{ (V\ q)$ @ $\tau \mid q\ \tau$. $q \in$ *reachable-states M* $\wedge\ \tau \in \{io@[(x,y)] \mid io\ x\ y$. $io \in$ *LS M q* $\wedge$ *length io* $\leq m-n \wedge x \in$ *inputs M* $\wedge\ y \in$ *outputs M*$\}\}$
$\cup (\bigcup\ q \in$ *reachable-states M* . $\bigcup\ \tau \in \{io@[(x,y)] \mid io\ x\ y$. $io \in$ *LS M q* $\wedge$ *length io* $\leq m-n \wedge x \in$ *inputs M* $\wedge\ y \in$ *outputs M*$\}$. $\{ (V\ q)$ @ $\tau' \mid \tau'$. $\tau' \in$ *list.set* (*prefixes* τ)$\} \times \{(V\ q)@\tau\})$
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* $\alpha \neq$ *after-initial M* β
shows $((\alpha,$*after-initial M* $\alpha),(\beta,$*after-initial M* $\beta)) \in$ *list.set* (*pairs-to-distinguish M V* $\mathcal{X}'$ *rstates*)
proof −

let $?\mathcal{X} = \lambda\ q$. $\{io@[(x,y)] \mid io\ x\ y$. $io \in$ *LS M q* $\wedge$ *length io* $\leq m-n \wedge x \in$ *inputs M* $\wedge\ y \in$ *outputs M*$\}$

define Π **where** Π: $\Pi =$ *map* $(\lambda q$. $(V\ q,q))$ *rstates*
moreover define A **where** A: $A =$ *List.product* $\Pi\ \Pi$
moreover define B **where** B: $B =$*List.product* Π (*concat* (*map* (λq . *map* (λ (τ,q') . $((V\ q)@$ *p-io* $\tau,q'))$ $(\mathcal{X}'\ q))$ *rstates*))
moreover define C **where** C: $C =$ *concat* (*map* (λq . *concat* (*map* (λ (τ',q'). *map* ($\lambda\tau''$. $(((V\ q)@$ *p-io* τ'', *target q* $\tau''),((V\ q)@$ *p-io* $\tau',q')))$ (*prefixes* τ')) $(\mathcal{X}'$ $q)))$ *rstates*)
ultimately have *pairs-def*: *pairs-to-distinguish M V* $\mathcal{X}'$ *rstates* = *filter* $(\lambda((\alpha,q'),(\beta,q''))$. $q' \neq q'')$ $(A@B@C)$
unfolding *pairs-to-distinguish.simps Let-def* **by** *force*

have *V-target*: $\bigwedge q$. $q \in$ *reachable-states M* $\Longrightarrow$ *after-initial M* $(V\ q) = q$
proof −
fix q **assume** $q \in$ *reachable-states M*
then have $q \in$ *io-targets M* $(V\ q)$ (*initial M*)
using *assms*(*2*) **by** *auto*
then have $V\ q \in L\ M$
unfolding *io-targets.simps*
by *force*

show *after-initial M* $(V\ q) = q$
by (*meson* ‹$q \in$ *reachable-states M*› *assms*(*1*) *assms*(*2*) *is-state-cover-assignment-observable-after*)
qed

have *V-path*: $\bigwedge$ *io q* . $q \in$ *reachable-states M* $\Longrightarrow io \in$ *LS M q* $\Longrightarrow \exists\ p$. *path M q p* $\wedge$ *p-io p* $= io \wedge$ *target q p* $=$ *after-initial M* $((V\ q)@io)$
proof −
fix *io q* **assume** $q \in$ *reachable-states M* **and** $io \in$ *LS M q*
then have *after-initial M* $(V\ q) = q$

using *V-target* **by** *auto*
then have $((V\ q)@io) \in L\ M$
using ‹$io \in LS\ M\ q$›
by (*meson* ‹$q \in$ *reachable-states* M› *assms*(*2*) *is-state-cover-assignment.simps language-io-target-append*)
then obtain p **where** *path* M (*initial* M) p **and** *p-io* $p = ((V\ q)@io)$
by *auto*
moreover have *target* (*initial* M) $p \in$ *io-targets* M $((V\ q)@io)$ (*initial* M)
using *calculation* **unfolding** *io-targets.simps* **by** *force*
ultimately have *target* (*initial* M) $p =$ *after-initial* M $((V\ q)@io)$
using *observable-io-targets*[*OF* ‹*observable* M› ‹$((V\ q)@io) \in L\ M$›]
unfolding *io-targets.simps*
by (*metis* (*mono-tags*, *lifting*) *after-path assms*(*1*))

have *path* M (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)
and *p-io* (*take* (*length* $(V\ q)$) p) $= V\ q$
and *path* M (*target* (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)) (*drop* (*length* $(V\ q)$) p)
and *p-io* (*drop* (*length* $(V\ q)$) p) $= io$
using *path-io-split*[*OF* ‹*path* M (*initial* M) p› ‹*p-io* $p = ((V\ q)@io)$›]
by *auto*

have *target* (*initial* M) $p =$ *target* (*target* (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)) (*drop* (*length* $(V\ q)$) p)
by (*metis append-take-drop-id path-append-target*)
moreover have (*target* (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)) $= q$
using ‹*p-io* (*take* (*length* $(V\ q)$) p) $= V\ q$› ‹*after-initial* M $(V\ q) = q$›
using ‹*path* M (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)› *after-path assms*(*1*) **by** *fastforce*
ultimately have *target* q (*drop* (*length* $(V\ q)$) p) $=$ *after-initial* M $((V\ q)@io)$
using ‹*target* (*initial* M) $p =$ *after-initial* M $((V\ q)@io)$›
by *presburger*
then show $\exists\ p$. *path* $M\ q\ p \wedge$ *p-io* $p = io \wedge$ *target* $q\ p =$ *after-initial* M $((V\ q)@io)$
using ‹*path* M (*target* (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)) (*drop* (*length* $(V\ q)$) p)› ‹*p-io* (*drop* (*length* $(V\ q)$) p) $= io$›
unfolding ‹(*target* (*FSM.initial* M) (*take* (*length* $(V\ q)$) p)) $= q$›
by *blast*
qed

consider $(\alpha,\beta) \in (V\ `\ reachable\text{-}states\ M) \times (V\ `\ reachable\text{-}states\ M)$ |
$(\alpha,\beta) \in (V\ `\ reachable\text{-}states\ M) \times \{\ (V\ q)\ @\ \tau \mid q\ \tau\ .\ q \in reachable\text{-}states\ M \wedge \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m-n \wedge x \in inputs\ M \wedge y \in outputs\ M\}\}$ |
$(\alpha,\beta) \in (\bigcup\ q \in reachable\text{-}states\ M\ .\ \bigcup\ \tau \in \{io@[(x,y)] \mid io\ x\ y\ .\ io \in LS\ M\ q \wedge length\ io \leq m-n \wedge x \in inputs\ M \wedge y \in outputs\ M\}\ .\ \{\ (V\ q)\ @\ \tau' \mid \tau'\ .\ \tau' \in list.set\ (prefixes\ \tau)\} \times \{(V\ q)@\tau\})$

using *assms*(*5*) **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then have $\alpha \in V$ ‘ *reachable-states M*
and $\beta \in V$ ‘ *reachable-states M*
by *auto*

have (α,*after-initial M* α) $\in$ *list.set* (*map* (λq . (V q,q)) *rstates*)
using ‹$\alpha \in V$ ‘ *reachable-states M*› *V-target assms*(*3*) **by** *force*
moreover have (β,*after-initial M* β) $\in$ *list.set* (*map* (λq . (V q,q)) *rstates*)
using ‹$\beta \in V$ ‘ *reachable-states M*› *V-target assms*(*3*) **by** *force*
ultimately have ((α,*after-initial M* α),(β,*after-initial M* β)) $\in$ *list.set A*
unfolding Π *A* **by** *auto*
then show *?thesis*
using ‹*after-initial M* $\alpha \neq$ *after-initial M* β›
unfolding *pairs-def* **by** *auto*
next
case *2*
then have $\alpha \in V$ ‘ *reachable-states M*
and $\beta \in \{$ (V q) @ τ | q τ . $q \in$ *reachable-states M* $\wedge$ $\tau \in \{$*io*@[(x,y)] | *io* x y . *io* $\in$ *LS M q* $\wedge$ *length io* $\leq m-n$ $\wedge$ $x \in$ *inputs M* $\wedge$ $y \in$ *outputs M*$\}\}$
by *auto*

have (α,*after-initial M* α) $\in$ *list.set* (*map* (λq . (V q,q)) *rstates*)
using ‹$\alpha \in V$ ‘ *reachable-states M*› *V-target assms*(*3*) **by** *force*

obtain *q io x y* **where** β = (V q) @ (*io*@[(x,y)])
and $q \in$ *reachable-states M*
and *length io* $\leq m-n$
using ‹$\beta \in \{$ (V q) @ τ | q τ . $q \in$ *reachable-states M* $\wedge$ $\tau \in \{$*io*@[(x,y)] | *io* x y . *io* $\in$ *LS M q* $\wedge$ *length io* $\leq m-n$ $\wedge$ $x \in$ *inputs M* $\wedge$ $y \in$ *outputs M*$\}\}$›
by *blast*

have (V q) @ (*io*@[(x,y)]) $\in$ *L M*
using ‹$\beta \in$ *L M*› **unfolding** ‹β = (V q) @ (*io*@[(x,y)])› **by** *simp*

have $q \in$ *io-targets M* (V q) (*initial M*)
using ‹$q \in$ *reachable-states M*› *assms*(*2*) **by** *auto*
then have *io*@[(x,y)] $\in$ *LS M q*
unfolding ‹β = (V q) @ (*io*@[(x,y)])›
using *observable-io-targets-language*[*OF* ‹(V q) @ (*io*@[(x,y)]) $\in$ *L M*› ‹*observable M*›]
by *auto*
then obtain *p* **where** *path M q p*
and *p-io p* = *io*@[(x,y)]
and *target q p* = *after-initial M* β
using *V-path*[*OF* ‹$q \in$ *reachable-states M*›]
unfolding ‹β = (V q) @ (*io*@[(x,y)])›

by *blast*
moreover have *length p* $\leq$ *m−n+1*
using *calculation* ‹*length io* $\leq$ *m−n*›
by (*metis* (*no-types*, *lifting*) *Suc-le-mono add.commute length-append-singleton length-map plus-1-eq-Suc*)
ultimately have (*p*,*after-initial M* β) $\in$ *list.set* ($\mathcal{X}'$ *q*)
using *assms*(*4*)[*OF* ‹*q* $\in$ *reachable-states M*›]
by *auto*
then have (β,*after-initial M* β) $\in$ *list.set* (*map* (λ (τ,*q*′) . ((*V q*)@ *p-io* τ,*q*′)) ($\mathcal{X}'$ *q*))
unfolding ‹β = (*V q*) @ (*io*@[(*x*,*y*)])› **using** ‹*p-io p* = *io*@[(*x*,*y*)]› **by** *force*
moreover have *q* $\in$ *list.set rstates*
using ‹*q* $\in$ *reachable-states M*› *assms*(*3*) **by** *auto*
ultimately have (β,*after-initial M* β) $\in$ *list.set* (*concat* (*map* (λ*q* . *map* (λ (τ,*q*′) . ((*V q*)@ *p-io* τ,*q*′)) ($\mathcal{X}'$ *q*)) *rstates*))
by *force*
then have ((α,*after-initial M* α),(β,*after-initial M* β)) $\in$ *list.set B*
using ‹(α,*after-initial M* α) $\in$ *list.set* (*map* (λ*q* . (*V q*,*q*)) *rstates*)›
unfolding *B* Π
by *auto*
then show *?thesis*
using ‹*after-initial M* α $\neq$ *after-initial M* β›
unfolding *pairs-def* **by** *auto*
next
case *3*
then obtain *q* τ' *io x y* **where** *q* $\in$ *reachable-states M*
and *io* $\in$ *LS M q*
and *length io* $\leq$ *m* − *n*
and *x* $\in$ *FSM.inputs M*
and *y* $\in$ *FSM.outputs M*
and α = *V q* @ τ'
and τ' $\in$ *list.set* (*prefixes* (*io* @ [(*x*, *y*)]))
and β = *V q* @ *io* @ [(*x*, *y*)]
by *blast*

have (*V q*) @ (*io*@[(*x*,*y*)]) $\in$ *L M*
using ‹β $\in$ *L M*› **unfolding** ‹β = (*V q*) @ (*io*@[(*x*,*y*)])› **by** *simp*

have *q* $\in$ *io-targets M* (*V q*) (*initial M*)
using ‹*q* $\in$ *reachable-states M*› *assms*(*2*) **by** *auto*
then have *io*@[(*x*,*y*)] $\in$ *LS M q*
unfolding ‹β = (*V q*) @ (*io*@[(*x*,*y*)])›
using *observable-io-targets-language*[*OF* ‹(*V q*) @ (*io*@[(*x*,*y*)]) $\in$ *L M*› ‹*observable M*›]
by *auto*
then obtain *p* **where** *path M q p*

and *p-io p* = *io*@[(*x*,*y*)]
and *target q p* = *after-initial M* β
using *V-path*[*OF* ‹*q* ∈ *reachable-states M*›]
unfolding ‹β = (*V q*) @ (*io*@[(*x*,*y*)])›
by *blast*
moreover have *length p* ≤ *m*−*n*+*1*
using *calculation* ‹*length io* ≤ *m*−*n*›
by (*metis* (*no-types*, *lifting*) *Suc-le-mono add.commute length-append-singleton length-map plus-1-eq-Suc*)
ultimately have (*p*,*after-initial M* β) ∈ *list.set* ($\mathcal{X}'$ *q*)
using *assms*(*4*)[*OF* ‹*q* ∈ *reachable-states M*›]
by *auto*

have *q* ∈ *list.set rstates*
using ‹*q* ∈ *reachable-states M*› *assms*(*3*) **by** *auto*

let *?*τ = *take* (*length* τ') (*io*@[(*x*,*y*)])
obtain τ'' **where** *io* @ [(*x*, *y*)] = τ' @ τ''
using ‹τ' ∈ *list.set* (*prefixes* (*io* @ [(*x*, *y*)]))›
using *prefixes-set-ob* **by** *blast*
then have τ' = *?*τ
by *auto*
then have *io*@[(*x*,*y*)] = τ' @ (*drop* (*length* τ') (*io*@[(*x*,*y*)]))
by (*metis append-take-drop-id*)
then have *p-io p* = τ' @ (*drop* (*length* τ') (*io*@[(*x*,*y*)]))
using ‹*p-io p* = *io*@[(*x*,*y*)]›
by *simp*

have *path M q* (*take* (*length* τ') *p*)
and *p-io* (*take* (*length* τ') *p*) = τ'
using *path-io-split*(*1*,*2*)[*OF* ‹*path M q p*› ‹*p-io p* = τ' @ (*drop* (*length* τ') (*io*@[(*x*,*y*)]))›]
by *auto*
then have τ' ∈ *LS M q*
using *language-intro* **by** *fastforce*

have (*V q*) @ τ' ∈ *L M*
using ‹(*V q*) @ (*io*@[(*x*,*y*)]) ∈ *L M*› **unfolding** ‹*io* @ [(*x*, *y*)] = τ' @ τ''›
using *language-prefix*[*of V q* @ τ' τ'' *M initial M*]
by *auto*

have (*FSM.after M* (*FSM.initial M*) (*V q*)) = *q*
using *V-target* ‹*q* ∈ *reachable-states M*› **by** *blast*
have *target q* (*take* (*length* τ') *p*) = *after-initial M* α
using *observable-after-target*[*OF* ‹*observable M*› ‹(*V q*) @ τ' ∈ *L M*› - ‹*p-io* (*take* (*length* τ') *p*) = τ'›]
using ‹*path M q* (*take* (*length* τ') *p*)›
unfolding ‹(*FSM.after M* (*FSM.initial M*) (*V q*)) = *q*› ‹α = *V q* @ τ'›

by *auto*

have $p = (take\ (length\ \tau')\ p)@(drop\ (length\ \tau')\ p)$
by *simp*
then have $(take\ (length\ \tau')\ p) \in list.set\ (prefixes\ p)$
unfolding *prefixes-set*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

have $(((V\ q)@\ p\text{-}io\ (take\ (length\ \tau')\ p),\ target\ q\ (take\ (length\ \tau')\ p)),((V\ q)@\ p\text{-}io\ p,after\text{-}initial\ M\ \beta)) \in list.set\ (\ (\lambda(\tau',\ q').\ map\ (\lambda\tau''.\ ((V\ q\ @\ p\text{-}io\ \tau'',\ target\ q\ \tau''),\ V\ q\ @\ p\text{-}io\ \tau',\ q'))\ (prefixes\ \tau'))\ (p,after\text{-}initial\ M\ \beta))$
using ‹$(take\ (length\ \tau')\ p) \in list.set\ (prefixes\ p)$›
by *auto*
then have $*$: $((\alpha,\ after\text{-}initial\ M\ \alpha),(\beta,after\text{-}initial\ M\ \beta)) \in list.set\ (\ (\lambda(\tau',\ q').\ map\ (\lambda\tau''.\ ((V\ q\ @\ p\text{-}io\ \tau'',\ target\ q\ \tau''),\ V\ q\ @\ p\text{-}io\ \tau',\ q'))\ (prefixes\ \tau'))\ (p,after\text{-}initial\ M\ \beta))$
unfolding ‹$\alpha = V\ q\ @\ \tau'$›
‹$\beta = V\ q\ @\ io\ @\ [(x,\ y)]$›
‹$target\ q\ (take\ (length\ \tau')\ p) = after\text{-}initial\ M\ \alpha$›
‹$p\text{-}io\ (take\ (length\ \tau')\ p) = \tau'$›
‹$p\text{-}io\ p = io@[(x,y)]$› **.**

have *scheme*: $\bigwedge x\ y\ ys\ f\ .\ x \in list.set\ (f\ y) \Longrightarrow y \in list.set\ ys \Longrightarrow x \in list.set\ (concat\ (map\ f\ ys))$
by *auto*

have $**$: $((\alpha,\ after\text{-}initial\ M\ \alpha),(\beta,after\text{-}initial\ M\ \beta)) \in list.set\ (concat\ (map\ (\lambda\ (\tau',q').\ map\ (\lambda\tau''\ .\ (((V\ q)@\ p\text{-}io\ \tau'',\ target\ q\ \tau''),((V\ q)@\ p\text{-}io\ \tau',q')))\ (prefixes\ \tau'))\ (\mathcal{X}'\ q)))$
using *scheme*[*of* - $(\lambda(\tau',\ q').\ map\ (\lambda\tau''.\ ((V\ q\ @\ p\text{-}io\ \tau'',\ target\ q\ \tau''),\ V\ q\ @\ p\text{-}io\ \tau',\ q'))\ (prefixes\ \tau'))$, *OF* $*$ ‹$(p,after\text{-}initial\ M\ \beta) \in list.set\ (\mathcal{X}'\ q)$›]
.

have $((\alpha,\ after\text{-}initial\ M\ \alpha),(\beta,after\text{-}initial\ M\ \beta)) \in list.set\ C$
unfolding C
using *scheme*[*of* - $(\lambda q\ .\ concat\ (map\ (\lambda\ (\tau',q').\ map\ (\lambda\tau''\ .\ (((V\ q)@\ p\text{-}io\ \tau'',\ target\ q\ \tau''),((V\ q)@\ p\text{-}io\ \tau',q')))\ (prefixes\ \tau'))\ (\mathcal{X}'\ q)))$, *OF* $**$ ‹$q \in list.set\ rstates$›]
.

then show *?thesis*
using ‹$after\text{-}initial\ M\ \alpha \neq after\text{-}initial\ M\ \beta$›
unfolding *pairs-def* **by** *auto*
qed
qed

20.3 Definition of the Pair-Framework

definition *pair-framework* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* ⇒
nat ⇒
(('a,'b,'c) *fsm* ⇒ *nat* ⇒ ('b×'c) *prefix-tree*) ⇒
(('a,'b,'c) *fsm* ⇒ *nat* ⇒ ((('b × 'c) *list* × 'a) × (('b × 'c) *list* × 'a)) *list*) ⇒
(('a,'b,'c) *fsm* ⇒ (('b × 'c) *list* × 'a) × ('b × 'c) *list* × 'a ⇒ ('b × 'c) *prefix-tree* ⇒ ('b × 'c) *prefix-tree*) ⇒
('b×'c) *prefix-tree*

where
pair-framework M m get-initial-test-suite get-pairs get-separating-traces =
(*let*
TS = *get-initial-test-suite M m*;
D = *get-pairs M m*;
dist-extension = (λ *t* ((α,q'),(β,q'')) . *let tDist* = *get-separating-traces M* ((α,q'),(β,q'')) *t*
in combine-after (*combine-after t* α *tDist*) β *tDist*)
in
foldl dist-extension TS D)

lemma *pair-framework-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
and *is-state-cover-assignment M V*
and {(*V q*)@*io*@[(*x*,*y*)] | *q io x y* . *q* ∈ *reachable-states M* ∧ *io* ∈ *LS M q* ∧ *length io* ≤ *m* − *size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} ⊆ *set* (*get-initial-test-suite M m*)
and ⋀ α β . (α,β) ∈ (*V* ‘ *reachable-states M*) × (*V* ‘ *reachable-states M*)
∪ (*V* ‘ *reachable-states M*) × { (*V q*) @ τ | *q* τ . *q* ∈ *reachable-states M* ∧ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}}
∪ (⋃ *q* ∈ *reachable-states M* . ⋃ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} . { (*V q*) @ τ' | τ' . τ' ∈ *list.set* (*prefixes* τ)} × {(*V q*)@τ}) ⟹
α ∈ *L M* ⟹ β ∈ *L M* ⟹ *after-initial M* α ≠ *after-initial M* β ⟹
((α,*after-initial M* α),(β,*after-initial M* β)) ∈ *list.set* (*get-pairs M m*)
and ⋀ α β *t* . α ∈ *L M* ⟹ β ∈ *L M* ⟹ *after-initial M* α ≠ *after-initial M* β ⟹ ∃ *io* ∈ *set* (*get-separating-traces M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α)

(*after-initial M β*) *io*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*pair-framework M m get-initial-test-suite get-pairs get-separating-traces*) = *L I* ∩ *set* (*pair-framework M m get-initial-test-suite get-pairs get-separating-traces*))
proof (*cases inputs M* = {} ∨ *outputs M* = {})
  case *True*
  then consider *inputs M* = {} | *outputs M* = {} **by** *blast*
  then show *?thesis* **proof** *cases*
    case *1*
    have *L M* = {[]}
      using *1 language-empty-IO* **by** *blast*
    moreover have *L I* = {[]}
      by (*metis 1 assms*(*6*) *language-empty-IO*)
    ultimately show *?thesis* **by** *blast*
  next
    case *2*
    have *L M* = {[]}
      using *language-io*(*2*)[*of - M initial M*] **unfolding** *2*
    by (*metis* (*no-types*, *opaque-lifting*) *ex-in-conv is-singletonI′ is-singleton-the-elem language-contains-empty-sequence set-empty2 singleton-iff surj-pair*)
    moreover have *L I* = {[]}
      using *language-io*(*2*)[*of - I initial I*] **unfolding** *2* ‹*outputs I* = *outputs M*›
    by (*metis* (*no-types*, *opaque-lifting*) *ex-in-conv is-singletonI′ is-singleton-the-elem language-contains-empty-sequence set-empty2 singleton-iff surj-pair*)
    ultimately show *?thesis* **by** *blast*
  qed
next
  case *False*

  define *T* **where** *T*: *T* = *get-initial-test-suite M m*
  moreover define *pairs* **where** *pairs*: *pairs* = *get-pairs M m*
  moreover define *distExtension* **where** *distExtension*: *distExtension* = (λ *t* ((α,*q′*),(β,*q″*)) . *let tDist* = *get-separating-traces M* ((α,*q′*),(β,*q″*)) *t*
in combine-after (*combine-after t α tDist*) *β tDist*)
  ultimately have *res-def*: *pair-framework M m get-initial-test-suite get-pairs get-separating-traces* = *foldl distExtension T pairs*
    unfolding *pair-framework-def Let-def* **by** *auto*

  define *T′* **where** *T′*: *T′* = *set* (*foldl distExtension T pairs*)
  then have *T′r*: *T′* = *set* (*foldr* (λ *x y* . *distExtension y x*) (*rev pairs*) *T*)
    by (*simp add*: *foldl-conv-foldr*)

  define Π **where** Π: Π = (*V* ‘ *reachable-states M*)
  define *n* **where** *n*: *n* = *size-r M*
  define 𝒳 **where** 𝒳: 𝒳 = (λ *q* . {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*n* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*})
  define *A* **where** *A*: *A* = Π × Π

define B **where** B: $B = \Pi \times \{ (V\ q) @ \tau \mid q\ \tau\ .\ q \in$ *reachable-states* $M \wedge \tau \in \mathcal{X}\ q\}$
define C **where** C: $C = (\bigcup\ q \in$ *reachable-states* $M\ .\ \bigcup\ \tau \in \mathcal{X}\ q\ .\ \{ (V\ q) @ \tau' \mid \tau'\ .\ \tau' \in$ *list.set* (*prefixes* τ)$\} \times \{(V\ q)@\tau\})$

have *satisfaction-conditions*: *is-state-cover-assignment* $M\ V \Longrightarrow$
$\Pi \subseteq T' \Longrightarrow$
$\{ (V\ q) @ \tau \mid q\ \tau\ .\ q \in$ *reachable-states* $M \wedge \tau \in \mathcal{X}\ q\} \subseteq T' \Longrightarrow$
$(\bigwedge\ \alpha\ \beta\ .\ (\alpha,\beta) \in A \cup B \cup C \Longrightarrow$
$\alpha \in L\ M \Longrightarrow$
$\beta \in L\ M \Longrightarrow$
after-initial $M\ \alpha \neq$ *after-initial* $M\ \beta \Longrightarrow$
$(\exists\ \omega\ .\ \alpha@\omega \in T' \wedge$
$\beta@\omega \in T' \wedge$
distinguishes M (*after-initial* $M\ \alpha$) (*after-initial* $M\ \beta$) $\omega)) \Longrightarrow$
satisfies-h-condition $M\ V\ T'\ m$
unfolding *satisfies-h-condition-def Let-def* $\Pi\ n\ \mathcal{X}\ A\ B\ C$
by *force*

have *c1*: *is-state-cover-assignment* $M\ V$
using *assms*(*8*) **.**

have *c2*: $\Pi \subseteq T'$ **and** *c3*: $\{ (V\ q) @ \tau \mid q\ \tau\ .\ q \in$ *reachable-states* $M \wedge \tau \in \mathcal{X}\ q\} \subseteq T'$
proof −
have *set* $T \subseteq T'$
unfolding T'
proof (*induction pairs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a pairs*)

obtain $\alpha\ q'\ \beta\ q''$ **where** $a = ((\alpha,q'),(\beta,q''))$
by (*metis prod.collapse*)

have *foldl distExtension* T (*pairs* @ [*a*]) = *distExtension* (*foldl distExtension* T *pairs*) a
by *simp*
moreover have $\bigwedge\ t\ .$ *set* $t \subseteq$ *set* (*distExtension* $t\ a$)
proof −
fix t
have *distExtension* $t\ a$ = *combine-after* (*combine-after* $t\ \alpha$ (*get-separating-traces* M $((\alpha,q'),(\beta,q''))$ t)) β (*get-separating-traces* M $((\alpha,q'),(\beta,q''))$ t)
unfolding *distExtension* ‹$a = ((\alpha,q'),(\beta,q''))$› *Let-def* **by** *auto*
moreover have $\bigwedge\ t'\ .$ *set* $t \subseteq$ *set* (*combine-after* (*combine-after* $t\ \alpha\ t'$) β t')
unfolding *combine-after-set* **by** *blast*

```
      ultimately show set t ⊆ set (distExtension t a)
        by simp
    qed
    ultimately show ?case
      using snoc.IH by auto
  qed

  have Π ⊆ set T
  proof
    fix io assume io ∈ Π
    then obtain q where io = (V q)
                  and q ∈ reachable-states M
      unfolding Π
      by blast

    obtain x y where x ∈ inputs M and y ∈ outputs M
      using False by blast
    moreover have [] ∈ LS M q
      using reachable-state-is-state[OF ‹q ∈ reachable-states M›] by auto
    ultimately have (V q) @ [(x,y)] ∈ set T
      using assms(9) ‹q ∈ reachable-states M›
      unfolding T[symmetric] by force
    then show io ∈ set T
      unfolding Π
      using ‹io = V q› set-prefix by auto
  qed
  then show Π ⊆ T'
    using ‹set T ⊆ T'› by blast


  have { (V q) @ τ | q τ . q ∈ reachable-states M ∧ τ ∈ 𝒳 q} ⊆ set T
    using assms(9)
    unfolding 𝒳 T[symmetric] n[symmetric] by force
  then show { (V q) @ τ | q τ . q ∈ reachable-states M ∧ τ ∈ 𝒳 q} ⊆ T'
    using ‹set T ⊆ T'› by blast
qed

have c4: (⋀ α β . (α,β) ∈ A ∪ B ∪ C ⟹
          α ∈ L M ⟹
          β ∈ L M ⟹
          after-initial M α ≠ after-initial M β ⟹
          (∃ ω . α@ω ∈ T' ∧
                β@ω ∈ T' ∧
                distinguishes M (after-initial M α) (after-initial M β) ω))
proof −
  fix α β assume (α,β) ∈ A ∪ B ∪ C
           and α ∈ L M
           and β ∈ L M
           and after-initial M α ≠ after-initial M β
```

have $((\alpha, FSM.after\ M\ (FSM.initial\ M)\ \alpha), \beta, FSM.after\ M\ (FSM.initial\ M)\ \beta) \in list.set\ pairs$
using ‹$(\alpha,\beta) \in A \cup B \cup C$›
unfolding $A\ B\ C\ \Pi\ \mathcal{X}\ pairs$
using ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$› ‹$after\text{-}initial\ M\ \alpha \neq after\text{-}initial\ M\ \beta$› $assms(10)$ n **by** *force*
moreover note ‹$\alpha \in L\ M$› ‹$\beta \in L\ M$› ‹$after\text{-}initial\ M\ \alpha \neq after\text{-}initial\ M\ \beta$›
moreover have $\bigwedge\ \alpha\ \beta\ .\ ((\alpha,after\text{-}initial\ M\ \alpha),(\beta,after\text{-}initial\ M\ \beta)) \in list.set\ pairs \Longrightarrow \alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow after\text{-}initial\ M\ \alpha \neq after\text{-}initial\ M\ \beta \Longrightarrow \exists\ io\ .\ io \in (LS\ M\ (after\text{-}initial\ M\ \alpha) - LS\ M\ (after\text{-}initial\ M\ \beta)) \cup (LS\ M\ (after\text{-}initial\ M\ \beta) - LS\ M\ (after\text{-}initial\ M\ \alpha)) \wedge \alpha@io \in T' \wedge \beta@io \in T'$
unfolding T' **proof** (*induction pairs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a pairs*)

obtain $\alpha a\ q'a\ \beta a\ q''a$ **where** $a = ((\alpha a,q'a),(\beta a,q''a))$
by (*metis prod.collapse*)

have $foldl\ distExtension\ T\ (pairs\ @\ [a]) = distExtension\ (foldl\ distExtension\ T\ pairs)\ a$
by *simp*
moreover have $\bigwedge\ t\ .\ set\ t \subseteq set\ (distExtension\ t\ a)$
proof −
fix t
have $distExtension\ t\ a = combine\text{-}after\ (combine\text{-}after\ t\ \alpha a\ (get\text{-}separating\text{-}traces\ M\ ((\alpha a,q'a),(\beta a,q''a))\ t))\ \beta a\ (get\text{-}separating\text{-}traces\ M\ ((\alpha a,q'a),(\beta a,q''a))\ t)$
unfolding $distExtension$ ‹$a = ((\alpha a,q'a),(\beta a,q''a))$› *Let-def* **by** *auto*
moreover have $\bigwedge\ t'\ .\ set\ t \subseteq set\ (combine\text{-}after\ (combine\text{-}after\ t\ \alpha a\ t')\ \beta a\ t')$
unfolding *combine-after-set* **by** *blast*
ultimately show $set\ t \subseteq set\ (distExtension\ t\ a)$
by *simp*
qed
ultimately have $set\ (foldl\ distExtension\ T\ pairs) \subseteq set\ (foldl\ distExtension\ T\ (pairs@[a]))$
by *auto*

let $?q' = after\text{-}initial\ M\ \alpha$
let $?q'' = after\text{-}initial\ M\ \beta$

show *?case* **proof** (*cases* $a = ((\alpha,?q'),(\beta,?q''))$)
case *True*
then have $foldl\ distExtension\ T\ (pairs\ @\ [a]) = distExtension\ (foldl\ distExtension\ T\ pairs)\ ((\alpha,?q'),(\beta,?q''))$
by *auto*

also have ... = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α (*get-separating-traces M* ((α, *?q'*), (β, *?q''*)) (*foldl distExtension T pairs*))) β (*get-separating-traces M* ((α, *?q'*), (β, *?q''*)) (*foldl distExtension T pairs*))
using *distExtension*
by (*metis* (*no-types*, *lifting*) *case-prod-conv*)
finally have *foldl distExtension T* (*pairs* @ [*a*]) = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α (*get-separating-traces M* ((α, *?q'*), (β, *?q''*)) (*foldl distExtension T pairs*))) β (*get-separating-traces M* ((α, *?q'*), (β, *?q''*)) (*foldl distExtension T pairs*))
.
moreover define *dist* **where** *dist*: *dist* = (*get-separating-traces M* ((α,*?q'*),(β,*?q''*)) (*foldl distExtension T pairs*))
ultimately have $*$: *foldl distExtension T* (*pairs* @ [*a*]) = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α *dist*) β *dist*
by *auto*

define *ta* **where** *ta* = (*after* (*foldl distExtension T pairs*) α)
define *tb* **where** *tb* = (*after* (*foldl distExtension T pairs*) β)
obtain *io* **where** *io* $\in$ *set dist* $\cup$ (*set ta* $\cap$ *set tb*) **and** *io* $\in$ (*LS M ?q'* − *LS M ?q''*) $\cup$ (*LS M ?q''* − *LS M ?q'*)
using *assms*(*11*)[*OF snoc.prems*(*2*,*3*,*4*), *of* (*foldl distExtension T pairs*)]
unfolding *dist distinguishes-def ta-def tb-def* **by** *blast*
then consider *io* $\in$ *set dist* | *io* $\in$ (*set ta* $\cap$ *set tb*)
by *blast*

then show *?thesis* **proof** *cases*
case *1*
then have α@*io* $\in$ *set* (*foldl distExtension T* (*pairs* @ [*a*])) **and** β@*io* $\in$ *set* (*foldl distExtension T* (*pairs* @ [*a*]))
unfolding $*$ **using** *combine-after-set* **by** *blast+*
then show *?thesis*
using ‹*io* $\in$ (*LS M ?q'* − *LS M ?q''*) $\cup$ (*LS M ?q''* − *LS M ?q'*)› **by** *auto*
next
case *2*
moreover have *io* $\neq$ []
using ‹*io* $\in$ (*LS M ?q'* − *LS M ?q''*) $\cup$ (*LS M ?q''* − *LS M ?q'*)›
after-is-state[*OF assms*(*1*) *snoc.prems*(*2*)]
after-is-state[*OF assms*(*1*) *snoc.prems*(*3*)]
by *auto*
ultimately have α@*io* $\in$ *set* (*foldl distExtension T pairs*) **and** β@*io* $\in$ *set* (*foldl distExtension T pairs*)
unfolding *ta-def tb-def after-set* **by** *blast+*
then show *?thesis*
using ‹*io* $\in$ (*LS M ?q'* − *LS M ?q''*) $\cup$ (*LS M ?q''* − *LS M ?q'*)›
‹*set* (*foldl distExtension T pairs*) $\subseteq$ *set* (*foldl distExtension T* (*pairs* @ [*a*]))›
by *auto*
qed
next

case *False*
then have $((\alpha,?q'),(\beta,?q'')) \in$ *list.set pairs*
using *snoc.prems*(*1*) **by** *auto*

show *?thesis*
using *snoc.IH*[*OF* ‹$((\alpha,?q'),(\beta,?q'')) \in$ *list.set pairs*› *snoc.prems*(*2*,*3*,*4*)]
‹*set* (*foldl distExtension T pairs*) $\subseteq$ *set* (*foldl distExtension T* (*pairs* @ [*a*]))›
by *auto*
qed
qed
ultimately show ($\exists$ ω . $\alpha@\omega \in T' \wedge \beta@\omega \in T' \wedge$ *distinguishes M* (*after-initial M* α) (*after-initial M* β) ω)
unfolding *distinguishes-def* **by** *blast*
qed

have *satisfies-h-condition M V T′ m*
using *satisfaction-conditions*[*OF c1 c2 c3 c4*]
by *blast*
then have *satisfies-h-condition M V* (*set* (*pair-framework M m get-initial-test-suite get-pairs get-separating-traces*)) *m*
unfolding *res-def T′* **.**
then show *?thesis*
using *h-condition-completeness*[*OF assms*(*1*−*7*)]
by *blast*
qed

lemma *pair-framework-finiteness* :
assumes $\bigwedge$ α β t . $\alpha \in L\ M \Longrightarrow \beta \in L\ M \Longrightarrow$ *after-initial M* $\alpha \neq$ *after-initial M* $\beta \Longrightarrow$ *finite-tree* (*get-separating-traces M* (α,*after-initial M* α),(β,*after-initial M* β)) *t*)
and *finite-tree* (*get-initial-test-suite M m*)
and $\bigwedge$ α q' β q'' . $((\alpha,q'),(\beta,q'')) \in$ *list.set* (*get-pairs M m*) $\Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge$ *after-initial M* $\alpha \neq$ *after-initial M* $\beta \wedge q' =$ *after-initial M* $\alpha \wedge q'' =$ *after-initial M* β
shows *finite-tree* (*pair-framework M m get-initial-test-suite get-pairs get-separating-traces*)
proof −

define *T* **where** *T*: *T* = *get-initial-test-suite M m*
moreover define *pairs* **where** *pairs*: *pairs* = *get-pairs M m*
moreover define *distExtension* **where** *distExtension*: *distExtension* = (λ *t* $((\alpha,q'),(\beta,q''))$. *let tDist* = *get-separating-traces M* $((\alpha,q'),(\beta,q''))$ *t*
in combine-after (*combine-after t* α *tDist*) β *tDist*)
ultimately have *res-def*: *pair-framework M m get-initial-test-suite get-pairs get-separating-traces* = *foldl distExtension T pairs*
unfolding *pair-framework-def Let-def* **by** *auto*

have $\bigwedge \alpha\ q'\ \beta\ q''$. $((\alpha,q'),(\beta,q'')) \in$ *list.set pairs* $\Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \wedge q' =$ *after-initial* $M\ \alpha \wedge q'' =$ *after-initial* $M\ \beta$
using *assms*(*3*) **unfolding** *pairs* **by** *auto*
then show *?thesis*
unfolding *res-def* **proof** (*induction pairs rule*: *rev-induct*)
case *Nil*
then show *?case*
using *from-list-finite-tree assms*(*2*)
unfolding *T*
by *auto*
next
case (*snoc a pairs*)
then have *p1*: $\bigwedge \alpha\ q'\ \beta\ q''$. $((\alpha,q'),(\beta,q'')) \in$ *list.set pairs* $\Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \wedge q' =$ *after-initial* $M\ \alpha \wedge q'' =$ *after-initial* $M\ \beta$
by (*metis butlast-snoc in-set-butlastD*)

obtain $\alpha\ q'\ \beta\ q''$ **where** $a = ((\alpha,q'),(\beta,q''))$
by (*metis prod.collapse*)
then have *foldl distExtension T* (*pairs* @ [*a*]) = *distExtension* (*foldl distExtension T pairs*) $((\alpha,q'),(\beta,q''))$
by *auto*
also have ... = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α (*get-separating-traces M* $((\alpha,q'),(\beta,q''))$ (*foldl distExtension T pairs*))) β (*get-separating-traces M* $((\alpha,q'),(\beta,q''))$ (*foldl distExtension T pairs*))
using *distExtension*
by (*metis* (*no-types*, *lifting*) *case-prod-conv*)
finally have *foldl distExtension T* (*pairs* @ [*a*]) = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α (*get-separating-traces M* $((\alpha,q'),(\beta,q''))$ (*foldl distExtension T pairs*))) β (*get-separating-traces M* $((\alpha,q'),(\beta,q''))$ (*foldl distExtension T pairs*))
.
moreover define *ta* **where** *ta*: *ta* = (*after* (*foldl distExtension T pairs*) α)
moreover define *tb* **where** *tb*: *tb* = (*after* (*foldl distExtension T pairs*) β)
moreover define *dist* **where** *dist*: *dist* = (*get-separating-traces M* $((\alpha,q'),(\beta,q''))$ (*foldl distExtension T pairs*))
ultimately have $*$: *foldl distExtension T* (*pairs* @ [*a*]) = *combine-after* (*combine-after* (*foldl distExtension T pairs*) α *dist*) β *dist*
by *auto*

have $((\alpha,q'),(\beta,q'')) \in$ *list.set* (*a*#*pairs*)
unfolding ‹$a = ((\alpha,q'),(\beta,q''))$› **by** *auto*
then have $\alpha \in L\ M \wedge \beta \in L\ M \wedge$ *after-initial* $M\ \alpha \neq$ *after-initial* $M\ \beta \wedge q' =$ *after-initial* $M\ \alpha \wedge q'' =$ *after-initial* $M\ \beta$
using *snoc.prems*
by *auto*
then have *finite-tree dist*

using *assms*(*1*) **unfolding** *dist* **by** *auto*
moreover have *finite-tree* (*foldl distExtension T pairs*)
using *snoc.IH*[*OF p1*] **by** *auto*
ultimately show *?case*
unfolding ∗
using *combine-after-finite-tree* **by** *blast*
qed
qed

end

21 Intermediate Implementations

This theory implements various functions to be supplied to the H, SPY, and Pair-Frameworks.

theory *Intermediate-Implementations*
imports *H-Framework SPY-Framework Pair-Framework ../Distinguishability Automatic-Refinement.Misc*
begin

21.1 Functions for the Pair Framework

definition *get-initial-test-suite-H* :: ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒

nat ⇒
('*b*×'*c*) *prefix-tree*
where
get-initial-test-suite-H V M m =
(*let*
rstates = *reachable-states-as-list M*;
n = *size-r M*;
iM = *inputs-as-list M*;
T = *from-list* (*concat* (*map* (λ*q* . *map* (λ*τ*. (*V q*)@*τ*) (*h-extensions M q* (*m*−*n*))) *rstates*))
in T)

lemma *get-initial-test-suite-H-set-and-finite* :
shows {(*V q*)@*io*@[(*x*,*y*)] | *q io x y* . *q* ∈ *reachable-states M* ∧ *io* ∈ *LS M q* ∧ *length io* ≤ *m* − *size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} ⊆ *set* (*get-initial-test-suite-H V M m*)
and *finite-tree* (*get-initial-test-suite-H V M m*)
proof −

define *rstates* **where** *rstates* = *reachable-states-as-list M*
moreover define *n* **where** *n* = *size-r M*
moreover define *iM* **where** *iM* = *inputs-as-list M*

moreover define T **where** T = *from-list* (*concat* (*map* (λq . *map* ($\lambda\tau$. (V q)@τ) (*h-extensions* M q ($m-n$))) *rstates*))
ultimately have *res*: *get-initial-test-suite-H* V M m = T
unfolding *get-initial-test-suite-H-def Let-def* **by** *auto*

define $\mathcal{X}$ **where** $\mathcal{X}$: $\mathcal{X}$ = (λ q . {*io*@[(x,y)] | *io* x y . *io* $\in$ *LS* M q $\wedge$ *length io* $\leq$ $m-n$ $\wedge$ x $\in$ *inputs* M $\wedge$ y $\in$ *outputs* M})

have *list.set rstates* = *reachable-states* M
unfolding *rstates-def reachable-states-as-list-set* **by** *simp*

have { (V q) @ τ | q τ . q $\in$ *reachable-states* M $\wedge$ τ $\in$ $\mathcal{X}$ q} $\subseteq$ *set* T
proof
fix *io* **assume** *io* $\in$ { (V q) @ τ | q τ . q $\in$ *reachable-states* M $\wedge$ τ $\in$ $\mathcal{X}$ q}
then obtain q τ **where** *io* = (V q) @ τ
and q $\in$ *reachable-states* M
and τ $\in$ $\mathcal{X}$ q
by *blast*

have τ $\in$ *list.set* (*h-extensions* M q (m − n))
using ‹τ $\in$ $\mathcal{X}$ q› **unfolding** $\mathcal{X}$
using *h-extensions-set*[*OF reachable-state-is-state*[*OF* ‹q $\in$ *reachable-states* M›]]
by *auto*
then have *io* $\in$ *list.set* (*map* ((@) (V q)) (*h-extensions* M q (m − n)))
unfolding ‹*io* = (V q) @ τ› **by** *auto*
moreover have q $\in$ *list.set rstates*
using ‹*list.set rstates* = *reachable-states* M› ‹q $\in$ *reachable-states* M› **by** *auto*
ultimately have *io* $\in$ *list.set* (*concat* (*map* (λq. *map* ((@) (V q)) (*h-extensions* M q (m − n))) *rstates*))
by *auto*
then show *io* $\in$ *set* T
unfolding *T-def from-list-set* **by** *blast*
qed
moreover have { (V q) @ τ | q τ . q $\in$ *reachable-states* M $\wedge$ τ $\in$ $\mathcal{X}$ q} = {(V q)@*io*@[(x,y)] | q *io* x y . q $\in$ *reachable-states* M $\wedge$ *io* $\in$ *LS* M q $\wedge$ *length io* $\leq$ m − *size-r* M $\wedge$ x $\in$ *inputs* M $\wedge$ y $\in$ *outputs* M}
unfolding $\mathcal{X}$ *n-def*[*symmetric*] **by** *force*
ultimately show {(V q)@*io*@[(x,y)] | q *io* x y . q $\in$ *reachable-states* M $\wedge$ *io* $\in$ *LS* M q $\wedge$ *length io* $\leq$ m − *size-r* M $\wedge$ x $\in$ *inputs* M $\wedge$ y $\in$ *outputs* M} $\subseteq$ *set* (*get-initial-test-suite-H* V M m)
unfolding *res* **by** *simp*

show *finite-tree* (*get-initial-test-suite-H* V M m)
unfolding *res T-def*
using *from-list-finite-tree* **by** *auto*
qed

fun *complete-inputs-to-tree* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ '*a* ⇒ '*c list* ⇒ '*b list* ⇒ ('*b* × '*c*) *prefix-tree* **where**
complete-inputs-to-tree M q ys [] = *Prefix-Tree.empty* |
complete-inputs-to-tree M q ys (*x*#*xs*) = *foldl* (λ *t y* . *case h-obs M q x y of None* ⇒ *insert t* [(*x*,*y*)] |
Some q' ⇒ *combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys xs*)) *Prefix-Tree.empty ys*

lemma *complete-inputs-to-tree-finite-tree* :
finite-tree (*complete-inputs-to-tree M q ys xs*)
proof (*induction xs arbitrary*: *q ys*)
case *Nil*
then show *?case* **using** *empty-finite-tree* **by** *auto*
next
case (*Cons x xs*)

define *ys'* **where** *ys'* = *ys*
moreover define *f* **where** *f* = (λ *t y* . *case h-obs M q x y of None* ⇒ *insert t* [(*x*,*y*)] | *Some q'* ⇒ *combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys' xs*))
ultimately have ∗:*complete-inputs-to-tree M q ys* (*x* # *xs*)
= *foldl f Prefix-Tree.empty ys*
by *auto*
moreover have *finite-tree* (*foldl f Prefix-Tree.empty ys*)
proof (*induction ys rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *empty-finite-tree* **by** *auto*
next
case (*snoc y ys*)

define *t* **where** *t* = *foldl* (λ *t y* . *case h-obs M q x y of None* ⇒ *insert t* [(*x*,*y*)] | *Some q'* ⇒ *combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys' xs*)) *Prefix-Tree.empty ys*
then have ∗:*foldl f Prefix-Tree.empty* (*ys*@[*y*])
= (*case h-obs M q x y of None* ⇒ *insert t* [(*x*,*y*)] | *Some q'* ⇒ *combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys' xs*))
unfolding *f-def* **by** *auto*

have *finite-tree t*
using *snoc* **unfolding** *t-def f-def* **by** *force*

have *finite-tree* (*insert t* [(*x*,*y*)])
using ‹*finite-tree t*› *insert-finite-tree* **by** *blast*
moreover have ⋀ *q'* . *finite-tree* (*combine-after t* [(*x*,*y*)] (*complete-inputs-to-tree M q' ys' xs*))
using ‹*finite-tree t*› ‹⋀ *q ys* . *finite-tree* (*complete-inputs-to-tree M q ys xs*)› *combine-after-finite-tree* **by** *blast*
ultimately show *?case*
unfolding ∗ **by** *auto*

```
  qed
  ultimately show ?case by auto
qed

fun complete-inputs-to-tree-initial :: ('a::linorder,'b::linorder,'c::linorder) fsm ⇒ 'b
list ⇒ ('b × 'c) prefix-tree where
  complete-inputs-to-tree-initial M xs = complete-inputs-to-tree M (initial M) (outputs-as-list
M) xs


definition get-initial-test-suite-H-2 :: bool ⇒ ('a,'b,'c) state-cover-assignment ⇒
                                        ('a::linorder,'b::linorder,'c::linorder) fsm ⇒

                                    nat ⇒
                                    ('b×'c) prefix-tree where
  get-initial-test-suite-H-2 c V M m =
    (if c then get-initial-test-suite-H V M m
       else let TS = get-initial-test-suite-H V M m;
                xss = map (map fst) (sorted-list-of-maximal-sequences-in-tree TS);
                ys  = outputs-as-list M
            in
             foldl (λ t xs . combine t (complete-inputs-to-tree-initial M xs)) TS xss)


lemma get-initial-test-suite-H-2-set-and-finite :
shows {(V q)@io@[(x,y)] | q io x y . q ∈ reachable-states M ∧ io ∈ LS M q ∧ length
io ≤ m − size-r M ∧ x ∈ inputs M ∧ y ∈ outputs M} ⊆ set (get-initial-test-suite-H-2
c V M m) (is ?P1)
and finite-tree (get-initial-test-suite-H-2 c V M m) (is ?P2)
proof −
  have ?P1 ∧ ?P2
  proof (cases c)
    case True
    then have get-initial-test-suite-H-2 c V M m = get-initial-test-suite-H V M m
      unfolding get-initial-test-suite-H-2-def by auto
    then show ?thesis
      using get-initial-test-suite-H-set-and-finite
      by fastforce
  next
    case False

    define TS where TS = get-initial-test-suite-H V M m
   moreover define xss where xss = map (map fst) (sorted-list-of-maximal-sequences-in-tree
TS)
    moreover define ys where ys  = outputs-as-list M
    ultimately have get-initial-test-suite-H-2 c V M m = foldl (λ t xs . combine
t (complete-inputs-to-tree M (initial M) ys xs)) TS xss
      unfolding get-initial-test-suite-H-2-def Let-def using False by auto
   moreover have set TS ⊆ set (foldl (λ t xs . combine t (complete-inputs-to-tree
```

M (*initial M*) *ys xs*)) *TS xss*)
using *combine-set* **by** (*induction xss rule*: *rev-induct*; *auto*)
moreover have *finite-tree* (*foldl* (λ *t xs* . *combine t* (*complete-inputs-to-tree M* (*initial M*) *ys xs*)) *TS xss*)
using *complete-inputs-to-tree-finite-tree get-initial-test-suite-H-set-and-finite*(*2*)[*of V M m*] *combine-finite-tree*
unfolding *TS-def*[*symmetric*] **by** (*induction xss rule*: *rev-induct*; *auto*; *blast*)

ultimately show *?thesis*
using *get-initial-test-suite-H-set-and-finite*(*1*)[*of V M m*] **unfolding** *TS-def*[*symmetric*]
by *force*
qed
then show *?P1* **and** *?P2*
by *blast*+
qed

definition *get-pairs-H* :: (*'a*,*'b*,*'c*) *state-cover-assignment* ⇒
(*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒
nat ⇒
(((*'b* × *'c*) *list* × *'a*) × ((*'b* × *'c*) *list* × *'a*)) *list*
where
get-pairs-H V M m =
(*let*
rstates = *reachable-states-as-list M*;
n = *size-r M*;
iM = *inputs-as-list M*;
hMap = *mapping-of* (*map* (λ(*q*,*x*) . ((*q*,*x*), *map* (λ(*y*,*q'*) . (*q*,*x*,*y*,*q'*)) (*sorted-list-of-set* (*h M* (*q*,*x*))))) (*List.product* (*states-as-list M*) *iM*));
hM = (λ *q x* . *case Mapping.lookup hMap* (*q*,*x*) *of Some ts* ⇒ *ts* | *None* ⇒ []);
pairs = *pairs-to-distinguish M V* (λ*q* . *paths-up-to-length-with-targets q hM iM* ((*m*−*n*)+*1*)) *rstates*
in
pairs)

lemma *get-pairs-H-set* :
assumes *observable M*
and *is-state-cover-assignment M V*
shows
⋀ α β . (α,β) ∈ (*V* ‘ *reachable-states M*) × (*V* ‘ *reachable-states M*)
∪ (*V* ‘ *reachable-states M*) × { (*V q*) @ τ | *q* τ . *q* ∈ *reachable-states M* ∧ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}}
∪ (⋃ *q* ∈ *reachable-states M* . ⋃ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} . { (*V q*) @ τ′ | τ′ . τ′ ∈ *list.set* (*prefixes* τ)} × {(*V q*)@τ}) ⟹
α ∈ *L M* ⟹ β ∈ *L M* ⟹ *after-initial M* α ≠ *after-initial M* β

$\Longrightarrow$
$((\alpha,\textit{after-initial } M\ \alpha),(\beta,\textit{after-initial } M\ \beta)) \in \textit{list.set } (\textit{get-pairs-H } V\ M\ m)$
and $\bigwedge \alpha\ q'\ \beta\ q''\ .\ ((\alpha,q'),(\beta,q'')) \in \textit{list.set } (\textit{get-pairs-H } V\ M\ m) \Longrightarrow \alpha \in L\ M \wedge \beta \in L\ M \wedge \textit{after-initial } M\ \alpha \neq \textit{after-initial } M\ \beta \wedge q' = \textit{after-initial } M\ \alpha \wedge q'' = \textit{after-initial } M\ \beta$
proof −

define *rstates* **where** *rstates* = *reachable-states-as-list M*
moreover define *n* **where** *n* = *size-r M*
moreover define *iM* **where** *iM* = *inputs-as-list M*
moreover define *hMap′* **where** *hMap′* = *mapping-of* (*map* (λ(*q*,*x*) . ((*q*,*x*), *map* (λ(*y*,*q′*) . (*q*,*x*,*y*,*q′*)) (*sorted-list-of-set* (*h M* (*q*,*x*))))) (*List.product* (*states-as-list M*) *iM*))
moreover define *hM′* **where** *hM′* = (λ *q x* . *case Mapping.lookup hMap′* (*q*,*x*) *of Some ts* ⇒ *ts* | *None* ⇒ []) **ultimately have** *get-pairs-H V M m* = *pairs-to-distinguish M V* (λ*q* . *paths-up-to-length-with-targets q hM′ iM* ((*m*−*n*)+*1*)) *rstates*
unfolding *get-pairs-H-def Let-def* **by** *force*

define *hMap* **where** *hMap* = *map-of* (*map* (λ(*q*,*x*) . ((*q*,*x*), *map* (λ(*y*,*q′*) . (*q*,*x*,*y*,*q′*)) (*sorted-list-of-set* (*h M* (*q*,*x*))))) (*List.product* (*states-as-list M*) *iM*))
define *hM* **where** *hM* = (λ *q x* . *case hMap* (*q*,*x*) *of Some ts* ⇒ *ts* | *None* ⇒ [])

have *distinct* (*List.product* (*states-as-list M*) *iM*)
using *states-as-list-distinct inputs-as-list-distinct distinct-product*
unfolding *iM-def*
by *blast*

then have *Mapping.lookup hMap′* = *hMap*
using *mapping-of-map-of*
unfolding *hMap-def hMap′-def*
using *map-pair-fst-helper*[*of* λ *q x* . *map* (λ(*y*,*q′*) . (*q*,*x*,*y*,*q′*)) (*sorted-list-of-set* (*h M* (*q*,*x*)))]
by (*metis* (*no-types*, *lifting*))
then have *hM′* = *hM*
unfolding *hM′-def hM-def*
by *meson*
moreover define *pairs* **where** *pairs* = *pairs-to-distinguish M V* (λ*q* . *paths-up-to-length-with-targets q hM iM* ((*m*−*n*)+*1*)) *rstates*
ultimately have *res*: *get-pairs-H V M m* = *pairs*
unfolding ‹*get-pairs-H V M m* = *pairs-to-distinguish M V* (λ*q* . *paths-up-to-length-with-targets q hM′ iM* ((*m*−*n*)+*1*)) *rstates*›
by *force*

have ∗:*list.set rstates* = *reachable-states M*

unfolding *rstates-def reachable-states-as-list-set* **by** *simp*

define $\mathcal{X}'$ **where** $\mathcal{X}'$: $\mathcal{X}' = (\lambda q\ .\ \textit{paths-up-to-length-with-targets}\ q\ hM\ iM\ ((m-n)+1))$

have $**$: $\bigwedge\ q\ p\ q'\ .\ q \in \textit{reachable-states}\ M \Longrightarrow (p,q') \in \textit{list.set}\ (\mathcal{X}'\ q) \longleftrightarrow \textit{path}\ M\ q\ p \wedge \textit{target}\ q\ p = q' \wedge \textit{length}\ p \leq m-n+1$
proof −
fix $q\ p\ q'$ **assume** $q \in \textit{reachable-states}\ M$

define *qxPairs* **where** *qxPairs*: $\textit{qxPairs} = (\textit{List.product}\ (\textit{states-as-list}\ M)\ iM)$
moreover define *mapList* **where** *mapList*: $\textit{mapList} = (\textit{map}\ (\lambda(q,x)\ .\ ((q,x), \textit{map}\ (\lambda(y,q')\ .\ (q,x,y,q'))\ (\textit{sorted-list-of-set}\ (h\ M\ (q,x)))))\ \textit{qxPairs})$
ultimately have *hMap'*: $hMap = \textit{map-of}\ \textit{mapList}$
unfolding *hMap-def* **by** *simp*

have $\textit{distinct}\ (\textit{states-as-list}\ M)$ **and** $\textit{distinct}\ iM$
unfolding *iM-def*
by *auto*
then have $\textit{distinct}\ \textit{qxPairs}$
unfolding *qxPairs* **by** (*simp add*: *distinct-product*)
moreover have $(\textit{map fst mapList}) = \textit{qxPairs}$
unfolding *mapList* **by** (*induction qxPairs*; *auto*)
ultimately have $\textit{distinct}\ (\textit{map fst mapList})$
by *auto*

have $\bigwedge\ q\ x\ .\ hM\ q\ x = \textit{map}\ (\lambda(y,\ q').\ (q,\ x,\ y,\ q'))\ (\textit{sorted-list-of-set}\ (h\ M\ (q,\ x)))$
proof −
fix $q\ x$
show $hM\ q\ x = \textit{map}\ (\lambda(y,\ q').\ (q,\ x,\ y,\ q'))\ (\textit{sorted-list-of-set}\ (h\ M\ (q,\ x)))$
proof $(\textit{cases}\ q \in \textit{states}\ M \wedge x \in \textit{inputs}\ M)$
case *False*
then have $(h\ M\ (q,\ x)) = \{\}$
unfolding *h-simps* **using** *fsm-transition-source fsm-transition-input* **by** *fastforce*
then have $\textit{map}\ (\lambda(y,\ q').\ (q,\ x,\ y,\ q'))\ (\textit{sorted-list-of-set}\ (h\ M\ (q,\ x))) = []$
by *auto*

have $q \notin \textit{list.set}\ (\textit{states-as-list}\ M) \vee x \notin \textit{list.set}\ iM$
using *False* **unfolding** *states-as-list-set iM-def inputs-as-list-set* **by** *simp*
then have $(q,x) \notin \textit{list.set}\ \textit{qxPairs}$
unfolding *qxPairs* **by** *auto*
then have $\nexists\ y\ .\ ((q,x),y) \in \textit{list.set}\ \textit{mapList}$
unfolding *mapList* **by** *auto*
then have $hMap\ (q,x) = \textit{None}$
unfolding *hMap'* **using** *map-of-eq-Some-iff*[*OF* ‹*distinct* (*map fst mapList*)›]
by (*meson option.exhaust*)

then show *?thesis*
using ‹*map* (λ(*y*, *q'*). (*q*, *x*, *y*, *q'*)) (*sorted-list-of-set* (*h M* (*q*, *x*))) = []›
unfolding *hM-def* **by** *auto*
next
case *True*
then have *q* ∈ *list.set* (*states-as-list M*) ∧ *x* ∈ *list.set iM*
unfolding *states-as-list-set iM-def inputs-as-list-set* **by** *simp*
then have (*q*,*x*) ∈ *list.set qxPairs*
unfolding *qxPairs* **by** *auto*
then have ((*q*,*x*),*map* (λ(*y*, *q'*). (*q*, *x*, *y*, *q'*)) (*sorted-list-of-set* (*h M* (*q*, *x*)))) ∈ *list.set mapList*
unfolding *mapList* **by** *auto*
then have *hMap* (*q*,*x*) = *Some* (*map* (λ(*y*, *q'*). (*q*, *x*, *y*, *q'*)) (*sorted-list-of-set* (*h M* (*q*, *x*))))
unfolding *hMap'* **using** *map-of-eq-Some-iff*[*OF* ‹*distinct* (*map fst mapList*)›]
by (*meson option.exhaust*)
then show *?thesis*
unfolding *hM-def* **by** *auto*
qed
qed
then have *hM-alt-def*: *hM* = (λ *q x* . *map* (λ(*y*, *q'*). (*q*, *x*, *y*, *q'*)) (*sorted-list-of-set* (*h M* (*q*, *x*))))
by *auto*

show (*p*,*q'*) ∈ *list.set* ($\mathcal{X}'$ *q*) ⟷ *path M q p* ∧ *target q p* = *q'* ∧ *length p* ≤ *m*−*n*+*1*
unfolding $\mathcal{X}'$ *hM-alt-def iM-def*
paths-up-to-length-with-targets-set[*OF reachable-state-is-state*[*OF* ‹*q* ∈ *reachable-states M*›]]
by *blast*
qed

show ⋀ α β . (α,β) ∈ (*V* ‘ *reachable-states M*) × (*V* ‘ *reachable-states M*)
∪ (*V* ‘ *reachable-states M*) × { (*V q*) @ τ | *q* τ . *q* ∈ *reachable-states M* ∧ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}}
∪ (⋃ *q* ∈ *reachable-states M* . ⋃ τ ∈ {*io*@[(*x*,*y*)] | *io x y* . *io* ∈ *LS M q* ∧ *length io* ≤ *m*−*size-r M* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*} . { (*V q*) @ τ' | τ' . τ' ∈ *list.set* (*prefixes* τ)} × {(*V q*)@τ}) ⟹
α ∈ *L M* ⟹ β ∈ *L M* ⟹ *after-initial M* α ≠ *after-initial M* β ⟹
((α,*after-initial M* α),(β,*after-initial M* β)) ∈ *list.set* (*get-pairs-H V M m*)
using *pairs-to-distinguish-containment*[*OF assms*(*1*,*2*) ∗ ∗∗]
unfolding *res pairs-def* $\mathcal{X}'$[*symmetric*] *n-def*[*symmetric*]
by *presburger*

show $\bigwedge$ α q' β q'' . $((\alpha,q'),(\beta,q'')) \in$ *list.set* (*get-pairs-H V M m*) $\Longrightarrow$ $\alpha \in L\ M$ $\wedge$ $\beta \in L\ M$ $\wedge$ *after-initial M* $\alpha \neq$ *after-initial M* β $\wedge$ $q' =$ *after-initial M* α $\wedge$ q'' $=$ *after-initial M* β
  using *pairs-to-distinguish-elems*(*3*,*4*,*5*,*6*,*7*)[*OF assms*(*1*,*2*) $*$ $**$]
  unfolding *res pairs-def* $\mathcal{X}'$[*symmetric*] *n-def*[*symmetric*]
  by *blast*
qed

21.2 Functions of the SPYH-Method

21.2.1 Heuristic Functions for Selecting Traces to Extend

fun *estimate-growth* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ (*'b* $\times$ *'c*) *list*) $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *'b* $\Rightarrow$ *'c* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
  estimate-growth M dist-fun q1 q2 x y errorValue= (*case h-obs M q1 x y of*
    None $\Rightarrow$ (*case h-obs M q1 x y of*
      None $\Rightarrow$ *errorValue* |
      Some q2' $\Rightarrow$ *1*) |
    Some q1' $\Rightarrow$ (*case h-obs M q2 x y of*
      None $\Rightarrow$ *1* |
      Some q2' $\Rightarrow$ *if q1'* $=$ *q2'* $\vee$ {*q1'*,*q2'*} $=$ {*q1*,*q2*}
        then errorValue
        else 1 $+$ *2* $*$ (*length* (*dist-fun q1 q2*))))

lemma *estimate-growth-result* :
  assumes *observable M*
  and *minimal M*
  and *q1* $\in$ *states M*
  and *q2* $\in$ *states M*
  and *estimate-growth M dist-fun q1 q2 x y errorValue* $<$ *errorValue*
shows $\exists$ γ . *distinguishes M q1 q2* ([(*x*,*y*)]@γ)
proof (*cases h-obs M q1 x y*)
  case *None*
  show *?thesis* **proof** (*cases h-obs M q2 x y*)
    case *None*
    then show *?thesis*
      using ‹*h-obs M q1 x y* $=$ *None*› *assms*(*5*)
      by *auto*
  next
    case (*Some a*)
    then have *distinguishes M q1 q2* [(*x*,*y*)]
      using *h-obs-distinguishes*[*OF assms*(*1*) - *None*] *distinguishes-sym*
      by *metis*
    then show *?thesis*
      by *auto*
  qed

```
next
  case (Some q1')
  show ?thesis proof (cases h-obs M q2 x y)
    case None
    then have distinguishes M q1 q2 [(x,y)]
      using h-obs-distinguishes[OF assms(1) Some]
      by metis
    then show ?thesis
      by auto
  next
    case (Some q2')
    then have q1' ≠ q2'
      using ‹h-obs M q1 x y = Some q1'› assms(5)
      by auto
    then obtain γ where distinguishes M q1' q2' γ
      using h-obs-state[OF ‹h-obs M q1 x y = Some q1'›]
      using h-obs-state[OF Some]
      using ‹minimal M› unfolding minimal.simps distinguishes-def
      by blast
    then have distinguishes M q1 q2 ([(x,y)]@γ)
      using h-obs-language-iff[OF assms(1), of x y γ]
      using ‹h-obs M q1 x y = Some q1'› Some
      unfolding distinguishes-def
      by force
    then show ?thesis
      by blast
  qed
qed


fun shortest-list-or-default :: 'a list list ⇒ 'a list ⇒ 'a list where
  shortest-list-or-default xs x = foldl (λ a b . if length a < length b then a else b)
x xs

lemma shortest-list-or-default-elem :
  shortest-list-or-default xs x ∈ Set.insert x (list.set xs)
  by (induction xs rule: rev-induct; auto)

fun shortest-list :: 'a list list ⇒ 'a list where
  shortest-list [] = undefined |
  shortest-list (x#xs) = shortest-list-or-default xs x

lemma shortest-list-elem :
  assumes xs ≠ []
shows shortest-list xs ∈ list.set xs
  using assms shortest-list-or-default-elem
  by (metis list.simps(15) shortest-list.elims)

fun shortest-list-in-tree-or-default :: 'a list list ⇒ 'a prefix-tree ⇒ 'a list ⇒ 'a list
```

where
shortest-list-in-tree-or-default xs T x = foldl (λ *a b . if isin T a* ∧ *length a* < *length b then a else b*) *x xs*

lemma *shortest-list-in-tree-or-default-elem* :
shortest-list-in-tree-or-default xs T x ∈ *Set.insert x* (*list.set xs*)
by (*induction xs rule*: *rev-induct*; *auto*)

fun *has-leaf* :: ('*b*×'*c*) *prefix-tree* ⇒ '*d* ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ('*b*×'*c*) *list* ⇒ *bool* **where**
has-leaf T G cg-lookup α =
(*find* (λ β . *is-maximal-in T* β) (α # *cg-lookup G* α) ≠ *None*)

fun *has-extension* :: ('*b*×'*c*) *prefix-tree* ⇒ '*d* ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ('*b*×'*c*) *list* ⇒ '*b* ⇒ '*c* ⇒ *bool* **where**
has-extension T G cg-lookup α *x y* =
(*find* (λ β . *isin T* (β@[(*x*,*y*)])) (α # *cg-lookup G* α) ≠ *None*)

fun *get-extension* :: ('*b*×'*c*) *prefix-tree* ⇒ '*d* ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ('*b*×'*c*) *list* ⇒ '*b* ⇒ '*c* ⇒ ('*b*×'*c*) *list option* **where**
get-extension T G cg-lookup α *x y* =
(*find* (λ β . *isin T* (β@[(*x*,*y*)])) (α # *cg-lookup G* α))

fun *get-prefix-of-separating-sequence* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ ('*b*×'*c*) *prefix-tree* ⇒ '*d* ⇒ ('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒ ('*a* ⇒ '*a* ⇒ ('*b*×'*c*) *list*) ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list* ⇒ *nat* ⇒ (*nat* × ('*b*×'*c*) *list*) **where**
get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v 0 = (*1*,[]) |
get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v (*Suc k*)= (*let*
u' = *shortest-list-or-default* (*cg-lookup G u*) *u*;
v' = *shortest-list-or-default* (*cg-lookup G v*) *v*;
su = *after-initial M u*;
sv = *after-initial M v*;
bestPrefix0 = *get-distinguishing-trace su sv*;
minEst0 = *length bestPrefix0* + (*if* (*has-leaf T G cg-lookup u'*) *then 0 else length u'*) + (*if* (*has-leaf T G cg-lookup v'*) *then 0 else length v'*);
errorValue = *Suc minEst0*;
XY = *List.product* (*inputs-as-list M*) (*outputs-as-list M*);
tryIO = (λ (*minEst*,*bestPrefix*) (*x*,*y*) .
if minEst = *0*
then (*minEst*,*bestPrefix*)
else (*case get-extension T G cg-lookup u' x y of*

Some u'' ⇒ (*case get-extension T G cg-lookup* v' *x y of*
Some v'' ⇒ *if* (*h-obs M su x y* = *None*) ≠ (*h-obs M sv x y* = *None*)
then (*0*,[])
else if h-obs M su x y = *h-obs M sv x y*
then (*minEst*,*bestPrefix*)
else (*let* (*e*,*w*) = *get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace* (u''@[(*x*,*y*)]) (v''@[(*x*,*y*)]) *k*
in if e = *0*
then (*0*,[])
else if e ≤ *minEst*
then (*e*,(*x*,*y*)#*w*)
else (*minEst*,*bestPrefix*)) |
None ⇒ (*let e* = *estimate-growth M get-distinguishing-trace su sv x y errorValue*;
e' = *if e* ≠ *1*
then if has-leaf T G cg-lookup u''
then e + *1*
else if ¬(*has-leaf T G cg-lookup* (u''@[(*x*,*y*)]))
then e + *length* u' + *1*
else e
else e;
e'' = e' + (*if* ¬(*has-leaf T G cg-lookup* v') *then length* v' *else 0*)
in if e'' ≤ *minEst*
then (e'',[(*x*,*y*)])
else (*minEst*,*bestPrefix*))) |
None ⇒ (*case get-extension T G cg-lookup* v' *x y of*
Some v'' ⇒ (*let e* = *estimate-growth M get-distinguishing-trace su sv x y errorValue*;
e' = *if e* ≠ *1*
then if has-leaf T G cg-lookup v''
then e + *1*
else if ¬(*has-leaf T G cg-lookup* (v''@[(*x*,*y*)]))
then e + *length* v' + *1*
else e
else e;
e'' = e' + (*if* ¬(*has-leaf T G cg-lookup* u') *then length* u' *else 0*)
in if e'' ≤ *minEst*
then (e'',[(*x*,*y*)])
else (*minEst*,*bestPrefix*)) |
None ⇒ (*minEst*,*bestPrefix*))))
in if ¬ *isin T* u' ∨ ¬ *isin T* v'
then (*errorValue*,[])
else foldl tryIO (*minEst0*,[]) *XY*)

lemma *estimate-growth-Suc* :

assumes *errorValue* > *0*
shows *estimate-growth M get-distinguishing-trace q1 q2 x y errorValue* > *0*
using *assms* **unfolding** *estimate-growth.simps*
by (*cases FSM.h-obs M q1 x y*; *cases FSM.h-obs M q2 x y*; *fastforce*)

lemma *get-extension-result*:
assumes $u \in L\ M1$ **and** $u \in L\ M2$
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *get-extension T G cg-lookup u x y* = *Some u'*
shows *converge M1 u u'* **and** $u' \in L\ M2 \Longrightarrow$ *converge M2 u u'* **and** $u'@[(x,y)] \in$ *set T*
proof −

have *find* ($\lambda\ \beta$. *isin T* (β@[(*x*,*y*)])) (*u* # *cg-lookup G u*) = *Some u'*
using *assms*(*4*)
by *auto*
then have *isin T* (*u'*@[(*x*,*y*)])
using *find-condition* **by** *metis*
then show *u'*@[(*x*,*y*)] ∈ *set T*
by *auto*

have *u'* ∈ *Set.insert u* (*list.set* (*cg-lookup G u*))
using ‹*find* ($\lambda\ \beta$. *isin T* (β@[(*x*,*y*)])) (*u* # *cg-lookup G u*) = *Some u'*›
by (*metis find-set list.simps*(*15*))
then show *converge M1 u u'* **and** $u' \in L\ M2 \Longrightarrow$ *converge M2 u u'*
using *assms*(*1*,*2*,*3*)
by (*metis converge.elims*(*3*) *convergence-graph-lookup-invar-def insert-iff*)+
qed

lemma *get-prefix-of-separating-sequence-result* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and $u \in L\ M1$ **and** $u \in L\ M2$
and $v \in L\ M1$ **and** $v \in L\ M2$
and *after-initial M1 u* ≠ *after-initial M1 v*
and $\bigwedge\ \alpha\ \beta\ q1\ q2$. $q1 \in$ *states M1* $\Longrightarrow q2 \in$ *states M1* $\Longrightarrow q1 \neq q2 \Longrightarrow$ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *L M1* ∩ *set T* = *L M2* ∩ *set T*
shows *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) = *0* $\Longrightarrow \neg$ *converge M2 u v*
and *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) ≠ *0* $\Longrightarrow$ $\exists\ \gamma$. *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) ((*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*))@γ)
proof −

have (*fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) = *0* ⟶ ¬ *converge M2 u v*)
∧ (*fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*) ≠ *0* ⟶ (∃ *γ* . *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) ((*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*))@*γ*)))
using *assms*(*4*,*5*,*6*,*7*,*8*)
proof (*induction k arbitrary*: *u v*)
case *0*

then have ∃ *γ* . *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) *γ*
using ‹*minimal M1*› **unfolding** *minimal.simps*
by (*meson after-is-state assms*(*1*) *assms*(*9*))
then show *?case*
unfolding *get-prefix-of-separating-sequence.simps fst-conv snd-conv*
by *auto*
next
case (*Suc k*)

define *u′* **where** *u′*: *u′* = *shortest-list-or-default* (*cg-lookup G u*) *u*
define *v′* **where** *v′*: *v′* = *shortest-list-or-default* (*cg-lookup G v*) *v*
define *su* **where** *su*: *su* = *after-initial M1 u*
define *sv* **where** *sv*: *sv* = *after-initial M1 v*
define *bestPrefix0* **where** *bestPrefix0*: *bestPrefix0* = *get-distinguishing-trace su sv*
define *minEst0* **where** *minEst0*: *minEst0* = *length bestPrefix0* + (*if* (*has-leaf T G cg-lookup u′*) *then 0 else length u′*) + (*if* (*has-leaf T G cg-lookup v′*) *then 0 else length v′*)
define *errorValue* **where** *errorValue*: *errorValue* = *Suc minEst0*
define *XY* **where** *XY*: *XY* = *List.product* (*inputs-as-list M1*) (*outputs-as-list M1*)
define *tryIO* **where** *tryIO*: *tryIO* = (*λ* (*minEst*,*bestPrefix*) (*x*,*y*) .
if minEst = *0*
then (*minEst*,*bestPrefix*)
else (*case get-extension T G cg-lookup u′ x y of*
Some u″ ⇒ (*case get-extension T G cg-lookup v′ x y of*
Some v″ ⇒ *if* (*h-obs M1 su x y* = *None*) ≠ (*h-obs M1 sv x y* = *None*)
then (*0*,[])
else if h-obs M1 su x y = *h-obs M1 sv x y*
then (*minEst*,*bestPrefix*)
else (*let* (*e*,*w*) = *get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (*u″*@[(*x*,*y*)]) (*v″*@[(*x*,*y*)]) *k*
in if e = *0*
then (*0*,[])
else if e ≤ *minEst*
then (*e*,(*x*,*y*)#*w*)
else (*minEst*,*bestPrefix*)) |
None ⇒ (*let e* = *estimate-growth M1 get-distinguishing-trace*

su sv x y errorValue;
$e' = \textit{if } e \neq 1$
$\textit{then if has-leaf } T\ G\ \textit{cg-lookup } u''$
$\textit{then } e + 1$
$\textit{else if } \neg(\textit{has-leaf } T\ G\ \textit{cg-lookup } (u''@[(x,y)]))$
$\textit{then } e + \textit{length } u' + 1$
$\textit{else } e$
$\textit{else } e;$
$e'' = e' + (\textit{if } \neg(\textit{has-leaf } T\ G\ \textit{cg-lookup } v')\ \textit{then length}$
$v'\ \textit{else } 0)$
$\textit{in if } e'' \leq \textit{minEst}$
$\textit{then } (e'',[(x,y)])$
$\textit{else } (\textit{minEst},\textit{bestPrefix}))) \mid$
$\textit{None} \Rightarrow (\textit{case get-extension } T\ G\ \textit{cg-lookup } v'\ x\ y\ \textit{of}$
$\textit{Some } v'' \Rightarrow (\textit{let } e = \textit{estimate-growth M1 get-distinguishing-trace}$
su sv x y errorValue;
$e' = \textit{if } e \neq 1$
$\textit{then if has-leaf } T\ G\ \textit{cg-lookup } v''$
$\textit{then } e + 1$
$\textit{else if } \neg(\textit{has-leaf } T\ G\ \textit{cg-lookup } (v''@[(x,y)]))$
$\textit{then } e + \textit{length } v' + 1$
$\textit{else } e$
$\textit{else } e;$
$e'' = e' + (\textit{if } \neg(\textit{has-leaf } T\ G\ \textit{cg-lookup } u')\ \textit{then length}$
$u'\ \textit{else } 0)$
$\textit{in if } e'' \leq \textit{minEst}$
$\textit{then } (e'',[(x,y)])$
$\textit{else } (\textit{minEst},\textit{bestPrefix})) \mid$
$\textit{None} \Rightarrow (\textit{minEst},\textit{bestPrefix}))))$

have *res'*: (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v (Suc k)*) =
(*if* ¬ *isin T u'* ∨ ¬ *isin T v' then* (*errorValue*,[]) *else foldl tryIO* (*minEst0*,[]) *XY*)
unfolding *tryIO XY errorValue minEst0 bestPrefix0 sv su v' u'*
unfolding *get-prefix-of-separating-sequence.simps Let-def*
by *force*

show *?case* **proof** (*cases* ¬ *isin T u'* ∨ ¬ *isin T v'*)
case *True*
then have ∗:(*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v (Suc k)*) = (*errorValue*,[])
using *res'* **by** *auto*

show *?thesis*
unfolding ∗ *fst-conv snd-conv errorValue*
by (*metis Suc.prems*(*1*,*3*,*5*) *Zero-not-Suc after-is-state append-Nil assms*(*1*)

```
assms(9))
  next
    case False

  then have res: (get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace
u v (Suc k)) = foldl tryIO (minEst0,[]) XY
      using res' by auto


    have converge M1 u u' and converge M2 u u'
      unfolding u'
    using shortest-list-or-default-elem[of cg-lookup G u u] assms(10) Suc.prems(1,2,3)
      by (metis converge.elims(3) convergence-graph-lookup-invar-def insertE)+

    have converge M1 v v' and converge M2 v v'
      unfolding v'
      using shortest-list-or-default-elem[of cg-lookup G v v] assms(10) Suc.prems
      by (metis converge.elims(3) convergence-graph-lookup-invar-def insertE)+

    have su ∈ states M1
      unfolding su
      using after-is-state[OF assms(1) Suc.prems(1)] .

    have sv ∈ states M1
      unfolding sv
      using after-is-state[OF assms(1) Suc.prems(3)] .

    define P where P: P = (λ (ew :: (nat × ('b × 'c) list)) .
                                (fst ew = 0 ⟶ ¬ converge M2 u v)
                                    ∧ (fst ew ≠ 0 ⟶  (∃ γ . distinguishes M1
(after-initial M1 u) (after-initial M1 v) ((snd ew)@γ))))

    have P (minEst0,[])
    proof −
      have distinguishes M1 (after-initial M1 u) (after-initial M1 v) bestPrefix0
        using assms(9)[of su sv]
        using ‹su ∈ states M1› ‹sv ∈ states M1›
        using Suc.prems(5)
        unfolding bestPrefix0 su sv
        by blast

      moreover have minEst0 ≠ 0
        unfolding minEst0
          using calculation distinguishes-not-Nil[OF - after-is-state[OF assms(1)
Suc.prems(1)] after-is-state[OF assms(1) Suc.prems(3)]]
        by auto
      ultimately show ?thesis
        unfolding P fst-conv snd-conv
        by (metis append.left-neutral)
```

qed

have *errorValue* > *0*
unfolding *errorValue* **by** *auto*

have $\bigwedge$ *x y e w* . *e* < *errorValue* $\Longrightarrow$ *P* (*e*,*w*) $\Longrightarrow$ *P* (*tryIO* (*e*,*w*) (*x*,*y*)) $\wedge$ *fst* (*tryIO* (*e*,*w*) (*x*,*y*)) $\leq$ *e*
proof −
fix *x y e w*
assume *e* < *errorValue* **and** *P* (*e*,*w*)

have ∗:$\bigwedge$ *x y a b f* . (*case* (*x*, *y*) *of* (*x*, *y*) $\Rightarrow$ (λ(*a*, *b*). *f x y a b*)) (*a*,*b*) = *f x y a b*
by *auto*

show *P* (*tryIO* (*e*,*w*) (*x*,*y*)) $\wedge$ *fst* (*tryIO* (*e*,*w*) (*x*,*y*)) $\leq$ *e*
proof (*cases e* = *0*)
case *True*
then have *tryIO* (*e*,*w*) (*x*,*y*) = (*e*,*w*)
unfolding *P tryIO fst-conv snd-conv case-prod-conv*
by *auto*
then show *?thesis*
using ‹*P* (*e*,*w*)›
by *auto*
next
case *False*
show *?thesis*
proof (*cases get-extension T G cg-lookup u′ x y*)
case *None*

show *?thesis*
proof (*cases get-extension T G cg-lookup v′ x y*)
case *None*
then have *tryIO* (*e*,*w*) (*x*,*y*) = (*e*,*w*)
using ‹*get-extension T G cg-lookup u′ x y* = *None*›
unfolding *tryIO* **by** *auto*
then show *?thesis*
using ‹*P* (*e*,*w*)›
by *auto*
next
case (*Some v″*)

define *c* **where** *c*: *c* = *estimate-growth M1 get-distinguishing-trace su sv x y errorValue*
define *c′* **where** *c′*: *c′* = (*if c* $\neq$ *1 then if has-leaf T G cg-lookup v″ then c* + *1 else if* $\neg$(*has-leaf T G cg-lookup* (*v″*@[(*x*,*y*)])) *then c* + *length v′* + *1 else c else c*)
define *c″* **where** *c″*: *c″* = *c′* + (*if* $\neg$(*has-leaf T G cg-lookup u′*) *then*

length u′ else 0)

have *tryIO* (*e*,*w*) (*x*,*y*) = (*if c*′′ ≤ *e then* (*c*′′,[(*x*,*y*)]) *else* (*e*,*w*))
unfolding *c c*′ *c*′′ *tryIO Let-def*
using *None Some False*
by *auto*

show *?thesis* **proof** (*cases c*′′ ≤ *e*)
case *True*
then have *c*′′ < *errorValue*
using ‹*e* < *errorValue*› **by** *auto*
then have *c*′ < *errorValue*
unfolding *c*′′ **by** *auto*
then have *estimate-growth M1 get-distinguishing-trace su sv x y errorValue* < *errorValue*
unfolding *c*′ *c*
using *add-lessD1* **by** *presburger*

have *c* > *0*
using *estimate-growth-Suc*[*OF* ‹*errorValue* > *0*›] **unfolding** *c*
by *blast*
then have *c*′′ > *0*
unfolding *c*′ *c*′′
using *add-gr-0* **by** *presburger*
then have *c*′′ ≠ *0*
by *auto*
then have *P* (*c*′′,[(*x*,*y*)])
using *True estimate-growth-result*[*OF assms*(*1*,*3*) ‹*su* ∈ *states M1*› ‹*sv* ∈ *states M1*› ‹*estimate-growth M1 get-distinguishing-trace su sv x y errorValue* < *errorValue*›]
unfolding *P fst-conv su sv snd-conv*
by *blast*
then show *?thesis*
using ‹*tryIO* (*e*,*w*) (*x*,*y*) = (*if c*′′ ≤ *e then* (*c*′′,[(*x*,*y*)]) *else* (*e*,*w*))› *True*
by *auto*
next
case *False*
then show *?thesis*
using ‹*tryIO* (*e*,*w*) (*x*,*y*) = (*if c*′′ ≤ *e then* (*c*′′,[(*x*,*y*)]) *else* (*e*,*w*))› ‹*P* (*e*,*w*)›
by *auto*
qed
qed
next
case (*Some u*′′)

show *?thesis* **proof** (*cases get-extension T G cg-lookup v*′ *x y*)
case *None*

define *c* **where** *c*: *c* = *estimate-growth M1 get-distinguishing-trace su sv x y errorValue*
define *c′* **where** *c′*: *c′* = (*if c* ≠ *1 then if has-leaf T G cg-lookup u″ then c* + *1 else if* ¬(*has-leaf T G cg-lookup* (*u″*@[(*x*,*y*)])) *then c* + *length u′* + *1 else c else c*)
define *c″* **where** *c″*: *c″* = *c′* + (*if* ¬(*has-leaf T G cg-lookup v′*) *then length v′ else 0*)

have *tryIO* (*e*,*w*) (*x*,*y*) = (*if c″* ≤ *e then* (*c″*,[(*x*,*y*)]) *else* (*e*,*w*))
unfolding *c c′ c″ tryIO Let-def*
using *None Some False*
by *auto*

show *?thesis* **proof** (*cases c″* ≤ *e*)
case *True*
then have *c″* < *errorValue*
using ‹*e* < *errorValue*› **by** *auto*
then have *c′* < *errorValue*
unfolding *c″* **by** *auto*
then have *estimate-growth M1 get-distinguishing-trace su sv x y errorValue* < *errorValue*
unfolding *c′ c*
using *add-lessD1* **by** *presburger*

have *c* > *0*
using *estimate-growth-Suc*[*OF* ‹*errorValue* > *0*›] **unfolding** *c*
by *blast*
then have *c″* > *0*
unfolding *c′ c″*
using *add-gr-0* **by** *presburger*
then have *c″* ≠ *0*
by *auto*
then have *P* (*c″*,[(*x*,*y*)])
using *True estimate-growth-result*[*OF assms*(*1*,*3*) ‹*su* ∈ *states M1*› ‹*sv* ∈ *states M1*› ‹*estimate-growth M1 get-distinguishing-trace su sv x y errorValue* < *errorValue*›]
unfolding *P fst-conv su sv snd-conv*
by *blast*
then show *?thesis*
using ‹*tryIO* (*e*,*w*) (*x*,*y*) = (*if c″* ≤ *e then* (*c″*,[(*x*,*y*)]) *else* (*e*,*w*))› *True*
by *auto*
next
case *False*
then show *?thesis*
using ‹*tryIO* (*e*,*w*) (*x*,*y*) = (*if c″* ≤ *e then* (*c″*,[(*x*,*y*)]) *else* (*e*,*w*))› ‹*P* (*e*,*w*)›
by *auto*

```
      qed

    next
      case (Some v'')

      have u' ∈ L M1
        using ‹converge M1 u u'› converge.simps by blast
      have v' ∈ L M1
        using ‹converge M1 v v'› converge.simps by blast
      have u' ∈ L M2
        using ‹converge M2 u u'› converge.simps by blast
      have v' ∈ L M2
        using ‹converge M2 v v'› converge.simps by blast


      have converge M1 u' u'' and u'' @ [(x, y)] ∈ set T
            using get-extension-result(1,3)[OF ‹u' ∈ L M1› ‹u' ∈ L M2›
assms(10) ‹get-extension T G cg-lookup u' x y = Some u''›]
        by blast+
      then have converge M1 u u''
        using ‹converge M1 u u'› by auto
      then have u'' ∈ set T ∩ L M1
        using set-prefix[OF ‹u'' @ [(x, y)] ∈ set T›] by auto

      have converge M1 v' v'' and v'' @ [(x, y)] ∈ set T
         using get-extension-result[OF ‹v' ∈ L M1› ‹v' ∈ L M2› assms(10)
‹get-extension T G cg-lookup v' x y = Some v''›]
        by blast+
      then have converge M1 v v''
        using ‹converge M1 v v'› by auto
      then have v'' ∈ set T ∩ L M1
        using set-prefix[OF ‹v'' @ [(x, y)] ∈ set T›] by auto

      show ?thesis proof (cases (h-obs M1 su x y = None) ≠ (h-obs M1 sv
x y = None))
        case True

        then have tryIO (e,w) (x,y) = (0,[])
          using Some ‹get-extension T G cg-lookup u' x y = Some u''› False
          unfolding tryIO Let-def by auto

        have ¬ converge M2 u v
        proof −
          note ‹L M1 ∩ set T = L M2 ∩ set T›

          then have u' ∈ L M2 and v' ∈ L M2
            using False ‹u' ∈ L M1› ‹v' ∈ L M1› ‹¬ (¬ isin T u' ∨ ¬ isin T
v')›
```

by *auto*

have $u'' \in L\ M2$
using ‹$L\ M1 \cap set\ T = L\ M2 \cap set\ T$› ‹$u'' \in set\ T \cap L\ M1$›
by *blast*
then have *converge M2 u′ u″*
using *get-extension-result*(*2*)[*OF* ‹$u' \in L\ M1$› ‹$u' \in L\ M2$› *assms*(*10*) ‹*get-extension T G cg-lookup u′ x y = Some u″*›]
by *blast*
moreover note ‹*converge M2 u u′*›
ultimately have *converge M2 u u″*
by *auto*

have $v'' \in L\ M2$
using ‹$L\ M1 \cap set\ T = L\ M2 \cap set\ T$› ‹$v'' \in set\ T \cap L\ M1$›
by *blast*
then have *converge M2 v′ v″*
using *get-extension-result*(*2*)[*OF* ‹$v' \in L\ M1$› ‹$v' \in L\ M2$› *assms*(*10*) ‹*get-extension T G cg-lookup v′ x y = Some v″*›]
by *blast*
moreover note ‹*converge M2 v v′*›
ultimately have *converge M2 v v″*
by *auto*

have *distinguishes M1 su sv* ([(*x*,*y*)])
using *h-obs-distinguishes*[*OF assms*(*1*), *of su x y - sv*]
using *distinguishes-sym*[*OF h-obs-distinguishes*[*OF assms*(*1*), *of sv x y - su*]]
using *True*
by (*cases h-obs M1 su x y*; *cases h-obs M1 sv x y*; *metis*)
then have *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) ([(*x*,*y*)])
unfolding *su sv* **by** *auto*

show ¬ *converge M2 u v*
using *distinguish-converge-diverge*[*OF assms*(*1*−*3*) - - ‹*converge M1 u u″*› ‹*converge M1 v v″*› ‹*converge M2 u u″*› ‹*converge M2 v v″*› ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) ([(*x*,*y*)])› ‹u'' @ [(*x*, *y*)] ∈ *set T*› ‹v'' @ [(*x*, *y*)] ∈ *set T*› ‹$L\ M1 \cap set\ T = L\ M2 \cap set\ T$›]
‹$u'' \in set\ T \cap L\ M1$› ‹$v'' \in set\ T \cap L\ M1$›
by *blast*
qed
then show *?thesis*
unfolding *P* ‹*tryIO* (*e*,*w*) (*x*,*y*) = (*0*,[])› *fst-conv snd-conv su sv*
by *blast*

next
case *False*

show *?thesis* **proof** (*cases h-obs M1 su x y = h-obs M1 sv x y*)
case *True*

then have *tryIO* (*e*,*w*) (*x*,*y*) = (*e*,*w*)
using ‹*get-extension T G cg-lookup u′ x y = Some u″*› *Some*
unfolding *tryIO* **by** *auto*
then show *?thesis*
using ‹*P* (*e*,*w*)›
by *auto*
next
case *False*

then have *h-obs M1 su x y* ≠ *None* **and** *h-obs M1 sv x y* ≠ *None*
using ‹¬ (*h-obs M1 su x y = None*) ≠ (*h-obs M1 sv x y = None*)›
by *metis+*

have *u″*@[(*x*,*y*)] ∈ *L M1*
by (*metis* ‹*converge M1 u u″*› ‹*h-obs M1 su x y* ≠ *None*› *after-language-iff assms*(*1*) *converge.elims*(*2*) *h-obs-language-single-transition-iff su*)

have *v″*@[(*x*,*y*)] ∈ *L M1*
by (*metis* ‹*converge M1 v v″*› ‹*h-obs M1 sv x y* ≠ *None*› *after-language-iff assms*(*1*) *converge.elims*(*2*) *h-obs-language-single-transition-iff sv*)

have *u″*@[(*x*,*y*)] ∈ *L M2*
using ‹*u″*@[(*x*,*y*)] ∈ *L M1*› ‹*u″*@[(*x*,*y*)] ∈ *set T*› ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*›
by *blast*
have *v″*@[(*x*,*y*)] ∈ *L M2*
using ‹*v″*@[(*x*,*y*)] ∈ *L M1*› ‹*v″*@[(*x*,*y*)] ∈ *set T*› ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*›
by *blast*

have *FSM.after M1* (*FSM.initial M1*) (*u″* @ [(*x*, *y*)]) ≠ *FSM.after M1* (*FSM.initial M1*) (*v″* @ [(*x*, *y*)])
using *False* ‹*converge M1 u u″*› ‹*converge M1 v v″*› **unfolding** *su sv*
proof −
assume *a1*: *h-obs M1* (*FSM.after M1* (*FSM.initial M1*) *u*) *x y* ≠ *h-obs M1* (*FSM.after M1* (*FSM.initial M1*) *v*) *x y*
have *f2*: ∀*f ps psa*. *converge* (*f*::(*′a*, *′b*, *′c*) *fsm*) *ps psa* = (*ps* ∈ *L f* ∧ *psa* ∈ *L f* ∧ *LS f* (*FSM.after f* (*FSM.initial f*) *ps*) = *LS f* (*FSM.after f* (*FSM.initial f*) *psa*))
by (*meson converge.simps*)
then have *f3*: *u* ∈ *L M1* ∧ *u″* ∈ *L M1* ∧ *LS M1* (*FSM.after M1* (*FSM.initial M1*) *u*) = *LS M1* (*FSM.after M1* (*FSM.initial M1*) *u″*)
using ‹*converge M1 u u″*› **by** *presburger*
have *f4*: ∀*f ps psa*. ¬ *minimal* (*f*::(*′a*, *′b*, *′c*) *fsm*) ∨ ¬ *observable*

f ∨ *ps* ∉ *L f* ∨ *psa* ∉ *L f* ∨ *converge f ps psa* = (*FSM.after f* (*FSM.initial f*) *ps* = *FSM.after f* (*FSM.initial f*) *psa*)
using *convergence-minimal* **by** *blast*
have *f5*: *v* ∈ *L M1* ∧ *v''* ∈ *L M1* ∧ *LS M1* (*FSM.after M1* (*FSM.initial M1*) *v*) = *LS M1* (*FSM.after M1* (*FSM.initial M1*) *v''*)
using *f2* ‹*converge M1 v v''*› **by** *blast*
then have *f6*: *FSM.after M1* (*FSM.initial M1*) *v* = *FSM.after M1* (*FSM.initial M1*) *v''*
using *f4* ‹*converge M1 v v''*› *assms*(*1*) *assms*(*3*) **by** *blast*
have *FSM.after M1* (*FSM.initial M1*) *u* = *FSM.after M1* (*FSM.initial M1*) *u''*
using *f4 f3* ‹*converge M1 u u''*› *assms*(*1*) *assms*(*3*) **by** *blast*
then show *?thesis*
using *f6 f5 f3 a1* **by** (*metis* (*no-types*) ‹*u''* @ [(*x*, *y*)] ∈ *L M1*› ‹*v''* @ [(*x*, *y*)] ∈ *L M1*› *after-h-obs after-language-iff after-split assms*(*1*) *h-obs-from-LS*)
qed

obtain *e' w'* **where** *get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (*u''*@[(*x*,*y*)]) (*v''*@[(*x*,*y*)]) *k* = (*e'*,*w'*)
using *prod.exhaust* **by** *metis*

then have *tryIO* (*e*,*w*) (*x*,*y*) = (*if e'* = *0 then* (*0*,[]) *else if e'* ≤ *e then* (*e'*,(*x*,*y*)#*w'*) *else* (*e*,*w*))
using ‹*get-extension T G cg-lookup u' x y* = *Some u''*› *Some False* ‹¬ (*h-obs M1 su x y* = *None*) ≠ (*h-obs M1 sv x y* = *None*)› ‹*e* ≠ *0*›
unfolding *tryIO Let-def* **by** *auto*

show *?thesis* **proof** (*cases e'* = *0*)
case *True*

have ¬ *converge M2 u v*
proof −
note ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*›
then have *u'* ∈ *L M2* **and** *v'* ∈ *L M2*
using ‹¬ (¬ *isin T u'* ∨ ¬ *isin T v'*)› ‹*u'* ∈ *L M1*› ‹*v'* ∈ *L M1*›
by *auto*

have *u''* ∈ *L M2*
using ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*› ‹*u''* ∈ *set T* ∩ *L M1*›
by *blast*
then have *converge M2 u' u''*
using *get-extension-result*(*2*)[*OF* ‹*u'* ∈ *L M1*› ‹*u'* ∈ *L M2*› *assms*(*10*) ‹*get-extension T G cg-lookup u' x y* = *Some u''*›]
by *blast*
moreover note ‹*converge M2 u u'*›
ultimately have *converge M2 u u''*
by *auto*

have $v'' \in L\ M2$
using ‹$L\ M1 \cap set\ T = L\ M2 \cap set\ T$› ‹$v'' \in set\ T \cap L\ M1$›
by *blast*
then have *converge M2 v′ v″*
using *get-extension-result*(*2*)[*OF* ‹$v' \in L\ M1$› ‹$v' \in L\ M2$› *assms*(*10*) ‹*get-extension T G cg-lookup v′ x y = Some v″*›]
by *blast*
moreover note ‹*converge M2 v v′*›
ultimately have *converge M2 v v″*
by *auto*

have *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u'' @ [(x, y)]) (v'' @ [(x, y)]) k) = *0*
using *True* ‹*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u''@[(x,y)]) (v''@[(x,y)]) k = (e',w')›
by *auto*
then have ¬ *converge M2* (u'' @ [(x, y)]) (v'' @ [(x, y)])
using *Suc.IH*[*OF* ‹u''@[(x,y)] ∈ *L M1*› ‹u''@[(x,y)] ∈ *L M2*› ‹v''@[(x,y)] ∈ *L M1*› ‹v''@[(x,y)] ∈ *L M2*› ‹*FSM.after M1* (*FSM.initial M1*) (u'' @ [(x, y)]) ≠ *FSM.after M1* (*FSM.initial M1*) (v'' @ [(x, y)])›]
using ‹$L\ M1 \cap$ *Prefix-Tree.set T* = $L\ M2 \cap$ *Prefix-Tree.set T*›
by *blast*
then have ¬ *converge M2 u″ v″*
using *diverge-prefix*[*OF assms*(*2*) ‹u''@[(x,y)] ∈ *L M2*› ‹v''@[(x,y)] ∈ *L M2*›]
by *blast*
then show ¬ *converge M2 u v*
using ‹*converge M2 u u″*› ‹*converge M2 v v″*›
by *fastforce*
qed
then show *?thesis*
unfolding *P* ‹*tryIO* (e,w) (x,y) = (*if* e' = *0 then* (*0*,[]) *else if* e' ≤ e *then* (e',(x,y)#w') *else* (e,w))› *True fst-conv snd-conv su sv*
by *simp*
next
case *False*

show *?thesis* **proof** (*cases* $e' \leq e$)
case *True*
then have *fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u'' @ [(x, y)]) (v'' @ [(x, y)]) k) ≠ *0*
using ‹*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u''@[(x,y)]) (v''@[(x,y)]) k = (e',w')› *False*
by *auto*
then have (∃γ. *distinguishes M1* (*FSM.after M1* (*FSM.initial M1*) (u'' @ [(x, y)])) (*FSM.after M1* (*FSM.initial M1*) (v'' @ [(x, y)]))
(*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u'' @ [(x, y)]) (v'' @ [(x, y)]) k) @ γ))
using *Suc.IH*[*OF* ‹u''@[(x,y)] ∈ *L M1*› ‹u''@[(x,y)] ∈ *L M2*›

‹v''@[(x,y)] ∈ L M1› ‹v''@[(x,y)] ∈ L M2› ‹FSM.after M1 (FSM.initial M1) (u'' @ [(x, y)]) ≠ FSM.after M1 (FSM.initial M1) (v'' @ [(x, y)])›]
by *blast*
then obtain γ **where** *distinguishes M1* (*FSM.after M1* (*FSM.initial M1*) (u'' @ [(x, y)])) (*FSM.after M1* (*FSM.initial M1*) (v'' @ [(x, y)])) (w'@γ)
unfolding ‹*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace* (u''@[(x,y)]) (v''@[(x,y)]) k = (e',w')› *snd-conv*
by *blast*
have *distinguishes M1* (*after-initial M1 u''*) (*after-initial M1 v''*) ((x,y)#(w'@γ))
using *distinguishes-after-initial-prepend*[*OF assms*(*1*) *language-prefix*[*OF* ‹u''@[(x,y)] ∈ L M1›] *language-prefix*[*OF* ‹v''@[(x,y)] ∈ L M1›]]
by (*metis Suc.prems*(*1*) *Suc.prems*(*3*) ‹*converge M1 u u'*› ‹*converge M1 u' u''*› ‹*converge M1 v v''*› ‹*distinguishes M1* (*after-initial M1* (u'' @ [(x, y)])) (*after-initial M1* (v'' @ [(x, y)])) (w' @ γ)› ‹*h-obs M1 su x y* ≠ *None*› ‹*h-obs M1 sv x y* ≠ *None*› ‹u' ∈ L M1› ‹u'' @ [(x, y)] ∈ L M1› ‹v'' @ [(x, y)] ∈ L M1› *assms*(*1*) *assms*(*3*) *convergence-minimal language-prefix su sv*)
then have *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) (((x,y)#w')@γ)
by (*metis Cons-eq-appendI Suc.prems*(*1*) *Suc.prems*(*3*) ‹*converge M1 u u''*› ‹*converge M1 v v''*› ‹u'' @ [(x, y)] ∈ L M1› ‹v'' @ [(x, y)] ∈ L M1› *assms*(*1*) *assms*(*3*) *convergence-minimal language-prefix*)

have *tryIO* (e,w) (x,y) = (e',(x,y)#w')
using ‹*tryIO* (e,w) (x,y) = (*if* e' = *0 then* (*0*,[]) *else if* e' ≤ e *then* (e',(x,y)#w') *else* (e,w))› *True False*
by *auto*

show *?thesis*
unfolding *P* ‹*tryIO* (e,w) (x,y) = (e',(x,y)#w')› *fst-conv snd-conv*
using ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) (((x,y)#w')@γ)›
False True
by *blast*
next
case *False*

then have *tryIO* (e,w) (x,y) = (e,w)
using ‹e' ≠ *0*› ‹*tryIO* (e,w) (x,y) = (*if* e' = *0 then* (*0*,[]) *else if* e' ≤ e *then* (e',(x,y)#w') *else* (e,w))›
by *auto*
then show *?thesis*
using ‹*P* (e,w)›
by *auto*
qed
qed
qed
qed

```
        qed
      qed
    qed
  qed

  have minEst0 < errorValue
    unfolding errorValue by auto

  have P (foldl tryIO (minEst0,[]) XY) ∧ fst (foldl tryIO (minEst0,[]) XY) ≤
minEst0
  proof (induction XY rule: rev-induct)
    case Nil
    then show ?case
      using ‹P (minEst0,[])›
      by auto
  next
    case (snoc a XY)

    obtain x y where a = (x,y)
      using prod.exhaust by metis
    moreover obtain e w where (foldl tryIO (minEst0,[]) XY) = (e,w)
      using prod.exhaust by metis
    ultimately have (foldl tryIO (minEst0, []) (XY@[a])) = tryIO (e,w) (x,y)
      by auto

    have P (e,w) and e ≤ minEst0 and e < errorValue
      using snoc.IH ‹minEst0 < errorValue›
      unfolding ‹(foldl tryIO (minEst0,[]) XY) = (e,w)›
      by auto

    then show ?case
      unfolding ‹(foldl tryIO (minEst0, []) (XY@[a])) = tryIO (e,w) (x,y)›
      using ‹⋀ x y e w . e < errorValue ⟹ P (e,w) ⟹ P (tryIO (e,w) (x,y))
∧ fst (tryIO (e,w) (x,y)) ≤ e›
      using dual-order.trans by blast
  qed

 then have P (get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace
u v (Suc k))
    unfolding res by blast
  then show ?thesis
    unfolding P by blast
 qed
qed

then show fst (get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace
u v k) = 0 ⟹ ¬ converge M2 u v
  and fst (get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace
u v k) ≠ 0 ⟹ ∃ γ . distinguishes M1 (after-initial M1 u) (after-initial M1 v)
```

((*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*))@γ)
by *blast+*
qed

21.2.2 Distributing Convergent Traces

fun *append-heuristic-io* :: ('b×'c) *prefix-tree* ⇒ ('b×'c) *list* ⇒ (('b×'c) *list* × *int*) ⇒ ('b×'c) *list* ⇒ (('b×'c) *list* × *int*) **where**
append-heuristic-io T w (*uBest*,*lBest*) *u'* = (*let t'* = *after T u'*;
w' = *maximum-prefix t' w*
in if w' = *w*
then (*u'*,*0*::*int*)
else if (*is-maximal-in t' w'* ∧ (*int* (*length w'*) > *lBest* ∨ (*int* (*length w'*) = *lBest* ∧ *length u'* < *length uBest*)))
then (*u'*, *int* (*length w'*))
else (*uBest*,*lBest*))

lemma *append-heuristic-io-in* :
fst (*append-heuristic-io T w* (*uBest*,*lBest*) *u'*) ∈ {*u'*,*uBest*}
unfolding *append-heuristic-io.simps Let-def* **by** *auto*

fun *append-heuristic-input* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* ⇒ ('b×'c) *prefix-tree* ⇒ ('b×'c) *list* ⇒ (('b×'c) *list* × *int*) ⇒ ('b×'c) *list* ⇒ (('b×'c) *list* × *int*) **where**
append-heuristic-input M T w (*uBest*,*lBest*) *u'* = (*let t'* = *after T u'*;
ws = *maximum-fst-prefixes t'* (*map fst w*) (*outputs-as-list M*)
in
foldr (λ *w'* (*uBest'*,*lBest'*::*int*) .
if w' = *w*
then (*u'*,*0*::*int*)
else if (*int* (*length w'*) > *lBest'* ∨ (*int* (*length w'*) = *lBest'* ∧ *length u'* < *length uBest'*))
then (*u'*,*int* (*length w'*))
else (*uBest'*,*lBest'*))
ws (*uBest*,*lBest*))

lemma *append-heuristic-input-in* :
fst (*append-heuristic-input M T w* (*uBest*,*lBest*) *u'*) ∈ {*u'*,*uBest*}
proof −
define *ws* **where** *ws*: *ws* = *maximum-fst-prefixes* (*after T u'*) (*map fst w*) (*outputs-as-list M*)
define *f* **where** *f*: *f* = (λ *w'* (*uBest'*,*lBest'*::*int*) .
if w' = *w*
then (*u'*,*0*::*int*)
else if (*int* (*length w'*) > *lBest'* ∨ (*int*

(*length w'*) = *lBest'* ∧ *length u'* < *length uBest'*))
then (*u'*,*int* (*length w'*))
else (*uBest'*,*lBest'*))

have ⋀ *w' b'* . *fst b'* ∈ {*u'*,*uBest*} ⟹ *fst* (*f w' b'*) ∈ {*u'*,*uBest*}
unfolding *f* **by** *auto*
then have *fst* (*foldr f ws* (*uBest*,*lBest*)) ∈ {*u'*,*uBest*}
by (*induction ws*; *auto*)
moreover have *append-heuristic-input M T w* (*uBest*,*lBest*) *u'* = *foldr f ws* (*uBest*,*lBest*)
unfolding *append-heuristic-input.simps Let-def ws f* **by** *force*
ultimately show *?thesis*
by *simp*
qed

fun *distribute-extension* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*'b*×*'c*) *prefix-tree* ⇒ *'d* ⇒ (*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list list*) ⇒ (*'d* ⇒ (*'b*×*'c*) *list* ⇒ *'d*) ⇒(*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list* ⇒ *bool* ⇒ ((*'b*×*'c*) *prefix-tree* ⇒ (*'b*×*'c*) *list* ⇒ ((*'b*×*'c*) *list* × *int*) ⇒ (*'b*×*'c*) *list* ⇒ ((*'b*×*'c*) *list* × *int*)) ⇒ ((*'b*×*'c*) *prefix-tree* ×*'d*) **where**
distribute-extension M T G cg-lookup cg-insert u w completeInputTraces append-heuristic = (*let*
cu = *cg-lookup G u*;
u0 = *shortest-list-in-tree-or-default cu T u*;
l0 = −*1*::*int*;
u' = *fst* ((*foldl* (*append-heuristic T w*) (*u0*,*l0*) (*filter* (*isin T*) *cu*)) :: ((*'b*×*'c*) *list* × *int*));
T' = *insert T* (*u'*@*w*);
G' = *cg-insert G* (*maximal-prefix-in-language M* (*initial M*) (*u'*@*w*))
in if completeInputTraces
then let TC = *complete-inputs-to-tree M* (*initial M*) (*outputs-as-list M*) (*map fst* (*u'*@*w*));
T'' = *Prefix-Tree.combine T' TC*
in (*T''*,*G'*)
else (*T'*,*G'*))

lemma *distribute-extension-subset* :
set T ⊆ *set* (*fst* (*distribute-extension M T G cg-lookup cg-insert u w b heuristic*))
proof −

define *u0* **where** *u0*: *u0* = *shortest-list-in-tree-or-default* (*cg-lookup G u*) *T u*
define *l0* **where** *l0*: *l0* = (−*1*::*int*)
define *u'* **where** *u'*: *u'* = *fst* (*foldl* (*heuristic T w*) (*u0*,*l0*) (*filter* (*isin T*) (*cg-lookup G u*)))
define *T'* **where** *T'*: *T'* = *insert T* (*u'*@*w*)

define G' **where** G': $G' = cg\text{-}insert\ G\ (maximal\text{-}prefix\text{-}in\text{-}language\ M\ (initial\ M)\ (u'@w))$

have $set\ T \subseteq set\ T'$
unfolding T' *insert-set*
by *blast*

show *?thesis* **proof** (*cases b*)
case *True*
then show *?thesis*
using ‹$set\ T \subseteq set\ T'$›
unfolding *distribute-extension.simps u0 l0 u′ T′ G′ Let-def*
using *combine-set*
by *force*
next
case *False*
then have $fst\ (distribute\text{-}extension\ M\ T\ G\ cg\text{-}lookup\ cg\text{-}insert\ u\ w\ b\ heuristic) = T'$
unfolding *distribute-extension.simps u0 l0 u′ T′ G′ Let-def* **by** *force*
then show *?thesis*
using ‹$set\ T \subseteq set\ T'$›
by *blast*
qed
qed

lemma *distribute-extension-finite* :
assumes *finite-tree T*
shows $finite\text{-}tree\ (fst\ (distribute\text{-}extension\ M\ T\ G\ cg\text{-}lookup\ cg\text{-}insert\ u\ w\ b\ heuristic))$
proof −

define *u0* **where** *u0*: $u0 = shortest\text{-}list\text{-}in\text{-}tree\text{-}or\text{-}default\ (cg\text{-}lookup\ G\ u)\ T\ u$
define *l0* **where** *l0*: $l0 = (-1::int)$
define u' **where** u': $u' = fst\ (foldl\ (heuristic\ T\ w)\ (u0,l0)\ (filter\ (isin\ T)\ (cg\text{-}lookup\ G\ u)))$
define T' **where** T': $T' = insert\ T\ (u'@w)$
define G' **where** G': $G' = cg\text{-}insert\ G\ (maximal\text{-}prefix\text{-}in\text{-}language\ M\ (initial\ M)\ (u'@w))$

have $finite\text{-}tree\ T'$
unfolding T'
using *insert-finite-tree*[*OF assms*]
by *blast*

show *?thesis* **proof** (*cases b*)
case *True*
then show *?thesis*
using ‹$finite\text{-}tree\ T'$›

unfolding *distribute-extension.simps u0 l0 u′ T′ G′ Let-def*
by (*simp add*: *combine-finite-tree complete-inputs-to-tree-finite-tree*)
next
case *False*
then have *fst* (*distribute-extension M T G cg-lookup cg-insert u w b heuristic*) = *T′*
unfolding *distribute-extension.simps u0 l0 u′ T′ G′ Let-def* **by** *force*
then show *?thesis*
using ‹*finite-tree T′*›
by *blast*
qed
qed

lemma *distribute-extension-adds-sequence* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
assumes *observable M1*
and *minimal M1*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and (*L M1* ∩ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)))
and ⋀ *u′ uBest lBest* . *fst* (*heuristic T w* (*uBest*,*lBest*) *u′*) ∈ {*u′*,*uBest*}
shows ∃ *u′* . *converge M1 u u′* ∧ *u′*@*w* ∈ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)) ∧ *converge M2 u u′*
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*))
proof −

define *u0* **where** *u0*: *u0* = *shortest-list-in-tree-or-default* (*cg-lookup G u*) *T u*
define *l0* **where** *l0*: *l0* = (−*1*::*int*)
define *u′* **where** *u′*: *u′* = *fst* (*foldl* (*heuristic T w*) (*u0*,*l0*) (*filter* (*isin T*) (*cg-lookup G u*)))
define *T′* **where** *T′*: *T′* = *insert T* (*u′*@*w*)
define *G′* **where** *G′*: *G′* = *cg-insert G* (*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*))

define *TC* **where** *TC*: *TC* = *complete-inputs-to-tree M1* (*initial M1*) (*outputs-as-list M1*) (*map fst* (*u′*@*w*))
define *T″* **where** *T″*: *T″* = *Prefix-Tree.combine T′ TC*

have *distribute-extension M1 T G cg-lookup cg-insert u w b heuristic* = (*T′*,*G′*) ∨
distribute-extension M1 T G cg-lookup cg-insert u w b heuristic = (*T″*,*G′*)
unfolding *distribute-extension.simps u0 l0 u′ T′ G′ TC T″ Let-def* **by** *force*
moreover have *set T′* ⊆ *set T″*
unfolding *T″ combine-set* **by** *blast*

ultimately have *set T′ ⊆ set (fst (distribute-extension M1 T G cg-lookup cg-insert u w b heuristic))*
by *force*

have ⋀ *xs . fst (foldl (heuristic T w) (u0,l0) xs) ∈ Set.insert u0 (list.set xs)*
proof −
fix *xs*

show *fst (foldl (heuristic T w) (u0,l0) xs) ∈ Set.insert u0 (list.set xs)*
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by *auto*
next
case (*snoc x xs*)
have ⋀ *u′ uBest lBest . (fst ((heuristic T w) (uBest,lBest) u′)) = u′ ∨ (fst ((heuristic T w) (uBest,lBest) u′)) = uBest*
using *assms*(*8*) **by** *blast*
then have (*fst ((heuristic T w) (foldl (heuristic T w) (u0, l0) xs) x)) = x ∨ (fst ((heuristic T w) (foldl (heuristic T w) (u0, l0) xs) x)) = fst (foldl (heuristic T w) (u0, l0) xs)*
by (*metis prod.exhaust-sel*)
then show *?case*
using *snoc.IH* **by** *auto*
qed
qed
then have *u′ ∈ Set.insert u0 (list.set (filter (isin T) (cg-lookup G u)))*
unfolding *u′*
by *blast*
then have *u′ ∈ Set.insert u0 (list.set (cg-lookup G u))*
by *auto*
moreover have *converge M1 u u0*
unfolding *u′*
using *shortest-list-in-tree-or-default-elem*[*of cg-lookup G u T u*]
by (*metis assms*(*1*−*5*) *convergence-graph-lookup-invar-def convergence-minimal insert-iff u0*)
moreover have ⋀ *u′ . u′ ∈ list.set (cg-lookup G u) ⟹ converge M1 u u′*
using *assms*(*3*,*4*,*5*)
by (*metis convergence-graph-lookup-invar-def*)
ultimately have *converge M1 u u′*
by *blast*
moreover have *u′@w ∈ set (fst (distribute-extension M1 T G cg-lookup cg-insert u w b heuristic))*
using ‹*set T′ ⊆ set (fst (distribute-extension M1 T G cg-lookup cg-insert u w b heuristic))*›
unfolding *T′ insert-set fst-conv*
by *blast*
moreover have *converge M2 u u′*
by (*metis* ‹*u′ ∈ Set.insert u0 (list.set (cg-lookup G u))*› *assms*(*3*) *assms*(*4*)

assms(*5*) *converge.elims*(*3*) *convergence-graph-lookup-invar-def insertE shortest-list-in-tree-or-default-elem u0*)
ultimately show ∃ *u′* . *converge M1 u u′* ∧ *u′*@*w* ∈ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*)) ∧ *converge M2 u u′*
by *blast*

have (*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *L M1*
and (*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *list.set* (*prefixes* (*u′*@*w*))
using *maximal-prefix-in-language-properties*[*OF assms*(*1*) *fsm-initial*]
by *auto*

moreover have (*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*))
using ‹*u′*@*w* ∈ *set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*))› *set-prefix*
by (*metis* (*no-types*, *lifting*) ‹*maximal-prefix-in-language M1* (*FSM.initial M1*) (*u′* @ *w*) ∈ *list.set* (*prefixes* (*u′* @ *w*))› *prefixes-set-ob*)
ultimately have (*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *L M2*
using *assms*(*7*)
by *blast*

have *convergence-graph-lookup-invar M1 M2 cg-lookup G′*
using *assms*(*5*,*6*) ‹(*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *L M1*› ‹(*maximal-prefix-in-language M1* (*initial M1*) (*u′*@*w*)) ∈ *L M2*›
unfolding *G′ convergence-graph-insert-invar-def*
by *blast*

show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distribute-extension M1 T G cg-lookup cg-insert u w b heuristic*))
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*›
unfolding *distribute-extension.simps u0 l0 u′ T′ G′ Let-def* **by** *force*
qed

21.2.3 Distinguishing a Trace from Other Traces

fun *spyh-distinguish* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ (*′b*×*′c*) *prefix-tree* ⇒ *′d* ⇒ (*′d* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list*) ⇒ (*′d* ⇒ (*′b*×*′c*) *list* ⇒ *′d*) ⇒ (*′a* ⇒ *′a* ⇒ (*′b*×*′c*) *list*) ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list* ⇒ *nat* ⇒ *bool* ⇒ ((*′b*×*′c*) *prefix-tree* ⇒ (*′b*×*′c*) *list* ⇒ ((*′b*×*′c*) *list* × *int*) ⇒ (*′b*×*′c*) *list* ⇒ ((*′b*×*′c*) *list* × *int*)) ⇒ ((*′b*×*′c*) *prefix-tree* × *′d*) **where**
spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic = (*let*
dist-helper = (λ (*T*,*G*) *v* . *if after-initial M u* = *after-initial M v*
then (*T*,*G*)
else (*let ew* = *get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v k*

in if fst ew = 0
then (T,G)
else (*let* $u' = (u@(snd\ ew))$;
$v' = (v@(snd\ ew))$;
$w' =$ *if does-distinguish M* (*after-initial M u*) (*after-initial M v*) (*snd ew*) *then* (*snd ew*) *else* (*snd ew*)@(*get-distinguishing-trace* (*after-initial M* u') (*after-initial M* v'));
$TG' =$ *distribute-extension M T G cg-lookup cg-insert u* w' *completeInputTraces append-heuristic*
in distribute-extension M (*fst* TG') (*snd* TG') *cg-lookup cg-insert v* w' *completeInputTraces append-heuristic*)))
in foldl dist-helper (T,G) *X*)

lemma *spyh-distinguish-subset* :
set T $\subseteq$ *set* (*fst* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
proof (*induction X rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a X*)

have *set* (*fst* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
$\subseteq$ *set* (*fst* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u* $(X@[a])$ *k completeInputTraces append-heuristic*))
proof −
define *dh* **where** *dh*: *dh* = (λ (T,G) *v* . *if after-initial M u = after-initial M v*
then (T,G)
else (*let ew = get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace u v k*
in if fst ew = 0
then (T,G)
else (*let* $u' = (u@(snd\ ew))$;
$v' = (v@(snd\ ew))$;
$w' =$ *if does-distinguish M* (*after-initial M u*) (*after-initial M v*) (*snd ew*) *then* (*snd ew*) *else* (*snd ew*)@(*get-distinguishing-trace* (*after-initial M* u') (*after-initial M* v'));
$TG' =$ *distribute-extension M T G cg-lookup cg-insert u* w' *completeInputTraces append-heuristic*
in distribute-extension M (*fst* TG') (*snd* TG') *cg-lookup cg-insert v* w' *completeInputTraces append-heuristic*)))

have *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u* $(X@[a])$ *k completeInputTraces append-heuristic*
= *dh* (*spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) *a*

```
    unfolding dh spyh-distinguish.simps Let-def
    unfolding foldl-append
    by auto

  moreover have ⋀ T G . set T ⊆ set (fst (dh (T,G) a))
  proof −
    fix T G
    show set T ⊆ set (fst (dh (T,G) a))
    proof (cases after-initial M u = after-initial M a)
      case True
      then show ?thesis using dh by auto
    next
      case False
      then show ?thesis proof (cases fst (get-prefix-of-separating-sequence M T
G cg-lookup get-distinguishing-trace u a k) = 0)
        case True
        then show ?thesis using False dh by auto
      next
        case False

        define u′ where u′: u′ = (u@(snd (get-prefix-of-separating-sequence M T
G cg-lookup get-distinguishing-trace u a k)))
        define v′ where v′: v′ = (a@(snd (get-prefix-of-separating-sequence M T
G cg-lookup get-distinguishing-trace u a k)))
          define w where w: w = get-distinguishing-trace (after-initial M u′)
(after-initial M v′)
          define w′ where w′: w′ = (if does-distinguish M (after-initial M u)
(after-initial M a) (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k)) then (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k)) else (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k))@w)
          define TG′ where TG′: TG′ = distribute-extension M T G cg-lookup
cg-insert u w′ completeInputTraces append-heuristic

      have dh (T,G) a = distribute-extension M (fst (distribute-extension M T G
cg-lookup cg-insert u w′ completeInputTraces append-heuristic)) (snd (distribute-extension
M T G cg-lookup cg-insert u w′ completeInputTraces append-heuristic)) cg-lookup
cg-insert a w′ completeInputTraces append-heuristic
        using False ‹FSM.after M (FSM.initial M) u ≠ FSM.after M (FSM.initial
M) a›
          unfolding dh u′ v′ w w′ TG′ Let-def case-prod-conv by metis

        then show ?thesis
          using distribute-extension-subset
          by (metis (no-types, lifting) subset-trans)
      qed
    qed
  qed
```

```
    ultimately show ?thesis
      by (metis eq-fst-iff)
  qed

  then show ?case
    using snoc.IH by blast
qed

lemma spyh-distinguish-finite :
  fixes T :: ('b::linorder×'c::linorder) prefix-tree
  assumes finite-tree T
 shows finite-tree (fst (spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace
u X k completeInputTraces append-heuristic))
proof (induction X rule: rev-induct)
  case Nil
  then show ?case using assms by auto
next
  case (snoc a X)


  define dh where dh: dh = (λ (T,G) v . if after-initial M u = after-initial M v
                                then (T,G)
                                 else (let ew = get-prefix-of-separating-sequence M T G
cg-lookup get-distinguishing-trace u v k
                                      in if fst ew = 0
                                          then (T,G)
                                          else (let u' = (u@(snd ew));
                                                  v' = (v@(snd ew));
                                           w' = if does-distinguish M (after-initial M u)
(after-initial M v) (snd ew) then (snd ew) else (snd ew)@(get-distinguishing-trace
(after-initial M u') (after-initial M v'));
                                                        TG' = distribute-extension M T G
cg-lookup cg-insert u w' completeInputTraces append-heuristic
                                              in distribute-extension M (fst TG') (snd
TG') cg-lookup cg-insert v w' completeInputTraces append-heuristic)))

   have ∗: spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u
(X@[a]) k completeInputTraces append-heuristic
            = dh (spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace
u X k completeInputTraces append-heuristic) a
      unfolding dh spyh-distinguish.simps Let-def
      unfolding foldl-append
      by auto

  have ∗∗:⋀ T G . finite-tree T ⟹ finite-tree (fst (dh (T,G) a))
  proof −
    fix T :: ('b×'c) prefix-tree
    fix G
    assume finite-tree T
```

```
    show finite-tree (fst (dh (T,G) a))
    proof (cases after-initial M u = after-initial M a)
      case True
      then show ?thesis using dh ‹finite-tree T› by auto
    next
      case False
     then show ?thesis proof (cases fst (get-prefix-of-separating-sequence M T G
cg-lookup get-distinguishing-trace u a k) = 0)
       case True
       then show ?thesis using False dh ‹finite-tree T› by auto
     next
       case False

       define u' where u': u' = (u@(snd (get-prefix-of-separating-sequence M T
G cg-lookup get-distinguishing-trace u a k)))
      define v' where v': v' = (a@(snd (get-prefix-of-separating-sequence M T G
cg-lookup get-distinguishing-trace u a k)))
          define w where w: w = get-distinguishing-trace (after-initial M u')
(after-initial M v')
         define w' where w': w' = (if does-distinguish M (after-initial M u)
(after-initial M a) (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k)) then (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k)) else (snd (get-prefix-of-separating-sequence M T G cg-lookup get-distinguishing-trace
u a k))@w)
         define TG' where TG': TG' = distribute-extension M T G cg-lookup
cg-insert u w'

      have *:dh (T,G) a = distribute-extension M (fst (distribute-extension M T G
cg-lookup cg-insert u w' completeInputTraces append-heuristic)) (snd (distribute-extension
M T G cg-lookup cg-insert u w' completeInputTraces append-heuristic)) cg-lookup
cg-insert a w' completeInputTraces append-heuristic
       using False ‹FSM.after M (FSM.initial M) u ≠ FSM.after M (FSM.initial
M) a›
         unfolding dh u' v' w w' TG' Let-def case-prod-conv by metis

       show ?thesis
         unfolding *
           using distribute-extension-finite[OF distribute-extension-finite[OF ‹fi-
nite-tree T›]]
         by metis
     qed
    qed
  qed

  show ?case
    unfolding *
    using **[OF snoc]
    by (metis eq-fst-iff)
qed
```

lemma *spyh-distinguish-establishes-divergence* :
fixes *M1* :: (*'a::linorder*,*'b::linorder*,*'c::linorder*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and $u \in L\ M1$ **and** $u \in L\ M2$
and $\bigwedge\ \alpha\ \beta\ q1\ q2\ .\ q1 \in states\ M1 \Longrightarrow q2 \in states\ M1 \Longrightarrow q1 \neq q2 \Longrightarrow$ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $list.set\ X \subseteq L\ M1$
and $list.set\ X \subseteq L\ M2$
and $L\ M1 \cap set$ (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) $= L\ M2 \cap set$ (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
and $\bigwedge\ T\ w\ u'\ uBest\ lBest\ .\ fst$ (*append-heuristic T w* (*uBest*,*lBest*) $u') \in$ $\{u',uBest\}$
shows $\forall\ v\ .\ v \in list.set\ X \longrightarrow \neg\ converge\ M1\ u\ v \longrightarrow \neg\ converge\ M2\ u\ v$
(**is** *?P1 X*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
(**is** *?P2 X*)
proof −
have *?P1 X* ∧ *?P2 X*
using *assms*(*10*,*11*,*12*)
proof (*induction X rule*: *rev-induct*)
case *Nil*

have ∗: *spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* [] *k completeInputTraces append-heuristic* = (*T*,*G*)
by *auto*

show *?case*
using *Nil assms*(*8*)
unfolding ∗ *fst-conv snd-conv* **by** *auto*
next
case (*snoc a X*)

define *dh* **where** *dh*: *dh* = (λ (*T*,*G*) *v* . *if after-initial M1 u* = *after-initial M1 v*
then (*T*,*G*)
else (*let ew* = *get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u v k*
in if fst ew = *0*

then (T,G)
else (*let* $u' = (u@(snd\ ew))$;
$v' = (v@(snd\ ew))$;
$w' =$ (*if does-distinguish M1* (*after-initial M1 u*) (*after-initial M1 v*) (*snd ew*) *then* (*snd ew*) *else* (*snd ew*)@(*get-distinguishing-trace* (*after-initial M1 u'*) (*after-initial M1 v'*)));
$TG' =$ *distribute-extension M1 T G cg-lookup cg-insert u w' completeInputTraces append-heuristic*
in distribute-extension M1 (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert v w' completeInputTraces append-heuristic*)))

have *spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X@[a]*) *k completeInputTraces append-heuristic*
= *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) *a*
unfolding *dh spyh-distinguish.simps Let-def*
unfolding *foldl-append*
by *auto*

have $\bigwedge$ *T G* . *set T* $\subseteq$ *set* (*fst* (*dh* (T,G) *a*))
proof −
fix *T G*
show *set T* $\subseteq$ *set* (*fst* (*dh* (T,G) *a*))
proof (*cases after-initial M1 u* = *after-initial M1 a*)
case *True*
then show *?thesis* **using** *dh* **by** *auto*
next
case *False*
then show *?thesis* **proof** (*cases fst* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u a k*) = *0*)
case *True*
then show *?thesis* **using** *False dh* **by** *auto*
next
case *False*

define *u'* **where** *u'*: $u' = (u@(snd\ (get\text{-}prefix\text{-}of\text{-}separating\text{-}sequence\ M1\ T\ G\ cg\text{-}lookup\ get\text{-}distinguishing\text{-}trace\ u\ a\ k)))$
define *v'* **where** *v'*: $v' = (a@(snd\ (get\text{-}prefix\text{-}of\text{-}separating\text{-}sequence\ M1\ T\ G\ cg\text{-}lookup\ get\text{-}distinguishing\text{-}trace\ u\ a\ k)))$
define *w* **where** *w*: $w =$ *get-distinguishing-trace* (*after-initial M1 u'*) (*after-initial M1 v'*)
define *w'* **where** *w'*: $w' =$ (*if does-distinguish M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u a k*)) *then* (*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u a k*)) *else* (*snd* (*get-prefix-of-separating-sequence M1 T G cg-lookup get-distinguishing-trace u a k*))@*w*)
define *TG'* **where** *TG'*: $TG' =$ *distribute-extension M1 T G cg-lookup*

cg-insert u w′

have *dh* (*T*,*G*) *a* = *distribute-extension M1* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)) (*snd* (*distribute-extension M1 T G cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)) *cg-lookup cg-insert a w′ completeInputTraces append-heuristic*
using *False* ‹*FSM.after M1* (*FSM.initial M1*) *u* ≠ *FSM.after M1* (*FSM.initial M1*) *a*›
unfolding *dh u′ v′ w w′ TG′ Let-def case-prod-conv* **by** *metis*

then show *?thesis*
using *distribute-extension-subset*
by (*metis* (*no-types*, *lifting*) *subset-trans*)
qed
qed
qed
then have *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) ⊆ *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic*))
unfolding ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic* = *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) *a*›
by (*metis prod.exhaust-sel*)
then have *L M1* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
using *snoc.prems*(*3*) **by** *blast*
moreover have *list.set X* ⊆ *L M1*
using *snoc.prems*(*1*) **by** *auto*
moreover have *list.set X* ⊆ *L M2*
using *snoc.prems*(*2*) **by** *auto*
ultimately have *?P1 X* **and** *?P2 X*
using *snoc.IH* **by** *blast+*

obtain *T′ G′* **where** (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) = (*T′*,*G′*)
using *prod.exhaust* **by** *metis*

then have *convergence-graph-lookup-invar M1 M2 cg-lookup G′*
using ‹*?P2 X*› **by** *auto*

have *L M1* ∩ *set T′* = *L M2* ∩ *set T′*
using ‹*L M1* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace*

u X k completeInputTraces append-heuristic))›
‹(*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) = (*T′*,*G′*)›
by *auto*

have ¬*converge M1 u a* ⟹ ¬*converge M2 u a* **and** *?P2* (*X*@[*a*])
proof −
have *a* ∈ *L M1*
using *snoc.prems*(*1*) **by** *auto*
then have ¬*converge M1 u a* ⟹ *after-initial M1 u* ≠ *after-initial M1 a*
using ‹*u* ∈ *L M1*›
using *assms*(*1*) *assms*(*3*) *convergence-minimal* **by** *blast*

have *a* ∈ *L M2*
using *snoc.prems*(*2*) **by** *auto*

define *ew* **where** *ew*: *ew* = *get-prefix-of-separating-sequence M1 T′ G′ cg-lookup get-distinguishing-trace u a k*

have (¬*converge M1 u a* ⟶ ¬*converge M2 u a*) ∧ *?P2* (*X*@[*a*])
proof (*cases fst ew* = *0*)
case *True*
then have ∗: *fst* (*get-prefix-of-separating-sequence M1 T′ G′ cg-lookup get-distinguishing-trace u a k*) = *0*
unfolding *ew* **by** *auto*

have *L M1* ∩ *Prefix-Tree.set T′* = *L M2* ∩ *Prefix-Tree.set T′* ⟹ ¬ *converge M1 u a* ⟹ ¬ *converge M2 u a*
using *get-prefix-of-separating-sequence-result*(*1*)[*OF assms*(*1*,*2*,*3*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› ‹*a* ∈ *L M1*› ‹*a* ∈ *L M2*› ‹¬*converge M1 u a* ⟹ *after-initial M1 u* ≠ *after-initial M1 a*› *assms*(*7*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*› - ∗]
by *fast*
then have (¬*converge M1 u a* ⟶ ¬*converge M2 u a*)
using ‹*L M1* ∩ *set T′* = *L M2* ∩ *set T′*›
by *blast*

have (*snd* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic*)) = (*snd* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
unfolding ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic* = *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) *a*›
unfolding ‹(*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) = (*T′*,*G′*)›
unfolding *dh case-prod-conv snd-conv*

using *True ew*
by *fastforce*
then have *?P2* (*X*@[*a*])
using ‹*?P2 X*›
by *auto*
then show *?thesis*
using ‹(¬*converge M1 u a* ⟶ ¬*converge M2 u a*)›
by *auto*
next
case *False*
then have ∗: *fst* (*get-prefix-of-separating-sequence M1 T′ G′ cg-lookup get-distinguishing-trace u a k*) ≠ *0*
unfolding *ew* **by** *auto*

define *w* **where** *w*: *w* = *get-distinguishing-trace* (*after-initial M1* (*u*@(*snd ew*))) (*after-initial M1* (*a*@(*snd ew*)))
define *w′* **where** *w′*: *w′* = (*if does-distinguish M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew*) *then* (*snd ew*) *else* (*snd ew*)@*w*)
define *TG′* **where** *TG′*: *TG′* = *distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*

show *?thesis* **proof** (*cases* ¬ *converge M1 u a*)
case *True*
then have *after-initial M1 u* ≠ *after-initial M1 a*
using ‹*u* ∈ *L M1*› ‹*a* ∈ *L M1*›
using *assms*(*1*) *assms*(*3*) *convergence-minimal* **by** *blast*

obtain *γ* **where** *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew* @ *γ*)
unfolding ‹*ew* = *get-prefix-of-separating-sequence M1 T′ G′ cg-lookup get-distinguishing-trace u a k*›
using *get-prefix-of-separating-sequence-result*(*2*)[*OF assms*(*1*,*2*,*3*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› ‹*a* ∈ *L M1*› ‹*a* ∈ *L M2*› ‹*after-initial M1 u* ≠ *after-initial M1 a*› *assms*(*7*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*› - ∗]
using ‹*L M1* ∩ *Prefix-Tree.set T′* = *L M2* ∩ *Prefix-Tree.set T′*› **by** *presburger*

have *dh* (*T′*,*G′*) *a* = *distribute-extension M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert a w′ completeInputTraces append-heuristic*
unfolding *dh w w′ TG′ case-prod-conv*
unfolding *ew*[*symmetric*] *Let-def*
using *ew False* ‹*after-initial M1 u* ≠ *after-initial M1 a*›
by *meson*

have *L M1* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*)) = *L M2* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*))
using *snoc.prems*(*3*)
using ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic* = *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)

a› ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic* = (*T′*, *G′*)›
by *auto*
moreover have *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)) ⊆ *set* (*fst* (*dh* (*T′*,*G′*) *a*))
by (*metis TG′* ‹*dh* (*T′*, *G′*) *a* = *distribute-extension M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert a w′ completeInputTraces append-heuristic*› *distribute-extension-subset*)
ultimately have (*L M1* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)))
by *blast*

obtain *u′* **where** *converge M1 u u′* **and** *converge M2 u u′*
and *u′* @ *w′* ∈ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)
using *distribute-extension-adds-sequence*[*OF assms*(*1*,*3*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*› ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*›, *of* - - *completeInputTraces append-heuristic*, *OF - assms*(*13*)]
‹(*L M1* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert u w′ completeInputTraces append-heuristic*)))›
unfolding *TG′*
by *blast*
then have *u′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))
unfolding ‹*dh* (*T′*,*G′*) *a* = *distribute-extension M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert a w′ completeInputTraces append-heuristic*›
by (*metis* (*no-types*, *opaque-lifting*) *TG′ distribute-extension-subset in-mono*)

obtain *a′* **where** *converge M1 a a′* **and** *converge M2 a a′*
and *a′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*dh* (*T′*,*G′*) *a*))
using *distribute-extension-adds-sequence*[*OF assms*(*1*,*3*) ‹*a* ∈ *L M1*› ‹*a* ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)› ‹*convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*›, *of fst TG′ w′ completeInputTraces append-heuristic*, *OF - assms*(*13*)]
‹*L M1* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*)) = *L M2* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*))›
unfolding ‹*dh* (*T′*,*G′*) *a* = *distribute-extension M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert a w′ completeInputTraces append-heuristic*›
by *blast*

have *u′* ∈ *L M1* **and** *a′* ∈ *L M1*
using ‹*converge M1 u u′*› ‹*converge M1 a a′*› **by** *auto*

have *?P2* (*X*@[*a*])
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*dh* (*T′*,*G′*) *a*))›
using *False* ‹*after-initial M1 u* ≠ *after-initial M1 a*›
using ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic* = *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*) *a*› ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*= (*T′*, *G′*)›
by *presburger*

show *?thesis* **proof** (*cases does-distinguish M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew*))
case *True*
then have *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) *w′*
using *does-distinguish-correctness*[*OF assms*(*1*) *after-is-state*[*OF assms*(*1*) ‹*u* ∈ *L M1*›] *after-is-state*[*OF assms*(*1*) ‹*a* ∈ *L M1*›]] *w′*
by *metis*

show *?thesis*
using *distinguish-converge-diverge*[*OF assms*(*1*,*2*,*3*) ‹*u′* ∈ *L M1*› ‹*a′* ∈ *L M1*› ‹*converge M1 u u′*› ‹*converge M1 a a′*› ‹*converge M2 u u′*› ‹*converge M2 a a′*› ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) *w′*› ‹*u′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))› ‹*a′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))› ‹*L M1* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*)) = *L M2* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*))›]
‹*?P2* (*X*@[*a*])›
by *blast*
next
case *False*
then have ¬ *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew*)
using *does-distinguish-correctness*[*OF assms*(*1*) *after-is-state*[*OF assms*(*1*) ‹*u* ∈ *L M1*›] *after-is-state*[*OF assms*(*1*) ‹*a* ∈ *L M1*›]]
by *blast*
then have *snd ew* ∈ *LS M1* (*after-initial M1 u*) = (*snd ew* ∈ *LS M1* (*after-initial M1 a*))
unfolding *distinguishes-def*
by *blast*
moreover have *snd ew* ∈ *LS M1* (*after-initial M1 u*) ∨ (*snd ew* ∈ *LS M1* (*after-initial M1 a*))
using ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew* @ *γ*)›
using *language-prefix*[*of snd ew γ*]
unfolding *distinguishes-def*
by *fast*
ultimately have *snd ew* ∈ *LS M1* (*after-initial M1 u*) **and** *snd ew* ∈ *LS M1* (*after-initial M1 a*)
by *auto*

have *after-initial M1* (*u* @ *snd ew*) ∈ *states M1*
using ‹*snd ew* ∈ *LS M1* (*after-initial M1 u*)› *after-is-state*[*OF assms*(*1*) ‹*u* ∈ *L M1*›]
by (*meson after-is-state after-language-iff assms*(*1*) *assms*(*5*))
moreover have *after-initial M1* (*a* @ *snd ew*) ∈ *states M1*
using ‹*snd ew* ∈ *LS M1* (*after-initial M1 a*)› *after-is-state*[*OF assms*(*1*) ‹*a* ∈ *L M1*›]
by (*meson* ‹*a* ∈ *L M1*› *after-is-state after-language-iff assms*(*1*))
moreover have *after-initial M1* (*u* @ *snd ew*) ≠ *after-initial M1* (*a* @ *snd ew*)
using ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) (*snd ew* @ *γ*)›
by (*metis* ‹*a* ∈ *L M1*› ‹*snd ew* ∈ *LS M1* (*after-initial M1 a*)› ‹*snd ew* ∈ *LS M1* (*after-initial M1 u*)› *after-distinguishes-language after-language-iff append.assoc assms*(*1*) *assms*(*5*))
ultimately have *distinguishes M1* (*after-initial M1* (*u* @ *snd ew*)) (*after-initial M1* (*a* @ *snd ew*)) *w*
unfolding *w* **using** *assms*(*7*)
by *blast*
moreover have *w′* = *snd ew* @ *w*
using *False w′* **by** *auto*
ultimately have *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) *w′*
using *distinguish-prepend-initial*[*OF assms*(*1*)]
by (*meson* ‹*a* ∈ *L M1*› ‹*snd ew* ∈ *LS M1* (*after-initial M1 a*)› ‹*snd ew* ∈ *LS M1* (*after-initial M1 u*)› *after-language-iff assms*(*1*) *assms*(*5*))

show *?thesis*
using *distinguish-converge-diverge*[*OF assms*(*1*,*2*,*3*) ‹*u′* ∈ *L M1*› ‹*a′* ∈ *L M1*› ‹*converge M1 u u′*› ‹*converge M1 a a′*› ‹*converge M2 u u′*› ‹*converge M2 a a′*› ‹*distinguishes M1* (*after-initial M1 u*) (*after-initial M1 a*) *w′*› ‹*u′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))› ‹*a′* @ *w′* ∈ *set* (*fst* (*dh* (*T′*,*G′*) *a*))› ‹*L M1* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*)) = *L M2* ∩ *set* (*fst* (*dh* (*T′*,*G′*) *a*))›]
‹*?P2* (*X*@[*a*])›
by *blast*
qed
next
case *False*
then have *after-initial M1 u* = *after-initial M1 a*
by (*meson* ‹*a* ∈ *L M1*› *assms*(*1*) *assms*(*3*) *assms*(*5*) *convergence-minimal*)
then have *dh* (*T′*,*G′*) *a* = (*T′*,*G′*)
unfolding *dh case-prod-conv*
by *auto*
then have *?P2* (*X*@[*a*])
using ‹*?P2 X*›
by (*metis* ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u* (*X*@[*a*]) *k completeInputTraces append-heuristic* = *dh* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)

a› ‹*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic* = (*T*′, *G*′)›)
then show *?thesis*
using *False*
by *blast*
qed
qed
then show ¬ *converge M1 u a* ⟹ ¬ *converge M2 u a*
and *?P2* (*X*@[*a*])
by *blast*+
qed

have *?P1* (*X*@[*a*])
proof −
have ⋀ *v* . *v* ∈ *list.set X* ⟹ ¬*converge M1 u v* ⟹ ¬*converge M2 u v*
using ‹*?P1 X*›
unfolding *preserves-divergence.simps*
using *Int-absorb2* ‹*list.set X* ⊆ *L M1*› *assms*(*5*) **by** *blast*
then show *?thesis*
using ‹¬ *converge M1 u a* ⟹ ¬ *converge M2 u a*› **by** *auto*
qed
then show *?case*
using ‹*?P2* (*X*@[*a*])› **by** *auto*
qed

then show *?P1 X* **and** *?P2 X*
by *auto*
qed

lemma *spyh-distinguish-preserves-divergence* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
and ⋀ *α β q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *list.set X* ⊆ *L M1*
and *list.set X* ⊆ *L M2*
and *L M1* ∩ *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) = *L M2* ∩ *set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
and ⋀ *T w u*′ *uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u*′) ∈ {*u*′,*uBest*}

and *preserves-divergence M1 M2* (*list.set X*)
shows *preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))
(**is** *?P1 X*)
using *spyh-distinguish-establishes-divergence*(*1*)[*OF assms*(*1*−*13*)]
using *assms*(*14*)
unfolding *preserves-divergence.simps*
by (*metis IntD2 Int-iff assms*(*10*) *converge.elims*(*2*) *converge.elims*(*3*) *inf.absorb-iff2 insert-iff*)

21.3 HandleIOPair

definition *handle-io-pair* :: *bool* ⇒ *bool* ⇒ (('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
('*a*,'*b*,'*c*) *state-cover-assignment* ⇒
('*b*×'*c*) *prefix-tree* ⇒
'*d* ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*) ⇒
('*d* ⇒ ('*b*×'*c*) *list* ⇒ ('*b*×'*c*) *list list*) ⇒
'*a* ⇒ '*b* ⇒ '*c* ⇒
(('*b*×'*c*) *prefix-tree* × '*d*)) **where**
handle-io-pair completeInputTraces useInputHeuristic M V T G cg-insert cg-lookup q x y =
distribute-extension M T G cg-lookup cg-insert (*V q*) [(*x*,*y*)] *completeInputTraces* (*if useInputHeuristic then append-heuristic-input M else append-heuristic-io*)

lemma *handle-io-pair-verifies-io-pair* : *verifies-io-pair* (*handle-io-pair b c*) *M1 M2 cg-lookup cg-insert*
proof −

have ∗:⋀ (*M*::('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*) *V T* (*G*::'*d*) *cg-insert cg-lookup q x y* . *set T* ⊆ *set* (*fst* (*handle-io-pair b c M V T G cg-insert cg-lookup q x y*))
using *distribute-extension-subset* **unfolding** *handle-io-pair-def*
by *metis*

have ∗∗∗:⋀ (*M*::('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*) *V T* (*G*::'*d*) *cg-insert cg-lookup q x y* . *finite-tree T* ⟶ *finite-tree* (*fst* (*handle-io-pair b c M V T G cg-insert cg-lookup q x y*))
using *distribute-extension-finite* **unfolding** *handle-io-pair-def*
by *metis*

have ∗∗:⋀ (*M1*::('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*) *V T* (*G*::'*d*) *cg-insert cg-lookup q x y*.
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
FSM.inputs M2 = *FSM.inputs M1* ⟹
FSM.outputs M2 = *FSM.outputs M1* ⟹

is-state-cover-assignment M1 V $\Longrightarrow$
L M1 $\cap$ *V ‘ reachable-states M1 = L M2* $\cap$ *V ‘ reachable-states M1* $\Longrightarrow$
q $\in$ *reachable-states M1* $\Longrightarrow$
x $\in$ *inputs M1* $\Longrightarrow$
y $\in$ *outputs M1* $\Longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
L M1 $\cap$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)) $\Longrightarrow$
($\exists$ α .
converge M1 α (*V q*) $\wedge$
converge M2 α (*V q*) $\wedge$
α $\in$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)) $\wedge$
α@[(*x*,*y*)] $\in$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)))
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*))
proof −
fix *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fix *G* :: *'d*
fix *V T cg-insert cg-lookup q x y*
assume *a01*: *observable M1*
assume *a02*: *observable M2*
assume *a03*: *minimal M1*
assume *a04*: *minimal M2*
assume *a05*: *FSM.inputs M2 = FSM.inputs M1*
assume *a06*: *FSM.outputs M2 = FSM.outputs M1*
assume *a07*: *is-state-cover-assignment M1 V*
assume *a09*: *L M1* $\cap$ *V ‘ reachable-states M1 = L M2* $\cap$ *V ‘ reachable-states M1*
assume *a10*: *q* $\in$ *reachable-states M1*
assume *a11*: *x* $\in$ *inputs M1*
assume *a12*: *y* $\in$ *outputs M1*
assume *a13*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a14*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
assume *a15*: *L M1* $\cap$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)) = *L M2* $\cap$ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*))

let *?heuristic* = (*if c then append-heuristic-input M1 else append-heuristic-io*)

have *d1*: *V q* $\in$ *L M1*
using *is-state-cover-assignment-language*[*OF a07 a10*] **by** *auto*
have *d2*: *V q* $\in$ *L M2*
using *is-state-cover-assignment-language*[*OF a07 a10*]
using *a09 a10* **by** *auto*

have *d3*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert* (*V q*) [(*x*,*y*)] *b ?heuristic*)) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*distribute-extension M1 T G cg-lookup cg-insert* (*V q*) [(*x*,*y*)] *b ?heuristic*))

using *a15* **unfolding** *handle-io-pair-def* **.**

have *d4*: ($\bigwedge$*T w u′ uBest lBest. fst* (*?heuristic T w* (*uBest*, *lBest*) *u′*) ∈ {*u′*, *uBest*})
using *append-heuristic-input-in*[*of M1*] *append-heuristic-io-in*
by *fastforce*

show (∃ *α* .
converge M1 α (*V q*) ∧
converge M2 α (*V q*) ∧
α ∈ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)) ∧
α@[(*x*,*y*)] ∈ *set* (*fst* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*)))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*handle-io-pair b c M1 V T G cg-insert cg-lookup q x y*))
using *distribute-extension-adds-sequence*[*OF a01 a03 d1 d2 a13 a14 d3 d4*]
unfolding *handle-io-pair-def*
by (*metis converge-sym set-prefix*)
qed

show *?thesis*
unfolding *verifies-io-pair-def*
using ∗ ∗∗∗ ∗∗ **by** *presburger*
qed

lemma *handle-io-pair-handles-io-pair* : *handles-io-pair* (*handle-io-pair b c*) *M1 M2 cg-lookup cg-insert*
using *verifies-io-pair-handled*[*OF handle-io-pair-verifies-io-pair*] **.**

21.4 HandleStateCover

21.4.1 Dynamic

fun *handle-state-cover-dynamic* :: *bool* ⇒
bool ⇒
(*′a* ⇒ *′a* ⇒ (*′b*×*′c*) *list*) ⇒
(*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒
(*′a*,*′b*,*′c*) *state-cover-assignment* ⇒
((*′a*,*′b*,*′c*) *fsm* ⇒ (*′b*×*′c*) *prefix-tree* ⇒ *′d*) ⇒
(*′d* ⇒ (*′b*×*′c*) *list* ⇒ *′d*) ⇒
(*′d* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list*) ⇒
((*′b*×*′c*) *prefix-tree* × *′d*)
where
handle-state-cover-dynamic completeInputTraces useInputHeuristic get-distinguishing-trace M V cg-initial cg-insert cg-lookup =
(*let*
k = (*2* ∗ *size M*);
heuristic = (*if useInputHeuristic then append-heuristic-input M else append-heuristic-io*);
rstates = *reachable-states-as-list M*;

T0′ = *from-list* (*map V rstates*);
T0 = (*if completeInputTraces*
then Prefix-Tree.combine T0′ (*from-list* (*concat* (*map* (λ *q* . *language-for-input M* (*initial M*) (*map fst* (*V q*))) *rstates*)))
else T0′);
G0 = *cg-initial M T0*;
separate-state = (λ (*X*,*T*,*G*) *q* . *let u* = *V q*;
TG′ = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces heuristic*;
X′ = *u*#*X*
in (*X′*,*TG′*))
in snd (*foldl separate-state* ([],*T0*,*G0*) *rstates*))

lemma *handle-state-cover-dynamic-separates-state-cover*:
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fixes *M2* :: (′*e*,′*b*,′*c*) *fsm*
fixes *cg-insert* :: (′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*)
assumes ⋀ α β *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*dist-fun q1 q2*)
shows *separates-state-cover* (*handle-state-cover-dynamic b c dist-fun*) *M1 M2 cg-initial cg-insert cg-lookup*
proof −

let *?f* = (*handle-state-cover-dynamic b c dist-fun*)

have ⋀ (*V* :: (′*a*,′*b*,′*c*) *state-cover-assignment*) .
(*V* ‘ *reachable-states M1* ⊆ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)))
∧ *finite-tree* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial ⟶
L M1 ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) ⟶
(*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*?f M1 V cg-initial cg-insert cg-lookup*)))) (**is** ⋀ *V* . *?P V*)
proof −
fix *V* :: (′*a*,′*b*,′*c*) *state-cover-assignment*

define *k* **where** *k* = *2* ∗ *size M1*
define *heuristic* **where** *heuristic* = (*if c then append-heuristic-input M1 else append-heuristic-io*)

define *separate-state* **where** *separate-state* = (λ (*X*,*T*,*G*::'*d*) *q* . *let u* = *V q*;
TG' = *spyh-distinguish M1 T G cg-lookup cg-insert dist-fun u X k b heuristic*;
X' = *u*#*X*
in (*X'*,*TG'*))
define *rstates* **where** *rstates* = *reachable-states-as-list M1*
define *T0'* **where** *T0'* = *from-list* (*map V rstates*)
define *T0* **where** *T0* = (*if b*
then Prefix-Tree.combine T0' (*from-list* (*concat* (*map* (λ *q* . *language-for-input M1* (*initial M1*) (*map fst* (*V q*))) *rstates*)))
else T0')
define *G0* **where** *G0* = *cg-initial M1 T0*

have ∗:(*?f M1 V cg-initial cg-insert cg-lookup*) = *snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)
unfolding *k-def separate-state-def rstates-def heuristic-def T0'-def T0-def G0-def handle-state-cover-dynamic.simps Let-def*
by *simp*

have *separate-state-subset* : ⋀ *q X T G* . *set T* ⊆ *set* (*fst* (*snd* (*separate-state* (*X*,*T*,*G*) *q*)))
using *spyh-distinguish-subset*
unfolding *separate-state-def case-prod-conv Let-def snd-conv*
by *metis*
then have *set T0* ⊆ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))
unfolding ∗
by (*induction rstates rule*: *rev-induct*; *auto*; *metis* (*mono-tags*, *opaque-lifting*) *Collect-mono-iff prod.exhaust-sel*)
moreover have *set T0'* ⊆ *set T0*
unfolding *T0-def* **using** *combine-set* **by** *auto*
moreover have *V* ‘ *reachable-states M1* ⊆ *set T0'*
unfolding *T0'-def rstates-def* **using** *from-list-subset*
by (*metis image-set reachable-states-as-list-set*)
ultimately have *p1*: *V* ‘ *reachable-states M1* ⊆ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))
by *blast*

have *finite-tree T0'*
unfolding *T0'-def* **using** *from-list-finite-tree* **by** *auto*
then have *finite-tree T0*
unfolding *T0-def* **using** *combine-finite-tree*[*OF - from-list-finite-tree*]
by *auto*

have *separate-state-finite* : ⋀ *q X T G* . *finite-tree T* ⟹ *finite-tree* (*fst* (*snd* (*separate-state* (*X*,*T*,*G*) *q*)))
using *spyh-distinguish-finite*
unfolding *separate-state-def case-prod-conv Let-def snd-conv*
by *metis*
have *p2*: *finite-tree* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))

unfolding $*$
proof (*induction rstates rule*: *rev-induct*)
case *Nil*
show *?case* **using** ‹*finite-tree T0*› **by** *auto*
next
case (*snoc a rstates*)
have $*$:*foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*a*]) = *separate-state* (*foldl separate-state* ([], *T0*, *G0*) *rstates*) *a*
by *auto*
show *?case*
using *separate-state-finite*[*OF snoc.IH*]
unfolding $*$
by (*metis prod.collapse*)
qed

have $\bigwedge$ *q X T G* . *fst* (*separate-state* (*X*,*T*,*G*) *q*) = *V q* # *X*
unfolding *separate-state-def case-prod-conv Let-def fst-conv* **by** *blast*

have *heuristic-prop*: ($\bigwedge$*T w u′ uBest lBest*. *fst* (*heuristic T w* (*uBest*, *lBest*) *u′*) ∈ {*u′*, *uBest*})
unfolding *heuristic-def*
using *append-heuristic-input-in*[*of M1*] *append-heuristic-io-in*
by *fastforce*

have *p3*: *observable M1* $\Longrightarrow$
observable M2 $\Longrightarrow$
minimal M1 $\Longrightarrow$
minimal M2 $\Longrightarrow$
inputs M2 = *inputs M1* $\Longrightarrow$
outputs M2 = *outputs M1* $\Longrightarrow$
is-state-cover-assignment M1 V $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial $\Longrightarrow$
L M1 ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) $\Longrightarrow$
(*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*?f M1 V cg-initial cg-insert cg-lookup*)))
proof −
assume *a0*: *observable M1*
and *a1*: *observable M2*
and *a2*: *minimal M1*
and *a3*: *minimal M2*
and *a4*: *inputs M2* = *inputs M1*
and *a5*: *outputs M2* = *outputs M1*
and *a6*: *is-state-cover-assignment M1 V*
and *a7*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *a8*: *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*

and *a9*: *L M1* ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))

have ⋀ *rstates* . (*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *V* ‘ *list.set rstates*
proof −
fix *rstates* **show** (*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *V* ‘ *list.set rstates*
proof (*induction rstates rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a rstates*)
have ∗:(*foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*a*])) = *separate-state* (*foldl separate-state* ([], *T0*, *G0*) *rstates*) *a*
by *auto*
have ∗∗: ⋀ *q XTG* . *fst* (*separate-state XTG q*) = *V q* # *fst XTG*
using ‹⋀ *q X T G* . *fst* (*separate-state* (*X*,*T*,*G*) *q*) = *V q* # *X*› **by** *auto*

show *?case*
unfolding ∗ ∗∗
using *snoc* **by** *auto*
qed
qed
then have (*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *V* ‘ *reachable-states M1*
by (*metis reachable-states-as-list-set rstates-def*)

have ⋀ *q* . *q* ∈ *reachable-states M1* ⟹ *V q* ∈ *set T0*
using ‹*Prefix-Tree.set T0′* ⊆ *Prefix-Tree.set T0*› ‹*V* ‘ *reachable-states M1* ⊆ *Prefix-Tree.set T0′*› **by** *auto*

have *list.set rstates* ⊆ *reachable-states M1*
unfolding *rstates-def*
using *reachable-states-as-list-set* **by** *auto*
moreover have *L M1* ∩ *set* (*fst* (*snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *L M2* ∩ *set* (*fst* (*snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)))
using ∗ *a9* **by** *presburger*
ultimately have *preserves-divergence M1 M2* (*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)))
proof (*induction rstates rule*: *rev-induct*)
case *Nil*
have *L M1* ∩ *set T0* = *L M2* ∩ *set T0*
using *a9*
using ‹*set T0* ⊆ *set* (*fst* (*handle-state-cover-dynamic b c dist-fun M1 V cg-initial cg-insert cg-lookup*))› **by** *blast*

then have *convergence-graph-lookup-invar M1 M2 cg-lookup G0*
using *a8* ‹*finite-tree T0*›
unfolding *G0-def convergence-graph-initial-invar-def*
by *blast*
then show *?case* **by** *auto*
next
case (*snoc q rstates*)

obtain X' T' G' **where** *foldl separate-state* ([],*T0*,*G0*) *rstates* = (X',T',G')
using *prod-cases3* **by** *blast*
then have T' = *fst* (*snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))
and X' = *fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)
by *auto*

define *u* **where** *u* = *V q*
define *TG''* **where** *TG''* = *spyh-distinguish M1 T' G' cg-lookup cg-insert dist-fun u X' k b heuristic*
define *X''* **where** *X''* = *u*#*X'*

have *foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*q*]) = *separate-state* (X',T',G') *q*
using ‹*foldl separate-state* ([],*T0*,*G0*) *rstates* = (X',T',G')› **by** *auto*
also have *separate-state* (X',T',G') *q* = (*X''*,*TG''*)
unfolding *separate-state-def u-def TG''-def X''-def case-prod-conv Let-def*
by *auto*
finally have *foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*q*]) = (*X''*,*TG''*) **.**

have *set T'* ⊆ *set* (*fst* (*snd* (*foldl separate-state* ([],*T0*,*G0*) (*rstates*@[*q*]))))
using *separate-state-subset*
unfolding ‹*foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*q*]) = *separate-state* (X',T',G') *q*› **by** *simp*
then have *L M1* ∩ *set T'* = *L M2* ∩ *set T'*
using *snoc.prems*(*2*) **by** *blast*

then have *preserves-divergence M1 M2* (*list.set X'*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G'*
using *snoc* **unfolding** ‹*foldl separate-state* ([],*T0*,*G0*) *rstates* = (X',T',G')›

by *auto*

have *set T0* ⊆ *set T'*
using *separate-state-subset*
unfolding ‹*T'* = *fst* (*snd* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))›
by (*induction rstates rule*: *rev-induct*; *auto*; *metis* (*mono-tags*, *opaque-lifting*) *Collect-mono-iff prod.collapse*)

have *V q* ∈ *set T0*
using *snoc.prems*

using ‹$\bigwedge q.\ q \in$ *reachable-states M1* $\Longrightarrow$ *V q* $\in$ *Prefix-Tree.set T0*› **by** *auto*
then have *V q* $\in$ *set T′*
using ‹*set T0* $\subseteq$ *set T′*› **by** *auto*
moreover have *V q* $\in$ *L M1*
proof −
have *q* $\in$ *reachable-states M1*
using *snoc.prems*(*1*) **by** *auto*
then show *?thesis*
using *is-state-cover-assignment-language*[*OF a6*] **by** *blast*
qed
ultimately have *V q* $\in$ *L M2*
using ‹*L M1* $\cap$ *set T′* = *L M2* $\cap$ *set T′*› **by** *blast*

have *list.set X′* = *V* ‘ *list.set rstates*
unfolding ‹*X′* = *fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*)›
using ‹$\bigwedge$ *rstates* . (*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *V* ‘ *list.set rstates*›
by *blast*
moreover have *list.set rstates* $\subseteq$ *reachable-states M1*
using *snoc.prems*(*1*) **by** *auto*
ultimately have *list.set X′* $\subseteq$ *set T′*
using ‹*set T0* $\subseteq$ *set T′*›
using ‹$\bigwedge q.\ q \in$ *reachable-states M1* $\Longrightarrow$ *V q* $\in$ *Prefix-Tree.set T0*› **by** *auto*
moreover have *list.set X′* $\subseteq$ *L M1*
using ‹*list.set X′* = *V* ‘ *list.set rstates*› ‹*list.set rstates* $\subseteq$ *reachable-states M1*› *a6*
by (*metis dual-order.trans image-mono state-cover-assignment-language*)
ultimately have *list.set X′* $\subseteq$ *L M2*
using ‹*L M1* $\cap$ *set T′* = *L M2* $\cap$ *set T′*› **by** *blast*

have $*$: *L M1* $\cap$ *set* (*fst* (*spyh-distinguish M1 T′ G′ cg-lookup cg-insert dist-fun* (*V q*) *X′ k b heuristic*)) =
L M2 $\cap$ *set* (*fst* (*spyh-distinguish M1 T′ G′ cg-lookup cg-insert dist-fun* (*V q*) *X′ k b heuristic*))
using *snoc.prems*(*2*) *TG″-def* ‹*foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*q*]) = *separate-state* (*X′*, *T′*, *G′*) *q*› ‹*separate-state* (*X′*, *T′*, *G′*) *q* = (*X″*, *TG″*)› *u-def* **by** *auto*

have *preserves-divergence M1 M2* (*Set.insert* (*V q*) (*list.set X′*))
using *spyh-distinguish-preserves-divergence*[*OF a0 a1 a2 a3* ‹*V q* $\in$ *L M1*› ‹*V q* $\in$ *L M2*› *assms*(*1*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*› *a7* ‹*list.set X′* $\subseteq$ *L M1*› ‹*list.set X′* $\subseteq$ *L M2*› $*$ *heuristic-prop* ‹*preserves-divergence M1 M2* (*list.set X′*)›]
by *presburger*
then have *preserves-divergence M1 M2* (*list.set X″*)

by (*metis X''-def list.simps*(*15*) *u-def*)

moreover have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)
using *spyh-distinguish-establishes-divergence*(*2*)[*OF a0 a1 a2 a3* ‹*V q* ∈ *L M1*› ‹*V q* ∈ *L M2*› *assms*(*1*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G'*› *a7* ‹*list.set X'* ⊆ *L M1*› ‹*list.set X'* ⊆ *L M2*› ∗ *heuristic-prop*]
unfolding *u-def*[*symmetric*] *TG''-def*[*symmetric*]
by *presburger*
ultimately show *?case*
unfolding ‹*foldl separate-state* ([], *T0*, *G0*) (*rstates*@[*q*]) = (*X''*,*TG''*)› *snd-conv fst-conv*
by *blast*
qed
then show *?thesis*
unfolding ‹(*list.set* (*fst* (*foldl separate-state* ([],*T0*,*G0*) *rstates*))) = *V* ‘ *reachable-states M1*›
unfolding ∗ **.**
qed

show *?P V*
using *p1 p2 p3* **by** *blast*
qed

then show *?thesis*
unfolding *separates-state-cover-def* **by** *blast*
qed

21.4.2 Static

```
fun handle-state-cover-static :: (nat ⇒ 'a ⇒ ('b×'c) prefix-tree) ⇒
                                 ('a::linorder,'b::linorder,'c::linorder) fsm ⇒
                                 ('a,'b,'c) state-cover-assignment ⇒
                                 (('a,'b,'c) fsm ⇒ ('b×'c) prefix-tree ⇒ 'd) ⇒
                                 ('d ⇒ ('b×'c) list ⇒ 'd) ⇒
                                 ('d ⇒ ('b×'c) list ⇒ ('b×'c) list list) ⇒
                                 (('b×'c) prefix-tree × 'd)
  where
  handle-state-cover-static dist-set M V cg-initial cg-insert cg-lookup  =
    (let
      separate-state = (λ T q . combine-after T (V q) (dist-set 0 q));
      T' = foldl separate-state empty (reachable-states-as-list M);
      G' = cg-initial M T'
    in (T',G'))
```

lemma *handle-state-cover-static-applies-dist-sets*:

assumes $q \in$ *reachable-states M1*
shows *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) (*V q*))
(**is** *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after ?T* (*V q*)))
proof −

define *k* **where** *k* = *2* ∗ *size M1*
define *separate-state* **where** *separate-state* = (λ *T q* . *combine-after T* (*V q*) (*dist-fun 0 q*))
define *rstates* **where** *rstates* = *reachable-states-as-list M1*
define *T* **where** *T* = *foldl separate-state empty rstates*
define *G* **where** *G* = *cg-initial M1 T*

have ∗:*?T* = *T*
unfolding *k-def separate-state-def rstates-def T-def G-def handle-state-cover-static.simps Let-def*
by *simp*

have *separate-state-subset* : ⋀ *q T* . *set T* $\subseteq$ *set* (*separate-state T q*)
unfolding *separate-state-def combine-after-set*
by *blast*

have ⋀ *q* . *q* $\in$ *list.set rstates* ⟹ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V q*))
proof −
fix *q* **assume** *q* $\in$ *list.set rstates*
then show *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V q*))
unfolding *T-def* **proof** (*induction rstates arbitrary*: *q rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a rstates*)
have ∗: *foldl separate-state empty* (*rstates*@[*a*]) = *separate-state* (*foldl separate-state empty rstates*) *a*
by *auto*
show *?case* **proof** (*cases q* = *a*)
case *True*
show *?thesis*
unfolding *True* **using** *separate-state-def combine-after-after-subset* **by** *force*
next
case *False*
then have ‹*q* $\in$ *list.set rstates*› **using** *snoc.prems* **by** *auto*
then have *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after* (*foldl separate-state empty rstates*) (*V q*))
using *snoc.IH* **by** *auto*
moreover have *set* (*after* (*foldl separate-state empty rstates*) (*V q*)) $\subseteq$ *set* (*after* (*foldl separate-state empty* (*rstates*@[*a*])) (*V q*))
unfolding ∗
using *subset-after-subset*[*OF separate-state-subset*] **by** *blast*

ultimately show *?thesis* **by** *blast*
qed
qed
qed

then show *?thesis*
unfolding *rstates-def* ‹*?T* = *T*› **using** *assms*
using *reachable-states-as-list-set* **by** *auto*
qed

lemma *handle-state-cover-static-separates-state-cover*:
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
fixes *cg-insert* :: ('*d* ⇒ ('*b*×'*c*) *list* ⇒ '*d*)
assumes *observable M1* ⟹ *minimal M1* ⟹ (⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*)
and ⋀ *k q* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows *separates-state-cover* (*handle-state-cover-static dist-fun*) *M1 M2 cg-initial cg-insert cg-lookup*
proof −

let *?f* = (*handle-state-cover-static dist-fun*)

have ⋀ (*V* :: ('*a*,'*b*,'*c*) *state-cover-assignment*) .
(*V* ‘ *reachable-states M1* ⊆ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)))
∧ *finite-tree* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))
∧ (*observable M1* ⟶
observable M2 ⟶
minimal M1 ⟶
minimal M2 ⟶
inputs M2 = *inputs M1* ⟶
outputs M2 = *outputs M1* ⟶
is-state-cover-assignment M1 V ⟶
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟶
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial ⟶
L M1 ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* ∩ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) ⟶
(*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*?f M1 V cg-initial cg-insert cg-lookup*)))) (**is** ⋀ *V* . *?P V*)
proof −
fix *V* :: ('*a*,'*b*,'*c*) *state-cover-assignment*

define *k* **where** *k* = *2* ∗ *size M1*
define *separate-state* **where** *separate-state* = (λ *T q* . *combine-after T* (*V q*) (*dist-fun 0 q*))

define *rstates* **where** *rstates* = *reachable-states-as-list M1*
define *T* **where** *T* = *foldl separate-state empty rstates*
define *G* **where** *G* = *cg-initial M1 T*

have ∗:(*?f M1 V cg-initial cg-insert cg-lookup*) = (*T*,*G*)
unfolding *k-def separate-state-def rstates-def T-def G-def handle-state-cover-static.simps Let-def*
by *simp*

have *separate-state-subset* : ⋀ *q T* . *set T* ⊆ *set* (*separate-state T q*)
unfolding *separate-state-def combine-after-set*
by *blast*

have *V* ‘ (*list.set rstates*) ⊆ *set T*
unfolding *T-def* **proof** (*induction rstates rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a rstates*)
have ∗: *foldl separate-state empty* (*rstates*@[*a*]) = *separate-state* (*foldl separate-state empty rstates*) *a*
by *auto*

have *V* ‘ (*list.set rstates*) ⊆ *set* (*foldl separate-state empty* (*rstates*@[*a*]))
using *snoc separate-state-subset* **by** *auto*
moreover have *V a* ∈ *set* (*separate-state* (*foldl separate-state empty rstates*) *a*)
unfolding *separate-state-def combine-after-set*
by *simp*
ultimately show *?case*
unfolding ∗ **by** *auto*
qed
then have *p1*: (*V* ‘ *reachable-states M1* ⊆ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)))
unfolding *rstates-def* ∗
using *reachable-states-as-list-set* **by** *auto*

have *separate-state-finite* : ⋀ *q X T G* . *q* ∈ *states M1* ⟹ *finite-tree T* ⟹ *finite-tree* (*separate-state T q*)
unfolding *separate-state-def* **using** *combine-after-finite-tree*[*OF - assms*(*2*)]
by *metis*
moreover have ⋀ *q* . *q* ∈ *list.set rstates* ⟹ *q* ∈ *states M1*
unfolding *rstates-def*
by (*metis reachable-state-is-state reachable-states-as-list-set*)
ultimately have *p2*: *finite-tree* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))
unfolding ∗ *fst-conv T-def* **using** *empty-finite-tree*
by (*induction rstates rule*: *rev-induct*; *auto*)

have *p3*: *observable M1* ⟹

observable M2 $\Longrightarrow$
minimal M1 $\Longrightarrow$
minimal M2 $\Longrightarrow$
inputs M2 = *inputs M1* $\Longrightarrow$
outputs M2 = *outputs M1* $\Longrightarrow$
is-state-cover-assignment M1 V $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
convergence-graph-initial-invar M1 M2 cg-lookup cg-initial $\Longrightarrow$
L M1 $\cap$ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* $\cap$ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) $\Longrightarrow$
(*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*?f M1 V cg-initial cg-insert cg-lookup*)))
proof −
assume *a0*: *observable M1*
and *a1*: *observable M2*
and *a2*: *minimal M1*
and *a3*: *minimal M2*
and *a4*: *inputs M2* = *inputs M1*
and *a5*: *outputs M2* = *outputs M1*
and *a6*: *is-state-cover-assignment M1 V*
and *a7*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and *a8*: *convergence-graph-initial-invar M1 M2 cg-lookup cg-initial*
and *a9*: *L M1* $\cap$ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*)) = *L M2* $\cap$ *set* (*fst* (*?f M1 V cg-initial cg-insert cg-lookup*))

have *L M1* $\cap$ *set T* = *L M2* $\cap$ *set T*
using *a9* **unfolding** $*$ **by** *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*?f M1 V cg-initial cg-insert cg-lookup*))
using *a8 p2*
unfolding $*$ *fst-conv snd-conv G-def convergence-graph-initial-invar-def*
by *blast*
moreover have *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
proof −
have $\bigwedge$ *u v* . *u*$\in$*L M1* $\cap$ *V* ‘ *reachable-states M1* $\Longrightarrow$ *v*$\in$*L M1* $\cap$ *V* ‘ *reachable-states M1* $\Longrightarrow$ $\neg$ *converge M1 u v* $\Longrightarrow$ $\neg$ *converge M2 u v*
proof −
fix *u v* **assume** *u*$\in$*L M1* $\cap$ *V* ‘ *reachable-states M1* **and** *v*$\in$*L M1* $\cap$ *V* ‘ *reachable-states M1* **and** $\neg$ *converge M1 u v*
then obtain *qv qu* **where** *qu* $\in$ *reachable-states M1* **and** *u* = *V qu*
qv $\in$ *reachable-states M1* **and** *v* = *V qv*
by *auto*
then have *u* $\in$ *L M1* **and** *v* $\in$ *L M1*
using *a6* **by** (*meson is-state-cover-assignment-language*)+
then have *qu* $\neq$ *qv*
using *a6* ‹$\neg$ *converge M1 u v*›
using ‹*u* = *V qu*› ‹*v* = *V qv*› *a0 a2 convergence-minimal* **by** *blast*
moreover have *qu* $\in$ *states M1* **and** *qv* $\in$ *states M1*

using ‹*qu* ∈ *reachable-states M1*› ‹*qv* ∈ *reachable-states M1*›
by (*simp add*: *reachable-state-is-state*)+
ultimately obtain *w* **where** *distinguishes M1 qu qv w* **and** *w* ∈ *set* (*dist-fun 0 qu*) **and** *w* ∈ *set* (*dist-fun 0 qv*)
using *assms*(*1*)[*OF a0 a2*]
by (*metis Int-iff*)
then have *w* ≠ []
by (*meson* ‹*qu* ∈ *FSM.states M1*› ‹*qv* ∈ *FSM.states M1*› *distinguishes-not-Nil*)

have (*u*@*w* ∈ *L M1*) ≠ (*v*@*w* ∈ *L M1*)
unfolding ‹*u* = *V qu*› ‹*v* = *V qv*›
using *state-cover-assignment-after*[*OF a0 a6* ‹*qu* ∈ *reachable-states M1*›]
using *state-cover-assignment-after*[*OF a0 a6* ‹*qv* ∈ *reachable-states M1*›]
by (*metis* ‹*distinguishes M1 qu qv w*› *a0 after-distinguishes-language*)
moreover have *u*@*w* ∈ *set T*
using *handle-state-cover-static-applies-dist-sets*[*OF* ‹*qu* ∈ *reachable-states M1*›, *of dist-fun V cg-initial cg-insert cg-lookup*] ‹*w* ∈ *set* (*dist-fun 0 qu*)› ‹*w* ≠ []›
unfolding ∗ *fst-conv after-set* ‹*u* = *V qu*› **by** *auto*
moreover have *v*@*w* ∈ *set T*
using *handle-state-cover-static-applies-dist-sets*[*OF* ‹*qv* ∈ *reachable-states M1*›, *of dist-fun V cg-initial cg-insert cg-lookup*] ‹*w* ∈ *set* (*dist-fun 0 qv*)› ‹*w* ≠ []›
unfolding ∗ *fst-conv after-set* ‹*v* = *V qv*› **by** *auto*
ultimately have (*u*@*w* ∈ *L M2*) ≠ (*v*@*w* ∈ *L M2*)
using ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*›
by *blast*
then show ¬ *converge M2 u v*
using *a1 converge-append-language-iff* **by** *blast*
qed
then show *?thesis*
unfolding *preserves-divergence.simps* **by** *blast*
qed
ultimately show *?thesis*
by *blast*
qed

show *?P V*
using *p1 p2 p3* **by** *blast*
qed

then show *?thesis*
unfolding *separates-state-cover-def* **by** *blast*
qed

21.5 Establishing Convergence of Traces

21.5.1 Dynamic

fun *distinguish-from-set* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*'a*,*'b*,*'c*) *state-cover-assignment* ⇒ (*'b*×*'c*) *prefix-tree* ⇒ *'d* ⇒ (*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*)

list list) ⇒ (′*d* ⇒ (′*b*×′*c*) *list* ⇒ ′*d*) ⇒ (′*a* ⇒ ′*a* ⇒ (′*b*×′*c*) *list*) ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list* ⇒ (′*b*×′*c*) *list list* ⇒ *nat* ⇒ *nat* ⇒ *bool* ⇒ ((′*b*×′*c*) *prefix-tree* ⇒ (′*b*×′*c*) *list* ⇒ ((′*b*×′*c*) *list* × *int*) ⇒ (′*b*×′*c*) *list* ⇒ ((′*b*×′*c*) *list* × *int*)) ⇒ *bool* ⇒ ((′*b*×′*c*) *prefix-tree* × ′*d*) **where**
distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v=
(*let TG′ = spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*;
vClass = Set.insert v (*list.set* (*cg-lookup* (*snd TG′*) *v*));
notReferenced = (¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*);
TG′′ = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG′*)
in if depth > *0*
then let X′ = if notReferenced then (*v*#*u*#*X*) *else* (*u*#*X*);
XY = List.product (*inputs-as-list M*) (*outputs-as-list M*);
handleIO = (λ (*T*,*G*) (*x*,*y*) . (*let TGu = distribute-extension M T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*;
TGv = if u-is-v then TGu else distribute-extension M (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*
in if is-in-language M (*initial M*) (*u*@[(*x*,*y*)])
then distinguish-from-set M V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k* (*depth* − *1*) *completeInputTraces append-heuristic u-is-v else TGv*))
in foldl handleIO TG′′ XY
else TG′′)

lemma *distinguish-from-set-subset* :
set T ⊆ *set* (*fst* (*distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v*))
proof (*induction depth arbitrary*: *T G u v X*)
case *0*

define *TG′* **where** *TG′*: *TG′ = spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass = Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = ((¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*))
define *TG′′* **where** *TG′′*: *TG′′* = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG′*)

have *distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic u-is-v = TG′′*
apply (*subst distinguish-from-set.simps*)
unfolding *TG′ vClass notReferenced TG′′ Let-def*

by *force*
moreover have *set T* ⊆ *set* (*fst* (*TG′*))
unfolding *TG′*
using *spyh-distinguish-subset*
by *metis*
moreover have *set* (*fst* (*TG′*)) ⊆ *set* (*fst* (*TG′′*))
unfolding *TG′′*
using *spyh-distinguish-subset*
by (*metis* (*mono-tags*, *lifting*) *equalityE*)
ultimately show *?case*
by *blast*
next
case (*Suc depth*)

have (*Suc depth* − *1*) = *depth*
by *auto*

define *TG′* **where** *TG′*: *TG′* = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass* = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = ((¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*))
define *TG′′* **where** *TG′′*: *TG′′* = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG′*)
define *X′* **where** *X′*: *X′* = (*if notReferenced then* (*v*#*u*#*X*) *else* (*u*#*X*))
define *XY* **where** *XY*: *XY* = *List.product* (*inputs-as-list M*) (*outputs-as-list M*)
define *handleIO* **where** *handleIO*: *handleIO* = (λ (*T*,*G*) (*x*,*y*) . (*let TGu* = *distribute-extension M T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*;
TGv = *if u-is-v then TGu else distribute-extension M* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*
in if is-in-language M (*initial M*) (*u*@[(*x*,*y*)])
then distinguish-from-set M V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k* (*depth*) *completeInputTraces append-heuristic u-is-v*
else TGv))

have ⋀ *x y T G* . *set T* ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
proof −
fix *x y T G*

define *TGu* **where** *TGu*: *TGu* = *distribute-extension M T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*

define *TGv* **where** *TGv*: *TGv* = (*if u-is-v then TGu else distribute-extension M* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*)
have ∗: *handleIO* (*T*,*G*) (*x*,*y*) = (*if is-in-language M* (*initial M*) (*u*@[(*x*,*y*)])
then distinguish-from-set M V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k* (*depth*) *completeInputTraces append-heuristic u-is-v*
else TGv)
unfolding *handleIO TGu TGv case-prod-conv Let-def*
by *auto*

have *set T* ⊆ *set* (*fst TGu*)
unfolding *TGu*
using *distribute-extension-subset*
by *metis*
moreover have *set* (*fst TGu*) ⊆ *set* (*fst TGv*)
unfolding *TGv*
using *distribute-extension-subset* **by** *force*
ultimately have *set T* ⊆ *set* (*fst TGv*)
by *blast*

show *set T* ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
unfolding ∗
using ‹*set T* ⊆ *set* (*fst TGv*)›
using *Suc.IH*[*of fst TGv snd TGv u*@[(*x*,*y*)] *v*@[(*x*,*y*)] *X′*]
by (*cases is-in-language M* (*initial M*) (*u*@[(*x*,*y*)]); *auto*)
qed

have *set* (*fst TG″*) ⊆ *set* (*fst* (*foldl handleIO TG″ XY*))
proof (*induction XY rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a XY*)
obtain *x y* **where** *a* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
then have ∗: (*foldl handleIO TG″* (*XY*@[*a*])) = *handleIO* (*fst* (*foldl handleIO TG″ XY*),*snd* (*foldl handleIO TG″ XY*)) (*x*,*y*)
by *auto*

show *?case*
using *snoc* **unfolding** ∗
using ‹⋀ *x y T G* . *set T* ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))›
by *blast*
qed
moreover have *set T* ⊆ *set* (*fst TG″*)
proof −
have *set T* ⊆ *set* (*fst TG′*)
unfolding *TG′*
using *spyh-distinguish-subset*

by *metis*
moreover have *set* (*fst TG′*) ⊆ *set* (*fst TG′′*)
unfolding *TG′′*
using *spyh-distinguish-subset*
by (*metis* (*mono-tags*, *lifting*) *order-refl*)
ultimately show *?thesis*
by *blast*
qed
moreover have *distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k* (*Suc depth*) *completeInputTraces append-heuristic u-is-v* = *foldl handleIO TG′′ XY*
apply (*subst distinguish-from-set.simps*)
unfolding *TG′ vClass notReferenced TG′′ Let-def X′ XY handleIO*
unfolding ‹(*Suc depth* − *1*) = *depth*›
by *force*

ultimately show *?case*
by (*metis* (*no-types*, *lifting*) *order-trans*)
qed

lemma *distinguish-from-set-finite* :
fixes *T* :: (*'b*::*linorder*×*'c*::*linorder*) *prefix-tree*
assumes *finite-tree T*
shows *finite-tree* (*fst* (*distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic u-is-v*))
using *assms* **proof** (*induction depth arbitrary*: *T G u v X*)
case *0*

define *TG′* **where** *TG′*: *TG′* = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass* = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = ((¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*))
define *TG′′* **where** *TG′′*: *TG′′* = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInput-Traces append-heuristic else TG′*)

have *finite-tree* (*fst* (*TG′*))
unfolding *TG′*
using *spyh-distinguish-finite 0*
by *metis*
then have *finite-tree* (*fst* (*TG′′*))
unfolding *TG′′*
using *spyh-distinguish-finite*[*OF* ‹*finite-tree* (*fst* (*TG′*))›, *of M snd TG′*]
by *auto*
moreover have *distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic u-is-v*= *TG′′*

apply (*subst distinguish-from-set.simps*)
unfolding *TG′ vClass notReferenced TG″ Let-def*
by *force*
ultimately show *?case*
by *blast*
next
case (*Suc depth*)

have (*Suc depth − 1*) = *depth*
by *auto*

define *TG′* **where** *TG′*: *TG′* = *spyh-distinguish M T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass* = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = ((¬ *u-is-v*) ∧ (∀ *q* ∈ *reachable-states M* . *V q* ∉ *vClass*))
define *TG″* **where** *TG″*: *TG″* = (*if notReferenced then spyh-distinguish M* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG′*)
define *X′* **where** *X′*: *X′* = (*if notReferenced then* (*v*#*u*#*X*) *else* (*u*#*X*))
define *XY* **where** *XY*: *XY* = *List.product* (*inputs-as-list M*) (*outputs-as-list M*)
define *handleIO* **where** *handleIO*: *handleIO* = (λ (*T*,*G*) (*x*,*y*) . (*let TGu* = *distribute-extension M T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*;
TGv = *if u-is-v then TGu else distribute-extension M* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*
in if is-in-language M (*initial M*) (*u*@[(*x*,*y*)])
then distinguish-from-set M V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k* (*depth*) *completeInputTraces append-heuristic u-is-v*
else TGv))

have ⋀ *x y T G* . *finite-tree T* ⟹ *finite-tree* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
proof −
fix *T* :: (*′b*::*linorder*×*′c*::*linorder*) *prefix-tree*
fix *x y G* **assume** *finite-tree T*

define *TGu* **where** *TGu*: *TGu* = *distribute-extension M T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*
define *TGv* **where** *TGv*: *TGv* = (*if u-is-v then TGu else distribute-extension M* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*)
have ∗: *handleIO* (*T*,*G*) (*x*,*y*) = (*if is-in-language M* (*initial M*) (*u*@[(*x*,*y*)])
then distinguish-from-set M V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k*

(*depth*) *completeInputTraces append-heuristic u-is-v*
else TGv)
unfolding *handleIO TGu TGv case-prod-conv Let-def*
by *auto*

have *finite-tree* (*fst TGu*)
unfolding *TGu*
using *distribute-extension-finite* ‹*finite-tree T*›
by *metis*
then have *finite-tree* (*fst TGv*)
unfolding *TGv*
using *distribute-extension-finite* **by** *force*
then show *finite-tree* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
unfolding ∗
using *Suc.IH*[*of fst TGv snd TGv u*@[(*x*,*y*)] *v*@[(*x*,*y*)] *X'*]
by (*cases is-in-language M* (*initial M*) (*u*@[(*x*,*y*)]); *auto*)
qed

have *finite-tree* (*fst TG'*)
unfolding *TG'*
using *spyh-distinguish-finite* ‹*finite-tree T*›
by *metis*
then have *finite-tree* (*fst TG''*)
unfolding *TG''*
using *spyh-distinguish-finite*[*OF* ‹*finite-tree* (*fst* (*TG'*))›, *of M snd TG'*]
by *auto*

have *finite-tree* (*fst* (*foldl handleIO TG'' XY*))
proof (*induction XY rule*: *rev-induct*)
case *Nil*
then show *?case* **using** ‹*finite-tree* (*fst TG''*)› **by** *auto*
next
case (*snoc a XY*)
obtain *x y* **where** *a* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
then have ∗: (*foldl handleIO TG''* (*XY*@[*a*])) = *handleIO* (*fst* (*foldl handleIO TG'' XY*),*snd* (*foldl handleIO TG'' XY*)) (*x*,*y*)
by *auto*

show *?case*
using *snoc* **unfolding** ∗
using ‹⋀ *x y T G* . *finite-tree T* ⟹ *finite-tree* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))›
by *blast*
qed
moreover have *distinguish-from-set M V T G cg-lookup cg-insert get-distinguishing-trace u v X k* (*Suc depth*) *completeInputTraces append-heuristic u-is-v* = *foldl handleIO TG'' XY*
apply (*subst distinguish-from-set.simps*)

unfolding *TG′ vClass notReferenced TG″ Let-def X′ XY handleIO*
unfolding ‹(*Suc depth* − *1*) = *depth*›
by *force*

ultimately show *?case*
by (*metis* (*no-types*, *lifting*))
qed

lemma *distinguish-from-set-properties* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and *is-state-cover-assignment M1 V*
and *V* ‘ *reachable-states M1* ⊆ *list.set X*
and *preserves-divergence M1 M2* (*list.set X*)
and ⋀ *w* . *w* ∈ *list.set X* ⟹ ∃ *w′* . *converge M1 w w′* ∧ *converge M2 w w′*
and *converge M1 u v*
and *u* ∈ *L M2*
and *v* ∈ *L M2*
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and ⋀ *α β q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *L M1* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*))) = *L M2* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*)))
and ⋀ *T w u′ uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u′*) ∈ {*u′*,*uBest*}
shows ∀ *γ x y* . *length* (*γ*@[(*x*,*y*)]) ≤ *depth* ⟶
γ ∈ *LS M1* (*after-initial M1 u*) ⟶
x ∈ *inputs M1* ⟶ *y* ∈ *outputs M1* ⟶
L M1 ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))}) = *L M2* ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))})
∧ *preserves-divergence M1 M2* (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ*@[(*x*,*y*)]))})
(**is** *?P1a X u v depth*)
and *preserves-divergence M1 M2* (*list.set X* ∪ {*u*,*v*})
(**is** *?P1b X u v*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k depth completeInputTraces append-heuristic* (*u* = *v*)))
(**is** *?P2 T G u v X depth*)
proof −

have *?P1a X u v depth* ∧ *?P1b X u v* ∧ *?P2 T G u v X depth*
using *assms*(*8*−*14*) *assms*(*17*)
proof (*induction depth arbitrary*: *T G u v X*)
case *0*

define *TG′* **where** *TG′*: *TG′* = *spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass* = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG′*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = ((¬ (*u* = *v*)) ∧ (∀ *q* ∈ *reachable-states M1* . *V q* ∉ *vClass*))
define *TG″* **where** *TG″*: *TG″* = (*if notReferenced then spyh-distinguish M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG′*)

have *distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*) = *TG″*
apply (*subst distinguish-from-set.simps*)
unfolding *TG′ vClass notReferenced TG″ Let-def*
by *force*

have *set T* ⊆ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*)))
using *distinguish-from-set-subset* **by** *metis*
then have *L M1* ∩ *set T* = *L M2* ∩ *set T*
using *0.prems*(*8*)
by *blast*

have *list.set X* ⊆ *L M1* **and** *list.set X* ⊆ *L M2*
using *0.prems*(*3*)
by (*meson converge.elims*(*2*) *subsetI*)+
have *set* (*fst TG′*) ⊆ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*)))
by (*metis TG″* ‹*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*) = *TG″*› *order-refl spyh-distinguish-subset*)

then have ∗: *L M1* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*)) =
L M2 ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*))
using *0.prems*(*8*) **unfolding** *TG′*
by *blast*
have *u* ∈ *L M1* **and** *v* ∈ *L M1*
using ‹*converge M1 u v*› **by** *auto*

have *preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))
using *spyh-distinguish-preserves-divergence*[*OF assms*(*1*−*4*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› *assms*(*16*) *0.prems*(*7*) *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*) *0.prems*(*2*)]

unfolding *TG′* **by** *presburger*

have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)
unfolding *TG′*
using *spyh-distinguish-establishes-divergence*[*OF assms*(*1−4*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› *assms*(*16*) *0.prems*(*7*) *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*)]
by *linarith*

have *L M1* ∩ *set* (*fst TG″*) = *L M2* ∩ *set* (*fst TG″*)
using *0.prems*(*8*)
unfolding ‹*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*) = *TG″*›
by *blast*

have *preserves-divergence M1 M2* (*Set.insert v* (*list.set X*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG″*)
proof −
have *preserves-divergence M1 M2* (*Set.insert v* (*list.set X*)) ∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG″*)
proof (*cases notReferenced*)
case *True*

then have *TG″* = *spyh-distinguish M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*
unfolding *TG″* **by** *auto*
then have ∗: *L M1* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*)) =
L M2 ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*))
using ‹*L M1* ∩ *set* (*fst TG″*) = *L M2* ∩ *set* (*fst TG″*)›
by *simp*

show *?thesis*
using *spyh-distinguish-preserves-divergence*[*OF assms*(*1−4*) ‹*v* ∈ *L M1*› ‹*v* ∈ *L M2*› *assms*(*16*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)› *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*) *0.prems*(*2*)]
using *spyh-distinguish-establishes-divergence*(*2*)[*OF assms*(*1−4*) ‹*v* ∈ *L M1*› ‹*v* ∈ *L M2*› *assms*(*16*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)› *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*)]
unfolding ‹*TG″* = *spyh-distinguish M1* (*fst TG′*) (*snd TG′*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*›
by *presburger*
next
case *False*
then consider *u* = *v* | (*u* ≠ *v*) ∧ ¬(∀ *q* ∈ *reachable-states M1* . *V q* ∉ *vClass*)

```
      unfolding notReferenced by blast
    then show ?thesis proof cases
      case 1
      then show ?thesis
       using False TG'' ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd
TG')› ‹preserves-divergence M1 M2 (Set.insert u (list.set X))› by presburger
    next
      case 2
      then have TG'' = TG'
        unfolding TG'' using False  by auto

      obtain q where q ∈ reachable-states M1
              and V q ∈ Set.insert v (list.set (cg-lookup (snd TG') v))
        using 2
        unfolding notReferenced vClass
        by blast

      have converge M1 (V q) v and converge M2 (V q) v
      proof −
        have converge M1 v (V q) ∧ converge M2 v (V q)
        proof (cases V q = v)
          case True
          then show ?thesis
            using ‹v ∈ L M1› ‹v ∈ L M2› by auto
        next
          case False
          then have V q ∈ list.set (cg-lookup (snd TG') v)
            using ‹V q ∈ Set.insert v (list.set (cg-lookup (snd TG') v))›
            by blast
          then show ?thesis
            using ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG')›
            unfolding convergence-graph-lookup-invar-def
            using 0.prems(6) ‹v ∈ L M1› by blast
        qed
        then show converge M1 (V q) v and converge M2 (V q) v
          by auto
      qed

      have V q ∈ Set.insert u (list.set X)
        using ‹q ∈ reachable-states M1› 0.prems(1) by blast

      have preserves-divergence M1 M2 (Set.insert v (list.set X))
          using preserves-divergence-converge-insert[OF assms(1−4) ‹converge
M1 (V q) v› ‹converge M2 (V q) v› ‹preserves-divergence M1 M2 (Set.insert u
(list.set X))› ‹V q ∈ Set.insert u (list.set X)›]
        unfolding preserves-divergence.simps by blast
      then show ?thesis
        unfolding ‹TG'' = TG'›
        using ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG')›
```

by *auto*
qed
qed
then show *preserves-divergence M1 M2* (*Set.insert v* (*list.set X*)) **and** *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)
by *auto*
qed

have *converge M1 u u* **and** *converge M1 v v* **and** *converge M1 v u* **and** *converge M1 u v*
using ‹*u* ∈ *L M1*› ‹*v* ∈ *L M1*› ‹*converge M1 u v*› **by** *auto*
then have *preserves-divergence M1 M2* (*Set.insert u* (*Set.insert v* (*list.set X*)))
using ‹*preserves-divergence M1 M2* (*Set.insert v* (*list.set X*))›
‹*preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))›
unfolding *preserves-divergence.simps*
by *blast*
then have *?P1b X u v*
by (*metis Un-insert-right sup-bot-right*)
moreover have *?P2 T G u v X 0*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)›
using ‹*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k 0 completeInputTraces append-heuristic* (*u* = *v*) = *TG''*› **by** *blast*
moreover have *P1*: *?P1a X u v 0*
by *auto*
ultimately show *?case*
by *blast*
next
case (*Suc depth*)
have *0* < *Suc depth* = *True*
by *auto*
have *Suc depth* − *1* = *depth*
by *auto*

have *u* ∈ *L M1* **and** *v* ∈ *L M1*
using ‹*converge M1 u v*› **by** *auto*

define *TG'* **where** *TG'*: *TG'* = *spyh-distinguish M1 T G cg-lookup cg-insert get-distinguishing-trace u X k completeInputTraces append-heuristic*
define *vClass* **where** *vClass*: *vClass* = *Set.insert v* (*list.set* (*cg-lookup* (*snd TG'*) *v*))
define *notReferenced* **where** *notReferenced*: *notReferenced* = (¬(*u* = *v*) ∧ (∀ *q* ∈ *reachable-states M1* . *V q* ∉ *vClass*))
define *TG''* **where** *TG''*: *TG''* = (*if notReferenced then spyh-distinguish M1* (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic else TG'*)
define *X'* **where** *X'*: *X'* = (*if notReferenced then* (*v*#*u*#*X*) *else* (*u*#*X*))
define *XY* **where** *XY*: *XY* = *List.product* (*inputs-as-list M1*) (*outputs-as-list M1*)
define *handleIO* **where** *handleIO*: *handleIO* = (λ (*T*,*G*) (*x*,*y*). (*let TGu* =

distribute-extension M1 T G cg-lookup cg-insert u [(*x*,*y*)] *completeInputTraces append-heuristic*;
TGv = *if* (*u* = *v*) *then TGu else distribute-extension M1* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*
in if is-in-language M1 (*initial M1*) (*u*@[(*x*,*y*)])
then distinguish-from-set M1 V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k depth completeInputTraces append-heuristic* (*u* = *v*)
else TGv))

have *result*: *distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v X k* (*Suc depth*) *completeInputTraces append-heuristic* (*u* = *v*) = *foldl handleIO TG″ XY*
apply (*subst distinguish-from-set.simps*)
unfolding *TG′ vClass notReferenced TG″ X′ XY handleIO* ‹*0* < *Suc depth* = *True*› *case-prod-conv* ‹*Suc depth* − *1* = *depth*› *if-True Let-def*
by *force*
then have *pass-result*: *L M1* ∩ *set* (*fst* (*foldl handleIO TG″ XY*)) = *L M2* ∩ *set* (*fst* (*foldl handleIO TG″ XY*))
using *Suc.prems*(*8*)
by *metis*

have *handleIO-subset* : ⋀ *x y T G* . *set T* ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
proof −
fix *x y T G*

define *TGu* **where** *TGu*: *TGu* = *distribute-extension M1 T G cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*
define *TGv* **where** *TGv*: *TGv* = (*if* (*u* = *v*) *then TGu else distribute-extension M1* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*)

have *handleIO*: *handleIO* (*T*,*G*) (*x*,*y*) = (*if is-in-language M1* (*initial M1*) (*u*@[(*x*,*y*)])
then distinguish-from-set M1 V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X′ k depth completeInputTraces append-heuristic* (*u* = *v*)
else TGv)
unfolding *handleIO TGu TGv case-prod-conv Let-def*
by *force*

have *set T* ⊆ *set* (*fst TGu*)
using *distribute-extension-subset*[*of T*]
unfolding *TGu* **by** *metis*
moreover have *set* (*fst TGu*) ⊆ *set* (*fst TGv*)
using *distribute-extension-subset*[*of fst TGu*]
unfolding *TGv* **by** *force*
moreover have *set* (*fst TGv*) ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))

unfolding *handleIO*
using *distinguish-from-set-subset*[*of fst TGv M1 V snd TGv cg-lookup cg-insert get-distinguishing-trace u@*[(*x*,*y*)] *v@*[(*x*,*y*)] *X′ k depth*]
by *auto*
ultimately show *set T* ⊆ *set* (*fst* (*handleIO* (*T*,*G*) (*x*,*y*)))
by *blast*
qed

have *result-subset*: *set* (*fst TG″*) ⊆ *set* (*fst* (*foldl handleIO TG″ XY*))
proof (*induction XY rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case*
using *handleIO-subset*[*of fst* (*foldl handleIO TG″ xs*) *snd* (*foldl handleIO TG″ xs*) *fst x snd x*]
by *force*
qed
then have *pass-TG″* : *L M1* ∩ *set* (*fst TG″*) = *L M2* ∩ *set* (*fst TG″*)
using *pass-result* **by** *blast*

have *set* (*fst TG′*) ⊆ *set* (*fst TG″*)
unfolding *TG″* **using** *spyh-distinguish-subset*
by (*metis* (*mono-tags*, *lifting*) *equalityE*)
then have *pass-TG′*: *L M1* ∩ *set* (*fst TG′*) = *L M2* ∩ *set* (*fst TG′*)
using *pass-TG″* **by** *blast*

have *set T* ⊆ *set* (*fst TG′*)
unfolding *TG′* **using** *spyh-distinguish-subset* **by** *metis*
then have *pass-T*: *L M1* ∩ *set T* = *L M2* ∩ *set T*
using *pass-TG′* **by** *blast*

have *list.set X* ⊆ *L M1* **and** *list.set X* ⊆ *L M2*
using *Suc.prems*(*3*) **by** *auto*

have *preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG′*)
using *spyh-distinguish-preserves-divergence*[*OF assms*(*1*−*4*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› *assms*(*16*) *Suc.prems*(*7*) *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› - -*Suc.prems*(*2*), *of T k completeInputTraces append-heuristic*, *OF* - - - - *assms*(*18*)]
spyh-distinguish-establishes-divergence(*2*)[*OF assms*(*1*−*4*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› *assms*(*16*) *Suc.prems*(*7*) *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*›, *of T k completeInputTraces append-heuristic*, *OF* - - - - *assms*(*18*)]
pass-TG′
unfolding *TG′*[*symmetric*]

by *linarith+*

have *preserves-divergence M1 M2* (*Set.insert v* (*list.set X*))
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)
proof −
have *preserves-divergence M1 M2* (*Set.insert v* (*list.set X*)) ∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)
proof (*cases notReferenced*)
case *True*

then have *TG'' = spyh-distinguish M1* (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*
unfolding *TG''* **by** *auto*
then have ∗: *L M1* ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1* (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*)) =
L M2 ∩ *Prefix-Tree.set* (*fst* (*spyh-distinguish M1* (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*))
using ‹*L M1* ∩ *set* (*fst TG''*) = *L M2* ∩ *set* (*fst TG''*)›
by *simp*

show *?thesis*
using *spyh-distinguish-preserves-divergence*[*OF assms*(*1*−*4*) ‹*v* ∈ *L M1*› ‹*v* ∈ *L M2*› *assms*(*16*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG'*)› *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*) *Suc.prems*(*2*)]
spyh-distinguish-establishes-divergence(*2*)[*OF assms*(*1*−*4*) ‹*v* ∈ *L M1*› ‹*v* ∈ *L M2*› *assms*(*16*) ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG'*)› *assms*(*15*) ‹*list.set X* ⊆ *L M1*› ‹*list.set X* ⊆ *L M2*› ∗ *assms*(*18*)]
unfolding ‹*TG'' = spyh-distinguish M1* (*fst TG'*) (*snd TG'*) *cg-lookup cg-insert get-distinguishing-trace v X k completeInputTraces append-heuristic*›
by *presburger*
next
case *False*
then consider *u = v* | (*u* ≠ *v*) ∧ ¬(∀ *q* ∈ *reachable-states M1* . *V q* ∉ *vClass*)
unfolding *notReferenced* **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis*
using *False TG''* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG'*)› ‹*preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))› **by** *presburger*
next
case *2*
then have *TG'' = TG'*
unfolding *TG''* **using** *False* **by** *auto*

```
obtain q where q ∈ reachable-states M1
          and V q ∈ Set.insert v (list.set (cg-lookup (snd TG′) v))
  using 2
  unfolding notReferenced vClass
  by blast

have converge M1 (V q) v and converge M2 (V q) v
proof −
  have converge M1 v (V q) ∧ converge M2 v (V q)
  proof (cases V q = v)
    case True
    then show ?thesis
      using ‹v ∈ L M1› ‹v ∈ L M2› by auto
  next
    case False
    then have V q ∈ list.set (cg-lookup (snd TG′) v)
      using ‹V q ∈ Set.insert v (list.set (cg-lookup (snd TG′) v))›
      by blast
    then show ?thesis
      using ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG′)›
      unfolding convergence-graph-lookup-invar-def
      using Suc.prems(6) ‹v ∈ L M1› by blast
  qed
  then show converge M1 (V q) v and converge M2 (V q) v
    by auto
qed

have V q ∈ Set.insert u (list.set X)
  using ‹q ∈ reachable-states M1› Suc.prems(1) by blast

have preserves-divergence M1 M2 (Set.insert v (list.set X))
    using preserves-divergence-converge-insert[OF assms(1−4) ‹converge
M1 (V q) v› ‹converge M2 (V q) v› ‹preserves-divergence M1 M2 (Set.insert u
(list.set X))› ‹V q ∈ Set.insert u (list.set X)›]
  unfolding preserves-divergence.simps by blast
then show ?thesis
  unfolding ‹TG′′ = TG′›
  using ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG′)›
  by auto
qed
qed
then show preserves-divergence M1 M2 (Set.insert v (list.set X)) and  con-
vergence-graph-lookup-invar M1 M2 cg-lookup (snd TG′′)
  by auto
qed

have converge M1 u u and converge M1 v v and converge M1 v u and converge
M1 u v
  using ‹u ∈ L M1› ‹v ∈ L M1› ‹converge M1 u v› by auto
```

then have *preserves-divergence M1 M2* (*Set.insert u* (*Set.insert v* (*list.set X*)))
using ‹*preserves-divergence M1 M2* (*Set.insert v* (*list.set X*))›
‹*preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))›
unfolding *preserves-divergence.simps*
by *blast*

have *IS1*: *V* ‘ *reachable-states M1* ⊆ *list.set X′*
using *Suc.prems*(*1*) **unfolding** *X′* **by** *auto*

have *IS2*: *preserves-divergence M1 M2* (*list.set X′*)
using ‹*preserves-divergence M1 M2* (*Set.insert u* (*Set.insert v* (*list.set X*)))›
‹*preserves-divergence M1 M2* (*Set.insert u* (*list.set X*))›
unfolding *X′*
by (*simp add*: *insert-commute*)

have *handleIO-props* : ⋀ *x y T′ G′* . *set T* ⊆ *set T′* ⟹ *convergence-graph-lookup-invar M1 M2 cg-lookup G′* ⟹ *L M1* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*))) ⟹
x ∈ *inputs M1* ⟹ *y* ∈ *outputs M1* ⟹
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*handleIO* (*T′*,*G′*) (*x*,*y*)))
∧ *L M1* ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* [(*x*,*y*)])}) = *L M2* ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* [(*x*,*y*)])})
∧ *preserves-divergence M1 M2* (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* [(*x*,*y*)])})
∧ (∀ *γ x′ y′* . *length* ((*x*,*y*)#*γ*@[(*x′*,*y′*)]) ≤ *Suc depth* ⟶
((*x*,*y*)#*γ*) ∈ *LS M1* (*after-initial M1 u*) ⟶
x′ ∈ *inputs M1* ⟶ *y′* ∈ *outputs M1* ⟶
L M1 ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* ((*x*,*y*)#*γ*@[(*x′*,*y′*)]))}) = *L M2* ∩ (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* ((*x*,*y*)#*γ*@[(*x′*,*y′*)]))})
∧ *preserves-divergence M1 M2* (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* ((*x*,*y*)#*γ*@[(*x′*,*y′*)]))}))
proof −
fix *x y T′ G′*

assume *convergence-graph-lookup-invar M1 M2 cg-lookup G′*
and *L M1* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*)))
and *x* ∈ *inputs M1*
and *y* ∈ *outputs M1*
and *set T* ⊆ *set T′*

define *TGu* **where** *TGu*: *TGu* = *distribute-extension M1 T′ G′ cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*

define *TGv* **where** *TGv*: *TGv* = (*if* (*u*=*v*) *then TGu else distribute-extension M1* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(*x*,*y*)] *completeInputTraces append-heuristic*)

have *handleIO*: *handleIO* (*T*′,*G*′) (*x*,*y*) = (*if is-in-language M1* (*initial M1*) (*u*@[(*x*,*y*)])
then distinguish-from-set M1 V (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)]) *X*′ *k depth completeInputTraces append-heuristic* (*u*=*v*)
else TGv)
unfolding *handleIO TGu TGv case-prod-conv Let-def*
by *force*

have *set T*′ ⊆ *set* (*fst TGu*)
using *distribute-extension-subset*[*of T*′]
unfolding *TGu* **by** *metis*
have *set* (*fst TGu*) ⊆ *set* (*fst TGv*)
using *distribute-extension-subset*[*of fst TGu*]
unfolding *TGv* **by** *force*
have *set* (*fst TGv*) ⊆ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*)))
unfolding *handleIO*
using *distinguish-from-set-subset*[*of fst TGv M1 V snd TGv cg-lookup cg-insert get-distinguishing-trace u*@[(*x*,*y*)] *v*@[(*x*,*y*)] *X*′ *k depth*]
by *auto*
then have *pass-TGv*: *L M1* ∩ *set* (*fst TGv*) = *L M2* ∩ *set* (*fst TGv*)
using ‹*L M1* ∩ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*)))› ‹*set* (*fst TGv*) ⊆ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*)))›
by *blast*

have ∗:*L M1* ∩ *set* (*fst* (*distribute-extension M1 T*′ *G*′ *cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1 T*′ *G*′ *cg-lookup cg-insert u* [(*x*,*y*)] *completeInputTraces append-heuristic*))
using ‹*L M1* ∩ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*)))› ‹*set* (*fst TGv*) ⊆ *set* (*fst* (*handleIO* (*T*′,*G*′) (*x*,*y*)))› ‹*set* (*fst TGu*) ⊆ *set* (*fst TGv*)›
unfolding *TGu*
by *blast*

obtain *u*′ **where** *converge M1 u u*′
u′ @ [(*x*, *y*)] ∈ *set* (*fst TGv*)
converge M2 u u′
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd TGu*)
using *distribute-extension-adds-sequence*[*OF assms*(*1*,*3*) ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G*′› *assms*(*15*) ∗ *assms*(*18*)]
using ‹*set* (*fst TGu*) ⊆ *set* (*fst TGv*)›
unfolding *TGu* **by** *blast*

have *u*′ ∈ *set* (*fst TGv*)

using ‹u' @ [(x, y)] ∈ *set* (*fst TGv*)› *set-prefix* **by** *metis*
have $u' \in L\ M1$
using ‹*converge M1 u u'*› **by** *auto*

have ∗:¬(u=v) ⟹ *L M1* ∩ *set* (*fst* (*distribute-extension M1* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(x,y)] *completeInputTraces append-heuristic*)) = *L M2* ∩ *set* (*fst* (*distribute-extension M1* (*fst TGu*) (*snd TGu*) *cg-lookup cg-insert v* [(x,y)] *completeInputTraces append-heuristic*))
using ‹*L M1* ∩ *set* (*fst* (*handleIO* (*T'*,*G'*) (x,y))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T'*,*G'*) (x,y)))› ‹*set* (*fst TGv*) ⊆ *set* (*fst* (*handleIO* (*T'*,*G'*) (x,y)))›
using *TGv pass-TGv* **by** *presburger*

obtain v' **where** *converge M1 v v'*
v' @ [(x, y)] ∈ *set* (*fst TGv*)
converge M2 v v'
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd TGv*)
u=v ⟹ *u' = v'*
proof (*cases* u=v)
case *True*
then have *TGv = TGu* **unfolding** *TGv* **by** *auto*
show *?thesis*
using *that*
using ‹*converge M1 u u'*› ‹*u'* @ [(x, y)] ∈ *set* (*fst TGv*)› ‹*converge M2 u u'*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TGu*)›
unfolding *True* ‹*TGv = TGu*› **by** *blast*
next
case *False*
then show *?thesis*
using *that*
using *distribute-extension-adds-sequence*[*OF assms*(*1*,*3*) ‹*v* ∈ *L M1*› ‹*v* ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TGu*)› *assms*(*15*) ∗[*OF False*] *assms*(*18*)]
unfolding *TGv* **by** *auto*
qed

have $v' \in$ *set* (*fst TGv*)
using ‹*v'* @ [(x, y)] ∈ *set* (*fst TGv*)› *set-prefix* **by** *metis*
have $v' \in L\ M1$
using ‹*converge M1 v v'*› **by** *auto*

have ∗: {ω@ω' | ω ω' . ω ∈ {u,v} ∧ ω' ∈ *list.set* (*prefixes* [(x,y)])} = {u,v,u@[(x,y)],v@[(x,y)]}
by *auto*

have u ∈ *L M1* = (u ∈ *L M2*)
using *Suc.prems*(*5*) ‹u ∈ *L M1*› **by** *auto*
moreover have v ∈ *L M1* = (v ∈ *L M2*)
using *Suc.prems*(*6*) ‹v ∈ *L M1*› **by** *auto*

moreover have u @ $[(x, y)] \in L\ M1 = (u$ @ $[(x, y)] \in L\ M2)$
proof −
have u @ $[(x, y)] \in L\ M1 = (u'$ @ $[(x, y)] \in L\ M1)$
using ‹*converge M1 u u'*› *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... $= (u'$ @ $[(x, y)] \in L\ M2)$
using *pass-TGv* ‹u' @ $[(x, y)] \in$ *set* (*fst TGv*)› **by** *blast*
also have ... $= (u$ @ $[(x, y)] \in L\ M2)$
using ‹*converge M2 u u'*› *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally show *?thesis* **.**
qed
moreover have v @ $[(x, y)] \in L\ M1 = (v$ @ $[(x, y)] \in L\ M2)$
proof −
have v @ $[(x, y)] \in L\ M1 = (v'$ @ $[(x, y)] \in L\ M1)$
using ‹*converge M1 v v'*› *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... $= (v'$ @ $[(x, y)] \in L\ M2)$
using *pass-TGv* ‹v' @ $[(x, y)] \in$ *set* (*fst TGv*)› **by** *blast*
also have ... $= (v$ @ $[(x, y)] \in L\ M2)$
using ‹*converge M2 v v'*› *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally show *?thesis* **.**
qed
moreover have $L\ M1 \cap$ *list.set* $X = (L\ M2 \cap$ *list.set* $X)$
using *Suc.prems*(*3*)
by *fastforce*
ultimately have *p2*: $L\ M1 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$) $= L\ M2 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$)
unfolding $*$ **by** *blast*

show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*handleIO* (T',G') (x,y)))
$\wedge\ L\ M1 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$) $= L\ M2 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$)
$\wedge$ *preserves-divergence M1 M2* (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$)
$\wedge\ (\forall\ \gamma\ x'\ y'$. *length* $((x,y)\#\gamma@[(x',y')]) \leq$ *Suc depth* $\longrightarrow$
$((x,y)\#\gamma) \in$ *LS M1* (*after-initial M1 u*) $\longrightarrow$
$x' \in$ *inputs M1* $\longrightarrow y' \in$ *outputs M1* $\longrightarrow$
$L\ M1 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x,y)\#\gamma@[(x',y')])$)$\}$) $= L\ M2 \cap$ (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x,y)\#\gamma@[(x',y')])$)$\}$)
$\wedge$ *preserves-divergence M1 M2* (*list.set* $X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x,y)\#\gamma@[(x',y')])$)$\}$))
proof (*cases is-in-language M1* (*initial M1*) $(u@[(x,y)])$)
case *False*

have $u@[(x,y)] \notin L\ M1$
using *False* **by** (*meson assms*(*1*) *fsm-initial is-in-language-iff*)
moreover have $v@[(x,y)] \notin L\ M1$
using *calculation Suc.prems*(*4*) *assms*(*1*) *converge-append-language-iff* **by** *blast*
moreover have *preserves-divergence M1 M2* (*list.set X* $\cup$ $\{u,v\}$)
by (*metis* (*no-types*) *Un-insert-right* ‹*preserves-divergence M1 M2* (*Set.insert u* (*Set.insert v* (*list.set X*)))› *sup-bot-right*)
ultimately have *p3*: *preserves-divergence M1 M2* (*list.set X* $\cup$ $\{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in$ *list.set* (*prefixes* $[(x,y)]$)$\}$)
unfolding $*$ *preserves-divergence.simps*
by *blast*

have *handleIO*: (*handleIO* (T',G') (x,y)) = *TGv*
using *handleIO False* **by** *auto*

have $\bigwedge$ *x xs* . *x* # *xs* = [*x*] @ *xs* **by** *auto*
then have $\bigwedge$ γ . (*x*, *y*) # $\gamma \notin$ *LS M1* (*after-initial M1 u*)
by (*metis* ‹*u* @ [(*x*, *y*)] $\notin$ *L M1*› ‹*u* $\in$ *L M1*› *after-language-iff assms*(*1*) *language-prefix*)

have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*handleIO* (T',G') (x,y)))
unfolding *handleIO*
by (*simp add*: ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TGv*)›)
moreover note *p2 p3* ‹$\bigwedge$ γ . (*x*, *y*) # $\gamma \notin$ *LS M1* (*after-initial M1 u*)›
ultimately show *?thesis*
by *presburger*
next
case *True*
then have *handleIO*: (*handleIO* (T',G') (x,y)) = *distinguish-from-set M1 V* (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u* @ [(*x*, *y*)]) (*v* @ [(*x*, *y*)]) *X′ k depth completeInputTraces append-heuristic* ($u@[(x,y)] = v@[(x,y)]$)
using *handleIO* **by** *auto*

have *converge M1* ($u@[(x,y)]$) ($v@[(x,y)]$)
by (*meson Suc.prems*(*4*) *True* ‹*v* $\in$ *L M1*› *assms*(*1*) *converge-append fsm-initial is-in-language-iff*)
then have ($u@[(x,y)]$) $\in$ *L M1* **and** ($v@[(x,y)]$) $\in$ *L M1*
by *auto*
have ($u@[(x,y)]$) $\in$ *L M2*
by (*meson True* ‹(*u* @ [(*x*, *y*)] $\in$ *L M1*) = (*u* @ [(*x*, *y*)] $\in$ *L M2*)› *assms*(*1*) *fsm-initial is-in-language-iff*)
have ($v@[(x,y)]$) $\in$ *L M2*
using *Suc.prems*(*4*) ‹(*u* @ [(*x*, *y*)] $\in$ *L M1*) = (*u* @ [(*x*, *y*)] $\in$ *L M2*)› ‹(*v* @ [(*x*, *y*)] $\in$ *L M1*) = (*v* @ [(*x*, *y*)] $\in$ *L M2*)› ‹*u* @ [(*x*, *y*)] $\in$ *L M2*› *assms*(*1*)

converge-append-language-iff **by** *blast*
have *preserves-divergence M1 M2* (*list.set X* ∪ {*u*,*v*})
by (*metis* (*no-types*) *Un-insert-right* ‹*preserves-divergence M1 M2* (*Set.insert u* (*Set.insert v* (*list.set X*)))› *sup-bot-right*)

have *IS3*: ⋀*w*. *w* ∈ *list.set X′* ⟹ ∃ *w′*. *converge M1 w w′* ∧ *converge M2 w w′*
unfolding *X′*
by (*metis* (*full-types*) *Suc.prems*(*3*) ‹*converge M1 u u′*› ‹*converge M1 v v′*› ‹*converge M2 u u′*› ‹*converge M2 v v′*› *set-ConsD*)

have (*u*@[(*x*,*y*)] = *v*@[(*x*,*y*)]) = (*u*=*v*)
by *auto*
have *IS4*: *L M1* ∩ *Prefix-Tree.set* (*fst* (*distinguish-from-set M1 V* (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u* @ [(*x*, *y*)]) (*v* @ [(*x*, *y*)]) *X′ k depth completeInputTraces append-heuristic* (*u*@[(*x*,*y*)] = *v*@[(*x*,*y*)]))) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*distinguish-from-set M1 V* (*fst TGv*) (*snd TGv*) *cg-lookup cg-insert get-distinguishing-trace* (*u* @ [(*x*, *y*)]) (*v* @ [(*x*, *y*)]) *X′ k depth completeInputTraces append-heuristic* (*u*@[(*x*,*y*)] = *v*@[(*x*,*y*)])))
using ‹*L M1* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*))) = *L M2* ∩ *set* (*fst* (*handleIO* (*T′*,*G′*) (*x*,*y*)))›
unfolding *handleIO* ‹(*u*@[(*x*,*y*)] = *v*@[(*x*,*y*)]) = (*u*=*v*)›
by *blast*

have *IH1*: ⋀ *γ xa ya*. *length* (*γ* @ [(*xa*, *ya*)]) ≤ *depth* ⟹
γ ∈ *LS M1* (*after-initial M1* (*u* @ [(*x*, *y*)])) ⟹
xa ∈ *FSM.inputs M1* ⟹
ya ∈ *FSM.outputs M1* ⟹
L M1 ∩ (*list.set X′* ∪ {*ω* @ *ω′* |*ω ω′*. *ω* ∈ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*xa*, *ya*)]))}) = *L M2* ∩ (*list.set X′* ∪ {*ω* @ *ω′* |*ω ω′*. *ω* ∈ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*xa*, *ya*)]))}) ∧
preserves-divergence M1 M2 (*list.set X′* ∪ {*ω* @ *ω′* |*ω ω′*. *ω* ∈ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*xa*, *ya*)]))})
and *IH2*: *preserves-divergence M1 M2* (*list.set X′* ∪ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]})
and *IH3*: *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*handleIO* (*T′*, *G′*) (*x*, *y*)))
using *Suc.IH*[*OF IS1 IS2 IS3* ‹*converge M1* (*u*@[(*x*,*y*)]) (*v*@[(*x*,*y*)])› ‹*u*@[(*x*,*y*)] ∈ *L M2*› ‹*v*@[(*x*,*y*)] ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TGv*)› *IS4*]
unfolding *handleIO*[*symmetric*]
by *blast*+

have *p3*: *preserves-divergence M1 M2* (*list.set X* ∪ {*ω*@*ω′* | *ω ω′* . *ω* ∈ {*u*,*v*} ∧ *ω′* ∈ *list.set* (*prefixes* [(*x*,*y*)])})
proof (*cases notReferenced*)

case *True*
then have *list.set X′ = list.set X ∪ {u,v}*
unfolding *X′* **by** *auto*
show *?thesis*
using *IH2*
unfolding ∗ *preserves-divergence.simps* ‹*list.set X′ = list.set X ∪ {u,v}*›
by *blast*
next
case *False*
then consider *u = v* | *(u ≠ v) ∧ ¬(∀ q ∈ reachable-states M1 . V q ∉ vClass)*
unfolding *notReferenced* **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) ∗ *False IH2 Un-insert-left Un-insert-right X′ insertI1 insert-absorb list.simps*(*15*))
next
case *2*

then have ∗∗:(*list.set X ∪ {ω@ω′ | ω ω′ . ω ∈ {u,v} ∧ ω′ ∈ list.set (prefixes [(x,y)])}) = (list.set X′ ∪ {u @ [(x, y)], v @ [(x, y)]}) ∪ {v}*
unfolding ∗ *X′*
by *auto*

obtain *q* **where** *q ∈ reachable-states M1* **and** *V q ∈ vClass*
using *2 notReferenced* **by** *blast*
then have *V q ∈ list.set (cg-lookup (snd TG′) v)*
unfolding *vClass*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG′)*› ‹*v ∈ L M1*› ‹*v ∈ L M2*›
unfolding *convergence-graph-lookup-invar-def* **by** *blast*
then have *converge M1 (V q) v* **and** *converge M2 (V q) v*
using *convergence-graph-lookup-invar-simp*[*OF* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup (snd TG′)*› ‹*v ∈ L M1*› ‹*v ∈ L M2*›, *of V q*]
by *auto*

have ⋀ *β . β ∈ L M1 ∩ (list.set X ∪ {ω@ω′ | ω ω′ . ω ∈ {u,v} ∧ ω′ ∈ list.set (prefixes [(x,y)])}) ⟹ ¬converge M1 v β ⟹ ¬converge M2 v β*
proof −
fix *β* **assume** *β ∈ L M1 ∩ (list.set X ∪ {ω@ω′ | ω ω′ . ω ∈ {u,v} ∧ ω′ ∈ list.set (prefixes [(x,y)])}) and ¬converge M1 v β*
then consider *β = v* | *β ∈ L M1 ∩ (list.set X′ ∪ {u @ [(x, y)], v @ [(x, y)]})*
unfolding ∗∗ **by** *blast*
then show *¬converge M2 v β*
proof *cases*
case *1*
then show *?thesis* **using** ‹*¬converge M1 v β*› ‹*v ∈ L M1*› **by** *auto*

next
case *2*
moreover have $\neg$*converge M1* (*V q*) β
using ‹*converge M1* (*V q*) *v*› ‹$\neg$*converge M1 v* β›
by *auto*
moreover have *V q* $\in$ *list.set X′*
using *Suc.prems*(*1*) ‹*q* $\in$ *reachable-states M1*›
unfolding *X′* **by** *auto*
moreover have *V q* $\in$ *L M1*
using ‹*converge M1* (*V q*) *v*› *converge.simps* **by** *blast*
ultimately have $\neg$*converge M2* (*V q*) β
using *IH2*
unfolding *preserves-divergence.simps*
by *blast*
then show *?thesis*
using ‹*converge M2* (*V q*) *v*› **unfolding** *converge.simps* **by** *force*
qed
qed

have $\bigwedge$ α β . α $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) $\Longrightarrow$ β $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) $\Longrightarrow$ $\neg$*converge M1* α β $\Longrightarrow$ $\neg$*converge M2* α β
proof −
fix α β **assume** α $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])})
β $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])})
$\neg$*converge M1* α β
then consider α = *v* $\wedge$ β $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) |
β = *v* $\wedge$ α $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) |
α $\in$ *L M1* $\cap$ (*list.set X′* $\cup$ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]}) $\wedge$ β $\in$ *L M1* $\cap$ (*list.set X′* $\cup$ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]})
unfolding ∗∗ **by** *auto*
then show $\neg$*converge M2* α β **proof** *cases*
case *1*
then show *?thesis* **using** ‹$\bigwedge$ β . β $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) $\Longrightarrow$ $\neg$*converge M1 v* β $\Longrightarrow$ $\neg$*converge M2 v* β›
using ‹$\neg$ *converge M1* α β› **by** *blast*
next
case *2*
then show *?thesis* **using** ‹$\bigwedge$ β . β $\in$ *L M1* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) $\Longrightarrow$ $\neg$*converge M1 v* β $\Longrightarrow$ $\neg$*converge M2 v* β›[*of* α]
using ‹$\neg$ *converge M1* α β›
unfolding *converge-sym*[*of* - α] **by** *blast*
next

case *3*
then show *?thesis*
using *IH2* ‹$\neg$ *converge M1* α β›
unfolding *preserves-divergence.simps* **by** *blast*
qed
qed
then show *?thesis*
unfolding *preserves-divergence.simps*
by *blast*
qed
qed

have *p4*: ($\bigwedge$ γ x' y'.
length ((*x*, *y*) # γ @ [(x', y')]) $\leq$ *Suc depth* $\Longrightarrow$
(*x*, *y*) # γ $\in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
x' $\in$ *FSM.inputs M1* $\Longrightarrow$
y' $\in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ (*list.set X* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u*, *v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(x', y')]))}) =
L M2 $\cap$ (*list.set X* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u*, *v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(x', y')]))}) $\wedge$
preserves-divergence M1 M2 (*list.set X* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u*, *v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(x', y')]))}))
proof −
fix γ x' y'
assume *length* ((*x*, *y*) # γ @ [(x', y')]) $\leq$ *Suc depth*
(*x*, *y*) # γ $\in$ *LS M1* (*after-initial M1 u*)
x' $\in$ *FSM.inputs M1*
y' $\in$ *FSM.outputs M1*

have *s1*: *length* (γ @ [(x', y')]) $\leq$ *depth*
using ‹*length* ((*x*, *y*) # γ @ [(x', y')]) $\leq$ *Suc depth*› **by** *auto*
have *s2*: γ $\in$ *LS M1* (*after-initial M1* (*u* @ [(*x*, *y*)]))
using ‹(*x*, *y*) # γ $\in$ *LS M1* (*after-initial M1 u*)›
by (*metis* ‹*u* @ [(*x*, *y*)] $\in$ *L M1*› *after-language-append-iff append-Cons assms*(*1*) *empty-append-eq-id*)

have *pass'*: *L M1* $\cap$ (*list.set X'* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(x', y')]))}) = *L M2* $\cap$ (*list.set X'* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(x', y')]))})
and *preserve'*: *preserves-divergence M1 M2* (*list.set X'* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*u* @ [(*x*, *y*)], *v* @ [(*x*, *y*)]} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(x', y')]))})
using *IH1*[*OF s1 s2* ‹x' $\in$ *FSM.inputs M1*› ‹y' $\in$ *FSM.outputs M1*›]
by *blast*+

have ***:{ω @ ω' |ω ω'. ω $\in$ {*u*, *v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(x', y')]))}

$= \{\omega \ @ \ \omega' \mid \omega \ \omega'. \ \omega \in \{u \ @ \ [(x, \ y)], \ v \ @ \ [(x, \ y)]\} \wedge \omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ (\gamma \ @ \ [(x', \ y')]))\} \cup \{u,v\}$
(**is** $?A = ?B$)
proof
show $?A \subseteq ?B$
proof
fix w **assume** $w \in ?A$
then obtain $\omega \ \omega'$ **where** $w = \omega \ @ \ \omega'$ **and** $\omega \in \{u, \ v\}$ **and** $\omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ ((x, \ y) \ \# \ \gamma \ @ \ [(x', \ y')]))$
by *blast*

show $w \in ?B$
proof (*cases* ω')
case *Nil*
then show *?thesis* **unfolding** ‹$w = \omega \ @ \ \omega'$› *prefixes-set* **using** ‹$\omega \in \{u, \ v\}$› **by** *auto*
next
case (*Cons a list*)
then have $a = (x,y)$ **and** $\mathit{list} \in \mathit{list.set}$ $(\mathit{prefixes} \ (\gamma \ @ \ [(x', \ y')]))$
using ‹$\omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ ((x, \ y) \ \# \ \gamma \ @ \ [(x', \ y')]))$›
by (*meson prefixes-Cons*)+
moreover have $\omega@[(x,y)] \in \{u \ @ \ [(x, \ y)], \ v \ @ \ [(x, \ y)]\}$
using ‹$\omega \in \{u, \ v\}$›
by *auto*
ultimately have $((\omega@[(x,y)])@\mathit{list}) \in \{\omega \ @ \ \omega' \mid \omega \ \omega'. \ \omega \in \{u \ @ \ [(x, \ y)], \ v \ @ \ [(x, \ y)]\} \wedge \omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ (\gamma \ @ \ [(x', \ y')]))\}$
by *blast*
then show *?thesis*
unfolding ‹$w = \omega \ @ \ \omega'$› *Cons* ‹$a = (x,y)$›
by *auto*
qed
qed
show $?B \subseteq ?A$
proof
fix w **assume** $w \in ?B$
then consider $w \in \{u,v\} \mid w \in \{\omega \ @ \ \omega' \mid \omega \ \omega'. \ \omega \in \{u \ @ \ [(x, \ y)], \ v \ @ \ [(x, \ y)]\} \wedge \omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ (\gamma \ @ \ [(x', \ y')]))\}$
by *blast*
then show $w \in ?A$ **proof** *cases*
case *1*
then show *?thesis* **using** *prefixes-set-Nil*[*of* $((x, \ y) \ \# \ \gamma \ @ \ [(x', \ y')])$]
using *append.right-neutral* **by** *blast*
next
case *2*
then obtain $\omega \ \omega'$ **where** $w = \omega \ @ \ \omega'$ **and** $\omega \in \{u \ @ \ [(x, \ y)], \ v \ @ \ [(x, \ y)]\}$ **and** $\omega' \in \mathit{list.set}$ $(\mathit{prefixes} \ (\gamma \ @ \ [(x', \ y')]))$
by *blast*

obtain ω'' **where** $\omega = \omega''@[(x,y)]$

using ‹$\omega \in \{u$ @ $[(x, y)], v$ @ $[(x, y)]\}$› **by** *auto*
then have $\omega'' \in \{u,v\}$
using ‹$\omega \in \{u$ @ $[(x, y)], v$ @ $[(x, y)]\}$› **by** *auto*
moreover have $[(x,y)]@\omega' \in$ *list.set* (*prefixes* $((x, y) \# \gamma$ @ $[(x', y')]))$
using *prefixes-prepend*[*OF* ‹$\omega' \in$ *list.set* (*prefixes* (γ @ $[(x', y')]$))›]
by (*metis append-Cons empty-append-eq-id*)
ultimately show $w \in$ *?A*
unfolding ‹$w = \omega$ @ ω'› ‹$\omega = \omega''@[(x,y)]$›
using *append-assoc* **by** *blast*
qed
qed
qed

have *list.set* $X \subseteq$ *list.set* X'
unfolding X' **by** *auto*
then have *pass''*: $L\ M1 \cap$ (*list.set* $X \cup \{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x, y) \# \gamma$ @ $[(x', y')]))\}) = L\ M2 \cap$ (*list.set* $X \cup \{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x, y) \# \gamma$ @ $[(x', y')]))\})$
using *pass'* ‹$u \in L\ M1$› ‹$v \in L\ M1$› ‹$u \in L\ M2$› ‹$v \in L\ M2$›
unfolding $***$
by *blast*

have *preserve''*: *preserves-divergence M1 M2* (*list.set* $X \cup \{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x, y) \# \gamma$ @ $[(x', y')]))\})$
proof (*cases notReferenced*)
case *True*
then have *list.set* $X' =$ *list.set* $X \cup \{u,v\}$
unfolding X' **by** *auto*
show *?thesis*
using *preserve'*
unfolding $***$ *preserves-divergence.simps* ‹*list.set* $X' =$ *list.set* $X \cup \{u,v\}$›
by *blast*
next
case *False*
then consider $u = v \mid (u \neq v) \wedge \neg(\forall\ q \in$ *reachable-states M1* . $V\ q \notin$ *vClass*)
unfolding *notReferenced* **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis*
using $***$ X' *preserve'* **by** *fastforce*
next
case *2*

then have $**$:(*list.set* $X \cup \{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* $((x, y) \# \gamma$ @ $[(x', y')]))\}) =$ (*list.set* $X' \cup \{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u$ @ $[(x, y)], v$ @ $[(x, y)]\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ $[(x', y')]$))$\}) \cup \{v\}$

unfolding $***$ X' **by** *auto*

obtain q **where** $q \in$ *reachable-states M1* **and** $V\ q \in$ *vClass*
using *2 notReferenced* **by** *blast*
then have $V\ q \in$ *list.set* (*cg-lookup* (*snd TG'*) v)
unfolding *vClass*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG'*)› ‹$v \in L\ M1$› ‹$v \in L\ M2$›
unfolding *convergence-graph-lookup-invar-def* **by** *blast*
then have *converge M1* ($V\ q$) v **and** *converge M2* ($V\ q$) v
using *convergence-graph-lookup-invar-simp*[*OF* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG'*)› ‹$v \in L\ M1$› ‹$v \in L\ M2$›, *of* $V\ q$]
by *auto*

have $\bigwedge \beta\ .\ \beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega'\ |\omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) \Longrightarrow \neg converge\ M1\ v\ \beta \Longrightarrow \neg converge\ M2\ v\ \beta$
proof −
fix β **assume** $\beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega'\ |\omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\})$ **and** $\neg converge\ M1\ v\ \beta$
then consider $\beta = v \mid \beta \in L\ M1 \cap (list.set\ X' \cup \{\omega\ @\ \omega'\ |\omega\ \omega'.\ \omega \in \{u\ @\ [(x,\ y)],\ v\ @\ [(x,\ y)]\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x',\ y')]))\})$
unfolding $**$ **by** *blast*
then show $\neg converge\ M2\ v\ \beta$
proof *cases*
case *1*
then show *?thesis* **using** ‹$\neg converge\ M1\ v\ \beta$› ‹$v \in L\ M1$› **by** *auto*
next
case *2*
moreover have $\neg converge\ M1\ (V\ q)\ \beta$
using ‹*converge M1* ($V\ q$) v› ‹$\neg converge\ M1\ v\ \beta$›
by *auto*
moreover have $V\ q \in list.set\ X'$
using *Suc.prems*(*1*) ‹$q \in$ *reachable-states M1*›
unfolding X' **by** *auto*
moreover have $V\ q \in L\ M1$
using ‹*converge M1* ($V\ q$) v› *converge.simps* **by** *blast*
ultimately have $\neg converge\ M2\ (V\ q)\ \beta$
using *preserve'*
unfolding *preserves-divergence.simps*
by *blast*
then show *?thesis*
using ‹*converge M2* ($V\ q$) v› **unfolding** *converge.simps* **by** *force*
qed
qed

have $\bigwedge \alpha\ \beta\ .\ \alpha \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega'\ |\omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) \Longrightarrow \beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega'\ |\omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) \Longrightarrow$

$\neg$*converge M1* α β $\Longrightarrow$ $\neg$*converge M2* α β
proof −
fix α β **assume** $\alpha \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\})$
$\beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\})$
$\neg$*converge M1* α β
then consider $\alpha = v \wedge \beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\})$ |
$\beta = v \wedge \alpha \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\})$ |
$\alpha \in L\ M1 \cap (list.set\ X' \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u\ @\ [(x, y)], v\ @\ [(x, y)]\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x', y')]))\}) \wedge \beta \in L\ M1 \cap (list.set\ X' \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u\ @\ [(x, y)], v\ @\ [(x, y)]\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x', y')]))\})$
unfolding $**$ **by** *auto*
then show $\neg$*converge M2* α β **proof** *cases*
case *1*
then show *?thesis* **using** ‹$\bigwedge \beta\ .\ \beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\}) \Longrightarrow \neg converge\ M1\ v\ \beta \Longrightarrow \neg converge\ M2\ v\ \beta$›
using ‹$\neg$ *converge M1* α β› **by** *blast*
next
case *2*
then show *?thesis* **using** ‹$\bigwedge \beta\ .\ \beta \in L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\}) \Longrightarrow \neg converge\ M1\ v\ \beta \Longrightarrow \neg converge\ M2\ v\ \beta$›[*of* α]
using ‹$\neg$ *converge M1* α β›
unfolding *converge-sym*[*of* - α] **by** *blast*
next
case *3*
then show *?thesis*
using *preserve′* ‹$\neg$ *converge M1* α β›
unfolding *preserves-divergence.simps* **by** *blast*
qed
qed
then show *?thesis*
unfolding *preserves-divergence.simps*
by *blast*
qed
qed

show $L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\}) =$
$L\ M2 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\}) \wedge$
preserves-divergence M1 M2 $(list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u, v\} \wedge \omega' \in list.set\ (prefixes\ ((x, y)\ \#\ \gamma\ @\ [(x', y')]))\})$
using *pass″ preserve″*

by *presburger*
qed

show *?thesis*
using *IH3 p2 p3 p4*
by *blast*
qed
qed

have *foldl-handleIO-subset*: $\bigwedge$ *XY T G* . *set T* $\subseteq$ *set* (*fst* (*foldl handleIO* (*T*,*G*) *XY*))
proof −
fix *XY T G*
show *set T* $\subseteq$ *set* (*fst* (*foldl handleIO* (*T*,*G*) *XY*))
proof (*induction XY rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case*
using *handleIO-subset*[*of fst* (*foldl handleIO* (*T*, *G*) *xs*) *snd* (*foldl handleIO* (*T*, *G*) *xs*) *fst x snd x*]
by *force*
qed
qed

have *list.set XY* = *inputs M1* $\times$ *outputs M1*
unfolding *XY*
by (*metis inputs-as-list-set outputs-as-list-set set-product*)
then have *list.set XY* $\subseteq$ *inputs M1* $\times$ *outputs M1*
by *auto*
moreover have *L M1* $\cap$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*)) = *L M2* $\cap$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))
using *pass-result* **by** *auto*
ultimately have *foldl-handleIO-props*: *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))
$\wedge$ ($\forall$ *x y* . (*x*,*y*) $\in$ *list.set XY* $\longrightarrow$
L M1 $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])}) = *L M2* $\cap$ (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])})
$\wedge$ *preserves-divergence M1 M2* (*list.set X* $\cup$ {ω@ω' | ω ω' . ω $\in$ {*u*,*v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* [(*x*,*y*)])})
$\wedge$ ($\forall$ γ *x' y'* . *length* ((*x*,*y*)#γ@[(*x'*,*y'*)]) $\leq$ *Suc depth* $\longrightarrow$
((*x*,*y*)#γ) $\in$ *LS M1* (*after-initial M1 u*) $\longrightarrow$
x' $\in$ *inputs M1* $\longrightarrow$ *y'* $\in$ *outputs M1* $\longrightarrow$

$L\ M1 \cap (list.set\ X \cup \{\omega@\omega' \mid \omega\ \omega' .$
$\omega \in \{u,v\} \wedge \omega' \in list.set\ (prefixes\ ((x,y)\#\gamma@[(x',y')]))\}) = L\ M2 \cap (list.set\ X \cup$
$\{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in list.set\ (prefixes\ ((x,y)\#\gamma@[(x',y')]))\})$
$\wedge\ preserves\text{-}divergence\ M1\ M2\ (list.set$
$X \cup \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{u,v\} \wedge \omega' \in list.set\ (prefixes\ ((x,y)\#\gamma@[(x',y')]))\}))$

proof (*induction XY rule*: *rev-induct*)
case *Nil*

have $*$:(*foldl handleIO* (*fst TG''*, *snd TG''*) []) = (*fst TG''*, *snd TG''*)
by *auto*

show *?case*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd TG''*)›
unfolding $*$ *snd-conv*
by *auto*
next
case (*snoc a XY*)
obtain x' y' **where** $a = (x',y')$
using *prod.exhaust* **by** *metis*
then have $x' \in$ *inputs M1* **and** $y' \in$ *outputs M1*
using *snoc.prems*(*1*) **by** *auto*

have *set T* $\subseteq$ *set* (*fst TG''*)
using ‹*Prefix-Tree.set* (*fst TG'*) $\subseteq$ *Prefix-Tree.set* (*fst TG''*)› ‹*Prefix-Tree.set T* $\subseteq$ *Prefix-Tree.set* (*fst TG'*)› **by** *auto*

have (*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])) = *handleIO* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*) (x',y')
unfolding ‹$a = (x',y')$› **by** *auto*
then have *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*)) $\subseteq$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])))
using *handleIO-subset*
by (*metis prod.collapse*)
then have *pass-XY*: *L M1* $\cap$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*)) = *L M2* $\cap$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))
using *snoc.prems*(*2*) **by** *blast*
have *set T* $\subseteq$ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))
using *foldl-handleIO-subset* ‹*set T* $\subseteq$ *set* (*fst TG''*)›
by *blast*

have *list.set XY* $\subseteq$ *FSM.inputs M1* $\times$ *FSM.outputs M1*
using *snoc.prems*(*1*) **by** *auto*
have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))
using *snoc.IH*[*OF* ‹*list.set XY* $\subseteq$ *FSM.inputs M1* $\times$ *FSM.outputs M1*› *pass-XY*] **by** *blast*
have *pass-aXY*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*handleIO* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*), *snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*)) (x',y'))) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*handleIO* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*)

XY), *snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*)) (*x'*,*y'*)))
using *snoc.prems*(*2*)
unfolding ‹(*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])) = *handleIO* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*) (*x'*,*y'*)›
unfolding *prod.collapse* **.**

show *?case* (**is** *?P1* ∧ *?P2*)
proof
show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])))
using *handleIO-props*[*OF* ‹*set T* ⊆ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))› *pass-aXY* ‹*x'* ∈ *inputs M1*› ‹*y'* ∈ *outputs M1*›]
unfolding ‹(*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])) = *handleIO* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*) (*x'*,*y'*)›
unfolding *prod.collapse*
by *blast*

have ⋀ *x y*. (*x*, *y*) ∈ *list.set* (*XY*@[*a*]) ⟹
L M1 ∩ (*list.set X* ∪ {ω@ω' | ω ω' . ω ∈ {*u*,*v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*,*y*)])}) = *L M2* ∩ (*list.set X* ∪ {ω@ω' | ω ω' . ω ∈ {*u*,*v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*,*y*)])}) ∧
preserves-divergence M1 M2 (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*, *y*)])}) ∧
(∀γ *x' y'*.
length ((*x*, *y*) # γ @ [(*x'*, *y'*)]) ≤ *Suc depth* ⟶
(*x*, *y*) # γ ∈ *LS M1* (*after-initial M1 u*) ⟶
x' ∈ *FSM.inputs M1* ⟶
y' ∈ *FSM.outputs M1* ⟶
L M1 ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(*x'*, *y'*)]))}) =
L M2 ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(*x'*, *y'*)]))}) ∧
preserves-divergence M1 M2 (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* ((*x*, *y*) # γ @ [(*x'*, *y'*)]))}))
proof −
fix *x y* **assume** (*x*, *y*) ∈ *list.set* (*XY*@[*a*])

show *L M1* ∩ (*list.set X* ∪ {ω@ω' | ω ω' . ω ∈ {*u*,*v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*,*y*)])}) = *L M2* ∩ (*list.set X* ∪ {ω@ω' | ω ω' . ω ∈ {*u*,*v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*,*y*)])}) ∧
preserves-divergence M1 M2 (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* [(*x*, *y*)])}) ∧
(∀γ *x' y'*.
length ((*x*, *y*) # γ @ [(*x'*, *y'*)]) ≤ *Suc depth* ⟶
(*x*, *y*) # γ ∈ *LS M1* (*after-initial M1 u*) ⟶
x' ∈ *FSM.inputs M1* ⟶
y' ∈ *FSM.outputs M1* ⟶

$L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) =$
$L\ M2 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) \wedge$
$preserves\text{-}divergence\ M1\ M2\ (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}))$

proof (*cases* $a = (x,y)$)
case *True*
then have $*$:$(x',y') = (x,y)$
using ‹$a = (x',y')$› **by** *auto*

show *?thesis*
using *handleIO-props*[*OF* ‹*set T* ⊆ *set* (*fst* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*))› *pass-aXY* ‹$x' \in$ *inputs M1*› ‹$y' \in$ *outputs M1*›]
unfolding ‹(*foldl handleIO* (*fst TG''*, *snd TG''*) (*XY*@[*a*])) = *handleIO* (*foldl handleIO* (*fst TG''*, *snd TG''*) *XY*) (x',y')›
unfolding *prod.collapse* $*$
by *presburger*
next
case *False*
then have $(x,y) \in$ *list.set XY*
using ‹$(x,\ y) \in$ *list.set* (*XY*@[*a*])› **by** *auto*

then show *?thesis*
using *snoc.IH*[*OF* ‹*list.set XY* ⊆ *FSM.inputs M1* × *FSM.outputs M1*› *pass-XY*]
by *presburger*
qed
qed
then show *?P2*
by *blast*
qed
qed

have $\bigwedge x\ y\ .\ (x,y) \in list.set\ XY = (x \in inputs\ M1 \wedge y \in outputs\ M1)$
unfolding ‹*list.set XY* = *inputs M1* × *outputs M1*› **by** *auto*

have *result-props-1*: $\bigwedge x\ y\ \gamma\ x'\ y'.\ x \in inputs\ M1 \Longrightarrow y \in outputs\ M1 \Longrightarrow$
$length\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]) \leq Suc\ depth \Longrightarrow$
$(x,\ y)\ \#\ \gamma \in LS\ M1\ (after\text{-}initial\ M1\ u) \Longrightarrow$
$x' \in FSM.inputs\ M1 \Longrightarrow$
$y' \in FSM.outputs\ M1 \Longrightarrow$
$L\ M1 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) =$
$L\ M2 \cap (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\}) \wedge$
$preserves\text{-}divergence\ M1\ M2\ (list.set\ X \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ ((x,\ y)\ \#\ \gamma\ @\ [(x',\ y')]))\})$

using *foldl-handleIO-props*
unfolding ‹$\bigwedge$ *x y* . (*x*,*y*) ∈ *list.set XY* = (*x* ∈ *inputs M1* ∧ *y* ∈ *outputs M1*)›
by *blast*

have *?P1a X u v* (*Suc depth*)
proof −
have $\bigwedge$ γ *x y*.
length (γ @ [(*x*, *y*)]) ≤ *Suc depth* ⟹
γ ∈ *LS M1* (*after-initial M1 u*) ⟹
x ∈ *FSM.inputs M1* ⟹
y ∈ *FSM.outputs M1* ⟹
L M1 ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) ∧
preserves-divergence M1 M2 (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
proof −
fix γ *x y*
assume *length* (γ @ [(*x*, *y*)]) ≤ *Suc depth*
γ ∈ *LS M1* (*after-initial M1 u*)
x ∈ *FSM.inputs M1*
y ∈ *FSM.outputs M1*

show *L M1* ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 ∩ (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) ∧
preserves-divergence M1 M2 (*list.set X* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
proof (*cases* γ)
case *Nil*
then have ∗:γ @ [(*x*,*y*)] = [(*x*,*y*)]
by *auto*
have (*x*,*y*) ∈ *list.set XY*
unfolding ‹*list.set XY* = *inputs M1* × *outputs M1*›
using ‹*x* ∈ *FSM.inputs M1*› ‹*y* ∈ *FSM.outputs M1*›
by *auto*
show *?thesis*
unfolding ∗
using *foldl-handleIO-props* ‹(*x*,*y*) ∈ *list.set XY*›
by *presburger*
next
case (*Cons a* γ')
obtain x' y' **where** *a* = (x',y')
using *prod.exhaust* **by** *metis*
then have ∗: γ = (x',y')#γ'
unfolding *Cons* **by** *auto*

```
        then have **: γ @ [(x, y)] = (x',y')#γ'@ [(x, y)]
          by auto

        have ‹x' ∈ inputs M1› ‹y' ∈ outputs M1›
          using language-io[OF ‹γ ∈ LS M1 (after-initial M1 u)›, of x' y']
          unfolding *
          by auto
        have length ((x', y') # (γ' @ [(x, y)])) ≤ Suc depth
          using ‹length (γ @ [(x, y)]) ≤ Suc depth› unfolding * by auto
        have (x', y') # γ' ∈ LS M1 (after-initial M1 u)
          using ‹γ ∈ LS M1 (after-initial M1 u)› unfolding * .

        show ?thesis
           using result-props-1[OF ‹x' ∈ inputs M1› ‹y' ∈ outputs M1› ‹length
((x', y') # (γ' @ [(x, y)])) ≤ Suc depth› ‹(x', y') # γ' ∈ LS M1 (after-initial M1
u)› ‹x ∈ FSM.inputs M1› ‹y ∈ outputs M1›]
           unfolding ** .
      qed
    qed
    then show ?thesis by blast
  qed

  moreover have ?P1b X u v
    using ‹preserves-divergence M1 M2 (Set.insert u (Set.insert v (list.set X)))›
by auto

  moreover have ?P2 T G u v X (Suc depth)
    using foldl-handleIO-props
    unfolding result prod.collapse
    by blast

  ultimately show ?case
    by blast
 qed

 then show ?P1a X u v depth and ?P1b X u v and ?P2 T G u v X depth
   by presburger+
qed


lemma distinguish-from-set-establishes-convergence :
 assumes observable M1
    and observable M2
    and minimal M1
    and minimal M2
    and size-r M1 ≤ m
    and size M2 ≤ m
    and inputs M2 = inputs M1
    and outputs M2 = outputs M1
```

and *is-state-cover-assignment M1 V*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *L M1* ∩ (*V* ‘ *reachable-states M1*) = *L M2* ∩ *V* ‘ *reachable-states M1*
and *converge M1 u v*
and *u* ∈ *L M2*
and *v* ∈ *L M2*
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M1 q1 q2* (*get-distinguishing-trace q1 q2*)
and *L M1* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*))) = *L M2* ∩ *set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
and ⋀ *T w u′ uBest lBest* . *fst* (*append-heuristic T w* (*uBest*,*lBest*) *u′*) ∈ {*u′*,*uBest*}
shows *converge M2 u v*
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
proof −
have *d1*: *V* ‘ *reachable-states M1* ⊆ *list.set* (*map V* (*reachable-states-as-list M1*))
using *reachable-states-as-list-set* **by** *auto*
have *d2*: *preserves-divergence M1 M2* (*list.set* (*map V* (*reachable-states-as-list M1*)))
using *assms*(*10*) *reachable-states-as-list-set*
by (*metis image-set*)
have *d3*: (⋀*w*. *w* ∈ *list.set* (*map V* (*reachable-states-as-list M1*)) ⟹ ∃ *w′*. *converge M1 w w′* ∧ *converge M2 w w′*)
proof −
fix *w* **assume** *w* ∈ *list.set* (*map V* (*reachable-states-as-list M1*))
then have *w* ∈ *V* ‘ *reachable-states M1*
using *reachable-states-as-list-set* **by** *auto*
moreover have *w* ∈ *L M1*
by (*metis assms*(*1*) *assms*(*9*) *calculation image-iff state-cover-assignment-after*(*1*))
ultimately have *w* ∈ *L M2*
using *assms*(*11*) **by** *blast*

have *converge M1 w w*
using ‹*w* ∈ *L M1*› **by** *auto*
moreover have *converge M2 w w*
using ‹*w* ∈ *L M2*› **by** *auto*
ultimately show ∃ *w′*. *converge M1 w w′* ∧ *converge M2 w w′*
by *blast*
qed

have *list.set* (*map V* (*reachable-states-as-list M1*)) = *V* ‘ *reachable-states M1*
using *reachable-states-as-list-set* **by** *auto*

have *prop1*: $\bigwedge\gamma\ x\ y.$
length $(\gamma$ @ $[(x,\ y)]) \leq (m -$ *size-r M1*$) \Longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
$x \in$ *FSM.inputs M1* $\Longrightarrow$
$y \in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ $\{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in$ *list.set* (*prefixes* $(\gamma$ @ $[(x,\ y)]))\}) =$
L M2 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ $\{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in$ *list.set* (*prefixes* $(\gamma$ @ $[(x,\ y)]))\}) \wedge$
preserves-divergence M1 M2
(*V* ‘ *reachable-states M1* $\cup$ $\{\omega$ @ $\omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in$ *list.set* (*prefixes* $(\gamma$ @ $[(x,\ y)]))\}$)
and *prop2*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ $\{u,\ v\}$)
and *prop3*: *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
using *distinguish-from-set-properties*[*OF assms*(*1*−*4*,*7*,*8*,*9*) *d1 d2 d3 assms*(*12*−*19*)]
unfolding ‹*list.set* (*map V* (*reachable-states-as-list M1*)) = *V* ‘ *reachable-states M1*›
by *presburger*+
then show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distinguish-from-set M1 V T G cg-lookup cg-insert get-distinguishing-trace u v* (*map V* (*reachable-states-as-list M1*)) *k* (*m* − *size-r M1*) *completeInputTraces append-heuristic* (*u*=*v*)))
by *presburger*

show *converge M2 u v*
using *establish-convergence-from-pass*[*OF assms*(*1*−*9*,*11*−*14*) *prop1 prop2*]
by *blast*
qed

definition *establish-convergence-dynamic* :: *bool* ⇒ *bool* ⇒ (*'a* ⇒ *'a* ⇒ (*'b* × *'c*) *list*) ⇒
(*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒
(*'a*,*'b*,*'c*) *state-cover-assignment* ⇒
(*'b*×*'c*) *prefix-tree* ⇒
'd ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ *'d*) ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list list*) ⇒
nat ⇒
(*'a*,*'b*,*'c*) *transition* ⇒
((*'b*×*'c*) *prefix-tree* × *'d*) **where**
establish-convergence-dynamic completeInputTraces useInputHeuristic dist-fun M1 V T G cg-insert cg-lookup m t =
distinguish-from-set M1 V T G cg-lookup cg-insert
dist-fun
((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])

(V (t-target t))
(map V (reachable-states-as-list M1))
(2 ∗ size M1)
(m − size-r M1)
completeInputTraces
(if useInputHeuristic then append-heuristic-input M1 else append-heuristic-io)
False

lemma *establish-convergence-dynamic-verifies-transition* :
assumes ⋀ q1 q2 . q1 ∈ states M1 ⟹ q2 ∈ states M1 ⟹ q1 ≠ q2 ⟹ distinguishes M1 q1 q2 (dist-fun q1 q2)
shows verifies-transition (establish-convergence-dynamic b c dist-fun) M1 M2 V T0 cg-insert cg-lookup
proof −
have ∗:⋀ (M1::('a::linorder,'b::linorder,'c::linorder) fsm) V T (G::'d) cg-insert cg-lookup m t. Prefix-Tree.set T ⊆ Prefix-Tree.set (fst (establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t))
using distinguish-from-set-subset **unfolding** establish-convergence-dynamic-def
by metis

have ∗∗∗:⋀ (M1::('a::linorder,'b::linorder,'c::linorder) fsm) V T (G::'d) cg-insert cg-lookup m t. finite-tree T ⟶ finite-tree (fst (establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t))
using distinguish-from-set-finite **unfolding** establish-convergence-dynamic-def
by metis

have ∗∗:⋀ V T (G::'d) cg-insert cg-lookup m t.
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ m ⟹
FSM.size M2 ≤ m ⟹
FSM.inputs M2 = FSM.inputs M1 ⟹
FSM.outputs M2 = FSM.outputs M1 ⟹
is-state-cover-assignment M1 V ⟹
preserves-divergence M1 M2 (V ‘ reachable-states M1) ⟹
V ‘ reachable-states M1 ⊆ set T ⟹
t ∈ FSM.transitions M1 ⟹
t-source t ∈ reachable-states M1 ⟹
((V (t-source t)) @ [(t-input t,t-output t)]) ≠ (V (t-target t)) ⟹
V (t-source t) @ [(t-input t, t-output t)] ∈ L M2 ⟹
convergence-graph-lookup-invar M1 M2 cg-lookup G ⟹
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert ⟹
L M1 ∩ Prefix-Tree.set (fst (establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t)) =
L M2 ∩ Prefix-Tree.set (fst (establish-convergence-dynamic b c dist-fun M1

V T G cg-insert cg-lookup m t)) $\Longrightarrow$
converge M2 (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)) $\wedge$
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t*))
proof −

fix *G* :: *'d*
fix *V T cg-insert cg-lookup m t*
assume *a01*: *observable M1*
assume *a02*: *observable M2*
assume *a03*: *minimal M1*
assume *a04*: *minimal M2*
assume *a05*: *size-r M1* $\leq$ *m*
assume *a06*: *FSM.size M2* $\leq$ *m*
assume *a07*: *FSM.inputs M2* = *FSM.inputs M1*
assume *a08*: *FSM.outputs M2* = *FSM.outputs M1*
assume *a09*: *is-state-cover-assignment M1 V*
assume *a10*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
assume *a11*: *V* ‘ *reachable-states M1* $\subseteq$ *set T*
assume *a12*: *t* $\in$ *FSM.transitions M1*
assume *a13*: *t-source t* $\in$ *reachable-states M1*
assume *a14*: *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*
assume *a15*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a16*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
assume *a17*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t*)) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t*))
assume *a18*: ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) $\neq$ (*V* (*t-target t*))

let *?heuristic* = (*if c then append-heuristic-input M1 else append-heuristic-io*)

have *d2*: *converge M1* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
using *state-cover-transition-converges*[*OF a01 a09 a12 a13*] **.**

have *d1*: *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
using *a11 a17* $*$[*of T M1 V G cg-insert cg-lookup m t*]
by *blast*

then have *d3*: *V* (*t-target t*) $\in$ *L M2*
using *a11 is-state-cover-assignment-language*[*OF a09*, *of t-target t*] *reachable-states-next*[*OF a13 a12*] **by** *auto*

have *d5*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert dist-fun* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)) (*map V* (*reachable-states-as-list M1*)) (*2* $*$ *size M1*) (*m* − *size-r M1*) *b ?heuristic* (((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) = (*V* (*t-target t*))))) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*distinguish-from-set M1 V T G cg-lookup cg-insert dist-fun* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)) (*map V* (*reachable-states-as-list*

M1)) (*2* ∗ *size M1*) (*m* − *size-r M1*) *b ?heuristic* (((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]) = (*V* (*t-target t*)))))
using *a17 a18* **unfolding** *establish-convergence-dynamic-def* **by** *force*

have *d6*: (⋀*T w u′ uBest lBest*. *fst* (*?heuristic T w* (*uBest*, *lBest*) *u′*) ∈ {*u′*, *uBest*})
using *append-heuristic-input-in*[*of M1*] *append-heuristic-io-in*
by *fastforce*

show *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)) ∧
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*establish-convergence-dynamic b c dist-fun M1 V T G cg-insert cg-lookup m t*))
using *distinguish-from-set-establishes-convergence*[*OF a01 a02 a03 a04 a05 a06 a07 a08 a09 a10 d1 d2 a14 d3 a15 a16 assms d5 d6*] *a18*
unfolding *establish-convergence-dynamic-def* **by** *force*
qed

show *?thesis*
unfolding *verifies-transition-def*
using ∗ ∗∗∗ ∗∗ **by** *presburger*
qed

definition *handleUT-dynamic* :: *bool* ⇒
bool ⇒
(*′a* ⇒ *′a* ⇒ (*′b* × *′c*) *list*) ⇒
((*′a*,*′b*,*′c*) *fsm* ⇒ (*′a*,*′b*,*′c*) *state-cover-assignment* ⇒ (*′a*,*′b*,*′c*) *transition* ⇒ (*′a*,*′b*,*′c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒
(*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒
(*′a*,*′b*,*′c*) *state-cover-assignment* ⇒
(*′b*×*′c*) *prefix-tree* ⇒
′d ⇒
(*′d* ⇒ (*′b*×*′c*) *list* ⇒ *′d*) ⇒
(*′d* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list list*) ⇒
(*′d* ⇒ (*′b*×*′c*) *list* ⇒ (*′b*×*′c*) *list* ⇒ *′d*) ⇒
nat ⇒
(*′a*,*′b*,*′c*) *transition* ⇒
(*′a*,*′b*,*′c*) *transition list* ⇒
((*′a*,*′b*,*′c*) *transition list* × (*′b*×*′c*) *prefix-tree* × *′d*)
where
handleUT-dynamic complete-input-traces
use-input-heuristic
dist-fun
do-establish-convergence
M
V
T
G

```
                cg-insert
                cg-lookup
                cg-merge
                m
                t
                X
=
(let k         = (2 * size M);
     l         = (m - size-r M);
     heuristic = (if use-input-heuristic then append-heuristic-input M
                                          else append-heuristic-io);
     rstates   = (map V (reachable-states-as-list M));
     (T1,G1)   = handle-io-pair complete-input-traces
                                use-input-heuristic
                                M
                                V
                                T
                                G
                                cg-insert
                                cg-lookup
                                (t-source t)
                                (t-input t)
                                (t-output t);
     u         = ((V (t-source t))@[(t-input t, t-output t)]);
     v         = (V (t-target t));
     X'        = butlast X
  in if (do-establish-convergence M V t X' l)
      then let (T2,G2) = distinguish-from-set M
                                              V
                                              T1
                                              G1
                                              cg-lookup
                                              cg-insert
                                              dist-fun
                                              u
                                              v
                                              rstates
                                              k
                                              l
                                              complete-input-traces
                                              heuristic
                                              False;
             G3 = cg-merge G2 u v
           in
             (X',T2,G3)
      else (X',distinguish-from-set M
                                    V
                                    T1
                                    G1
```

cg-lookup
cg-insert
dist-fun
u
u
rstates
k
l
complete-input-traces
heuristic
True))

lemma *handleUT-dynamic-handles-transition* :
fixes *M1*::('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2*::('*e*,'*b*,'*c*) *fsm*
assumes $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M1 q1 q2* (*dist-fun q1 q2*)
shows *handles-transition* (*handleUT-dynamic b c dist-fun d*) *M1 M2 V T0 cg-insert cg-lookup cg-merge*
proof −

have $\bigwedge$ *T G m t X* .
Prefix-Tree.set T $\subseteq$ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-dynamic b c dist-fun d M1 V T G cg-insert cg-lookup cg-merge m t X*))) $\wedge$
(*finite-tree T* $\longrightarrow$ *finite-tree* (*fst* (*snd* (*handleUT-dynamic b c dist-fun d M1 V T G cg-insert cg-lookup cg-merge m t X*)))) $\wedge$
(*observable M1* $\longrightarrow$
observable M2 $\longrightarrow$
minimal M1 $\longrightarrow$
minimal M2 $\longrightarrow$
size-r M1 $\leq$ *m* $\longrightarrow$
FSM.size M2 $\leq$ *m* $\longrightarrow$
FSM.inputs M2 = *FSM.inputs M1* $\longrightarrow$
FSM.outputs M2 = *FSM.outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\longrightarrow$
V ‘ *reachable-states M1* $\subseteq$ *Prefix-Tree.set T* $\longrightarrow$
t $\in$ *FSM.transitions M1* $\longrightarrow$
t-source t $\in$ *reachable-states M1* $\longrightarrow$
V (*t-source t*) @ [(*t-input t*, *t-output t*)] $\neq$ *V* (*t-target t*) $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge $\longrightarrow$
L M1 $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-dynamic b c dist-fun d M1 V T G cg-insert cg-lookup cg-merge m t X*))) =
L M2 $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-dynamic b c dist-fun d M1 V T G cg-insert cg-lookup cg-merge m t X*))) $\longrightarrow$
Prefix-Tree.set T0 $\subseteq$ *Prefix-Tree.set T* $\longrightarrow$

$(\forall \gamma.\ length\ \gamma \leq m - size\text{-}r\ M1 \wedge list.set\ \gamma \subseteq FSM.inputs\ M1 \times FSM.outputs\ M1 \wedge butlast\ \gamma \in LS\ M1\ (t\text{-}target\ t) \longrightarrow$
$L\ M1 \cap (V\ `\ reachable\text{-}states\ M1 \cup \{(V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)])\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}) =$
$L\ M2 \cap (V\ `\ reachable\text{-}states\ M1 \cup \{(V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)])\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}) \wedge$
$preserves\text{-}divergence\ M1\ M2\ (V\ `\ reachable\text{-}states\ M1 \cup \{(V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)])\ @\ \omega'\ |\omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\})) \wedge$
$convergence\text{-}graph\text{-}lookup\text{-}invar\ M1\ M2\ cg\text{-}lookup\ (snd\ (snd\ (handleUT\text{-}dynamic\ b\ c\ dist\text{-}fun\ d\ M1\ V\ T\ G\ cg\text{-}insert\ cg\text{-}lookup\ cg\text{-}merge\ m\ t\ X))))$
(**is** $\bigwedge\ T\ G\ m\ t\ X\ .\ ?P\ T\ G\ m\ t\ X$)
proof −

fix $T :: ('b \times 'c)\ prefix\text{-}tree$
fix $G :: 'd$
fix $m :: nat$
fix $t :: ('a,'b,'c)\ transition$
fix $X :: ('a,'b,'c)\ transition\ list$

let $?TG = snd\ (handleUT\text{-}dynamic\ b\ c\ dist\text{-}fun\ d\ M1\ V\ T\ G\ cg\text{-}insert\ cg\text{-}lookup\ cg\text{-}merge\ m\ t\ X)$

define k **where** $k = (2 * size\ M1)$
define l **where** $l = (m - size\text{-}r\ M1)$
define X' **where** $X' = butlast\ X$
define $heuristic$ **where** $heuristic = (if\ c\ then\ append\text{-}heuristic\text{-}input\ M1\ else\ append\text{-}heuristic\text{-}io)$
define $rstates$ **where** $rstates = (map\ V\ (reachable\text{-}states\text{-}as\text{-}list\ M1))$
obtain $T1\ G1$ **where** $(T1,G1) = handle\text{-}io\text{-}pair\ b\ c\ M1\ V\ T\ G\ cg\text{-}insert\ cg\text{-}lookup\ (t\text{-}source\ t)\ (t\text{-}input\ t)\ (t\text{-}output\ t)$
using $prod.collapse$ **by** $blast$
then have $T1\text{-}def$: $T1 = fst\ (handle\text{-}io\text{-}pair\ b\ c\ M1\ V\ T\ G\ cg\text{-}insert\ cg\text{-}lookup\ (t\text{-}source\ t)\ (t\text{-}input\ t)\ (t\text{-}output\ t))$
and $G1\text{-}def$: $G1 = snd\ (handle\text{-}io\text{-}pair\ b\ c\ M1\ V\ T\ G\ cg\text{-}insert\ cg\text{-}lookup\ (t\text{-}source\ t)\ (t\text{-}input\ t)\ (t\text{-}output\ t))$
using $fst\text{-}conv[of\ T1\ G1]\ snd\text{-}conv[of\ T1\ G1]$ **by** $force+$
define u **where** $u = ((V\ (t\text{-}source\ t))@[(t\text{-}input\ t,\ t\text{-}output\ t)])$
define v **where** $v = (V\ (t\text{-}target\ t))$

obtain $T2\ G2$ **where** $(T2,G2) = distinguish\text{-}from\text{-}set\ M1\ V\ T1\ G1\ cg\text{-}lookup\ cg\text{-}insert\ dist\text{-}fun\ u\ v\ rstates\ k\ l\ b\ heuristic\ False$
using $prod.collapse$ **by** $blast$
then have $T2\text{-}def$: $T2 = fst\ (distinguish\text{-}from\text{-}set\ M1\ V\ T1\ G1\ cg\text{-}lookup\ cg\text{-}insert\ dist\text{-}fun\ u\ v\ rstates\ k\ l\ b\ heuristic\ False)$
and $G2\text{-}def$: $G2 = snd\ (distinguish\text{-}from\text{-}set\ M1\ V\ T1\ G1\ cg\text{-}lookup\ cg\text{-}insert\ dist\text{-}fun\ u\ v\ rstates\ k\ l\ b\ heuristic\ False)$
using $fst\text{-}conv[of\ T2\ G2]\ snd\text{-}conv[of\ T2\ G2]$ **by** $force+$

define $G3$ **where** $G3 = cg\text{-}merge\ G2\ u\ v$

obtain *TH GH* **where** (*TH*,*GH*) = *distinguish-from-set M1 V T1 G1 cg-lookup cg-insert dist-fun u u rstates k l b heuristic True*
using *prod.collapse* **by** *blast*
then have *TH-def*: *TH* = *fst* (*distinguish-from-set M1 V T1 G1 cg-lookup cg-insert dist-fun u u rstates k l b heuristic True*)
and *GH-def*: *GH* = *snd* (*distinguish-from-set M1 V T1 G1 cg-lookup cg-insert dist-fun u u rstates k l b heuristic True*)
using *fst-conv*[*of TH GH*] *snd-conv*[*of TH GH*] **by** *force*+

have *TG-cases*: *?TG* = (*if* (*d M1 V t X′ l*) *then* (*T2*,*G3*) *else* (*TH*,*GH*))
unfolding *handleUT-dynamic-def Let-def*
unfolding *u-def*[*symmetric*] *v-def*[*symmetric*] *rstates-def*[*symmetric*] *k-def*[*symmetric*] *l-def*[*symmetric*] *heuristic-def*[*symmetric*]
unfolding ‹(*T1*,*G1*) = *handle-io-pair b c M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)›[*symmetric*] *case-prod-conv*
unfolding ‹(*T2*,*G2*) = *distinguish-from-set M1 V T1 G1 cg-lookup cg-insert dist-fun u v rstates k l b heuristic False*›[*symmetric*] *case-prod-conv*
unfolding *G3-def*[*symmetric*]
unfolding ‹(*TH*,*GH*) = *distinguish-from-set M1 V T1 G1 cg-lookup cg-insert dist-fun u u rstates k l b heuristic True*›[*symmetric*]
unfolding *X′-def*[*symmetric*]
by *auto*
then have *TG-cases-fst*: *fst ?TG* = (*if* (*d M1 V t X′ l*) *then T2 else TH*)
and *TG-cases-snd*: *snd ?TG* = (*if* (*d M1 V t X′ l*) *then G3 else GH*)
by *auto*

have *set T* ⊆ *set T1*
unfolding *T1-def handle-io-pair-def*
by (*metis distribute-extension-subset*)
moreover have *set T1* ⊆ *set T2*
unfolding *T2-def*
by (*meson distinguish-from-set-subset*)
moreover have *set T1* ⊆ *set TH*
unfolding *TH-def*
by (*meson distinguish-from-set-subset*)
ultimately have ∗:*set T* ⊆ *set* (*fst ?TG*)
using *TG-cases* **by** *auto*

have *finite-tree T* ⟹ *finite-tree T1*
unfolding *T1-def handle-io-pair-def*
by (*metis distribute-extension-finite*)
moreover have *finite-tree T1* ⟹ *finite-tree T2*
unfolding *T2-def*
by (*meson distinguish-from-set-finite*)
moreover have *finite-tree T1* ⟹ *finite-tree TH*
unfolding *TH-def*
by (*meson distinguish-from-set-finite*)

ultimately have $**$:*finite-tree T* $\Longrightarrow$ *finite-tree* (*fst ?TG*)
using *TG-cases* **by** *auto*

have $***$: *observable M1* $\Longrightarrow$
observable M2 $\Longrightarrow$
minimal M1 $\Longrightarrow$
minimal M2 $\Longrightarrow$
size-r M1 $\leq$ *m* $\Longrightarrow$
size M2 $\leq$ *m* $\Longrightarrow$
inputs M2 = *inputs M1* $\Longrightarrow$
outputs M2 = *outputs M1* $\Longrightarrow$
is-state-cover-assignment M1 V $\Longrightarrow$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\Longrightarrow$
V ‘ *reachable-states M1* $\subseteq$ *set T* $\Longrightarrow$
t $\in$ *transitions M1* $\Longrightarrow$
t-source t $\in$ *reachable-states M1* $\Longrightarrow$
V (*t-source t*) @ [(*t-input t*, *t-output t*)] $\neq$ *V* (*t-target t*) $\Longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge $\Longrightarrow$
L M1 $\cap$ *set* (*fst ?TG*) = *L M2* $\cap$ *set* (*fst ?TG*) $\Longrightarrow$
(*set T0* $\subseteq$ *set T*) $\Longrightarrow$
($\forall$ γ . (*length* γ $\leq$ (*m*−*size-r M1*) $\wedge$ *list.set* γ $\subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* γ $\in$ *LS M1* (*t-target t*))
$\longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})))

$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)
proof −
assume *a01* : *observable M1*
assume *a02* : *observable M2*
assume *a03* : *minimal M1*
assume *a04* : *minimal M2*
assume *a05* : *size-r M1* $\leq$ *m*
assume *a06* : *size M2* $\leq$ *m*
assume *a07* : *inputs M2* = *inputs M1*
assume *a08* : *outputs M2* = *outputs M1*
assume *a09* : *is-state-cover-assignment M1 V*
assume *a10* : *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
assume *a11* : *V* ‘ *reachable-states M1* $\subseteq$ *set T*
assume *a12* : *t* $\in$ *transitions M1*
assume *a13* : *t-source t* $\in$ *reachable-states M1*
assume *a14* : *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a15* : *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
assume *a16* : *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*

assume *a17* : *L M1* ∩ *set* (*fst ?TG*) = *L M2* ∩ *set* (*fst ?TG*)
assume *a18* : (*set T0* ⊆ *set T*)
assume *a19* : *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] ≠ *V* (*t-target t*)

have *pass-T1* : *L M1* ∩ *set T1* = *L M2* ∩ *set T1*
using *a17* ‹*set T1* ⊆ *set T2*› ‹*set T1* ⊆ *set TH*› **unfolding** *TG-cases-fst*
by (*cases d M1 V t X′ l*; *auto*)
then have *pass-T* : *L M1* ∩ *set T* = *L M2* ∩ *set T*
using ‹*set T* ⊆ *set T1*› **by** *blast*

have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*OF a13 a12*] **by** *auto*
then have (*V* (*t-target t*)) ∈ *L M1*
using *is-state-cover-assignment-language*[*OF a09*] **by** *blast*
moreover have (*V* (*t-target t*)) ∈ *set T*
using *a11* ‹*t-target t* ∈ *reachable-states M1*› **by** *blast*
ultimately have (*V* (*t-target t*)) ∈ *L M2*
using *pass-T* **by** *blast*
then have *v* ∈ *L M2*
unfolding *v-def* **.**

have (*V* (*t-source t*)) ∈ *L M1*
using *is-state-cover-assignment-language*[*OF a09 a13*] **by** *blast*
moreover have (*V* (*t-source t*)) ∈ *set T*
using *a11 a13* **by** *blast*
ultimately have (*V* (*t-source t*)) ∈ *L M2*
using *pass-T* **by** *blast*
have *u* ∈ *L M1*
unfolding *u-def*
using *a01 a09 a12 a13 converge.simps state-cover-transition-converges* **by** *blast*

have *heuristic-prop*: (⋀*T w u′ uBest lBest*. *fst* (*heuristic T w* (*uBest*, *lBest*) *u′*) ∈ {*u′*, *uBest*})
unfolding *heuristic-def*
using *append-heuristic-input-in append-heuristic-io-in*
by *fastforce*

have *convergence-graph-lookup-invar M1 M2 cg-lookup G1*
using *distribute-extension-adds-sequence*(*2*)[*OF a01 a03* ‹(*V* (*t-source t*)) ∈ *L M1*› ‹(*V* (*t-source t*)) ∈ *L M2*› *a14 a15*, *of T* [(*t-input t*, *t-output t*)] *b heuristic*, *OF - heuristic-prop*]
using *pass-T1*
unfolding *T1-def G1-def handle-io-pair-def*
unfolding *heuristic-def*[*symmetric*]
by *blast*

have *list.set rstates* = *V* ‘ *reachable-states M1*
unfolding *rstates-def*
using *reachable-states-as-list-set* **by** *auto*
then have *V* ‘ *reachable-states M1* ⊆ *list.set rstates*
by *auto*
have *preserves-divergence M1 M2* (*list.set rstates*)
unfolding *rstates-def*
using *a10*
by (*metis image-set reachable-states-as-list-set*)
then have *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
unfolding ‹*list.set rstates* = *V* ‘ *reachable-states M1*› **.**
have (⋀*w*. *w* ∈ *list.set rstates* ⟹ ∃ *w*′. *converge M1 w w*′ ∧ *converge M2 w w*′)
proof −
fix *w* **assume** *w* ∈ *list.set rstates*
then obtain *q* **where** *w* = *V q* **and** *q* ∈ *reachable-states M1*
unfolding *rstates-def*
using *reachable-states-as-list-set* **by** *auto*
then have *w* ∈ *L M1* **and** *w* ∈ *set T*
using *is-state-cover-assignment-language*[*OF a09*] *a11* **by** *blast*+
then have *w* ∈ *L M2*
using *pass-T* **by** *blast*
then have *converge M1 w w* **and** *converge M2 w w*
using ‹*w* ∈ *L M1*› **by** *auto*
then show ∃ *w*′. *converge M1 w w*′ ∧ *converge M2 w w*′
by *blast*
qed
have *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
by (*meson a11 inter-eq-subsetI pass-T*)

have *converge M1 u v*
unfolding *u-def v-def*
using *a01 a09 a12 a13 state-cover-transition-converges* **by** *blast*
have *u* ∈ *L M2*
using *distribute-extension-adds-sequence*(*1*)[*OF a01 a03* ‹(*V* (*t-source t*)) ∈ *L M1*› ‹(*V* (*t-source t*)) ∈ *L M2*› *a14 a15*, *of T* [(*t-input t*, *t-output t*)] *b heuristic*, *OF - heuristic-prop*]
using *pass-T1*
unfolding *T1-def G1-def handle-io-pair-def*
unfolding *heuristic-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*) *Int-iff* ‹*V* (*t-target t*) ∈ *L M1*› ‹*converge M1 u v*› *a01 a02 append-Nil2 converge-append-language-iff u-def v-def*)

have (*u* = *v*) = *False*
unfolding *u-def v-def* **using** *a19* **by** *simp*

have *after-initial M1 u = t-target t*
using *a09* **unfolding** *u-def*
by (*metis* ‹*converge M1 u v*› ‹*t-target t* ∈ *reachable-states M1*› *a01 a03 converge.elims*(*2*) *convergence-minimal is-state-cover-assignment-observable-after u-def v-def*)

have ⋀ *γ x y* . {*u* @ *ω′* | *ω′*. *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*x*, *y*)]))} ⊆ {*ω* @ *ω′* |*ω ω′*. *ω* ∈ {*u*, *v*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*x*, *y*)]))}
by *blast*

show (∀ *γ* . (*length γ* ≤ (*m*−*size-r M1*) ∧ *list.set γ* ⊆ *inputs M1* × *outputs M1* ∧ *butlast γ* ∈ *LS M1* (*t-target t*))
⟶ ((*L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ *ω′* | *ω′*. *ω′* ∈ *list.set* (*prefixes γ*)})
= *L M2* ∩ (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ *ω′* | *ω′*. *ω′* ∈ *list.set* (*prefixes γ*)}))
∧ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* ∪ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ *ω′* | *ω′*. *ω′* ∈ *list.set* (*prefixes γ*)})))

∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)
proof (*cases d M1 V t X′ l*)
case *True*
then have *?TG* = (*T2*,*G3*)
unfolding *TG-cases* **by** *auto*

have *pass-T2*: *L M1* ∩ *set T2* = *L M2* ∩ *set T2*
using *a17* **unfolding** ‹*?TG* = (*T2*,*G3*)› **by** *auto*

have *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
and *converge M2 u v*
using *pass-T2*
using *distinguish-from-set-establishes-convergence*[*OF a01 a02 a03 a04 a05 a06 a07 a08 a09* ‹*preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)› ‹*L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*› ‹*converge M1 u v*› ‹*u* ∈ *L M2*› ‹*v* ∈ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G1*› *a15 assms*, *of T1 k b heuristic*, *OF - - - - heuristic-prop*]
unfolding *G2-def T2-def* ‹(*u* = *v*) = *False*› *rstates-def*[*symmetric*] *l-def*[*symmetric*]
by *blast*+
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)
unfolding ‹*?TG* = (*T2*,*G3*)› *G3-def snd-conv* **using** *a16*
by (*meson* ‹*converge M1 u v*› *convergence-graph-merge-invar-def*)

have *cons-prop*: ⋀*γ x y*.
length (*γ* @ [(*x*, *y*)]) ≤ *l* ⟹
γ ∈ *LS M1* (*after-initial M1 u*) ⟹
x ∈ *FSM.inputs M1* ⟹
y ∈ *FSM.outputs M1* ⟹

$L\ M1 \cap (list.set\ rstates \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x,\ y)]))\}) =$
$L\ M2 \cap (list.set\ rstates \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x,\ y)]))\}) \wedge$
$preserves\text{-}divergence\ M1\ M2\ (list.set\ rstates \cup \{\omega\ @\ \omega' \mid \omega\ \omega'.\ \omega \in \{u,\ v\} \wedge \omega' \in list.set\ (prefixes\ (\gamma\ @\ [(x,\ y)]))\})$
and *nil-prop*: $preserves\text{-}divergence\ M1\ M2\ (list.set\ rstates \cup \{u,\ v\})$
using *pass-T2*
using *distinguish-from-set-properties*(1,2)[*OF a01 a02 a03 a04 a07 a08 a09* ‹V ‘ *reachable-states M1* $\subseteq$ *list.set rstates*› ‹*preserves-divergence M1 M2* (*list.set rstates*)› ‹($\bigwedge w.\ w \in list.set\ rstates \Longrightarrow \exists\, w'.\ converge\ M1\ w\ w' \wedge converge\ M2\ w\ w'$)› ‹*converge M1 u v*› ‹$u \in L\ M2$› ‹$v \in L\ M2$› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G1*› *a15 assms*, *of T1 k l b heuristic*, *OF - - - - - heuristic-prop*]
unfolding *G2-def T2-def* ‹$(u = v) = False$›
by *presburger+*
have $\bigwedge \gamma\ .\ (length\ \gamma \leq (m{-}size\text{-}r\ M1) \wedge list.set\ \gamma \subseteq inputs\ M1 \times outputs\ M1 \wedge butlast\ \gamma \in LS\ M1\ (t\text{-}target\ t))$
$\Longrightarrow ((L\ M1 \cap (V\ ‘\ reachable\text{-}states\ M1 \cup \{((V\ (t\text{-}source\ t))@[(t\text{-}input\ t, t\text{-}output\ t)])\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\})$
$= L\ M2 \cap (V\ ‘\ reachable\text{-}states\ M1 \cup \{((V\ (t\text{-}source\ t))@[(t\text{-}input\ t, t\text{-}output\ t)])\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}))$
$\wedge\ preserves\text{-}divergence\ M1\ M2\ (V\ ‘\ reachable\text{-}states\ M1 \cup \{((V\ (t\text{-}source\ t))@[(t\text{-}input\ t, t\text{-}output\ t)])\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\}))$
(**is** $\bigwedge \gamma\ .\ (length\ \gamma \leq (m{-}size\text{-}r\ M1) \wedge list.set\ \gamma \subseteq inputs\ M1 \times outputs\ M1 \wedge butlast\ \gamma \in LS\ M1\ (t\text{-}target\ t)) \Longrightarrow ?P1\ \gamma \wedge ?P2\ \gamma$)
proof −
fix γ **assume** *assm*:$(length\ \gamma \leq (m{-}size\text{-}r\ M1) \wedge list.set\ \gamma \subseteq inputs\ M1 \times outputs\ M1 \wedge butlast\ \gamma \in LS\ M1\ (t\text{-}target\ t))$
show $?P1\ \gamma \wedge ?P2\ \gamma$
proof (*cases* γ *rule*: *rev-cases*)
case *Nil*
have $*$: $(V\ ‘\ reachable\text{-}states\ M1 \cup \{((V\ (t\text{-}source\ t))@[(t\text{-}input\ t, t\text{-}output\ t)])\ @\ \omega' \mid \omega'.\ \omega' \in list.set\ (prefixes\ \gamma)\})$
$= (V\ ‘\ reachable\text{-}states\ M1 \cup \{u\})$
unfolding *u-def*[*symmetric*] ‹$list.set\ rstates = V\ ‘\ reachable\text{-}states\ M1$› *Nil* **by** *auto*

have $?P1\ \gamma$
using ‹$L\ M1 \cap V\ ‘\ reachable\text{-}states\ M1 = L\ M2 \cap V\ ‘\ reachable\text{-}states\ M1$›
‹$u \in L\ M1$› ‹$u \in L\ M2$›
unfolding $*$ **by** *blast*
moreover have $?P2\ \gamma$
using *preserves-divergence-subset*[*OF nil-prop*]
unfolding $*$ ‹$list.set\ rstates = V\ ‘\ reachable\text{-}states\ M1$›
by (*metis Un-empty-right Un-insert-right Un-upper1 insertI1 insert-subsetI*)
ultimately show *?thesis*
by *simp*

next
case (*snoc* γ' *xy*)
moreover obtain *x y* **where** *xy* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
ultimately have $\gamma = \gamma'$@[(*x*,*y*)]
by *auto*

have ∗: (*V* ‘ *reachable-states M1* ∪ {*u* @ ω' |ω'. ω' ∈ *list.set* (*prefixes* γ)})
⊆ (*V* ‘ *reachable-states M1* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* γ)})
by *blast*

have *length* (γ' @ [(*x*, *y*)]) ≤ *l*
using *assm* **unfolding** *l-def* ‹$\gamma = \gamma'$@[(*x*,*y*)]› **by** *auto*
moreover have γ' ∈ *LS M1* (*after-initial M1 u*)
using *assm* **unfolding** *l-def* ‹$\gamma = \gamma'$@[(*x*,*y*)]›
by (*simp add*: ‹*after-initial M1 u* = *t-target t*›)
moreover have *x* ∈ *FSM.inputs M1* **and** *y* ∈ *FSM.outputs M1*
using *assm* **unfolding** ‹$\gamma = \gamma'$@[(*x*,*y*)]› **by** *auto*
ultimately show *?thesis*
using *cons-prop*[*of* γ' *x y*] *preserves-divergence-subset*[*of M1 M2* (*V* ‘ *reachable-states M1* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*, *v*} ∧ ω' ∈ *list.set* (*prefixes* γ)}), *OF* - ∗]
unfolding ‹$\gamma = \gamma'$@[(*x*,*y*)]›[*symmetric*] *u-def*[*symmetric*] ‹*list.set rstates* = *V* ‘ *reachable-states M1*›
by *blast*
qed
qed
then show *?thesis*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)›
by *presburger*
next
case *False*

then have *?TG* = (*TH*,*GH*)
unfolding *TG-cases* **by** *auto*

have *pass-TH*: *L M1* ∩ *set TH* = *L M2* ∩ *set TH*
using *a17* **unfolding** ‹*?TG* = (*TH*,*GH*)› **by** *auto*

have *converge M1 u u*
using ‹*u* ∈ *L M1*› **by** *auto*

have *cons-prop*: ⋀γ *x y*.
length (γ @ [(*x*, *y*)]) ≤ *l* ⟹
γ ∈ *LS M1* (*t-target t*) ⟹
x ∈ *FSM.inputs M1* ⟹
y ∈ *FSM.outputs M1* ⟹
L M1 ∩ (*V* ‘ *reachable-states M1* ∪ {ω @ ω' |ω ω'. ω ∈ {*u*,

$u\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}) =
L M2 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, u\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}) $\wedge$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in \{u, u\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))})
and *nil-prop*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*u*,*u*})

and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)
using *pass-TH*
using *distinguish-from-set-properties*[*OF a01 a02 a03 a04 a07 a08 a09* ‹*V* ‘ *reachable-states M1* $\subseteq$ *list.set rstates*› ‹*preserves-divergence M1 M2* (*list.set rstates*)› ‹($\bigwedge$*w*. *w* $\in$ *list.set rstates* $\Longrightarrow$ $\exists$ *w'*. *converge M1 w w'* $\wedge$ *converge M2 w w'*)› ‹*converge M1 u u*› ‹*u* $\in$ *L M2*› ‹*u* $\in$ *L M2*› ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G1*› *a15 assms*, *of T1 k l b heuristic*, *OF - - - - - heuristic-prop*]
unfolding ‹*?TG* = (*TH*,*GH*)› *snd-conv*
unfolding *GH-def TH-def* ‹*list.set rstates* = *V* ‘ *reachable-states M1*› ‹*after-initial M1 u* = *t-target t*›
by *presburger*+

have $\bigwedge \gamma$. (*length* $\gamma \leq$ (*m*−*size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*))
$\Longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
(**is** $\bigwedge \gamma$. (*length* $\gamma \leq$ (*m*−*size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*)) $\Longrightarrow$ *?P1* γ $\wedge$ *?P2* γ)
proof −
fix γ **assume** *assm*:(*length* $\gamma \leq$ (*m*−*size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*))
show *?P1* γ $\wedge$ *?P2* γ
proof (*cases* γ *rule*: *rev-cases*)
case *Nil*
have $*$: (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= (*V* ‘ *reachable-states M1* $\cup$ {*u*})
unfolding *u-def*[*symmetric*] ‹*list.set rstates* = *V* ‘ *reachable-states M1*› *Nil* **by** *auto*

have *?P1* γ
using ‹*L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*›
‹*u* $\in$ *L M1*› ‹*u* $\in$ *L M2*›
unfolding $*$ **by** *blast*
moreover have *?P2* γ
using *nil-prop*
unfolding $*$ **by** *auto*

```
          ultimately show ?thesis
            by simp
        next
          case (snoc γ' xy)
          moreover obtain x y where xy = (x,y)
            using prod.exhaust by metis
          ultimately have γ = γ'@[(x,y)]
            by auto

          have ∗: {ω @ ω' |ω ω'. ω ∈ {u, u} ∧ ω' ∈ list.set (prefixes γ)} = {u @
ω' |ω'. ω' ∈ list.set (prefixes γ)}
            by blast

          have length (γ' @ [(x, y)]) ≤ l
            using assm unfolding l-def ‹γ = γ'@[(x,y)]› by auto
          moreover have γ' ∈ LS M1 (t-target t)
            using assm unfolding l-def ‹γ = γ'@[(x,y)]›
            by simp
          moreover have x ∈ FSM.inputs M1 and y ∈ FSM.outputs M1
            using assm unfolding ‹γ = γ'@[(x,y)]› by auto
          ultimately show ?thesis
            using cons-prop[of γ' x y]
          unfolding ‹γ = γ'@[(x,y)]›[symmetric] u-def[symmetric] ‹list.set rstates
= V ‘ reachable-states M1› ∗
            by blast
        qed
      qed
      then show ?thesis
        using ‹convergence-graph-lookup-invar M1 M2 cg-lookup (snd ?TG)›
        by presburger
    qed
  qed



  show ?P T G m t X
    using ∗ ∗∗ ∗∗∗ by blast
 qed
 then show ?thesis
  unfolding handles-transition-def
  by blast
qed
```

21.5.2 Static

```
fun traces-to-check :: ('a,'b::linorder,'c::linorder) fsm ⇒ 'a ⇒ nat ⇒ ('b×'c) list
list where
  traces-to-check M q 0 = [] |
  traces-to-check M q (Suc k) = (let
```

ios = *List.product* (*inputs-as-list M*) (*outputs-as-list M*)
in concat (*map* (λ(*x*,*y*) . *case h-obs M q x y of None* ⇒ [[(*x*,*y*)]] | *Some q′* ⇒ [(*x*,*y*)] # (*map* ((#) (*x*,*y*)) (*traces-to-check M q′ k*))) *ios*))

lemma *traces-to-check-set* :
fixes *M* :: (′*a*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
assumes *observable M*
and *q* ∈ *states M*
shows *list.set* (*traces-to-check M q k*) = {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ *k* ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
using *assms*(*2*) **proof** (*induction k arbitrary*: *q*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)

define *ios* **where** *ios*: *ios* = *List.product* (*inputs-as-list M*) (*outputs-as-list M*)
define *f* **where** *f*: *f* = (λ(*x*,*y*) . *case h-obs M q x y of None* ⇒ [[(*x*,*y*)]] | *Some q′* ⇒ [(*x*,*y*)] # (*map* ((#) (*x*,*y*)) (*traces-to-check M q′ k*)))

have *list.set ios* = *inputs M* × *outputs M*
using *inputs-as-list-set outputs-as-list-set* **unfolding** *ios* **by** *auto*
moreover have *traces-to-check M q* (*Suc k*) = *concat* (*map f ios*)
unfolding *f ios* **by** *auto*
ultimately have *in-ex* : ⋀ *io* . *io* ∈ *list.set* (*traces-to-check M q* (*Suc k*)) ⟷ (∃ *x y* . *x* ∈ *inputs M* ∧ *y* ∈ *outputs M* ∧ *io* ∈ *list.set* (*f* (*x*,*y*)))
by *auto*

show *?case*
proof
show *list.set* (*traces-to-check M q* (*Suc k*)) ⊆ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ (*Suc k*) ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
proof
fix *io* **assume** *io* ∈ *list.set* (*traces-to-check M q* (*Suc k*))
then obtain *x y* **where** *x* ∈ *inputs M* **and** *y* ∈ *outputs M*
and *io* ∈ *list.set* (*f* (*x*,*y*))
using *in-ex* **by** *blast*

have [(*x*,*y*)] ∈ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ (*Suc k*) ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
proof −
have *length* ([] @ [(*x*, *y*)]) ≤ *Suc k*
by *auto*
moreover have [] ∈ *LS M q*
using *Suc.prems* **by** *auto*
ultimately show *?thesis*
using ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*› **by** *blast*
qed

show *io* ∈ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ (*Suc k*) ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
proof (*cases h-obs M q x y*)
case *None*
then have *io* = [(*x*,*y*)]
using ‹*io* ∈ *list.set* (*f* (*x*,*y*))› **unfolding** *f* **by** *auto*
then show *?thesis*
using ‹[(*x*,*y*)] ∈ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ (*Suc k*) ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}›
by *blast*
next
case (*Some q′*)
then consider *io* = [(*x*,*y*)] | *io* ∈ *list.set* (*map* ((#) (*x*,*y*)) (*traces-to-check M q′ k*))
using ‹*io* ∈ *list.set* (*f* (*x*,*y*))› **unfolding** *f* **by** *auto*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis*
using ‹[(*x*,*y*)] ∈ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ (*Suc k*) ∧ γ ∈ *LS M q* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}›
by *blast*
next
case *2*
then obtain *io′* **where** *io* = (*x*,*y*)#*io′* **and** *io′* ∈ *list.set* (*traces-to-check M q′ k*)
by *auto*
then have *io′* ∈ {(γ @ [(*x*, *y*)]) | γ *x y* . *length* (γ @ [(*x*, *y*)]) ≤ *k* ∧ γ ∈ *LS M q′* ∧ *x* ∈ *inputs M* ∧ *y* ∈ *outputs M*}
using *Suc.IH*[*OF h-obs-state*[*OF Some*]] **by** *blast*
then obtain γ *x′ y′* **where** *io′* = (γ @ [(*x′*, *y′*)]) **and** *length* (γ @ [(*x′*, *y′*)]) ≤ *k* **and** γ ∈ *LS M q′* **and** *x′* ∈ *inputs M* **and** *y′* ∈ *outputs M*
by *auto*

have *length* (((*x*,*y*)#γ) @ [(*x′*, *y′*)]) ≤ *Suc k*
using ‹*length* (γ @ [(*x′*, *y′*)]) ≤ *k*› **by** *auto*
moreover have ((*x*,*y*)#γ) ∈ *LS M q*
using ‹γ ∈ *LS M q′*› *Some assms*(*1*)
by (*meson h-obs-language-iff*)
ultimately show *?thesis*
using ‹*x′* ∈ *inputs M*› ‹*y′* ∈ *outputs M*› **unfolding** ‹*io* = (*x*,*y*)#*io′*› ‹*io′* = (γ @ [(*x′*, *y′*)])›
by *auto*
qed
qed
qed

show {γ @ [(*x*, *y*)] |γ *x y*. *length* (γ @ [(*x*, *y*)]) ≤ *Suc k* ∧ γ ∈ *LS M q* ∧ *x* ∈ *FSM.inputs M* ∧ *y* ∈ *FSM.outputs M*} ⊆ *list.set* (*traces-to-check M q* (*Suc k*))

proof
fix *io* **assume** *io* ∈ {γ @ [(x, y)] |γ x y. *length* (γ @ [(x, y)]) ≤ *Suc k* ∧ γ ∈ *LS M q* ∧ x ∈ *FSM.inputs M* ∧ y ∈ *FSM.outputs M*}
then obtain γ x′ y′ **where** *io* = (γ @ [(x′, y′)]) **and** *length* (γ @ [(x′, y′)]) ≤ *Suc k* **and** γ ∈ *LS M q* **and** x′ ∈ *inputs M* **and** y′ ∈ *outputs M*
by *auto*
show *io* ∈ *list.set* (*traces-to-check M q* (*Suc k*))
proof (*cases* γ)
case *Nil*
then have *io* = [(x′,y′)]
using ‹*io* = (γ @ [(x′, y′)])› **by** *auto*
have *io* ∈ *list.set* (*f* (x′,y′))
unfolding *f case-prod-conv* ‹*io* = [(x′,y′)]›
by (*cases FSM.h-obs M q* x′ y′; *auto*)
then show *?thesis*
using *in-ex*[*of io*] ‹x′ ∈ *inputs M*› ‹y′ ∈ *outputs M*› **by** *blast*
next
case (*Cons xy* γ′)

obtain *x y* **where** *xy* = (*x*,*y*)
using *prod.exhaust* **by** *metis*

obtain q′ **where** *h-obs M q x y* = *Some* q′ **and** x ∈ *inputs M* **and** y ∈ *outputs M* **and** γ′ ∈ *LS M* q′
using ‹γ ∈ *LS M q*› **unfolding** *Cons* ‹*xy* = (*x*,*y*)›
by (*meson assms*(*1*) *h-obs-language-iff language-io*(*1*) *language-io*(*2*) *list.set-intros*(*1*))
then have γ′@[(x′,y′)] ∈ {γ @ [(x, y)] |γ x y. *length* (γ @ [(x, y)]) ≤ k ∧ γ ∈ *LS M* q′ ∧ x ∈ *FSM.inputs M* ∧ y ∈ *FSM.outputs M*}
using ‹*length* (γ @ [(x′, y′)]) ≤ *Suc k*› ‹x′ ∈ *inputs M*› ‹y′ ∈ *outputs M*› **unfolding** *Cons* **by** *auto*
then have γ′@[(x′,y′)] ∈ *list.set* (*traces-to-check M* q′ *k*)
using *Suc.IH*[*OF h-obs-state*[*OF* ‹*h-obs M q x y* = *Some* q′›]] **by** *blast*
then have *io* ∈ *list.set* (*f* (*x*,*y*))
unfolding *f case-prod-conv* ‹*h-obs M q x y* = *Some* q′› **unfolding** ‹*io* = (γ @ [(x′, y′)])› *Cons* ‹*xy* = (*x*,*y*)›
by *auto*
then show *?thesis*
using *in-ex*[*of io*] ‹x ∈ *inputs M*› ‹y ∈ *outputs M*› **by** *blast*
qed
qed
qed
qed

fun *establish-convergence-static* :: (*nat* ⇒ ′*a* ⇒ (′*b*×′*c*) *prefix-tree*) ⇒
(′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒
(′*a*,′*b*,′*c*) *state-cover-assignment* ⇒
(′*b*×′*c*) *prefix-tree* ⇒
′*d* ⇒

$('d \Rightarrow ('b \times 'c)\ list \Rightarrow 'd) \Rightarrow$
$('d \Rightarrow ('b \times 'c)\ list \Rightarrow ('b \times 'c)\ list\ list) \Rightarrow$
$nat \Rightarrow$
$('a,'b,'c)\ transition \Rightarrow$
$(('b \times 'c)\ prefix\text{-}tree \times 'd)$

where
establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t =
(*let*
α = *V* (*t-source t*);
xy = (*t-input t*, *t-output t*);
β = *V* (*t-target t*);
qSource = (*after-initial M* (*V* (*t-source t*)));
qTarget = (*after-initial M* (*V* (*t-target t*)));
k = *m* − *size-r M*;
ttc = [] # *traces-to-check M qTarget k*;
handleTrace = (λ (*T*,*G*) *u* .
if is-in-language M qTarget u
then let
qu = *FSM.after M qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(*T'*,*G'*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' cg-lookup cg-insert β (*u*@*w*) *False* (*append-heuristic-input M*))
in foldl appendDistTrace (*T*,*G*) *ws*
else let
(*T'*,*G'*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' cg-lookup cg-insert β *u False* (*append-heuristic-input M*))
in
foldl handleTrace (*T*,*G*) *ttc*)

lemma *appendDistTrace-subset-helper* :
assumes *appendDistTrace* = (λ (*T*,*G*) *w* . *let*
(*T'*,*G'*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' cg-lookup cg-insert β (*u*@*w*) *False* (*append-heuristic-input M*))
shows *set T* ⊆ *set* (*fst* (*appendDistTrace* (*T*,*G*) *w*))
proof −
obtain *T' G'* **where** ∗∗∗: *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*) = (*T'*,*G'*)
using *prod.exhaust* **by** *metis*

show *set T* ⊆ *set* (*fst* (*appendDistTrace* (*T*,*G*) *w*))
using *distribute-extension-subset*[*of T M G cg-lookup cg-insert* α *xy*#*u*@*w False* (*append-heuristic-input M*)]
using *distribute-extension-subset*[*of T′ M G′ cg-lookup cg-insert* β *u*@*w False* (*append-heuristic-input M*)]
unfolding *assms case-prod-conv* ∗∗∗ *Let-def fst-conv*
by *blast*
qed

lemma *handleTrace-subset-helper* :
assumes *handleTrace* = (λ (*T*,*G*) *u* .
if is-in-language M qTarget u
then let
qu = *FSM.after M qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(*T′*,*G′*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*)
in distribute-extension M T′ G′ cg-lookup cg-insert β (*u*@*w*) *False* (*append-heuristic-input M*))
in foldl appendDistTrace (*T*,*G*) *ws*
else let
(*T′*,*G′*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*) *False* (*append-heuristic-input M*)
in distribute-extension M T′ G′ cg-lookup cg-insert β *u False* (*append-heuristic-input M*))
shows *set T* ⊆ *set* (*fst* (*handleTrace* (*T*,*G*) *u*))
proof (*cases is-in-language M qTarget u*)
case *True*

define *qu* **where** *qu*: *qu* = *FSM.after M qTarget u*
define *ws* **where** *ws*: *ws* = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*)
define *appendDistTrace* **where** *appendDistTrace*: *appendDistTrace* = (λ (*T*,*G*) *w* . *let*
(*T′*,*G′*) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*)
in distribute-extension M T′ G′ cg-lookup cg-insert β (*u*@*w*) *False* (*append-heuristic-input M*))

have ∗∗: *handleTrace* (*T*,*G*) *u* = *foldl appendDistTrace* (*T*,*G*) *ws*
unfolding *qu ws appendDistTrace Let-def case-prod-conv assms* **using** *True* **by** *force*

show *?thesis*
using *appendDistTrace-subset-helper*[*OF appendDistTrace*]
unfolding ∗∗
apply (*induction ws rule*: *rev-induct*; *simp*)

by (*metis* (*no-types*, *opaque-lifting*) *Collect-mono-iff fst-conv old.prod.exhaust*)
next
case *False*

obtain T' G' **where** $***$: *distribute-extension M T G cg-lookup cg-insert* α (*xy#u*) *False* (*append-heuristic-input M*) = (T',G')
using *prod.exhaust* **by** *metis*

show *set T* $\subseteq$ *set* (*fst* (*handleTrace* (*T*, *G*) *u*))
using *distribute-extension-subset*[*of T M G cg-lookup cg-insert* α *xy#u False* (*append-heuristic-input M*)]
using *distribute-extension-subset*[*of* T' *M* G' *cg-lookup cg-insert* β *u False* (*append-heuristic-input M*)]
using *False*
unfolding *case-prod-conv* $***$ *Let-def fst-conv assms*
by *force*
qed

lemma *establish-convergence-static-subset* :
set T $\subseteq$ *set* (*fst* (*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t*))
proof −
define α **where** α: α = *V* (*t-source t*)
define *xy* **where** *xy*: *xy* = (*t-input t*, *t-output t*)
define β **where** β: β = *V* (*t-target t*)
define *qSource* **where** *qSource*: *qSource* = (*after-initial M* (*V* (*t-source t*)))
define *qTarget* **where** *qTarget*: *qTarget* = (*after-initial M* (*V* (*t-target t*)))
define *k* **where** *k*: *k* = *m* − *size-r M*
define *ttc* **where** *ttc* : *ttc* = [] # *traces-to-check M qTarget k*
define *handleTrace* **where** *handleTrace*: *handleTrace* = (λ (*T*,*G*) *u* .
if is-in-language M qTarget u
then let
qu = *FSM.after M qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(T',G') = *distribute-extension M T G cg-lookup cg-insert* α (*xy#u@w*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' *cg-lookup cg-insert* β (*u@w*) *False* (*append-heuristic-input M*))
in foldl appendDistTrace (*T*,*G*) *ws*
else let
(T',G') = *distribute-extension M T G cg-lookup cg-insert* α (*xy#u*) *False* (*append-heuristic-input M*)
in distribute-extension M T' G' *cg-lookup cg-insert* β *u False* (*append-heuristic-input M*))

have $*$:*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t* =

foldl handleTrace (*T*,*G*) *ttc*
unfolding *establish-convergence-static.simps* α *xy* β *qSource qTarget k ttc handleTrace Let-def* **by** *force*

show *?thesis*
unfolding $*$ **proof** (*induction ttc rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc io ttc*)

have $*$:*foldl handleTrace* (*T*, *G*) (*ttc*@[*io*]) = *handleTrace* (*foldl handleTrace* (*T*,*G*) *ttc*) *io*
by *auto*

have $\bigwedge$ *u T G* . *set T* $\subseteq$ *set* (*fst* (*handleTrace* (*T*,*G*) *u*))
using *handleTrace-subset-helper*[*of handleTrace*] *handleTrace*
unfolding α *xy* β *qSource qTarget k ttc* **by** *blast*
then show *?case*
unfolding $*$
by (*metis* (*no-types*, *opaque-lifting*) *snoc.IH dual-order.trans fst-conv old.prod.exhaust*)

qed
qed

lemma *establish-convergence-static-finite* :
fixes *M* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
assumes *finite-tree T*
shows *finite-tree* (*fst* (*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t*))
proof −
define α **where** α: α = *V* (*t-source t*)
define *xy* **where** *xy*: *xy* = (*t-input t*, *t-output t*)
define β **where** β: β = *V* (*t-target t*)
define *qSource* **where** *qSource*: *qSource* = (*after-initial M* (*V* (*t-source t*)))
define *qTarget* **where** *qTarget*: *qTarget* = (*after-initial M* (*V* (*t-target t*)))
define *k* **where** *k*: *k* = *m* − *size-r M*
define *ttc* **where** *ttc* : *ttc* = [] # *traces-to-check M qTarget k*
define *handleTrace* **where** *handleTrace*: *handleTrace* = (λ (*T*,*G*) *u* .
if is-in-language M qTarget u
then let
qu = *FSM.after M qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(*T*′,*G*′) = *distribute-extension M T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M*)

in distribute-extension M T′ G′ cg-lookup cg-insert β (u@w) False (append-heuristic-input M))
in foldl appendDistTrace (T,G) ws
else let
(T′,G′) = distribute-extension M T G cg-lookup cg-insert α (xy#u) False (append-heuristic-input M)
in distribute-extension M T′ G′ cg-lookup cg-insert β u False (append-heuristic-input M))

have ∗:*establish-convergence-static dist-fun M V T G cg-insert cg-lookup m t = foldl handleTrace (T,G) ttc*
unfolding *establish-convergence-static.simps α xy β qSource qTarget k ttc handleTrace Let-def* **by** *force*

show *?thesis*
unfolding ∗ **proof** (*induction ttc rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *assms* **by** *auto*
next
case (*snoc io ttc*)

have ∗:*foldl handleTrace (T, G) (ttc@[io]) = handleTrace (foldl handleTrace (T,G) ttc) io*
by *auto*

have ⋀ *u T G . finite-tree T* ⟹ *finite-tree (fst (handleTrace (T,G) u))*
proof −
fix *T* :: (′*b*×′*c*) *prefix-tree*
fix *u G* **assume** *finite-tree T*
show *finite-tree (fst (handleTrace (T,G) u))* **proof** (*cases is-in-language M qTarget u*)
case *True*

define *qu* **where** *qu*: *qu = FSM.after M qTarget u*
define *ws* **where** *ws*: *ws = sorted-list-of-maximal-sequences-in-tree (dist-fun (Suc (length u)) qu)*
define *appendDistTrace* **where** *appendDistTrace*: *appendDistTrace = (λ (T,G) w . let*
(T′,G′) = distribute-extension M T G cg-lookup cg-insert α (xy#u@w) False (append-heuristic-input M)
in distribute-extension M T′ G′ cg-lookup cg-insert β (u@w) False (append-heuristic-input M))

have ∗∗: *handleTrace (T,G) u = foldl appendDistTrace (T,G) ws*
unfolding *handleTrace qu ws appendDistTrace Let-def case-prod-conv* **using** *True* **by** *force*

have ⋀ *w T G . finite-tree T* ⟹ *finite-tree (fst (appendDistTrace (T,G)*

```
w))
      proof −
        fix T :: ('b×'c) prefix-tree
        fix w G assume finite-tree T

        obtain T' G' where ∗∗∗: distribute-extension M T G cg-lookup cg-insert
α (xy#u@w) False (append-heuristic-input M) = (T',G')
          using prod.exhaust by metis

        show finite-tree (fst (appendDistTrace (T,G) w))
         using distribute-extension-finite[of T M G cg-lookup cg-insert α xy#u@w
False (append-heuristic-input M), OF ‹finite-tree T›]
           using distribute-extension-finite[of T' M G' cg-lookup cg-insert β u@w
False (append-heuristic-input M)]
          unfolding appendDistTrace case-prod-conv ∗∗∗ Let-def fst-conv
          by blast
      qed
      then show ?thesis
        unfolding ∗∗ using ‹finite-tree T›
        apply (induction ws rule: rev-induct; simp)
        by (metis (no-types, opaque-lifting) fst-conv old.prod.exhaust)
    next
      case False

      obtain T' G' where ∗∗∗: distribute-extension M T G cg-lookup cg-insert α
(xy#u) False (append-heuristic-input M) = (T',G')
          using prod.exhaust by metis

      show finite-tree (fst (handleTrace (T, G) u))
       using distribute-extension-finite[of T M G cg-lookup cg-insert α xy#u False
(append-heuristic-input M), OF ‹finite-tree T›]
        using distribute-extension-finite[of T' M G' cg-lookup cg-insert β u False
(append-heuristic-input M)]
        using False
        unfolding case-prod-conv ∗∗∗ Let-def fst-conv handleTrace
        by force
    qed
  qed

  then show ?case
    unfolding ∗
    by (metis (no-types, opaque-lifting) snoc.IH fst-conv old.prod.exhaust)
 qed
qed


lemma establish-convergence-static-properties :
 assumes observable M1
    and observable M2
```

and *minimal M1*
and *minimal M2*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $t \in$ *transitions M1*
and *t-source t* $\in$ *reachable-states M1*
and *is-state-cover-assignment M1 V*
and *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*
and *V* ‘ *reachable-states M1* $\subseteq$ *set T*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow \exists$ *io* . $\forall$ *k1 k2* . *io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V q*))
and $\bigwedge$ *q k* . *q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)
and *L M1* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*)) = *L M2* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
shows $\forall$ γ *x y* . *length* (γ@[(*x*,*y*)]) $\leq$ *m* − *size-r M1* $\longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)])) $\longrightarrow$
x $\in$ *inputs M1* $\longrightarrow$ *y* $\in$ *outputs M1* $\longrightarrow$
L M1 $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))}) = *L M2* $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))})
$\wedge$ *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))})
(**is** *?P1a*)
and *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) $\cup$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))})
(**is** *?P1b*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
(**is** *?P2*)
proof −

define α **where** α: α = *V* (*t-source t*)
define *xy* **where** *xy*: *xy* = (*t-input t*, *t-output t*)
define β **where** β: β = *V* (*t-target t*)
define *qSource* **where** *qSource*: *qSource* = (*after-initial M1* (*V* (*t-source t*)))
define *qTarget* **where** *qTarget*: *qTarget* = (*after-initial M1* (*V* (*t-target t*)))
define *k* **where** *k*: *k* = *m* − *size-r M1*

define *ttc* **where** *ttc* : *ttc* = [] # *traces-to-check M1 qTarget k*
define *handleTrace* **where** *handleTrace*: *handleTrace* = (λ (*T*,*G*) *u* .
if is-in-language M1 qTarget u
then let
qu = *FSM.after M1 qTarget u*;
ws = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length u*)) *qu*);
appendDistTrace = (λ (*T*,*G*) *w* . *let*
(*T*′,*G*′) = *distribute-extension M1 T G cg-lookup cg-insert* α (*xy*#*u*@*w*) *False* (*append-heuristic-input M1*)
in distribute-extension M1 T′ *G*′ *cg-lookup cg-insert* β (*u*@*w*) *False* (*append-heuristic-input M1*))
in foldl appendDistTrace (*T*,*G*) *ws*
else let
(*T*′,*G*′) = *distribute-extension M1 T G cg-lookup cg-insert* α (*xy*#*u*) *False* (*append-heuristic-input M1*)
in distribute-extension M1 T′ *G*′ *cg-lookup cg-insert* β *u False* (*append-heuristic-input M1*))

have *result*: *establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t* = *foldl handleTrace* (*T*,*G*) *ttc*
unfolding *establish-convergence-static.simps* α *xy* β *qSource qTarget k ttc handleTrace Let-def* **by** *force*
then have *result-pass*: *L M1* ∩ *set* (*fst* (*foldl handleTrace* (*T*,*G*) *ttc*)) = *L M2* ∩ *set* (*fst* (*foldl handleTrace* (*T*,*G*) *ttc*))
using *assms*(*18*) **by** *auto*

have *V* (*t-source t*) ∈ *L M1* **and** *t-source t* = *qSource*
using *state-cover-assignment-after*[*OF assms*(*1*,*9*,*8*)] **unfolding** *qSource* **by** *auto*
then have *qSource* ∈ *states M1*
unfolding *qSource*
by (*simp add*: *assms*(*8*) *reachable-state-is-state*)
have α ∈ *L M1*
using ‹*V* (*t-source t*) ∈ *L M1*› **unfolding** α **by** *auto*
have α ∈ *L M2*
by (*metis* α *assms*(*10*) *language-prefix*)

have *qTarget* ∈ *reachable-states M1*
using *reachable-states-next*[*OF assms*(*8*,*7*)] **unfolding** *qTarget*
by (*metis assms*(*1*) *assms*(*9*) *is-state-cover-assignment-observable-after*)
then have *qTarget* ∈ *states M1*
using *reachable-state-is-state* **by** *metis*
have *V* (*t-target t*) ∈ *L M1*
by (*meson assms*(*7*) *assms*(*8*) *assms*(*9*) *is-state-cover-assignment-language reachable-states-next*)
then have β ∈ *L M1*

unfolding β **by** *auto*
have *t-target t* = *qTarget*
by (*metis assms*(*1*) *assms*(*7*) *assms*(*8*) *assms*(*9*) *is-state-cover-assignment-observable-after qTarget reachable-states-next*)
have *converge M1* (α@[*xy*]) β
using *state-cover-transition-converges*[*OF assms*(*1*,*9*,*7*,*8*)]
unfolding α *xy* β **.**
then have α@[*xy*] $\in$ *L M1*
by *auto*

have *L M1* $\cap$ *set T* = *L M2* $\cap$ *set T*
using *assms*(*18*) *establish-convergence-static-subset*[*of T dist-fun M1 V G cg-insert cg-lookup m t*]
by *blast*
then have $\beta \in$ *L M2*
using *reachable-states-next*[*OF assms*(*8*,*7*)] *assms*(*11*) ‹$\beta \in$ *L M1*›
unfolding β *qTarget* **by** *blast*

have ($\forall$ *u w* . *u* $\in$ *list.set ttc* $\longrightarrow$ *u* $\in$ *LS M1 qTarget* $\longrightarrow$ *w* $\in$ *set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*)) $\longrightarrow$ *L M1* $\cap$ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* $\cap$ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*})
$\wedge$ ($\forall$ *u w* . *u* $\in$ *list.set ttc* $\longrightarrow$ *u* $\notin$ *LS M1 qTarget* $\longrightarrow$ *L M1* $\cap$ {α@[*xy*]@*u*,β@*u*} = *L M2* $\cap$ {α@[*xy*]@*u*,β@*u*})
$\wedge$ (*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleTrace* (*T*,*G*) *ttc*)))
using *result-pass*
proof (*induction ttc rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *assms*(*13*) **by** *auto*
next
case (*snoc a ttc*)

have $*$:*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]) = *handleTrace* (*foldl handleTrace* (*T*,*G*) *ttc*) *a*
by *auto*
have *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) *ttc*)) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) *ttc*))
using *snoc.prems handleTrace-subset-helper*[*of handleTrace M1 qTarget dist-fun cg-lookup cg-insert*, *OF handleTrace*]
unfolding $*$
by (*metis* (*no-types*, *opaque-lifting*) *fst-conv inter-eq-subsetI old.prod.exhaust*)
then have *IH1*: $\bigwedge$ *u w*. *u* $\in$ *list.set ttc* $\Longrightarrow$ *u* $\in$ *LS M1 qTarget* $\Longrightarrow$ *w* $\in$ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*)) $\Longrightarrow$ *L M1* $\cap$ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* $\cap$ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*}
and *IH2*: $\bigwedge$*u w*. *u* $\in$ *list.set ttc* $\Longrightarrow$ *u* $\notin$ *LS M1 qTarget* $\Longrightarrow$ *L M1* $\cap$ {α @ [*xy*] @ *u*, β @ *u*} = *L M2* $\cap$ {α @ [*xy*] @ *u*, β @ *u*}
and *IH3*: *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleTrace* (*T*, *G*) *ttc*))
using *snoc.IH*

by *presburger+*

show *?case* **proof** (*cases is-in-language M1 qTarget a*)
case *True*

define *qa* **where** *qa*: *qa* = *FSM.after M1 qTarget a*
define *ws* **where** *ws*: *ws* = *sorted-list-of-maximal-sequences-in-tree* (*dist-fun* (*Suc* (*length a*)) *qa*)
define *appendDistTrace* **where** *appendDistTrace*: *appendDistTrace* = (λ (*T*,*G*) *w* . *let*
(*T′*,*G′*) = *distribute-extension M1 T G cg-lookup cg-insert* α (*xy*#*a*@*w*) *False* (*append-heuristic-input M1*)
in distribute-extension M1 T′ G′ cg-lookup cg-insert β (*a*@*w*) *False* (*append-heuristic-input M1*))

have ∗∗: ⋀ *TG* . *handleTrace TG a* = *foldl appendDistTrace TG ws*
using ‹*is-in-language M1 qTarget a*›
unfolding *qa ws appendDistTrace Let-def case-prod-conv assms True handleTrace* **by** *force*
have *foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]) = *foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*
unfolding ∗
unfolding *True*
unfolding ∗∗ **by** *auto*
then have *L M1* ∩ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) = *L M2* ∩ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*))
using *snoc.prems* **by** *metis*

then have *handleTrace-props*: (∀ *w* . *w* ∈ *list.set ws* ⟶ ((∃ α′ . *converge M1* α α′ ∧ (α′@[*xy*]@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) ∧ *converge M2* α α′)
∧ (∃ β′ . *converge M1* β β′ ∧ (β′@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) ∧ *converge M2* β β′)))
∧ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*))
proof (*induction ws rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *IH3* **by** *auto*
next
case (*snoc v ws*)

have ∗:*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]) = *appendDistTrace* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*) *v*
by *auto*

define *Tws* **where** *Tws*: *Tws* = *fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)
define *Gws* **where** *Gws*: *Gws* = *snd* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)

have (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*) = (*Tws*,*Gws*)
unfolding *Tws Gws* **by** *auto*

obtain *T′ G′* **where** *distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*) = (*T′*,*G′*)
using *prod.exhaust* **by** *metis*

have ∗∗: *appendDistTrace* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*) *v*
= *distribute-extension M1 T′ G′ cg-lookup cg-insert* β (*a*@*v*) *False* (*append-heuristic-input M1*)
using ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy* # *a* @ *v*) *False* (*append-heuristic-input M1*) = (*T′*, *G′*)› ‹*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws* = (*Tws*, *Gws*)› *appendDistTrace* **by** *auto*

have *pass-outer* : *L M1* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert* β (*a*@*v*) *False* (*append-heuristic-input M1*)))
= *L M2* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert* β (*a*@*v*) *False* (*append-heuristic-input M1*)))
using *snoc.prems* **unfolding** ∗ ∗∗ **.**
moreover have *set* (*fst* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*))) ⊆ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert* β (*a*@*v*) *False* (*append-heuristic-input M1*)))
using *distribute-extension-subset*[*of T′ M1 G′ cg-lookup cg-insert* β (*a*@*v*) *False* (*append-heuristic-input M1*)]
using ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*) = (*T′*,*G′*)›
by (*metis fst-conv*)
ultimately have *pass-inner*: *L M1* ∩ *set* (*fst* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*)))
= *L M2* ∩ *set* (*fst* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*)))
by *blast*
then have *pass-ws*: *L M1* ∩ *Prefix-Tree.set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) =
L M2 ∩ *Prefix-Tree.set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*))
using *distribute-extension-subset*[*of Tws M1 Gws cg-lookup cg-insert*]
unfolding *Tws Gws*
by *blast*

have *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) ⊆ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*])))

using *appendDistTrace-subset-helper*[*OF appendDistTrace*]
by (*metis* ∗ *Tws* ‹*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws* = (*Tws*, *Gws*)›)

have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*))
using *snoc.IH*[*OF pass-ws*] **by** *auto*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*)))
using *distribute-extension-adds-sequence*(*2*)[*OF assms*(*1*,*3*) ‹α ∈ *L M1*› ‹α ∈ *L M2*› - *assms*(*14*) *pass-inner append-heuristic-input-in*]
unfolding *Gws* **by** *blast*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*appendDistTrace* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*) *v*))
unfolding ∗∗ ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*) = (*T*′,*G*′)› *snd-conv*
using *distribute-extension-adds-sequence*(*2*)[*OF assms*(*1*,*3*) ‹β ∈ *L M1*› ‹β ∈ *L M2*› - *assms*(*14*) *pass-outer append-heuristic-input-in*]
by *blast*
moreover have ⋀ *w* . *w* ∈ *list.set* (*ws*@[*v*]) ⟹ ((∃ α′ . *converge M1* α α′ ∧ (α′@[*xy*]@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* α α′)
∧ (∃ β′ . *converge M1* β β′ ∧ (β′@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* β β′))
proof −
fix *w* **assume** *w* ∈ *list.set* (*ws*@[*v*])
then consider *w* ∈ *list.set ws* | *v* = *w*
by *auto*
then show ((∃ α′ . *converge M1* α α′ ∧ (α′@[*xy*]@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* α α′)
∧ (∃ β′ . *converge M1* β β′ ∧ (β′@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* β β′))
proof *cases*
case *1*
then show *?thesis* **using** *snoc.IH*[*OF pass-ws*]
using ‹*set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*)) ⊆ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))))›
by *blast*
next
case *2*

have ∃ *u*′. *converge M1* α *u*′ ∧ *u*′ @ *xy* # *a* @ *w* ∈ *set T*′ ∧ *converge M2* α *u*′
using *distribute-extension-adds-sequence*(*1*)[*OF assms*(*1*,*3*) ‹α ∈ *L M1*› ‹α ∈ *L M2*› - *assms*(*14*) *pass-inner append-heuristic-input-in*]
‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) *ws*))›

unfolding *Gws*[*symmetric*]
unfolding ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*) = (*T*′,*G*′)›
unfolding *2 fst-conv*
by *blast*
then have (∃ α' . *converge M1* α α' ∧ (α'@[*xy*]@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* α α')
using ∗∗ ‹*Prefix-Tree.set* (*fst* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy* # *a* @ *v*) *False* (*append-heuristic-input M1*))) ⊆ *Prefix-Tree.set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert* β (*a* @ *v*) *False* (*append-heuristic-input M1*)))› ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy* # *a* @ *v*) *False* (*append-heuristic-input M1*) = (*T*′, *G*′)› **by** *auto*
moreover have (∃ β' . *converge M1* β β' ∧ (β'@*a*@*w*) ∈ *set* (*fst* (*foldl appendDistTrace* (*foldl handleTrace* (*T*, *G*) *ttc*) (*ws*@[*v*]))) ∧ *converge M2* β β')
using *distribute-extension-adds-sequence*(*1*)[*OF assms*(*1*,*3*) ‹β ∈ *L M1*› ‹β ∈ *L M2*› - *assms*(*14*) *pass-outer append-heuristic-input-in*]
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*)))›
unfolding ‹*distribute-extension M1 Tws Gws cg-lookup cg-insert* α (*xy*#*a*@*v*) *False* (*append-heuristic-input M1*) = (*T*′,*G*′)› *snd-conv*
unfolding ∗ ∗∗
unfolding *2*
by *blast*
ultimately show *?thesis* **by** *blast*
qed
qed
ultimately show *?case*
by *fastforce*
qed

have ⋀ *u w*. *u* ∈ *list.set* (*ttc*@[*a*]) ⟹ *u* ∈ *LS M1 qTarget* ⟹ *w* ∈ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*)) ⟹ *L M1* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*}
proof −
fix *u w* **assume** *u* ∈ *list.set* (*ttc*@[*a*]) **and** *a1*:*u* ∈ *LS M1 qTarget* **and** *a2*:*w* ∈ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*))
then consider *u* ∈ *list.set ttc* | *a* = *u*
by *auto*
then show *L M1* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*}
proof *cases*
case *1*
then show *?thesis*
using *IH1*[*OF - a1 a2*] **by** *blast*
next
case *2*

obtain *w*′ **where** *w*@*w*′ ∈ *list.set ws*
proof −

have *qa* ∈ *reachable-states M1*
using ‹*qTarget* ∈ *reachable-states M1*› ‹*u* ∈ *LS M1 qTarget*›
by (*metis 2 after-reachable assms*(*1*) *qa*)
then have *finite-tree* (*dist-fun* (*Suc* (*length u*)) *qa*)
using ‹⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)› *reachable-state-is-state*[*of qa M1*]
by *blast*
moreover have *w* ∈ *set* (*dist-fun* (*Suc* (*length u*)) *qa*)
using ‹*w* ∈ *set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*))›
unfolding *qa 2* **.**
ultimately show *?thesis*
using *sorted-list-of-maximal-sequences-in-tree-ob*[*of dist-fun* (*Suc* (*length u*)) *qa w*]
using *that* **unfolding** *ws 2* **by** *blast*
qed
then obtain α' β' **where** *converge M1* α α' **and** α' @ [*xy*] @ *a* @ *w*@*w'* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]))) **and** *converge M2* α α'
and *converge M1* β β' **and** β' @ *a* @ *w*@*w'* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]))) **and** *converge M2* β β'
using *handleTrace-props*
unfolding ∗∗[*symmetric*] ∗[*symmetric*]
by *blast*
then have α' @ [*xy*] @ *a* @ *w* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*u*])))
and β' @ *a* @ *w* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*u*])))
using *set-prefix*[*of* α' @ [*xy*] @ *a* @ *w w'*]
using *set-prefix*[*of* β' @ *a* @ *w w'*]
unfolding *2*
by *auto*

have α @ [*xy*] @ *u* @ *w* ∈ *L M1* = (α' @ [*xy*] @ *u* @ *w* ∈ *L M1*)
using ‹*converge M1* α α'›
using *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... = (α' @ [*xy*] @ *u* @ *w* ∈ *L M2*)
using ‹α' @ [*xy*] @ *a* @ *w* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*u*]))›
using *snoc.prems* **unfolding** *2*
by *blast*
also have ... = (α @ [*xy*] @ *u* @ *w* ∈ *L M2*)
using ‹*converge M2* α α'›
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally have α @ [*xy*] @ *u* @ *w* ∈ *L M1* = (α @ [*xy*] @ *u* @ *w* ∈ *L M2*) **.**

have β @ *u* @ *w* ∈ *L M1* = (β' @ *u* @ *w* ∈ *L M1*)
using ‹*converge M1* β β'›
using *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... = (β' @ *u* @ *w* ∈ *L M2*)

using ‹β' @ a @ $w \in$ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (T, G) (*ttc*@[u]))›
using *snoc.prems* **unfolding** *2*
by *blast*
also have ... = (β @ u @ $w \in L$ *M2*)
using ‹*converge M2* β β'›
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally have β @ u @ $w \in L$ *M1* = (β @ u @ $w \in L$ *M2*) **.**

then show *?thesis*
using ‹α @ [*xy*] @ u @ $w \in L$ *M1* = (α @ [*xy*] @ u @ $w \in L$ *M2*)›
by *blast*
qed
qed
moreover have $\bigwedge$ u w . $u \in$ *list.set* (*ttc*@[a]) $\Longrightarrow$ $u \notin$ *LS M1 qTarget* $\Longrightarrow$ *L M1* $\cap$ {α @ [*xy*] @ u, β @ u} = *L M2* $\cap$ {α @ [*xy*] @ u, β @ u}
proof −
fix u w **assume** $u \in$ *list.set* (*ttc*@[a]) **and** $u \notin$ *LS M1 qTarget*

then have $u \neq a$
using *True*
unfolding *is-in-language-iff*[*OF assms*(*1*) ‹*qTarget* $\in$ *states M1*›]
by *auto*
then have $u \in$ *list.set ttc*
using ‹$u \in$ *list.set* (*ttc*@[a])› **by** *auto*
then show *L M1* $\cap$ {α @ [*xy*] @ u, β @ u} = *L M2* $\cap$ {α @ [*xy*] @ u, β @ u}
using *IH2*[*OF* - ‹$u \notin$ *LS M1 qTarget*›] **by** *blast*
qed
moreover have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleTrace* (T, G) (*ttc*@[a])))
using *handleTrace-props* **unfolding** $*$ $**$ **by** *blast*
ultimately show *?thesis*
by *blast*
next
case *False*

define *Tc* **where** *Tc*: *Tc* = *fst* (*foldl handleTrace* (T, G) *ttc*)
define *Gc* **where** *Gc*: *Gc* = *snd* (*foldl handleTrace* (T, G) *ttc*)

have (*foldl handleTrace* (T, G) *ttc*) = (*Tc*,*Gc*)
unfolding *Tc Gc* **by** *auto*

define T' **where** T': T' = *fst* (*distribute-extension M1 Tc Gc cg-lookup cg-insert* α (*xy*#*a*) *False* (*append-heuristic-input M1*))
define G' **where** G': G' = *snd* (*distribute-extension M1 Tc Gc cg-lookup cg-insert* α (*xy*#*a*) *False* (*append-heuristic-input M1*))

have $**$: *handleTrace* (*foldl handleTrace* (T,G) *ttc*) a = *distribute-extension*

M1 T′ G′ cg-lookup cg-insert β a False (*append-heuristic-input M1*)
using *False*
unfolding ‹(*foldl handleTrace* (*T*, *G*) *ttc*) = (*Tc*,*Gc*)›
unfolding *handleTrace*
unfolding *case-prod-conv Let-def*
unfolding *T′ G′ Tc Gc*
by (*meson case-prod-beta′*)

have *pass-outer* : *L M1* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert β a False* (*append-heuristic-input M1*)))
= *L M2* ∩ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert β a False* (*append-heuristic-input M1*)))
using *snoc.prems* **unfolding** ∗ ∗∗ **.**
moreover have *set* (*fst* (*distribute-extension M1 Tc Gc cg-lookup cg-insert α* (*xy*#*a*) *False* (*append-heuristic-input M1*))) ⊆ *set* (*fst* (*distribute-extension M1 T′ G′ cg-lookup cg-insert β* (*a*) *False* (*append-heuristic-input M1*)))
using *distribute-extension-subset*[*of T′ M1 G′ cg-lookup cg-insert β a False* (*append-heuristic-input M1*)]
using ‹(*foldl handleTrace* (*T*, *G*) *ttc*) = (*Tc*,*Gc*)›
using *T′* **by** *blast*
ultimately have *pass-inner*: *L M1* ∩ *set* (*fst* (*distribute-extension M1 Tc Gc cg-lookup cg-insert α* (*xy*#*a*) *False* (*append-heuristic-input M1*)))
= *L M2* ∩ *set* (*fst* (*distribute-extension M1 Tc Gc cg-lookup cg-insert α* (*xy*#*a*) *False* (*append-heuristic-input M1*)))
by *blast*

have *convergence-graph-lookup-invar M1 M2 cg-lookup Gc*
using *snoc.IH*[*OF* ‹*L M1* ∩ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) *ttc*)) = *L M2* ∩ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) *ttc*))›]
unfolding *Gc* **by** *blast*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup G′*
using *distribute-extension-adds-sequence*(*2*)[*OF assms*(*1*,*3*) ‹*α* ∈ *L M1*› ‹*α* ∈ *L M2*› - *assms*(*14*) *pass-inner append-heuristic-input-in*]
unfolding *G′* **by** *blast*
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*])))
unfolding ∗ ∗∗
using *distribute-extension-adds-sequence*(*2*)[*OF assms*(*1*,*3*) ‹*β* ∈ *L M1*› ‹*β* ∈ *L M2*› - *assms*(*14*) *pass-outer append-heuristic-input-in*]
by *blast*
moreover have ⋀ *u w*. *u* ∈ *list.set* (*ttc*@[*a*]) ⟹ *u* ∈ *LS M1 qTarget* ⟹ *w* ∈ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*)) ⟹ *L M1* ∩ {*α* @ [*xy*] @ *u* @ *w*, *β* @ *u* @ *w*} = *L M2* ∩ {*α* @ [*xy*] @ *u* @ *w*, *β* @ *u* @ *w*}
proof −
fix *u w* **assume** *u* ∈ *list.set* (*ttc*@[*a*]) **and** *a1*:*u* ∈ *LS M1 qTarget* **and** *a2*:*w* ∈ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*))
then have *u* ≠ *a*

using *False*
unfolding *is-in-language-iff*[*OF assms*(*1*) ‹*qTarget* ∈ *states M1*›]
by *auto*
then have $u \in$ *list.set ttc*
using ‹$u \in$ *list.set* (*ttc*@[*a*])› **by** *auto*
then show *L M1* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*}
using *IH1*[*OF - a1 a2*]
by *blast*
qed
moreover have ⋀ *u w* . $u \in$ *list.set* (*ttc*@[*a*]) ⟹ $u \notin$ *LS M1 qTarget* ⟹ *L M1* ∩ {α @ [*xy*] @ *u*, β @ *u*} = *L M2* ∩ {α @ [*xy*] @ *u*, β @ *u*}
proof −
fix *u w* **assume** $u \in$ *list.set* (*ttc*@[*a*]) **and** $u \notin$ *LS M1 qTarget*
then consider $u \in$ *list.set ttc* | *a* = *u*
by *auto*
then show *L M1* ∩ {α @ [*xy*] @ *u*, β @ *u*} = *L M2* ∩ {α @ [*xy*] @ *u*, β @ *u*} **proof** *cases*
case *1*
then show *?thesis*
using *IH2*[*OF* - ‹$u \notin$ *LS M1 qTarget*›] **by** *blast*
next
case *2*

obtain α′ **where** *converge M1* α α′ **and** α′ @ *xy* # *a* ∈ *set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]))) **and** *converge M2* α α′
using *distribute-extension-adds-sequence*(*1*)[*OF assms*(*1*,*3*) ‹α ∈ *L M1*› ‹α ∈ *L M2*› - *assms*(*14*) *pass-inner append-heuristic-input-in*] ‹*convergence-graph-lookup-invar M1 M2 cg-lookup Gc*›
unfolding *T′*[*symmetric*]
using *distribute-extension-subset*[*of T′ M1 G′ cg-lookup cg-insert* β *a False* (*append-heuristic-input M1*)]
unfolding ∗ ∗∗ **by** *blast*
have ⋀ α′ . α′ @ *xy* # *u* = α′ @ [*xy*] @ *u*
by *auto*

obtain β′ **where** *converge M1* β β′ **and** β′ @ *a* ∈ *set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]))) **and** *converge M2* β β′
using *distribute-extension-adds-sequence*(*1*)[*OF assms*(*1*,*3*) ‹β ∈ *L M1*› ‹β ∈ *L M2*› - *assms*(*14*) *pass-outer append-heuristic-input-in*] ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G′*›
unfolding ∗ ∗∗ **by** *blast*

have α @ [*xy*] @ *u* ∈ *L M1* = (α′ @ [*xy*] @ *u* ∈ *L M1*)
using ‹*converge M1* α α′›
using *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... = (α′ @ [*xy*] @ *u* ∈ *L M2*)
using ‹α′ @ *xy* # *a* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*]))›

using *snoc.prems* **unfolding** *2* ‹$\bigwedge \alpha'$. α' @ *xy* # *u* = α' @ [*xy*] @ *u*›
by *blast*
also have ... = (α @ [*xy*] @ *u* ∈ *L M2*)
using ‹*converge M2* α α'›
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally have α @ [*xy*] @ *u* ∈ *L M1* = (α @ [*xy*] @ *u* ∈ *L M2*) **.**

have β @ *u* ∈ *L M1* = (β' @ *u* ∈ *L M1*)
using ‹*converge M1* β β'›
using *assms*(*1*) *converge-append-language-iff* **by** *blast*
also have ... = (β' @ *u* ∈ *L M2*)
using ‹β' @ *a* ∈ *Prefix-Tree.set* (*fst* (*foldl handleTrace* (*T*, *G*) (*ttc*@[*a*])))›
using *snoc.prems* **unfolding** *2*
by *blast*
also have ... = (β @ *u* ∈ *L M2*)
using ‹*converge M2* β β'›
using *assms*(*2*) *converge-append-language-iff* **by** *blast*
finally have β @ *u* ∈ *L M1* = (β @ *u* ∈ *L M2*) **.**
then show *?thesis*
using ‹α @ [*xy*] @ *u* ∈ *L M1* = (α @ [*xy*] @ *u* ∈ *L M2*)›
by *blast*
qed
qed
ultimately show *?thesis*
by *blast*
qed
qed

then have *handleTrace-foldl-props-1*: $\bigwedge$ *u w*. *u* ∈ *list.set ttc* ⟹
u ∈ *LS M1 qTarget* ⟹
w ∈ *Prefix-Tree.set* (*dist-fun* (*Suc* (*length u*)) (*FSM.after M1 qTarget u*))
⟹
L M1 ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*} = *L M2* ∩ {α @ [*xy*] @ *u* @ *w*, β @ *u* @ *w*}
and *handleTrace-foldl-props-2*: $\bigwedge$*u w*. *u* ∈ *list.set ttc* ⟹ *u* ∉ *LS M1 qTarget* ⟹ *L M1* ∩ {α @ [*xy*] @ *u*, β @ *u*} = *L M2* ∩ {α @ [*xy*] @ *u*, β @ *u*}
and *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*foldl handleTrace* (*T*, *G*) *ttc*))
by *presburger+*

then show *?P2*
unfolding *result* **by** *blast*

show *preserves-divergence M1 M2* ((*V* ‘ *reachable-states M1*) ∪ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))})
proof −
let *?w* = ((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)])

have V (*t-target t*) $\in$ (V ‘ *reachable-states M1*)
by (*simp add*: ‹*qTarget* $\in$ *reachable-states M1*› ‹*t-target t* = *qTarget*›)
then have ((V ‘ *reachable-states M1*) $\cup$ {((V (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (V (*t-target t*))}) = *Set.insert* ((V (*t-source t*)) @ [(*t-input t*,*t-output t*)]) (V ‘ *reachable-states M1*)
by *blast*
moreover have *Set.insert ?w* (V ‘ *reachable-states M1*) $\subseteq$ *L M1*
using *state-cover-assignment-language*[*OF assms*(*9*)]
using α ‹*converge M1* (α @ [*xy*]) β› *xy* **by** *auto*
ultimately have $*$:*L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {V (*t-source t*) @ [(*t-input t*, *t-output t*)], V (*t-target t*)}) = *Set.insert ?w* (V ‘ *reachable-states M1*)
and $**$:*L M1* $\cap$ (V ‘ *reachable-states M1*) = (V ‘ *reachable-states M1*)
by *blast+*

have $\bigwedge$ *u* . *u* $\in$ *Set.insert ?w* (V ‘ *reachable-states M1*) $\Longrightarrow$ $\neg$*converge M1 u ?w* $\Longrightarrow$ $\neg$*converge M2 u ?w*
proof −
fix *u* **assume** *u* $\in$ *Set.insert ?w* (V ‘ *reachable-states M1*) **and** $\neg$*converge M1 u ?w*
moreover have *converge M1 ?w ?w*
using ‹α@[*xy*] $\in$ *L M1*› **unfolding** α *xy* **by** *auto*
ultimately have *u* $\in$ (V ‘ *reachable-states M1*)
by *auto*

have $\neg$*converge M1 u* β
using ‹$\neg$*converge M1 u ?w*› ‹*converge M1* (α@[*xy*]) β› **unfolding** α *xy* β
by *auto*

have β = V *qTarget*
by (*simp add*: β ‹*t-target t* = *qTarget*›)

obtain *qU* **where** *qU* $\in$ *reachable-states M1* **and** *u* = V *qU*
using ‹*u* $\in$ (V ‘ *reachable-states M1*)› **by** *blast*
then have *qU* = *after-initial M1 u*
using *state-cover-assignment-after*[*OF assms*(*1*,*9*)] **by** *metis*
then have *qU* $\neq$ *qTarget*
using ‹$\neg$*converge M1 u* β›
using β ‹β $\in$ *L M1*› ‹*t-target t* = *qTarget*› ‹*u* = V *qU*› **by** *fastforce*

then obtain *w* **where** $\forall$ *k1 k2* . *w* $\in$ *set* (*dist-fun k1 qU*) $\cap$ *set* (*dist-fun k2 qTarget*) **and** *distinguishes M1 qU qTarget w*
using *assms*(*15*)[*OF reachable-state-is-state*[*OF* ‹*qU* $\in$ *reachable-states M1*›] ‹*qTarget* $\in$ *states M1*›]
by *blast*
then have *w* $\in$ *set* (*after T* (V *qU*)) **and** *w* $\in$ *set* (*after T* (V *qTarget*))
using *assms*(*16*)[*OF* ‹*qU* $\in$ *reachable-states M1*›]

using *assms*(*16*)[*OF* ‹*qTarget* ∈ *reachable-states M1*›]
by *blast*+

have [] ∈ *list.set ttc*
unfolding *ttc* **by** *auto*
moreover have [] ∈ *LS M1 qTarget*
using ‹*qTarget* ∈ *states M1*› **by** *auto*
moreover have *w* ∈ *set* (*dist-fun* (*Suc* (*length* [])) (*FSM.after M1 qTarget* []))
using ‹∀ *k1 k2* . *w* ∈ *set* (*dist-fun k1 qU*) ∩ *set* (*dist-fun k2 qTarget*)› **by** *auto*
ultimately have *L M1* ∩ {*?w* @ *w*, *β* @ *w*} = *L M2* ∩ {*?w* @ *w*, *β* @ *w*}
using *handleTrace-foldl-props-1*[*of* [] *w*]
unfolding *α xy*
by *auto*
moreover have (*?w* @ *w* ∈ *L M1*) = (*β*@*w* ∈ *L M1*)
using *converge-extend*[*OF assms*(*1*) ‹*converge M1* (*α*@[*xy*]) *β*› - ‹*β* ∈ *L M1*›, *of w*]
using *converge-extend*[*OF assms*(*1*) - - ‹*α*@[*xy*] ∈ *L M1*›, *of β w*]
using ‹*converge M1* (*α*@[*xy*]) *β*› **unfolding** *converge-sym*[**where** *u*=*β*]
unfolding *α*[*symmetric*] *xy*[*symmetric*]
by *blast*
ultimately have (*?w* @ *w* ∈ *L M2*) = (*β*@*w* ∈ *L M2*)
by *blast*

have (*w* ∈ *LS M1 qU*) ≠ (*w* ∈ *LS M1 qTarget*)
using ‹*distinguishes M1 qU qTarget w*›
unfolding *distinguishes-def*
by *blast*
moreover have (*w* ∈ *LS M1 qU*) = (*u*@*w* ∈ *L M1*)
by (*metis* ∗∗ *IntD1* ‹*qU* = *after-initial M1 u*› ‹*u* ∈ *V* ‘ *reachable-states M1*› *after-language-iff assms*(*1*))
moreover have (*w* ∈ *LS M1 qTarget*) = (*β*@*w* ∈ *L M1*)
by (*metis* ‹*β* = *V qTarget*› ‹*β* ∈ *L M1*› ‹*qTarget* ∈ *reachable-states M1*› *after-language-iff assms*(*1*) *assms*(*9*) *is-state-cover-assignment-observable-after*)
ultimately have (*u*@*w* ∈ *L M1*) ≠ (*β*@*w* ∈ *L M1*)
by *blast*
moreover have *u*@*w* ∈ *set T*
using ‹*w* ∈ *set* (*after T* (*V qU*))›
unfolding *after-set* ‹*u* = *V qU*›[*symmetric*]
using ‹*u* ∈ *V* ‘ *reachable-states M1*› *assms*(*11*) **by** *auto*
moreover have *β*@*w* ∈ *set T*
using ‹*w* ∈ *set* (*after T* (*V qTarget*))›
unfolding *after-set* ‹*β* = *V qTarget*›
using ‹*qTarget* ∈ *reachable-states M1*› *assms*(*11*) **by** *auto*
ultimately have (*u*@*w* ∈ *L M2*) ≠ (*β*@*w* ∈ *L M2*)
using ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*› **by** *blast*

then have $(u@w \in L\ M2) \neq (?w@w \in L\ M2)$
unfolding ‹$(?w\ @\ w \in L\ M2) = (\beta@w \in L\ M2)$› **.**
moreover have $(u@w \in L\ M2) = (w \in LS\ M2\ (after\text{-}initial\ M2\ u))$
by (*metis* (*no-types*, *lifting*) ** *Int-iff* ‹$L\ M1 \cap Prefix\text{-}Tree.set\ T = L\ M2 \cap Prefix\text{-}Tree.set\ T$› ‹$u \in V$ ‘ *reachable-states M1*› *after-language-iff assms*(*11*) *assms*(*2*) *inter-eq-subsetI*)
moreover have $(?w@w \in L\ M2) = (w \in LS\ M2\ (after\text{-}initial\ M2\ ?w))$
using *assms*(*10*) **unfolding** α[*symmetric*] *xy*[*symmetric*]
by (*metis assms*(*2*) *observable-after-language-append observable-after-language-none*)

ultimately show $\neg converge\ M2\ u\ ?w$
using *converge.elims*(*2*) **by** *blast*
qed
moreover have $\bigwedge\ v\ .\ v \in (V$ ‘ *reachable-states M1*$) \Longrightarrow \neg converge\ M1\ ?w\ v \Longrightarrow \neg converge\ M2\ ?w\ v$
using *calculation* **unfolding** *converge-sym*[**where** *v*=*?w*]
by *blast*
ultimately show *?thesis*
using *assms*(*12*)
unfolding *preserves-divergence.simps*
unfolding * **
by *blast*
qed

have $\bigwedge\ \gamma\ x\ y\ .\ length\ (\gamma@[(x,y)]) \leq m - size\text{-}r\ M1 \Longrightarrow$
$\gamma \in LS\ M1\ (after\text{-}initial\ M1\ (V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)])) \Longrightarrow$
$x \in inputs\ M1 \Longrightarrow y \in outputs\ M1 \Longrightarrow$
$L\ M1 \cap ((V$ ‘ *reachable-states M1*$) \cup \{\omega@\omega' \mid \omega\ \omega'\ .\ \omega \in \{((V\ (t\text{-}source\ t))\ @\ [(t\text{-}input\ t,t\text{-}output\ t)]),\ (V\ (t\text{-}target\ t))\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\}) = L\ M2 \cap\ ((V$ ‘ *reachable-states M1*$) \cup \{\omega@\omega' \mid \omega\ \omega'\ .\ \omega \in \{((V\ (t\text{-}source\ t))\ @\ [(t\text{-}input\ t,t\text{-}output\ t)]),\ (V\ (t\text{-}target\ t))\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\})$
$\wedge\ preserves\text{-}divergence\ M1\ M2\ ((V$ ‘ *reachable-states M1*$) \cup \{\omega@\omega' \mid \omega\ \omega'\ .\ \omega \in \{((V\ (t\text{-}source\ t))\ @\ [(t\text{-}input\ t,t\text{-}output\ t)]),\ (V\ (t\text{-}target\ t))\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\})$
proof
fix $\gamma\ x\ y$
assume $length\ (\gamma@[(x,y)]) \leq m - size\text{-}r\ M1$
and $\gamma \in LS\ M1\ (after\text{-}initial\ M1\ (V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)]))$
and $x \in inputs\ M1$
and $y \in outputs\ M1$

have $(after\text{-}initial\ M1\ (V\ (t\text{-}source\ t)\ @\ [(t\text{-}input\ t,\ t\text{-}output\ t)])) = qTarget$
using ‹$converge\ M1\ (\alpha@[xy])\ \beta$›
unfolding α[*symmetric*] *xy*[*symmetric*] *qTarget* β[*symmetric*]
using ‹$\alpha\ @\ [xy] \in L\ M1$› ‹$\beta \in L\ M1$› *assms*(*1*) *assms*(*3*) *convergence-minimal* **by** *blast*
then have $\gamma \in LS\ M1\ qTarget$

using ‹$\gamma \in$ *LS M1* (*after-initial M1* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]))›
by *auto*

then have γ@[(*x*,*y*)] $\in$ *list.set* (*traces-to-check M1 qTarget k*)
unfolding *traces-to-check-set*[*OF assms*(*1*) ‹*qTarget* $\in$ *states M1*›] *k*
using ‹*length* (γ@[(*x*,*y*)]) $\leq$ *m* − *size-r M1*› ‹*x* $\in$ *inputs M1*› ‹*y* $\in$ *outputs M1*›
by *blast*
then have (γ@[(*x*,*y*)]) $\in$ *list.set ttc*
unfolding *ttc* **by** *auto*

have $\bigwedge \gamma' . \gamma' \in$ *list.set* (*prefixes* γ) $\Longrightarrow \gamma' \in$ *list.set ttc* $\wedge \gamma' \in$ *LS M1 qTarget*
proof
fix γ' **assume** $\gamma' \in$ *list.set* (*prefixes* γ)
then obtain γ'' **where** $\gamma = \gamma'$@γ''
using *prefixes-set-ob* **by** *blast*
then show $\gamma' \in$ *LS M1 qTarget*
using ‹$\gamma \in$ *LS M1 qTarget*› *language-prefix* **by** *metis*

show $\gamma' \in$ *list.set ttc* **proof** (*cases* γ' *rule*: *rev-cases*)
case *Nil*
then show *?thesis* **unfolding** *ttc* **by** *auto*
next
case (*snoc ioI ioL*)
then obtain *xL yL* **where** $\gamma' =$ *ioI*@[(*xL*,*yL*)]
using *prod.exhaust* **by** *metis*
then have *xL* $\in$ *inputs M1* **and** *yL* $\in$ *outputs M1*
using *language-io*[*OF* ‹$\gamma' \in$ *LS M1 qTarget*›, *of xL yL*]
by *auto*
moreover have *length* $\gamma' \leq$ *m* − *size-r M1*
using ‹*length* (γ@[(*x*,*y*)]) $\leq$ *m* − *size-r M1*› ‹$\gamma = \gamma'$@γ''› **by** *auto*
moreover have *ioI* $\in$ *LS M1 qTarget*
using ‹$\gamma' \in$ *LS M1 qTarget*› ‹$\gamma' =$ *ioI*@[(*xL*,*yL*)]› *language-prefix* **by** *metis*
ultimately have $\gamma' \in$ *list.set* (*traces-to-check M1 qTarget k*)
unfolding *traces-to-check-set*[*OF assms*(*1*) ‹*qTarget* $\in$ *states M1*›] *k* ‹$\gamma' =$ *ioI*@[(*xL*,*yL*)]›
by *blast*
then show *?thesis*
unfolding *ttc* **by** *auto*
qed
qed

show *L M1* $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge \omega' \in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))}) = *L M2* $\cap$ ((*V* ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((*V* (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (*V* (*t-target t*))} $\wedge \omega' \in$ *list.set* (*prefixes* (γ@[(*x*,*y*)]))})

proof −
have $L\ M1 \cap (V\ `\ reachable\text{-}states\ M1) = L\ M2 \cap (V\ `\ reachable\text{-}states\ M1)$
using *assms*(*11*) ‹$L\ M1 \cap set\ T = L\ M2 \cap set\ T$›
by *blast*
moreover have $L\ M1 \cap \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{((V\ (t\text{-}source\ t))\ @\ [(t\text{-}input\ t, t\text{-}output\ t)]),\ (V\ (t\text{-}target\ t))\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\} = L\ M2 \cap \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{((V\ (t\text{-}source\ t))\ @\ [(t\text{-}input\ t, t\text{-}output\ t)]),\ (V\ (t\text{-}target\ t))\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\}$
proof −
have $*$:$\{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ (\gamma@[(x,y)]))\}$
$= \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ \gamma)\} \cup \{(\alpha@[xy])@(\gamma@[(x,y)]),\beta@(\gamma@[(x,y)])\}$
unfolding *prefixes-set-Cons-insert* **by** *blast*
have $L\ M1 \cap \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ \gamma)\} = L\ M2 \cap \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ \gamma)\}$
proof −
have $\bigwedge io .\ io \in \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ \gamma)\} \Longrightarrow (io \in L\ M1) = (io \in L\ M2)$
proof −
fix *io* **assume** $io \in \{\omega@\omega' \mid \omega\ \omega' .\ \omega \in \{\alpha@[xy],\beta\} \wedge \omega' \in list.set\ (prefixes\ \gamma)\}$
then obtain γ' **where** $io \in \{\alpha@[xy]@\gamma',\beta@\gamma'\}$ **and** $\gamma' \in list.set\ (prefixes\ \gamma)$
by *force*
then have $\gamma' \in list.set\ ttc$ **and** $\gamma' \in LS\ M1\ qTarget$
using ‹$\bigwedge \gamma' .\ \gamma' \in list.set\ (prefixes\ \gamma) \Longrightarrow \gamma' \in list.set\ ttc \wedge \gamma' \in LS\ M1\ qTarget$›
by *blast+*
moreover have $[] \in Prefix\text{-}Tree.set\ (dist\text{-}fun\ (length\ \gamma')\ (FSM.after\ M1\ qTarget\ \gamma'))$
by *simp*
ultimately have $L\ M1 \cap \{\alpha@[xy]@\gamma',\beta@\gamma'\} = L\ M2 \cap \{\alpha@[xy]@\gamma',\beta@\gamma'\}$
using *handleTrace-foldl-props-1*[*of* γ' []]
by *auto*
then show $(io \in L\ M1) = (io \in L\ M2)$
using ‹$io \in \{\alpha@[xy]@\gamma',\beta@\gamma'\}$› **by** *blast*
qed
then show *?thesis* **by** *blast*
qed
moreover have $L\ M1 \cap \{(\alpha@[xy])@(\gamma@[(x,y)]),\beta@(\gamma@[(x,y)])\} = L\ M2 \cap \{(\alpha@[xy])@(\gamma@[(x,y)]),\beta@(\gamma@[(x,y)])\}$
proof (*cases* $(\gamma@[(x,y)]) \in LS\ M1\ qTarget$)
case *True*
show *?thesis*
using *handleTrace-foldl-props-1*[*OF* ‹$(\gamma@[(x,y)]) \in list.set\ ttc$› *True*, *of* []]
by *auto*
next
case *False*

show *?thesis*
using *handleTrace-foldl-props-2*[*OF* ‹(γ@[(x,y)]) $\in$ *list.set ttc*› *False*]
by *auto*
qed
ultimately show *?thesis*
unfolding α[*symmetric*] *xy*[*symmetric*] β[*symmetric*] $*$
by (*metis* (*no-types*, *lifting*) *Int-Un-distrib*)
qed
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *Int-Un-distrib*)
qed

show *preserves-divergence M1 M2* ((V ‘ *reachable-states M1*) $\cup$ {ω@ω' | ω ω' . $\omega \in$ {((V (*t-source t*)) @ [(*t-input t*,*t-output t*)]), (V (*t-target t*))} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ@[(x,y)]))})
proof −

have $\bigwedge$ u v . $u \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}) $\Longrightarrow$
$v \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}) $\Longrightarrow$
$\neg$ *converge M1 u v* $\Longrightarrow$
$\neg$ *converge M2 u v*
proof −
fix u v **assume** $u \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))})
and $v \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))})
and $\neg$ *converge M1 u v*

then have $u \in$ *L M1* **and** $v \in$ *L M1* **and** *after-initial M1 u* $\neq$ *after-initial M1 v*
by *auto*
then have *after-initial M1 u* $\in$ *states M1*
and *after-initial M1 v* $\in$ *states M1*
using *after-is-state*[*OF assms*(*1*)] **by** *auto*

have *pass-dist*: $\bigwedge$ u . $u \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}) $\Longrightarrow$
($\exists$ k . $\forall$ $w \in$ *Prefix-Tree.set* (*dist-fun k* (*after-initial M1 u*)) . (u@$w \in$ *L M1*) = (u@$w \in$ *L M2*))
proof −
fix u **assume** $u \in$ *L M1* $\cap$ (V ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))})
then consider $u \in$ V ‘ *reachable-states M1* | $u \in$ {ω @ ω' |ω ω'. $\omega \in$ {α @ [xy], β} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(x, y)]))}
by *blast*
then show ($\exists$ k . $\forall$ $w \in$ *Prefix-Tree.set* (*dist-fun k* (*after-initial M1 u*)) . (u@$w \in$ *L M1*) = (u@$w \in$ *L M2*))

proof *cases*
case *1*
then obtain *qU* **where** *qU* ∈ *reachable-states M1* **and** *V qU* = *u*
by *blast*
have *after-initial M1 u* = *qU*
by (*metis* ‹*V qU* = *u*› ‹*qU* ∈ *reachable-states M1*› *assms*(*1*) *assms*(*9*) *is-state-cover-assignment-observable-after*)

have ⋀*w* . *w* ∈ *Prefix-Tree.set* (*dist-fun 0* (*after-initial M1 u*)) ⟹ (*u*@*w* ∈ *L M1*) = (*u*@*w* ∈ *L M2*)
proof −
fix *w* **assume** *w* ∈ *Prefix-Tree.set* (*dist-fun 0* (*after-initial M1 u*))
then have *w* ∈ *Prefix-Tree.set* (*Prefix-Tree.after T u*)
using *assms*(*16*)[*OF* ‹*qU* ∈ *reachable-states M1*›]
unfolding ‹*V qU* = *u*› ‹*after-initial M1 u* = *qU*›
by *blast*
moreover have *u* ∈ *set T*
using *1 assms*(*11*) **by** *auto*
ultimately have *u*@*w* ∈ *set T*
unfolding *after-set*
by *auto*
then show (*u*@*w* ∈ *L M1*) = (*u*@*w* ∈ *L M2*)
using ‹*L M1* ∩ *set T* = *L M2* ∩ *set T*› **by** *blast*
qed
then show *?thesis*
by *blast*
next
case *2*
then obtain *γ′* **where** *u* ∈ {(*α* @ [*xy*]) @ *γ′*, *β* @ *γ′*} **and** *γ′* ∈ *list.set* (*prefixes* (*γ* @ [(*x*, *y*)]))
by *blast*
then have *γ′* ∈ *list.set ttc*
using ‹(*γ* @ [(*x*, *y*)]) ∈ *list.set ttc*› ‹⋀ *γ′* . *γ′* ∈ *list.set* (*prefixes γ*) ⟹ *γ′* ∈ *list.set ttc* ∧ *γ′* ∈ *LS M1 qTarget*›
unfolding *prefixes-set-Cons-insert* **by** *blast*

have *γ′* ∈ *LS M1 qTarget*
proof −
have *u* ∈ *L M1*
using ‹*u* ∈ *L M1* ∩ (*V* ‘ *reachable-states M1* ∪ {*ω* @ *ω′* |*ω ω′*. *ω* ∈ {*α* @ [*xy*], *β*} ∧ *ω′* ∈ *list.set* (*prefixes* (*γ* @ [(*x*, *y*)]))})› **by** *blast*
then show *?thesis*
using ‹*u* ∈ {(*α* @ [*xy*]) @ *γ′*, *β* @ *γ′*}› ‹*converge M1* (*α* @ [*xy*]) *β*›
unfolding *qTarget β*[*symmetric*]
by (*metis* ‹*β* ∈ *L M1*› *assms*(*1*) *converge-append-language-iff insert-iff observable-after-language-none singleton-iff*)
qed

then have (*FSM.after M1 qTarget γ′*) = (*after-initial M1 u*)

using ‹$u \in \{(\alpha$ @ $[xy])$ @ γ', β @ $\gamma'\}$› ‹*converge M1* $(\alpha$ @ $[xy])$ β›
unfolding *qTarget* β[*symmetric*]
by (*metis* ‹α @ $[xy] \in L$ *M1*› ‹$\beta \in L$ *M1*› *after-split assms*(*1*) *assms*(*3*) *convergence-minimal insert-iff observable-after-language-append singleton-iff*)
have $\bigwedge w$. $\{\alpha$ @ $[xy]$ @ γ' @ w, β @ γ' @ $w\} = \{((\alpha$ @ $[xy])$ @ $\gamma')$ @ $w, (\beta$ @ $\gamma')$ @ $w\}$
by *auto*

have $\bigwedge w$. $w \in$ *set* (*dist-fun* (*Suc* (*length* γ')) (*after-initial M1 u*)) $\Longrightarrow$ $(u$ @ $w \in L$ *M1*$) = (u$ @ $w \in L$ *M2*)
using *handleTrace-foldl-props-1*[*OF* ‹$\gamma' \in$ *list.set ttc*› ‹$\gamma' \in$ *LS M1 qTarget*›]
unfolding ‹(*FSM.after M1 qTarget* γ') = (*after-initial M1 u*)›
using ‹$u \in \{(\alpha$ @ $[xy])$ @ γ', β @ $\gamma'\}$›
unfolding ‹$\bigwedge w$. $\{\alpha$ @ $[xy]$ @ γ' @ w, β @ γ' @ $w\} = \{((\alpha$ @ $[xy])$ @ $\gamma')$ @ $w, (\beta$ @ $\gamma')$ @ $w\}$› **by** *blast*

then show *?thesis*
by *blast*
qed
qed

obtain *ku* **where** $\bigwedge w$. $w \in$ *set* (*dist-fun ku* (*after-initial M1 u*)) $\Longrightarrow$ $(u@w \in L$ *M1*$) = (u@w \in L$ *M2*)
using *pass-dist*[*OF* ‹$u \in L$ *M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup \{\omega$ @ $\omega' | \omega$ ω'. $\omega \in \{\alpha$ @ $[xy], \beta\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ $[(x, y)]$))$\}$)›]
by *blast*

obtain *kv* **where** $\bigwedge w$. $w \in$ *set* (*dist-fun kv* (*after-initial M1 v*)) $\Longrightarrow$ $(v@w \in L$ *M1*$) = (v@w \in L$ *M2*)
using *pass-dist*[*OF* ‹$v \in L$ *M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup \{\omega$ @ $\omega' | \omega$ ω'. $\omega \in \{\alpha$ @ $[xy], \beta\} \wedge \omega' \in$ *list.set* (*prefixes* (γ @ $[(x, y)]$))$\}$)›]
by *blast*

obtain *w* **where** $w \in$ *set* (*dist-fun ku* (*after-initial M1 u*))
and $w \in$ *set* (*dist-fun kv* (*after-initial M1 v*))
and *distinguishes M1* (*after-initial M1 u*) (*after-initial M1 v*) *w*
using *assms*(*15*)[*OF* ‹*after-initial M1 u* $\in$ *states M1*› ‹*after-initial M1 v* $\in$ *states M1*› ‹*after-initial M1 u* $\neq$ *after-initial M1 v*›]
by *blast*

then have ($w \in$ *LS M1* (*after-initial M1 u*)) $\neq$ ($w \in$ *LS M1* (*after-initial M1 v*))
unfolding *distinguishes-def* **by** *blast*
moreover have $w \in$ *LS M1* (*after-initial M1 u*) = ($w \in$ *LS M2* (*after-initial M2 u*))
by (*metis* ‹$\bigwedge w$. $w \in$ *Prefix-Tree.set* (*dist-fun ku* (*after-initial M1 u*)) $\Longrightarrow$ $(u$ @ $w \in L$ *M1*$) = (u$ @ $w \in L$ *M2*)› ‹$u \in L$ *M1*› ‹$w \in$ *Prefix-Tree.set* (*dist-fun ku* (*after-initial M1 u*))› *append-Nil2 assms*(*1*) *assms*(*2*) *observable-after-language-append*

observable-after-language-none set-Nil)
moreover have $w \in$ *LS M1* (*after-initial M1 v*) = ($w \in$ *LS M2* (*after-initial M2 v*))
by (*metis* ‹$\bigwedge w.$ $w \in$ *Prefix-Tree.set* (*dist-fun kv* (*after-initial M1 v*)) $\Longrightarrow$ (v @ $w \in$ *L M1*) = (v @ $w \in$ *L M2*)› ‹$v \in$ *L M1*› ‹$w \in$ *Prefix-Tree.set* (*dist-fun kv* (*after-initial M1 v*))› *append-Nil2 assms*(*1*) *assms*(*2*) *observable-after-language-append observable-after-language-none set-Nil*)
ultimately have ($w \in$ *LS M2* (*after-initial M2 u*)) $\neq$ ($w \in$ *LS M2* (*after-initial M2 v*))
by *blast*
then have *after-initial M2 u* $\neq$ *after-initial M2 v*
by *auto*
then show $\neg$ *converge M2 u v*
using *assms*(*2*) *assms*(*4*) *converge.simps convergence-minimal* **by** *blast*
qed

then show *?thesis*
unfolding *preserves-divergence.simps* α[*symmetric*] *xy*[*symmetric*] β[*symmetric*]
by *blast*
qed
qed
then show *?P1a*
by *blast*
qed

lemma *establish-convergence-static-establishes-convergence* :
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $t \in$ *transitions M1*
and *t-source t* $\in$ *reachable-states M1*
and *is-state-cover-assignment M1 V*
and *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*
and *V* ‘ *reachable-states M1* $\subseteq$ *set T*
and *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
and *convergence-graph-lookup-invar M1 M2 cg-lookup G*
and *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ $\exists$ *io* . $\forall$ *k1 k2* . *io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V*

q))
and $\bigwedge$ *q k* . *q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)
and *L M1* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*)) = *L M2* $\cap$ *set* (*fst* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
shows *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
(**is** *converge M2 ?u ?v*)
proof −

have *prop1*: $\bigwedge\gamma$ *x y*.
length (γ @ [(*x*, *y*)]) $\leq$ (*m* − *size-r M1*) $\Longrightarrow$
γ $\in$ *LS M1* (*after-initial M1 ?u*) $\Longrightarrow$
x $\in$ *FSM.inputs M1* $\Longrightarrow$
y $\in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*?u*, *?v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*?u*, *?v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) $\wedge$
preserves-divergence M1 M2
(*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. ω $\in$ {*?u*, *?v*} $\wedge$ ω' $\in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
and *prop2*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*?u*, *?v*})
using *establish-convergence-static-properties*(*1*,*2*)[*OF assms*(*1*−*4*,*7*−*20*)]
by *presburger*+

have *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
using *assms*(*13*,*20*)
using *establish-convergence-static-subset*[*of T dist-fun M1 V G cg-insert cg-lookup m t*]
by *blast*
then have *V* (*t-target t*) $\in$ *L M2*
by (*metis Int-iff assms*(*10*) *assms*(*11*) *assms*(*9*) *imageI is-state-cover-assignment-language reachable-states-next*)

have *converge M1 ?u ?v*
using *state-cover-transition-converges*[*OF assms*(*1*,*11*,*9*,*10*)] **.**

show *?thesis*
using *establish-convergence-from-pass*[*OF assms*(*1*−*8*,*11*) ‹*L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*› ‹*converge M1 ?u ?v*› ‹*V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*› ‹*V* (*t-target t*) $\in$ *L M2*› *prop1 prop2*]
by *blast*
qed

lemma *establish-convergence-static-verifies-transition* :

assumes $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ $\exists$ *io* . $\forall$ *k1 k2* . *io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*

and $\bigwedge$ *q k* . *q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)

shows *verifies-transition* (*establish-convergence-static dist-fun*) *M1 M2 V* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup*

proof −

have $*$:$\bigwedge$ *V T* (*G*::*'d*) *m t*. *set T* $\subseteq$ *set* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*))

using *establish-convergence-static-subset*

by *metis*

have $***$:$\bigwedge$ *V T* (*G*::*'d*) *m t*. *finite-tree T* $\longrightarrow$ *finite-tree* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*))

using *establish-convergence-static-finite*

by *metis*

let *?distinguish-traces* = (λ α *t' q'* β *t'' g''* . *dist-fun 0 q'*)

have $**$:$\bigwedge$ *T* (*G*::*'d*) *m t*.

observable M1 $\Longrightarrow$

observable M2 $\Longrightarrow$

minimal M1 $\Longrightarrow$

minimal M2 $\Longrightarrow$

size-r M1 $\leq$ *m* $\Longrightarrow$

FSM.size M2 $\leq$ *m* $\Longrightarrow$

FSM.inputs M2 = *FSM.inputs M1* $\Longrightarrow$

FSM.outputs M2 = *FSM.outputs M1* $\Longrightarrow$

is-state-cover-assignment M1 V $\Longrightarrow$

preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\Longrightarrow$

V ‘ *reachable-states M1* $\subseteq$ *set T* $\Longrightarrow$

t $\in$ *FSM.transitions M1* $\Longrightarrow$

t-source t $\in$ *reachable-states M1* $\Longrightarrow$

V (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2* $\Longrightarrow$

convergence-graph-lookup-invar M1 M2 cg-lookup G $\Longrightarrow$

convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$

set (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) $\subseteq$ *set T* $\Longrightarrow$

L M1 $\cap$ *Prefix-Tree.set* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*)) =

L M2 $\cap$ *Prefix-Tree.set* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*)) $\Longrightarrow$

converge M2 (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*)) $\wedge$

convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*))

proof

fix *G* :: *'d*

fix *T m t*

assume *a01*: *observable M1*

assume *a02*: *observable M2*
assume *a03*: *minimal M1*
assume *a04*: *minimal M2*
assume *a05*: *size-r M1* $\leq$ *m*
assume *a06*: *FSM.size M2* $\leq$ *m*
assume *a07*: *FSM.inputs M2* = *FSM.inputs M1*
assume *a08*: *FSM.outputs M2* = *FSM.outputs M1*
assume *a09*: *is-state-cover-assignment M1 V*
assume *a10*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
assume *a11*: *V* ‘ *reachable-states M1* $\subseteq$ *set T*
assume *a12*: *t* $\in$ *FSM.transitions M1*
assume *a13*: *t-source t* $\in$ *reachable-states M1*
assume *a14*: *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*
assume *a15*: *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a16*: *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
assume *a17*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*)) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* ((*establish-convergence-static dist-fun*) *M1 V T G cg-insert cg-lookup m t*))
assume *a18*: *set* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) $\subseteq$ *set T*

have *L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*
using *a11 a17* $*$
by *blast*
then have *d2*: *V* (*t-target t*) $\in$ *L M2*
using *a11 is-state-cover-assignment-language*[*OF a09*, *of t-target t*] *reachable-states-next*[*OF a13 a12*]
by *blast*

have *d1*: $\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T* (*V q*))
using *handle-state-cover-static-applies-dist-sets*[*of - M1 dist-fun V cg-initial cg-insert cg-lookup*] *a18*
by (*meson in-mono subsetI subset-after-subset*)

show *converge M2* (*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
using *establish-convergence-static-establishes-convergence*[**where** *dist-fun*=*dist-fun*, *OF a01 a02 a03 a04 a05 a06 a07 a08 a12 a13 a09 a14 a11 a10 a15 a16 assms*(*1*) *d1 assms*(*2*) *a17*]
by *force*

show *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd* (*establish-convergence-static dist-fun M1 V T G cg-insert cg-lookup m t*))
using *establish-convergence-static-properties*(*3*)[**where** *dist-fun*=*dist-fun*, *OF a01 a02 a03 a04 a07 a08 a12 a13 a09 a14 a11 a10 a15 a16 assms*(*1*) *d1 assms*(*2*) *a17*]
by *blast*
qed

show *?thesis*
unfolding *verifies-transition-def*
using $*$ *** **
by *presburger*
qed

definition *handleUT-static* :: (*nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree*) ⇒
((*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒
(*'a*,*'b*,*'c*) *state-cover-assignment* ⇒
(*'b*×*'c*) *prefix-tree* ⇒
'd ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ *'d*) ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list list*) ⇒
(*'d* ⇒ (*'b*×*'c*) *list* ⇒ (*'b*×*'c*) *list* ⇒ *'d*) ⇒
nat ⇒
(*'a*,*'b*,*'c*) *transition* ⇒
(*'a*,*'b*,*'c*) *transition list* ⇒
((*'a*,*'b*,*'c*) *transition list* × (*'b*×*'c*) *prefix-tree* × *'d*))
where
handleUT-static dist-fun M V T G cg-insert cg-lookup cg-merge l t X = (*let*
(*T1*,*G1*) = *handle-io-pair False False M V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*);
(*T2*,*G2*) = *establish-convergence-static dist-fun M V T1 G1 cg-insert cg-lookup l t*;
G3 = *cg-merge G2* ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)]) (*V* (*t-target t*))
in (*X*,*T2*,*G3*))

lemma *handleUT-static-handles-transition* :
fixes *M1*::(*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2*::(*'e*,*'b*,*'c*) *fsm*
assumes ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows *handles-transition* (*handleUT-static dist-fun*) *M1 M2 V* (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*)) *cg-insert cg-lookup cg-merge*
proof −

let *?T0* = (*fst* (*handle-state-cover-static dist-fun M1 V cg-initial cg-insert cg-lookup*))

have ⋀ *T G m t X* .
Prefix-Tree.set T ⊆ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*))) ∧
(*finite-tree T* ⟶ *finite-tree* (*fst* (*snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*)))) ∧

(*observable M1* $\longrightarrow$
observable M2 $\longrightarrow$
minimal M1 $\longrightarrow$
minimal M2 $\longrightarrow$
size-r M1 $\leq$ *m* $\longrightarrow$
FSM.size M2 $\leq$ *m* $\longrightarrow$
FSM.inputs M2 = *FSM.inputs M1* $\longrightarrow$
FSM.outputs M2 = *FSM.outputs M1* $\longrightarrow$
is-state-cover-assignment M1 V $\longrightarrow$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\longrightarrow$
V ‘ *reachable-states M1* $\subseteq$ *Prefix-Tree.set T* $\longrightarrow$
t $\in$ *FSM.transitions M1* $\longrightarrow$
t-source t $\in$ *reachable-states M1* $\longrightarrow$
V (*t-source t*) @ [(*t-input t*, *t-output t*)] $\neq$ *V* (*t-target t*) $\longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\longrightarrow$
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge $\longrightarrow$
L M1 $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*))) =
L M2 $\cap$ *Prefix-Tree.set* (*fst* (*snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*))) $\longrightarrow$
Prefix-Tree.set ?T0 $\subseteq$ *Prefix-Tree.set T* $\longrightarrow$
($\forall \gamma$. *length* $\gamma \leq m -$ *size-r M1* $\wedge$ *list.set* $\gamma \subseteq$ *FSM.inputs M1* $\times$ *FSM.outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*) $\longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)}) =
L M2 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)}) $\wedge$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1* $\cup$ {(*V* (*t-source t*) @ [(*t-input t*, *t-output t*)]) @ ω' |ω'. $\omega' \in$ *list.set* (*prefixes* γ)})) $\wedge$
convergence-graph-lookup-invar M1 M2 cg-lookup (*snd* (*snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*))))
(**is** $\bigwedge$ *T G m t X* . *?P T G m t X*)
proof −

fix *T* :: ($'b \times 'c$) *prefix-tree*
fix *G* :: $'d$
fix *m* :: *nat*
fix *t* :: ($'a$,$'b$,$'c$) *transition*
fix *X* :: ($'a$,$'b$,$'c$) *transition list*

let *?TG* = *snd* (*handleUT-static dist-fun M1 V T G cg-insert cg-lookup cg-merge m t X*)

obtain *T1 G1* **where** (*T1*,*G1*) = *handle-io-pair False False M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)
using *prod.collapse* **by** *blast*
then have *T1-def*: *T1* = *fst* (*handle-io-pair False False M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*))

and *G1-def*: *G1* = *snd* (*handle-io-pair False False M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*))
using *fst-conv*[*of T1 G1*] *snd-conv*[*of T1 G1*] **by** *force+*

obtain *T2 G2* **where** (*T2*,*G2*) = *establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*
using *prod.collapse* **by** *blast*
have *T2-def*: *T2* = *fst* (*establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*)
and *G2-def*: *G2* = *snd* (*establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*)
unfolding ‹(*T2*,*G2*) = *establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*›[*symmetric*] **by** *auto*
define *u* **where** *u* = ((*V* (*t-source t*))@[(*t-input t*, *t-output t*)])
define *v* **where** *v* = (*V* (*t-target t*))

define *G3* **where** *G3* = *cg-merge G2 u v*

have *TG-cases*: *?TG* = (*T2*,*G3*)
unfolding *handleUT-static-def Let-def*
unfolding ‹(*T1*,*G1*) = *handle-io-pair False False M1 V T G cg-insert cg-lookup* (*t-source t*) (*t-input t*) (*t-output t*)›[*symmetric*] *case-prod-conv*
unfolding ‹(*T2*,*G2*) = *establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*›[*symmetric*] *case-prod-conv*
unfolding *G3-def u-def v-def*
by *simp*

have *set T1* ⊆ *set T2*
and *finite-tree T1* ⟹ *finite-tree T2*
using *establish-convergence-static-verifies-transition*[*OF assms*, *of M2 V cg-initial cg-insert cg-lookup*]
unfolding *T2-def verifies-transition-def* **by** *blast+*
moreover have *set T* ⊆ *set T1*
and *finite-tree T* ⟹ *finite-tree T1*
using *handle-io-pair-verifies-io-pair*[*of False False M1 M2 cg-insert cg-lookup*]
unfolding *T1-def verifies-io-pair-def*
by *blast+*
ultimately have ∗:*set T* ⊆ *set* (*fst ?TG*)
and ∗∗:*finite-tree T* ⟹ *finite-tree* (*fst ?TG*)
using *TG-cases* **by** *auto*

have ∗∗∗: *observable M1* ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ *m* ⟹
size M2 ≤ *m* ⟹

inputs M2 = *inputs M1* $\Longrightarrow$
outputs M2 = *outputs M1* $\Longrightarrow$
is-state-cover-assignment M1 V $\Longrightarrow$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1*) $\Longrightarrow$
V ‘ *reachable-states M1* $\subseteq$ *set T* $\Longrightarrow$
t $\in$ *transitions M1* $\Longrightarrow$
t-source t $\in$ *reachable-states M1* $\Longrightarrow$
V (*t-source t*) @ [(*t-input t*, *t-output t*)] $\neq$ *V* (*t-target t*) $\Longrightarrow$
convergence-graph-lookup-invar M1 M2 cg-lookup G $\Longrightarrow$
convergence-graph-insert-invar M1 M2 cg-lookup cg-insert $\Longrightarrow$
convergence-graph-merge-invar M1 M2 cg-lookup cg-merge $\Longrightarrow$
L M1 $\cap$ *set* (*fst ?TG*) = *L M2* $\cap$ *set* (*fst ?TG*) $\Longrightarrow$
(*set ?T0* $\subseteq$ *set T*) $\Longrightarrow$
($\forall$ γ . (*length* γ $\leq$ (*m*−*size-r M1*) $\wedge$ *list.set* γ $\subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* γ $\in$ *LS M1* (*t-target t*))
$\longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. ω' $\in$ *list.set* (*prefixes* γ)})))

$\wedge$ *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)

proof −
assume *a01* : *observable M1*
assume *a02* : *observable M2*
assume *a03* : *minimal M1*
assume *a04* : *minimal M2*
assume *a05* : *size-r M1* $\leq$ *m*
assume *a06* : *size M2* $\leq$ *m*
assume *a07* : *inputs M2* = *inputs M1*
assume *a08* : *outputs M2* = *outputs M1*
assume *a09* : *is-state-cover-assignment M1 V*
assume *a10* : *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1*)
assume *a11* : *V* ‘ *reachable-states M1* $\subseteq$ *set T*
assume *a12* : *t* $\in$ *transitions M1*
assume *a13* : *t-source t* $\in$ *reachable-states M1*
assume *a14* : *convergence-graph-lookup-invar M1 M2 cg-lookup G*
assume *a15* : *convergence-graph-insert-invar M1 M2 cg-lookup cg-insert*
assume *a16* : *convergence-graph-merge-invar M1 M2 cg-lookup cg-merge*
assume *a17* : *L M1* $\cap$ *set* (*fst ?TG*) = *L M2* $\cap$ *set* (*fst ?TG*)
assume *a18* : (*set ?T0* $\subseteq$ *set T*)
assume *a19* : *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\neq$ *V* (*t-target t*)

have *pass-T1* : *L M1* $\cap$ *set T1* = *L M2* $\cap$ *set T1*
using *a17* ‹*set T1* $\subseteq$ *set T2*› **unfolding** *TG-cases* **by** *auto*
then have *pass-T* : *L M1* $\cap$ *set T* = *L M2* $\cap$ *set T*
using ‹*set T* $\subseteq$ *set T1*› **by** *blast*

have *t-target t* ∈ *reachable-states M1*
using *reachable-states-next*[*OF a13 a12*] **by** *auto*
then have (*V* (*t-target t*)) ∈ *L M1*
using *is-state-cover-assignment-language*[*OF a09*] **by** *blast*
moreover have (*V* (*t-target t*)) ∈ *set T*
using *a11* ‹*t-target t* ∈ *reachable-states M1*› **by** *blast*
ultimately have (*V* (*t-target t*)) ∈ *L M2*
using *pass-T* **by** *blast*
then have *v* ∈ *L M2*
unfolding *v-def* **.**

have (*V* (*t-source t*)) ∈ *L M1*
using *is-state-cover-assignment-language*[*OF a09 a13*] **by** *blast*
moreover have (*V* (*t-source t*)) ∈ *set T*
using *a11 a13* **by** *blast*
ultimately have (*V* (*t-source t*)) ∈ *L M2*
using *pass-T* **by** *blast*
have *u* ∈ *L M1*
unfolding *u-def*
using *a01 a09 a12 a13 converge.simps state-cover-transition-converges* **by** *blast*

have *after-initial M1 u* = *t-target t*
using *a09* **unfolding** *u-def*
by (*metis* ‹*u* ∈ *L M1*› *a01 a12 a13 after-split after-transition-exhaust is-state-cover-assignment-observable-after u-def*)

have *u* ∈ *L M2*
using *distribute-extension-adds-sequence*(*1*)[*OF a01 a03* ‹(*V* (*t-source t*)) ∈ *L M1*› ‹(*V* (*t-source t*)) ∈ *L M2*› *a14 a15*, *of T* [(*t-input t*, *t-output t*)], *of False* (*if False then append-heuristic-input M1 else append-heuristic-io*)]
using *pass-T1 append-heuristic-io-in*
unfolding *T1-def G1-def handle-io-pair-def u-def*
by (*metis* (*no-types*, *lifting*) *Int-iff* ‹*u* ∈ *L M1*› *a01 a02 converge-append-language-iff u-def*)
then have *V* (*t-source t*) @ [(*t-input t*, *t-output t*)] ∈ *L M2*
unfolding *u-def* **.**
have *L M1* ∩ *V* ‘ *reachable-states M1* = *L M2* ∩ *V* ‘ *reachable-states M1*
using *a11 a17* ∗
by *blast*
have *V* ‘ *reachable-states M1* ⊆ *set T1*
using *a11* ‹*set T* ⊆ *set T1*› **by** *blast*
have ⋀ *q* . *q* ∈ *reachable-states M1* ⟹ *set* (*dist-fun 0 q*) ⊆ *set* (*after T* (*V q*))
using *handle-state-cover-static-applies-dist-sets*[*of - M1 dist-fun V cg-initial cg-insert cg-lookup*] *a18*
by (*meson in-mono subsetI subset-after-subset*)

then have $\bigwedge q$. $q \in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T1* (*V q*))
using ‹*set T* $\subseteq$ *set T1*›
by (*meson dual-order.trans subset-after-subset*)

have *pass-T2*: *L M1* $\cap$ *Prefix-Tree.set* (*fst* (*establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*)) = *L M2* $\cap$ *Prefix-Tree.set* (*fst* (*establish-convergence-static dist-fun M1 V T1 G1 cg-insert cg-lookup m t*))
using *a17* **unfolding** *TG-cases T2-def fst-conv* **.**
have *convergence-graph-lookup-invar M1 M2 cg-lookup G1*
using *handle-io-pair-verifies-io-pair*[*of False False M1 M2 cg-insert cg-lookup*]

using *a01 a02 a03 a04 a07 a08 a09* ‹*L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*› *pass-T1 a13 fsm-transition-input*[*OF a12*] *fsm-transition-output*[*OF a12*] *a14 a15*
unfolding *T1-def G1-def verifies-io-pair-def*
by *blast*

have *cons-prop*: $\bigwedge\gamma$ *x y*.
length (γ @ [(*x*, *y*)]) $\leq$ *m−size-r M1* $\Longrightarrow$
$\gamma \in$ *LS M1* (*after-initial M1 u*) $\Longrightarrow$
x $\in$ *FSM.inputs M1* $\Longrightarrow$
y $\in$ *FSM.outputs M1* $\Longrightarrow$
L M1 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) =
L M2 $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))}) $\wedge$
preserves-divergence M1 M2 (*V* ‘ *reachable-states M1* $\cup$ {ω @ ω' |ω ω'. $\omega \in$ {*u*, *v*} $\wedge$ $\omega' \in$ *list.set* (*prefixes* (γ @ [(*x*, *y*)]))})
and *nil-prop*: *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {*u*, *v*})
and *conv-G2*: *convergence-graph-lookup-invar M1 M2 cg-lookup G2*
using *establish-convergence-static-properties*[*OF a01 a02 a03 a04 a07 a08 a12 a13 a09* ‹*V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*› ‹*V* ‘ *reachable-states M1* $\subseteq$ *set T1*› *a10* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G1*› *a15 assms*(*1*) ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T1*(*V q*))› *assms*(*2*) *pass-T2*]
unfolding *G2-def*[*symmetric*] *u-def*[*symmetric*] *v-def*[*symmetric*]
by *blast+*

have *converge M2 u v*
using *establish-convergence-static-establishes-convergence*[*OF a01 a02 a03 a04 a05 a06 a07 a08 a12 a13 a09* ‹*V* (*t-source t*) @ [(*t-input t*, *t-output t*)] $\in$ *L M2*› ‹*V* ‘ *reachable-states M1* $\subseteq$ *set T1*› *a10* ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G1*› *a15 assms*(*1*) ‹$\bigwedge$ *q* . *q* $\in$ *reachable-states M1* $\Longrightarrow$ *set* (*dist-fun 0 q*) $\subseteq$ *set* (*after T1*(*V q*))› *assms*(*2*) *pass-T2*]
unfolding *u-def v-def* **by** *blast*
moreover have *converge M1 u v*

unfolding *u-def v-def* **using** *a09 a12 a13*
using *a01 state-cover-transition-converges* **by** *blast*
ultimately have *convergence-graph-lookup-invar M1 M2 cg-lookup G3*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup G2*› *a16*
unfolding *G3-def*
by (*meson convergence-graph-merge-invar-def*)
then have *convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)
unfolding *TG-cases* **by** *auto*

moreover have $\bigwedge \gamma$. (*length* $\gamma \leq$ (*m−size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*))
$\Longrightarrow$ ((*L M1* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= *L M2* $\cap$ (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
$\wedge$ *preserves-divergence M1 M2* (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)}))
(**is** $\bigwedge \gamma$. (*length* $\gamma \leq$ (*m−size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*)) $\Longrightarrow$ *?P1* γ $\wedge$ *?P2* γ)
proof −
fix γ **assume** *assm*:(*length* $\gamma \leq$ (*m−size-r M1*) $\wedge$ *list.set* $\gamma \subseteq$ *inputs M1* $\times$ *outputs M1* $\wedge$ *butlast* $\gamma \in$ *LS M1* (*t-target t*))
show *?P1* γ $\wedge$ *?P2* γ
proof (*cases* γ *rule*: *rev-cases*)
case *Nil*
have $*$: (*V* ‘ *reachable-states M1* $\cup$ {((*V* (*t-source t*))@[(*t-input t*,*t-output t*)]) @ ω' | ω'. $\omega' \in$ *list.set* (*prefixes* γ)})
= (*V* ‘ *reachable-states M1* $\cup$ {*u*})
unfolding *u-def*[*symmetric*] *Nil* **by** *auto*

have *?P1* γ
using ‹*L M1* $\cap$ *V* ‘ *reachable-states M1* = *L M2* $\cap$ *V* ‘ *reachable-states M1*›
‹*u* $\in$ *L M1*› ‹*u* $\in$ *L M2*›
unfolding $*$ **by** *blast*
moreover have *?P2* γ
using *preserves-divergence-subset*[*OF nil-prop*]
unfolding $*$
by (*metis Un-empty-right Un-insert-right Un-upper1 insertI1 insert-subsetI*)
ultimately show *?thesis*
by *simp*
next
case (*snoc* γ' *xy*)
moreover obtain *x y* **where** *xy* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
ultimately have $\gamma = \gamma'$@[(*x*,*y*)]
by *auto*

have $*$: (V ‘ *reachable-states M1* $\cup$ $\{u$ @ ω' $|\omega'.$ $\omega' \in$ *list.set* (*prefixes* γ)$\}$) $\subseteq$ (V ‘ *reachable-states M1* $\cup$ $\{\omega$ @ ω' $|\omega$ $\omega'.$ $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* γ)$\}$)
by *blast*

have *length* (γ' @ $[(x, y)]$) $\leq m -$ *size-r M1*
using *assm* **unfolding** ‹$\gamma = \gamma'$@$[(x,y)]$› **by** *auto*
moreover have $\gamma' \in$ *LS M1* (*after-initial M1 u*)
using *assm* **unfolding** ‹$\gamma = \gamma'$@$[(x,y)]$›
by (*simp add*: ‹*after-initial M1 u* = *t-target t*›)
moreover have $x \in$ *FSM.inputs M1* **and** $y \in$ *FSM.outputs M1*
using *assm* **unfolding** ‹$\gamma = \gamma'$@$[(x,y)]$› **by** *auto*
ultimately show *?thesis*
using *cons-prop*[*of* γ' *x y*] *preserves-divergence-subset*[*of M1 M2* (V ‘ *reachable-states M1* $\cup$ $\{\omega$ @ ω' $|\omega$ $\omega'.$ $\omega \in \{u, v\} \wedge \omega' \in$ *list.set* (*prefixes* γ)$\}$), *OF* - $*$]
unfolding ‹$\gamma = \gamma'$@$[(x,y)]$›[*symmetric*] *u-def*[*symmetric*]
by *blast*
qed
qed
then show *?thesis*
using ‹*convergence-graph-lookup-invar M1 M2 cg-lookup* (*snd ?TG*)›
by *presburger*
qed
show *?P T G m t X*
using $*$ $**$ $***$ **by** *blast*
qed
then show *?thesis*
unfolding *handles-transition-def*
by *blast*
qed

21.6 Distinguishing Traces

21.6.1 Symmetry

The following lemmata serve to show that the function to choose distinguishing sequences returns the same sequence for reversed pairs, thus ensuring that the HSIs do not contain two sequences for the same pair of states.

lemma *select-diverging-ofsm-table-io-sym* :
assumes *observable M*
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *ofsm-table M* (λq . *states M*) (*Suc k*) *q1* $\neq$ *ofsm-table M* (λq . *states M*) (*Suc k*) *q2*
assumes (*select-diverging-ofsm-table-io M q1 q2* (*Suc k*)) = (*io*,(*a*,*b*))
shows (*select-diverging-ofsm-table-io M q2 q1* (*Suc k*)) = (*io*,(*b*,*a*))
proof –
define *xs* **where** *xs*: *xs* = (*List.product* (*inputs-as-list M*) (*outputs-as-list M*))

define *f1′* **where** *f1′*: *f1′* = (λ(*x*, *y*) ⇒ (*case* (*h-obs M q1 x y*, *h-obs M q2 x y*) *of*
(*None*, *None*) ⇒ *None* |
(*None*, *Some q2′*) ⇒ *Some* ((*x*, *y*), *None*, *Some q2′*) |
(*Some q1′*, *None*) ⇒ *Some* ((*x*, *y*), *Some q1′*, *None*) |
(*Some q1′*, *Some q2′*) ⇒ (*if ofsm-table M* (λ*q* . *states M*) ((*Suc k*) − *1*) *q1′* ≠ *ofsm-table M* (λ*q* . *states M*) ((*Suc k*) − *1*) *q2′ then Some* ((*x*, *y*), *Some q1′*, *Some q2′*) *else None*)))
define *f1* **where** *f1*: *f1* = (λ*xs* . (*hd* (*List.map-filter f1′ xs*)))

define *f2′* **where** *f2′*: *f2′* = (λ(*x*, *y*) ⇒ (*case* (*h-obs M q2 x y*, *h-obs M q1 x y*) *of*
(*None*, *None*) ⇒ *None* |
(*None*, *Some q2′*) ⇒ *Some* ((*x*, *y*), *None*, *Some q2′*) |
(*Some q1′*, *None*) ⇒ *Some* ((*x*, *y*), *Some q1′*, *None*) |
(*Some q1′*, *Some q2′*) ⇒ (*if ofsm-table M* (λ*q* . *states M*) ((*Suc k*) − *1*) *q1′* ≠ *ofsm-table M* (λ*q* . *states M*) ((*Suc k*) − *1*) *q2′ then Some* ((*x*, *y*), *Some q1′*, *Some q2′*) *else None*)))
define *f2* **where** *f2*: *f2* = (λ*xs* . (*hd* (*List.map-filter f2′ xs*)))

obtain *x y* **where** *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = ((*x*,*y*),(*h-obs M q1 x y*, *h-obs M q2 x y*))
using *select-diverging-ofsm-table-io-Some*(*1*)[*OF assms*(*1*−*4*)]
by *meson*

have ⋀ *xy io a b* . *f1′ xy* = *Some* (*io*,(*a*,*b*)) ⟹ *f2′ xy* = *Some* (*io*,(*b*,*a*))
proof −
fix *xy io a b* **assume** ∗: *f1′ xy* = *Some* (*io*,(*a*,*b*))
obtain *x y* **where** *xy* = (*x*,*y*)
using *prod.exhaust* **by** *metis*

show *f2′ xy* = *Some* (*io*,(*b*,*a*))
proof (*cases h-obs M q1 x y*)
case *None*
show *?thesis* **proof** (*cases h-obs M q2 x y*)
case *None*
then show *?thesis* **using** ‹*h-obs M q1 x y* = *None*› ∗ **unfolding** *f1′ f2′* ‹*xy* = (*x*,*y*)› **by** *auto*
next
case (*Some q2′*)

show *?thesis* **using** ∗ **unfolding** *f1′ f2′*
unfolding *case-prod-conv None Some* ‹*xy* = (*x*,*y*)› **by** *auto*
qed
next
case (*Some q1′*)
show *?thesis* **proof** (*cases h-obs M q2 x y*)

```
      case None
      show ?thesis using ∗ unfolding f1′ f2′
        unfolding case-prod-conv None Some ‹xy = (x,y)› by auto
    next
      case (Some q2′)
      have ofsm-table M (λq . states M) ((Suc k) − 1) q2′ ≠ ofsm-table M (λq
. states M) ((Suc k) − 1) q1′
       using ∗ unfolding f1′ case-prod-conv ‹h-obs M q1 x y = Some q1′› Some
‹xy = (x,y)› by auto
      then have f1′ (x,y) = Some ((x,y),(h-obs M q1 x y,h-obs M q2 x y))
        unfolding f1′ case-prod-conv ‹h-obs M q1 x y = Some q1′› Some by auto
      then have io = (x,y) and b = h-obs M q2 x y and a = h-obs M q1 x y
        using ∗ ‹xy = (x,y)› by auto

      show ?thesis unfolding f2′
       unfolding case-prod-conv ‹h-obs M q1 x y = Some q1′› Some ‹io = (x,y)›
‹b = h-obs M q2 x y› ‹a = h-obs M q1 x y› ‹xy = (x,y)›
         using ‹ofsm-table M (λq . states M) ((Suc k) − 1) q2′ ≠ ofsm-table M
(λq . states M) ((Suc k) − 1) q1′› by simp
    qed
  qed
 qed
 moreover have ⋀ xy io a b . f2′ xy = Some (io,(a,b)) ⟹ f1′ xy = Some
(io,(b,a))
 proof −
  fix xy io a b assume ∗: f2′ xy = Some (io,(a,b))
  obtain x y where xy = (x,y)
    using prod.exhaust by metis

  show f1′ xy = Some (io,(b,a))
  proof (cases h-obs M q1 x y)
    case None
    show ?thesis proof (cases h-obs M q2 x y)
      case None
      then show ?thesis using ‹h-obs M q1 x y = None› ∗ unfolding f1′ f2′ ‹xy
= (x,y)› by auto
    next
      case (Some q2′)

      show ?thesis using ∗ unfolding f1′ f2′
        unfolding case-prod-conv None Some ‹xy = (x,y)› by auto
    qed
  next
    case (Some q1′)
    show ?thesis proof (cases h-obs M q2 x y)
      case None
      show ?thesis using ∗ unfolding f1′ f2′
        unfolding case-prod-conv None Some ‹xy = (x,y)› by auto
    next
```

case (*Some q2'*)
have *ofsm-table M* (λ*q . states M*) ((*Suc k*) − *1*) *q2'* ≠ *ofsm-table M* (λ*q . states M*) ((*Suc k*) − *1*) *q1'*
using ∗ **unfolding** *f2' case-prod-conv* ‹*h-obs M q1 x y = Some q1'*› *Some* ‹*xy* = (*x*,*y*)› **by** *auto*
then have *f2'* (*x*,*y*) = *Some* ((*x*,*y*),(*h-obs M q2 x y*,*h-obs M q1 x y*))
unfolding *f2' case-prod-conv* ‹*h-obs M q1 x y = Some q1'*› *Some* **by** *auto*
then have *io* = (*x*,*y*) **and** *b* = *h-obs M q1 x y* **and** *a* = *h-obs M q2 x y*
using ∗ ‹*xy* = (*x*,*y*)› **by** *auto*

show *?thesis* **unfolding** *f1'*
unfolding *case-prod-conv* ‹*h-obs M q1 x y = Some q1'*› *Some* ‹*io* = (*x*,*y*)› ‹*b = h-obs M q1 x y*› ‹*a = h-obs M q2 x y*› ‹*xy* = (*x*,*y*)›
using ‹*ofsm-table M* (λ*q . states M*) ((*Suc k*) − *1*) *q2'* ≠ *ofsm-table M* (λ*q . states M*) ((*Suc k*) − *1*) *q1'*› **by** *simp*
qed
qed
qed
ultimately have $\bigwedge$ *xy io a b . f2' xy = Some* (*io*,(*a*,*b*)) ⟷ *f1' xy = Some* (*io*,(*b*,*a*))
by *blast*

moreover have $\bigwedge$ *xs* . ($\bigwedge$ *xy io a b . f1' xy = Some* (*io*,(*a*,*b*)) ⟷ *f2' xy = Some* (*io*,(*b*,*a*))) ⟹ ∃ *xy* ∈ *list.set xs . f1' xy* ≠ *None* ⟹ *f1 xs* = (*io*,(*a*,*b*)) ⟹ *f2 xs* = (*io*,(*b*,*a*))
proof −
fix *xs* **assume** ($\bigwedge$ *xy io a b . f1' xy = Some* (*io*,(*a*,*b*)) ⟷ *f2' xy = Some* (*io*,(*b*,*a*)))
∃ *xy* ∈ *list.set xs . f1' xy* ≠ *None*
f1 xs = (*io*,(*a*,*b*))
then show *f2 xs* = (*io*,(*b*,*a*))
proof (*induction xs*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons xy xs*)
show *?case* **proof** (*cases f1' xy*)
case *None*
then have ∄ *io a b . f1' xy = Some* (*io*,(*a*,*b*))
by *auto*
then have *f2' xy = None*
using *Cons.prems*(*1*)
by (*metis option.exhaust prod-cases3*)
then have *f2* (*xy*#*xs*) = *f2 xs*
unfolding *f2 map-filter-simps* **by** *auto*
moreover have *f1* (*xy*#*xs*) = *f1 xs*
using *None* **unfolding** *f1 map-filter-simps* **by** *auto*
ultimately show *?thesis*
using *Cons.IH Cons.prems*(*1*) *Cons.prems*(*2*) *Cons.prems*(*3*) *None* **by**

```
fastforce
      next
        case (Some ioab)
        then have f1 (xy#xs) = ioab
          unfolding f1 map-filter-simps
          by simp
        then have ioab = (io,(a,b))
          using Cons.prems(3) by auto
        then have f2' xy = Some (io,(b,a))
          using Cons.prems(1) Some by auto
        then show f2 (xy#xs) = (io,(b,a))
          unfolding f2 map-filter-simps by auto
      qed
    qed
  qed

  moreover have f1 xs = (io,(a,b))
    using ‹(select-diverging-ofsm-table-io M q1 q2 (Suc k)) = (io,(a,b))›
    unfolding select-diverging-ofsm-table-io.simps f1 f1' xs Let-def by auto

  moreover have ∃ xy ∈ list.set xs . f1' xy ≠ None
  proof −
    let ?P = ∀ x y . x ∈ inputs M ⟶ y ∈ outputs M ⟶ (h-obs M q1 x y = None
⟷ h-obs M q2 x y = None)
    show ?thesis proof (cases ?P)
      case False
      then obtain x y where x ∈ inputs M and y ∈ outputs M and ¬ (h-obs M
q1 x y = None ⟷ h-obs M q2 x y = None)
        by blast
      then consider h-obs M q1 x y = None ∧ (∃ q2' . h-obs M q2 x y = Some
q2') |
                    h-obs M q2 x y = None ∧ (∃ q1' . h-obs M q1 x y = Some q1')
        by fastforce
      then show ?thesis proof cases
        case 1
         then obtain q2' where h-obs M q1 x y = None and h-obs M q2 x y =
Some q2' by blast
        then have f1' (x,y) = Some ((x,y),(None, Some q2'))
          unfolding f1' by force
        moreover have (x,y) ∈ list.set xs
          unfolding xs
          using ‹y ∈ outputs M› outputs-as-list-set[of M]
          using ‹x ∈ inputs M› inputs-as-list-set[of M]
          using image-iff by fastforce
        ultimately show ?thesis
          by blast
      next
        case 2
         then obtain q1' where h-obs M q2 x y = None and h-obs M q1 x y =
```

Some q1' **by** *blast*
then have *f1'* (*x*,*y*) = *Some* ((*x*,*y*),(*Some q1'*, *None*))
unfolding *f1'* **by** *force*
moreover have (*x*,*y*) ∈ *list.set xs*
unfolding *xs*
using ‹*y* ∈ *outputs M*› *outputs-as-list-set*[*of M*]
using ‹*x* ∈ *inputs M*› *inputs-as-list-set*[*of M*]
using *image-iff* **by** *fastforce*
ultimately show *?thesis*
by *blast*
qed
next
case *True*

obtain *io* **where** *length io* ≤ *Suc k* **and** *io* ∈ *LS M q1* ∪ *LS M q2* **and** *io* ∉ *LS M q1* ∩ *LS M q2*
using ‹*ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q1* ≠ *ofsm-table M* (λ*q* . *states M*) (*Suc k*) *q2*›
unfolding *ofsm-table-set-observable*[*OF assms*(*1*,*2*) *minimise-initial-partition*] *ofsm-table-set-observable*[*OF assms*(*1*,*3*) *minimise-initial-partition*] **by** *blast*

then have *io* ≠ []
using *assms*(*2*) *assms*(*3*) **by** *auto*
then have *io* = [*hd io*] @ *tl io*
by (*metis append.left-neutral append-Cons list.exhaust-sel*)
then obtain *x y* **where** *hd io* = (*x*,*y*)
by (*meson prod.exhaust-sel*)

have [(*x*,*y*)] ∈ *LS M q1* ∩ *LS M q2*
proof −
have [(*x*,*y*)] ∈ *LS M q1* ∪ *LS M q2*
using ‹*io* ∈ *LS M q1* ∪ *LS M q2*› *language-prefix* ‹*hd io* = (*x*,*y*)› ‹*io* = [*hd io*] @ *tl io*›
by (*metis Un-iff*)
then have *x* ∈ *inputs M* **and** *y* ∈ *outputs M*
by *auto*

consider [(*x*,*y*)] ∈ *LS M q1* | [(*x*,*y*)] ∈ *LS M q2*
using ‹[(*x*,*y*)] ∈ *LS M q1* ∪ *LS M q2*› **by** *blast*
then show *?thesis*
proof *cases*
case *1*
then have *h-obs M q1 x y* ≠ *None*
using *h-obs-None*[*OF* ‹*observable M*›] **unfolding** *LS-single-transition* **by** *auto*
then have *h-obs M q2 x y* ≠ *None*
using *True* ‹*x* ∈ *inputs M*› ‹*y* ∈ *outputs M*› **by** *meson*
then show *?thesis*
using *1 h-obs-None*[*OF* ‹*observable M*›]

by (*metis IntI LS-single-transition fst-conv snd-conv*)
next
case *2*
then have *h-obs M q2 x y* $\neq$ *None*
using *h-obs-None*[*OF* ‹*observable M*›] **unfolding** *LS-single-transition* **by** *auto*
then have *h-obs M q1 x y* $\neq$ *None*
using *True* ‹*x* $\in$ *inputs M*› ‹*y* $\in$ *outputs M*› **by** *meson*
then show *?thesis*
using *2 h-obs-None*[*OF* ‹*observable M*›]
by (*metis IntI LS-single-transition fst-conv snd-conv*)
qed
qed
then obtain *q1′ q2′* **where** (*q1*,*x*,*y*,*q1′*) $\in$ *transitions M*
and (*q2*,*x*,*y*,*q2′*) $\in$ *transitions M*
using *LS-single-transition* **by** *force*
then have *q1′* $\in$ *states M* **and** *q2′* $\in$ *states M* **using** *fsm-transition-target* **by** *auto*

have *tl io* $\in$ *LS M q1′* $\cup$ *LS M q2′*
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹(*q1*,*x*,*y*,*q1′*) $\in$ *transitions M*›]
observable-language-transition-target[*OF* ‹*observable M*› ‹(*q2*,*x*,*y*,*q2′*) $\in$ *transitions M*›]
‹*io* $\in$ *LS M q1* $\cup$ *LS M q2*›
unfolding *fst-conv snd-conv*
by (*metis Un-iff* ‹*hd io* = (*x*, *y*)› ‹*io* = [*hd io*] @ *tl io*› *append-Cons append-Nil*)
moreover have *tl io* $\notin$ *LS M q1′* $\cap$ *LS M q2′*
using *observable-language-transition-target*[*OF* ‹*observable M*› ‹(*q1*,*x*,*y*,*q1′*) $\in$ *transitions M*›]
observable-language-transition-target[*OF* ‹*observable M*› ‹(*q2*,*x*,*y*,*q2′*) $\in$ *transitions M*›]
‹*io* $\in$ *LS M q1* $\cup$ *LS M q2*›
unfolding *fst-conv snd-conv*
by (*metis Int-iff LS-prepend-transition* ‹(*q1*, *x*, *y*, *q1′*) $\in$ *FSM.transitions M*› ‹(*q2*, *x*, *y*, *q2′*) $\in$ *FSM.transitions M*› ‹*hd io* = (*x*, *y*)› ‹*io* $\neq$ []› ‹*io* $\notin$ *LS M q1* $\cap$ *LS M q2*› *fst-conv list.collapse snd-conv*)
ultimately have ((*tl io*) $\in$ *LS M q1′*) $\neq$ (*tl io* $\in$ *LS M q2′*)
by *blast*
moreover have *length* (*tl io*) $\leq$ *k*
using ‹*length io* $\leq$ *Suc k*› **by** *auto*
ultimately have *q2′* $\notin$ *ofsm-table M* (λ*q* . *states M*) *k q1′*
unfolding *ofsm-table-set-observable*[*OF assms*(*1*) ‹*q1′* $\in$ *states M*› *minimise-initial-partition*]
by *blast*
then have *ofsm-table M* (λ*q* . *states M*) *k q1′* $\neq$ *ofsm-table M* (λ*q* . *states M*) *k q2′*
by (*metis* ‹*q2′* $\in$ *FSM.states M*› *ofsm-table-containment*)

moreover have *h-obs M q1 x y = Some q1′*
using ‹(*q1*,*x*,*y*,*q1′*) ∈ *transitions M*› ‹*observable M*› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] *observable-alt-def* **by** *auto*
moreover have *h-obs M q2 x y = Some q2′*
using ‹(*q2*,*x*,*y*,*q2′*) ∈ *transitions M*› ‹*observable M*› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›] *observable-alt-def* **by** *auto*
ultimately have *f1′* (*x*,*y*) = *Some* ((*x*,*y*),(*Some q1′*, *Some q2′*))
unfolding *f1′* **by** *force*

moreover have (*x*,*y*) ∈ *list.set xs*
unfolding *xs*
using *fsm-transition-output*[*OF* ‹(*q1*,*x*,*y*,*q1′*) ∈ *transitions M*›] *outputs-as-list-set*[*of M*]
using *fsm-transition-input*[*OF* ‹(*q1*,*x*,*y*,*q1′*) ∈ *transitions M*›] *inputs-as-list-set*[*of M*]
using *image-iff* **by** *fastforce*
ultimately show *?thesis*
by *blast*
qed
qed

ultimately have *f2 xs* = (*io*,(*b*,*a*))
by *blast*
then show *?thesis*
unfolding *select-diverging-ofsm-table-io.simps f2 f2′ xs Let-def* **by** *auto*
qed

lemma *assemble-distinguishing-sequence-from-ofsm-table-sym* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *ofsm-table M* (λ*q* . *states M*) *k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *k q2*
shows *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 k* = *assemble-distinguishing-sequence-from-of M q2 q1 k*
using *assms*(*2*,*3*,*4*) **proof** (*induction k arbitrary*: *q1 q2*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
obtain *xy a b* **where** *select-diverging-ofsm-table-io M q1 q2* (*Suc k*) = (*xy*,(*a*,*b*))
using *prod-cases3* **by** *blast*
then have *select-diverging-ofsm-table-io M q2 q1* (*Suc k*) = (*xy*,(*b*, *a*))
using *select-diverging-ofsm-table-io-sym*[*OF assms*(*1*) *Suc.prems*] **by** *auto*

consider ∃ *q1′ q2′* . *a* = *Some q1′* ∧ *b* = *Some q2′* | *a* = *None* ∨ *b* = *None*
using *option.exhaust-sel* **by** *auto*
then show *?case* **proof** *cases*
case *1*

then obtain *q1′ q2′* **where** *select-diverging-ofsm-table-io M q1 q2 (Suc k) = (xy,(Some q1′, Some q2′))*
using ‹*select-diverging-ofsm-table-io M q1 q2 (Suc k) = (xy,(a,b))*› **by** *auto*
then have *select-diverging-ofsm-table-io M q2 q1 (Suc k) = (xy,(Some q2′, Some q1′))*
using *select-diverging-ofsm-table-io-sym*[*OF assms*(*1*) *Suc.prems*] **by** *auto*

obtain *x y* **where** *select-diverging-ofsm-table-io M q1 q2 (Suc k) = ((x,y),(h-obs M q1 x y, h-obs M q2 x y))*
and $\bigwedge$ *q1′ q2′ . h-obs M q1 x y = Some q1′ ⟹ h-obs M q2 x y = Some q2′ ⟹ ofsm-table M (λq . states M) k q1′ ≠ ofsm-table M (λq . states M) k q2′*
and *h-obs M q1 x y ≠ None ∨ h-obs M q2 x y ≠ None*
using *select-diverging-ofsm-table-io-Some*(*1*)[*OF assms*(*1*) *Suc.prems*]
by *blast*
then have *xy = (x,y)* **and** *h-obs M q1 x y = Some q1′* **and** *h-obs M q2 x y = Some q2′*
using ‹*select-diverging-ofsm-table-io M q1 q2 (Suc k) = (xy,(Some q1′, Some q2′))*› **by** *auto*
then have *q1′ ∈ states M* **and** *q2′ ∈ states M*
unfolding *h-obs-Some*[*OF assms*(*1*)] **using** *fsm-transition-target* **by** *fast-force+*
moreover have *ofsm-table M (λq . states M) k q1′ ≠ ofsm-table M (λq . states M) k q2′*
using ‹*h-obs M q1 x y = Some q1′*› ‹*h-obs M q2 x y = Some q2′*› ‹$\bigwedge$ *q1′ q2′ . h-obs M q1 x y = Some q1′ ⟹ h-obs M q2 x y = Some q2′ ⟹ ofsm-table M (λq . states M) k q1′ ≠ ofsm-table M (λq . states M) k q2′*›
by *blast*
ultimately have *assemble-distinguishing-sequence-from-ofsm-table M q1′ q2′ k = assemble-distinguishing-sequence-from-ofsm-table M q2′ q1′ k*
using *Suc.IH* **by** *auto*
then show *?thesis*
using ‹*select-diverging-ofsm-table-io M q1 q2 (Suc k) = (xy,(Some q1′, Some q2′))*›
‹*select-diverging-ofsm-table-io M q2 q1 (Suc k) = (xy,(Some q2′, Some q1′))*›
by *auto*
next
case *2*

obtain *x y* **where** *xy = (x,y)*
using *prod.exhaust* **by** *metis*
have *helper*: $\bigwedge$ *f f1 f2 .(case ((x,y),(a,b)) of ((x,y),(Some a′,Some b′)) ⇒ f1 x y a′ b′ | ((x,y),-) ⇒ f2 x y) = f2 x y*
using *2* **by** (*metis case-prod-conv option.case-eq-if*)
have *helper2*: $\bigwedge$ *f f1 f2 .(case ((x,y),(b,a)) of ((x,y),(Some a′,Some b′)) ⇒ f1 x y a′ b′ | ((x,y),-) ⇒ f2 x y) = f2 x y*
using *2* **by** (*metis case-prod-conv option.case-eq-if*)

have *assemble-distinguishing-sequence-from-ofsm-table M q1 q2 (Suc k) = [xy]*
unfolding *assemble-distinguishing-sequence-from-ofsm-table.simps*
‹*select-diverging-ofsm-table-io M q1 q2 (Suc k) = (xy,(a, b))*› ‹*xy = (x,y)*› *helper*
by *simp*
moreover have *assemble-distinguishing-sequence-from-ofsm-table M q2 q1 (Suc k) = [xy]*
unfolding *assemble-distinguishing-sequence-from-ofsm-table.simps*
‹*select-diverging-ofsm-table-io M q2 q1 (Suc k) = (xy,(b, a))*› ‹*xy = (x,y)*› *helper2*
by *simp*
ultimately show *?thesis*
by *simp*
qed
qed

lemma *find-first-distinct-ofsm-table-sym* :
assumes *q1 ∈ FSM.states M*
and *q2 ∈ FSM.states M*
and *ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2*
shows *find-first-distinct-ofsm-table M q1 q2 = find-first-distinct-ofsm-table M q2 q1*
proof −
have *⋀ q1 q2 . q1 ∈ FSM.states M ⟹ q2 ∈ FSM.states M ⟹ ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2 ⟹ find-first-distinct-ofsm-table M q2 q1 < find-first-distinct-ofsm-table M q1 q2 ⟹ False*
proof −
fix *q1 q2* **assume** *q1 ∈ FSM.states M* **and** *q2 ∈ FSM.states M*
and *ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2*
and *find-first-distinct-ofsm-table M q2 q1 < find-first-distinct-ofsm-table M q1 q2*

show *False*
using *find-first-distinct-ofsm-table-is-first(1)[OF ‹q1 ∈ FSM.states M› ‹q2 ∈ FSM.states M› ‹ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2›]*
find-first-distinct-ofsm-table-is-first(2)[OF ‹q1 ∈ FSM.states M› ‹q2 ∈ FSM.states M› ‹ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2› ‹find-first-distinct-ofsm-table M q2 q1 < find-first-distinct-ofsm-table M q1 q2›]
‹*find-first-distinct-ofsm-table M q2 q1 < find-first-distinct-ofsm-table M q1 q2*›
by (*metis ‹ofsm-table-fix M (λq . states M) 0 q1 ≠ ofsm-table-fix M (λq . states M) 0 q2› ‹q1 ∈ FSM.states M› ‹q2 ∈ FSM.states M› find-first-distinct-ofsm-table-gt-is-first-gt(1)*)
qed
then show *?thesis*

using *assms*
by (*metis linorder-neqE-nat*)
qed

lemma *get-distinguishing-sequence-from-ofsm-tables-sym* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2* = *get-distinguishing-sequence-from-ofsm-tables M q2 q1*
proof −

have *ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*
using ‹*minimal M*› **unfolding** *minimal-observable-code*[*OF assms*(*1*)]
using *assms*(*3*,*4*,*5*) **by** *blast*

let *?k* = *find-first-distinct-ofsm-table-gt M q1 q2 0*
have *ofsm-table M* (λ*q* . *states M*) *?k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *?k q2*
using *find-first-distinct-ofsm-table-is-first*(*1*)[*OF assms*(*3*,*4*) ‹*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*›] **.**

show *?thesis*
using *assemble-distinguishing-sequence-from-ofsm-table-sym*[*OF assms*(*1*,*3*,*4*) ‹*ofsm-table M* (λ*q* . *states M*) *?k q1* ≠ *ofsm-table M* (λ*q* . *states M*) *?k q2*›]
unfolding *get-distinguishing-sequence-from-ofsm-tables.simps Let-def*
find-first-distinct-ofsm-table-sym[*OF assms*(*3*,*4*) ‹*ofsm-table-fix M* (λ*q* . *states M*) *0 q1* ≠ *ofsm-table-fix M* (λ*q* . *states M*) *0 q2*›] **.**
qed

21.6.2 Harmonised State Identifiers

fun *add-distinguishing-sequence* :: (*'a*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ ((*'b*×*'c*) *list* × *'a*) × ((*'b*×*'c*) *list* × *'a*) ⇒ (*'b*×*'c*) *prefix-tree* ⇒ (*'b*×*'c*) *prefix-tree* **where**
add-distinguishing-sequence M ((α,*q1*), (β,*q2*)) *t* = *insert empty* (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*)

lemma *add-distinguishing-sequence-distinguishes* :
assumes *observable M*
and *minimal M*
and α ∈ *L M*
and β ∈ *L M*
and *after-initial M* α ≠ *after-initial M* β
shows ∃ *io* ∈ *set* (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α)

(*after-initial M β*) *io*
proof −
have *set* (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = *set* (*insert empty* (*get-distinguishing-sequence-from-ofsm-tables M* (*after-initial M* α) (*after-initial M* β)))
by *auto*
then have *get-distinguishing-sequence-from-ofsm-tables M* (*after-initial M* α) (*after-initial M* β) ∈ *set* (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β))
unfolding *insert-set* **by** *auto*
then show *?thesis*
using *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace*(*1*,*2*)[*OF assms*(*1*,*2*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*)]
by (*meson distinguishes-def*)
qed

lemma *add-distinguishing-sequence-finite* :
finite-tree (*add-distinguishing-sequence M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
unfolding *add-distinguishing-sequence.simps*
using *insert-finite-tree*[*OF empty-finite-tree*] **by** *metis*

fun *get-HSI* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ ′*a* ⇒ (′*b*×′*c*) *prefix-tree*
where
get-HSI M q = *from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* ((≠) *q*) (*states-as-list M*)))

lemma *get-HSI-elem* :
assumes *q2* ∈ *states M*
and *q2* ≠ *q1*
shows *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *set* (*get-HSI M q1*)
proof −
have *q2* ∈ *list.set* (*filter* ((≠) *q1*) (*states-as-list M*))
using *assms* **unfolding** *states-as-list-set*[*of M*,*symmetric*] **by** *auto*
then have ∗:*get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *list.set* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q1 q*′) (*filter* ((≠) *q1*) (*states-as-list M*)))
by *auto*
show *?thesis*
using *from-list-set-elem*[*OF* ∗]
unfolding *get-HSI.simps* **.**
qed

lemma *get-HSI-distinguishes* :
assumes *observable M*
and *minimal M*

and $q1 \in states\ M$ **and** $q2 \in states\ M$ **and** $q1 \neq q2$
shows $\exists$ $io \in set\ (get\text{-}HSI\ M\ q1) \cap set\ (get\text{-}HSI\ M\ q2)$. $distinguishes\ M\ q1\ q2\ io$
proof −

have *get-distinguishing-sequence-from-ofsm-tables M q2 q1* $\in$ *set* (*get-HSI M q1*)
using *get-HSI-elem*[*OF assms*(*4*), *of q1*] *assms*(*5*)
unfolding *get-distinguishing-sequence-from-ofsm-tables-sym*[*OF assms*]
by *metis*
moreover have *get-distinguishing-sequence-from-ofsm-tables M q2 q1* $\in$ *set* (*get-HSI M q2*)
using *get-HSI-elem*[*OF assms*(*3*)] *assms*(*5*) **by** *metis*
moreover have *distinguishes M q1 q2* (*get-distinguishing-sequence-from-ofsm-tables M q2 q1*)
using *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace*(*1*,*2*)[*OF assms*]
unfolding *get-distinguishing-sequence-from-ofsm-tables-sym*[*OF assms*]
unfolding *distinguishes-def*
by *blast*
ultimately show *?thesis*
by *blast*
qed

lemma *get-HSI-finite* :
finite-tree (*get-HSI M q*)
unfolding *get-HSI.simps* **using** *from-list-finite-tree* **by** *metis*

21.6.3 Distinguishing Sets

fun *distinguishing-set* :: ('*a* :: *linorder*, '*b* :: *linorder*, '*c* :: *linorder*) *fsm* $\Rightarrow$ ('*b* $\times$ '*c*) *prefix-tree* **where**
distinguishing-set M = (*let*
pairs = *filter* (λ (*x*,*y*) . $x \neq y$) (*list-ordered-pairs* (*states-as-list M*))
in from-list (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*))

lemma *distinguishing-set-distinguishes* :
assumes *observable M*
and *minimal M*
and $q1 \in states\ M$
and $q2 \in states\ M$
and $q1 \neq q2$
shows $\exists$ $io \in set\ (distinguishing\text{-}set\ M)$. $distinguishes\ M\ q1\ q2\ io$
proof −
define *pairs* **where** *pairs*: *pairs* = *filter* (λ (*x*,*y*) . $x \neq y$) (*list-ordered-pairs* (*states-as-list M*))
then have $*$: *distinguishing-set M* = *from-list* (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*)

by *auto*

have *q1* ∈ *list.set* (*states-as-list M*) **and** *q2* ∈ *list.set* (*states-as-list M*)
unfolding *states-as-list-set* **using** *assms*(*3*,*4*) **by** *blast*+
then have (*q1*,*q2*) ∈ *list.set pairs* ∨ (*q2*,*q1*) ∈ *list.set pairs*
using *list-ordered-pairs-set-containment*[*OF* - - *assms*(*5*)] *assms*(*5*) **unfolding** *pairs* **by** *auto*
then have *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *list.set* (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*)
| *get-distinguishing-sequence-from-ofsm-tables M q2 q1* ∈ *list.set* (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*)
by (*metis image-iff old.prod.case set-map*)
then have *get-distinguishing-sequence-from-ofsm-tables M q1 q2* ∈ *set* (*distinguishing-set M*)
∨ *get-distinguishing-sequence-from-ofsm-tables M q2 q1* ∈ *set* (*distinguishing-set M*)
unfolding ∗ *from-list-set* **by** *blast*
then show *?thesis*
using *get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace*(*1*,*2*)[*OF assms*]
get-distinguishing-sequence-from-ofsm-tables-is-distinguishing-trace(*1*,*2*)[*OF assms*(*1*,*2*,*4*,*3*)] *assms*(*5*)
unfolding *distinguishes-def* **by** *blast*
qed

lemma *distinguishing-set-finite* :
finite-tree (*distinguishing-set M*)
unfolding *distinguishing-set.simps Let-def*
using *from-list-finite-tree* **by** *metis*

function (*domintros*) *intersection-is-distinguishing* :: (′*a*,′*b*,′*c*) *fsm* ⇒ (′*b* × ′*c*) *prefix-tree* ⇒ ′*a* ⇒ (′*b* × ′*c*) *prefix-tree* ⇒ ′*a* ⇒ *bool* **where**
intersection-is-distinguishing M (*PT t1*) *q1* (*PT t2*) *q2* =
(∃ (*x*,*y*) ∈ *dom t1* ∩ *dom t2* .
case h-obs M q1 x y of
None ⇒ *h-obs M q2 x y* ≠ *None* |
Some q1′ ⇒ (*case h-obs M q2 x y of*
None ⇒ *True* |
Some q2′ ⇒ *intersection-is-distinguishing M* (*the* (*t1* (*x*,*y*))) *q1′* (*the* (*t2* (*x*,*y*))) *q2′*))
by *pat-completeness auto*
termination
proof −
{
fix *M* :: (′*a*,′*b*,′*c*) *fsm*

```
    fix t1
    fix q1
    fix t2
    fix q2

    have intersection-is-distinguishing-dom (M, t1,q1, t2,q2)
    proof (induction t1 arbitrary: t2 q1 q2)
      case (PT m1)

      obtain m2 where t2 = PT m2
        by (metis prefix-tree.exhaust)

    have (⋀xy t1' t2' q1' q2'. m1 xy = Some t1' ⟹ intersection-is-distinguishing-dom
(M, t1', q1', t2', q2'))
      proof −
        fix xy t1' t2' q1' q2' assume m1 xy = Some t1'

        then have Some t1' ∈ range m1
          by (metis  range-eqI)

        show intersection-is-distinguishing-dom (M, t1', q1', t2', q2')
          using PT(1)[OF ‹Some t1' ∈ range m1›]
          by simp
      qed

      then show ?case
       using intersection-is-distinguishing.domintros[of q1 M q2 m1 m2] unfolding
‹t2 = PT m2› by blast
    qed
  }
  then show ?thesis by auto
qed



lemma intersection-is-distinguishing-code[code] :
  intersection-is-distinguishing M (MPT t1) q1 (MPT t2) q2 =
    (∃ (x,y) ∈ Mapping.keys t1 ∩ Mapping.keys t2 .
      case h-obs M q1 x y of
        None ⇒ h-obs M q2 x y ≠ None |
        Some q1' ⇒ (case h-obs M q2 x y of
          None ⇒ True |
             Some q2' ⇒ intersection-is-distinguishing M (the (Mapping.lookup t1
(x,y))) q1' (the (Mapping.lookup  t2 (x,y))) q2'))
  unfolding intersection-is-distinguishing.simps MPT-def
  by (simp add: keys-dom-lookup)

lemma intersection-is-distinguishing-correctness :
```

```
  assumes observable M
  and     q1 ∈ states M
  and     q2 ∈ states M
shows intersection-is-distinguishing M t1 q1 t2 q2 = (∃ io . isin t1 io ∧ isin t2 io
∧ distinguishes M q1 q2 io)
  (is ?P1 = ?P2)
proof
  show ?P1 ⟹ ?P2
  proof (induction t1 arbitrary: t2 q1 q2)
    case (PT m1)

    obtain m2 where t2 = PT m2
      using prefix-tree.exhaust by blast
    then obtain x y where (x,y) ∈ dom m1 and (x,y) ∈ dom m2
                          and ∗: case h-obs M q1 x y of
                                 None ⇒ h-obs M q2 x y ≠ None |
                                 Some q1' ⇒ (case h-obs M q2 x y of
                                   None ⇒ True |
                                     Some q2' ⇒ intersection-is-distinguishing M (the
(m1 (x,y))) q1' (the (m2 (x,y))) q2')
      using PT.prems(1) intersection-is-distinguishing.simps by force

    obtain t1' where m1 (x,y) = Some t1'
      using ‹(x,y) ∈ dom m1› by auto
    then have isin (PT m1) [(x,y)]
      by auto

    obtain t2' where m2 (x,y) = Some t2'
      using ‹(x,y) ∈ dom m2› by auto
    then have isin t2 [(x,y)]
      unfolding ‹t2 = PT m2› by auto

    show ?case proof (cases h-obs M q1 x y)
      case None
      then have h-obs M q2 x y ≠ None
        using ∗ by auto
      then have [(x,y)] ∈ LS M q2
        unfolding LS-single-transition h-obs-None[OF ‹observable M›]
        by fastforce
      moreover have [(x,y)] ∉ LS M q1
       using None unfolding LS-single-transition h-obs-None[OF ‹observable M›]
        by auto
      ultimately have distinguishes M q1 q2 [(x,y)]
        unfolding distinguishes-def by blast
      then show ?thesis
        using ‹isin (PT m1) [(x,y)]› ‹isin t2 [(x,y)]› by blast
    next
      case (Some q1')
      then have [(x,y)] ∈ LS M q1
```

```
      unfolding LS-single-transition h-obs-Some[OF ‹observable M›]
      using insert-compr by fastforce


    show ?thesis proof (cases h-obs M q2 x y)
      case None
      then have [(x,y)] ∉ LS M q2
        unfolding LS-single-transition h-obs-None[OF ‹observable M›]
        by auto
      then have distinguishes M q1 q2 [(x,y)]
        using ‹[(x,y)] ∈ LS M q1› unfolding distinguishes-def by blast
      then show ?thesis
        using ‹isin (PT m1) [(x,y)]› ‹isin t2 [(x,y)]› by blast
    next
      case (Some q2')
       then have intersection-is-distinguishing M (the (m1 (x,y))) q1' (the (m2
(x,y))) q2'
        using ‹h-obs M q1 x y = Some q1'› * by auto
      moreover have (the (m1 (x,y))) = t1'
        using ‹m1 (x,y) = Some t1'› by auto
      moreover have (the (m2 (x,y))) = t2'
        using ‹m2 (x,y) = Some t2'› by auto
      ultimately have intersection-is-distinguishing M t1' q1' t2' q2'
        by simp
      then have ∃io. isin t1' io ∧ isin t2' io ∧ distinguishes M q1' q2' io
        using PT.IH[of Some t1' t1' q1' t2' q2']
        by (metis ‹m1 (x, y) = Some t1'› option.set-intros rangeI)
      then obtain io where isin t1' io
                    and isin t2' io
                    and distinguishes M q1' q2' io
        by blast

      have isin (PT m1) ((x,y)#io)
        using ‹m1 (x, y) = Some t1'› ‹isin t1' io› by auto
      moreover have isin t2 ((x,y)#io)
        using ‹t2 = PT m2› ‹m2 (x, y) = Some t2'› ‹isin t2' io› by auto
      moreover have distinguishes M q1 q2 ((x,y)#io)
          using h-obs-language-iff[OF ‹observable M›, of x y io  q1] unfolding
‹h-obs M q1 x y = Some q1'›
          using h-obs-language-iff[OF ‹observable M›, of x y io  q2] unfolding
Some
        using ‹distinguishes M q1' q2' io›
        unfolding distinguishes-def
        by auto
      ultimately show ?thesis
        by blast
    qed
  qed
 qed
```

```
show ?P2 ⟹ ?P1
proof −
  assume ?P2
  then obtain io where isin t1 io
                  and isin t2 io
                  and distinguishes M q1 q2 io
    by blast
  then show ?P1
  using assms(2,3) proof (induction io arbitrary: t1 t2 q1 q2)
    case Nil
    then have [] ∈ LS M q1 ∩ LS M q2
      by auto
    then have ¬ distinguishes M q1 q2 []
      unfolding distinguishes-def by blast
    then show ?case
      using ‹distinguishes M q1 q2 []› by simp
  next
    case (Cons a io)

    obtain x y where a = (x,y)
      by fastforce

    obtain m1 where t1 = PT m1
      using prefix-tree.exhaust by blast
    obtain t1' where m1 (x,y) = Some t1'
                and isin t1' io
      using ‹isin t1 (a # io)› unfolding ‹a = (x,y)› ‹t1 = PT m1› isin.simps
      using case-optionE by blast

    obtain m2 where t2 = PT m2
      using prefix-tree.exhaust by blast
    obtain t2' where m2 (x,y) = Some t2'
                and isin t2' io
      using ‹isin t2 (a # io)› unfolding ‹a = (x,y)› ‹t2 = PT m2› isin.simps
      using case-optionE by blast
    then have (x,y) ∈ dom m1 ∩ dom m2
      using ‹m1 (x,y) = Some t1'› by auto


    show ?case proof (cases h-obs M q1 x y)
      case None
      then have [(x,y)] ∉ LS M q1
        unfolding LS-single-transition h-obs-None[OF ‹observable M›]
        by auto
      then have a#io ∉ LS M q1
        unfolding ‹a = (x,y)›
        by (metis None assms(1) h-obs-language-iff option.distinct(1))
      then have a#io ∈ LS M q2
```

using ‹*distinguishes M q1 q2 (a#io)*› **unfolding** *distinguishes-def* **by** *blast*
then have [(*x*,*y*)] ∈ *LS M q2*
unfolding ‹*a* = (*x*,*y*)›
using *language-prefix*
by (*metis append-Cons append-Nil*)
then have *h-obs M q2 x y* ≠ *None*
unfolding *h-obs-None*[*OF* ‹*observable M*›] *LS-single-transition* **by** *force*
then show *?thesis*
using *None* ‹(*x*,*y*) ∈ *dom m1* ∩ *dom m2*› **unfolding** ‹*t1* = *PT m1*› ‹*t2* = *PT m2*›
by *force*
next
case (*Some q1'*)
then have [(*x*,*y*)] ∈ *LS M q1*
unfolding *LS-single-transition h-obs-Some*[*OF* ‹*observable M*›]
by (*metis Some assms*(*1*) *fst-conv h-obs-None option.distinct*(*1*) *snd-conv*)

show *?thesis* **proof** (*cases h-obs M q2 x y*)
case *None*
then show *?thesis*
using *Some* ‹(*x*,*y*) ∈ *dom m1* ∩ *dom m2*› **unfolding** ‹*t1* = *PT m1*› ‹*t2* = *PT m2*›
unfolding *intersection-is-distinguishing.simps*
by (*metis* (*no-types*, *lifting*) *case-prodI option.case-eq-if option.distinct*(*1*))

next
case (*Some q2'*)

have *distinguishes M q1' q2' io*
using *h-obs-language-iff*[*OF* ‹*observable M*›, *of x y io q1*] **unfolding** ‹*h-obs M q1 x y* = *Some q1'*›
using *h-obs-language-iff*[*OF* ‹*observable M*›, *of x y io q2*] **unfolding** *Some*
using ‹*distinguishes M q1 q2 (a#io)*› **unfolding** ‹*a* = (*x*,*y*)› *distinguishes-def*
by *blast*
moreover have *q1'* ∈ *states M* **and** *q2'* ∈ *states M*
using *Some* ‹*h-obs M q1 x y* = *Some q1'*› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
using *fsm-transition-target*[**where** *M*=*M*]
by *fastforce*+
ultimately have *intersection-is-distinguishing M t1' q1' t2' q2'*
using *Cons.IH*[*OF* ‹*isin t1' io*› ‹*isin t2' io*›]
by *auto*
then show *?thesis*
using ‹(*x*,*y*) ∈ *dom m1* ∩ *dom m2*› *Some* ‹*h-obs M q1 x y* = *Some q1'*›
unfolding ‹*t1* = *PT m1*› ‹*t2* = *PT m2*›
unfolding *intersection-is-distinguishing.simps*

by (*metis* (*no-types*, *lifting*) ‹*m1* (*x*, *y*) = *Some t1'*› ‹*m2* (*x*, *y*) = *Some t2'*› *case-prodI option.case-eq-if option.distinct*(*1*) *option.sel*)
qed
qed
qed
qed
qed

fun *contains-distinguishing-trace* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
contains-distinguishing-trace M T q1 q2 = *intersection-is-distinguishing M T q1 T q2*

lemma *contains-distinguishing-trace-code*[*code*] :
contains-distinguishing-trace M (*MPT t1*) *q1 q2* =
(∃ (*x*,*y*) ∈ *Mapping.keys t1*.
case h-obs M q1 x y of
None ⇒ *h-obs M q2 x y* ≠ *None* |
Some q1' ⇒ (*case h-obs M q2 x y of*
None ⇒ *True* |
Some q2' ⇒ *contains-distinguishing-trace M* (*the* (*Mapping.lookup t1* (*x*,*y*))) *q1' q2'*))
unfolding *intersection-is-distinguishing.simps MPT-def*
by (*simp add*: *keys-dom-lookup*)

lemma *contains-distinguishing-trace-correctness* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *contains-distinguishing-trace M t q1 q2* = (∃ *io* . *isin t io* ∧ *distinguishes M q1 q2 io*)
using *intersection-is-distinguishing-correctness*[*OF assms*]
by *simp*

fun *distinguishing-set-reduced* :: ('*a* :: *linorder*, '*b* :: *linorder*, '*c* :: *linorder*) *fsm* ⇒ ('*b* × '*c*) *prefix-tree* **where**
distinguishing-set-reduced M = (*let*
pairs = *filter* (λ (*q*,*q'*) . *q* ≠ *q'*) (*list-ordered-pairs* (*states-as-list M*));
handlePair = (λ *W* (*q*,*q'*) . *if contains-distinguishing-trace M W q q'*
then W
else insert W (*get-distinguishing-sequence-from-ofsm-tables M q q'*))
in foldl handlePair empty pairs)

lemma *distinguishing-set-reduced-distinguishes* :
assumes *observable M*

and *minimal M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows ∃ *io* ∈ *set* (*distinguishing-set-reduced M*) . *distinguishes M q1 q2 io*
proof −
define *pairs* **where** *pairs*: *pairs* = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*))

define *handlePair* **where** *handlePair* = (λ *W* (*q*,*q′*) . *if contains-distinguishing-trace M W q q′*
then W
else insert W (*get-distinguishing-sequence-from-ofsm-tables M q q′*))

have *distinguishing-set-reduced M* = *foldl handlePair empty pairs*
unfolding *distinguishing-set-reduced.simps handlePair-def pairs Let-def* **by** *metis*

have *handlePair-subset*: ⋀ *W q q′* . *set W* ⊆ *set* (*handlePair W* (*q*,*q′*))
unfolding *handlePair-def*
using *insert-set* **unfolding** *case-prod-conv*
by (*metis* (*mono-tags*) *Un-upper1 order-refl*)

have *q1* ∈ *list.set* (*states-as-list M*) **and** *q2* ∈ *list.set* (*states-as-list M*)
unfolding *states-as-list-set* **using** *assms*(*3*,*4*) **by** *blast*+
then have (*q1*,*q2*) ∈ *list.set pairs* ∨ (*q2*,*q1*) ∈ *list.set pairs*
using *list-ordered-pairs-set-containment*[*OF* - - *assms*(*5*)] *assms*(*5*) **unfolding** *pairs* **by** *auto*
moreover have ⋀ *pairs′* . *list.set pairs′* ⊆ *list.set pairs* ⟹ (*q1*,*q2*) ∈ *list.set pairs′* ∨ (*q2*,*q1*) ∈ *list.set pairs′* ⟹ (∃ *io* ∈ *set* (*foldl handlePair empty pairs′*) . *distinguishes M q1 q2 io*)
proof −
fix *pairs′* **assume** *list.set pairs′* ⊆ *list.set pairs* **and** (*q1*,*q2*) ∈ *list.set pairs′* ∨ (*q2*,*q1*) ∈ *list.set pairs′*
then show (∃ *io* ∈ *set* (*foldl handlePair empty pairs′*) . *distinguishes M q1 q2 io*)
proof (*induction pairs′ rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc qq qqs*)

define *W* **where** *W* = *foldl handlePair empty qqs*
have *foldl handlePair empty* (*qqs*@[*qq*]) = *handlePair W qq*
unfolding *W-def* **by** *auto*
then have *W-subset*: *set W* ⊆ *set* (*foldl handlePair empty* (*qqs*@[*qq*]))

```
      by (metis handlePair-subset prod.collapse)


    have handlePair-sym : handlePair W (q1,q2) = handlePair W (q2,q1)
      unfolding handlePair-def case-prod-conv
      unfolding contains-distinguishing-trace-correctness[OF assms(1,3,4)] con-
tains-distinguishing-trace-correctness[OF assms(1,4,3)]
      unfolding get-distinguishing-sequence-from-ofsm-tables-sym[OF assms]
      using distinguishes-sym
      by metis

    show ?case proof (cases qq = (q1,q2) ∨ qq = (q2,q1))
      case True
      then have foldl handlePair empty (qqs@[qq]) = handlePair W (q1,q2)
        unfolding ‹foldl handlePair empty (qqs@[qq]) = handlePair W qq›
        using handlePair-sym
        by auto

      show ?thesis proof (cases contains-distinguishing-trace M W q1 q2)
        case True
        then show ?thesis
          unfolding contains-distinguishing-trace-correctness[OF assms(1,3,4)]
          using W-subset
          by auto
      next
        case False
      then have foldl handlePair empty (qqs@[qq]) = insert W (get-distinguishing-sequence-from-ofsm-tables
M q1 q2)
          unfolding ‹foldl handlePair empty (qqs@[qq]) = handlePair W (q1,q2)›
          unfolding handlePair-def case-prod-conv
          by auto
          then have get-distinguishing-sequence-from-ofsm-tables M q1 q2 ∈ set
(foldl handlePair empty (qqs@[qq]))
          using insert-isin
          by metis
        then show ?thesis
       using get-distinguishing-sequence-from-ofsm-tables-distinguishes[OF assms]
          by blast
      qed
    next
      case False
      then have (q1, q2) ∈ list.set qqs ∨ (q2, q1) ∈ list.set qqs
        using snoc.prems by auto
      then show ?thesis using snoc W-subset unfolding W-def
        by (meson dual-order.trans list-prefix-subset subsetD)
    qed
  qed
qed
ultimately show ?thesis
```

unfolding ‹*distinguishing-set-reduced M = foldl handlePair empty pairs*›
by *blast*
qed

lemma *distinguishing-set-reduced-finite* :
finite-tree (*distinguishing-set-reduced M*)
proof −
define *pairs* **where** *pairs*: *pairs* = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*))

define *handlePair* **where** *handlePair* = (λ *W* (*q*,*q*′) . *if contains-distinguishing-trace M W q q*′
then W
else insert W (*get-distinguishing-sequence-from-ofsm-tables M q q*′))

have *distinguishing-set-reduced M = foldl handlePair empty pairs*
unfolding *distinguishing-set-reduced.simps handlePair-def pairs Let-def* **by** *metis*

show *?thesis*
unfolding ‹*distinguishing-set-reduced M = foldl handlePair empty pairs*›
proof (*induction pairs rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *empty-finite-tree* **by** *auto*
next
case (*snoc qq qqs*)
define *W* **where** *W = foldl handlePair empty qqs*
have *foldl handlePair empty* (*qqs*@[*qq*]) = *handlePair W qq*
unfolding *W-def* **by** *auto*

have *finite-tree W*
using *snoc W-def* **by** *auto*
then show *?case*
unfolding ‹*foldl handlePair empty* (*qqs*@[*qq*]) = *handlePair W qq*›
unfolding *handlePair-def*
using *insert-finite-tree*[*of W*]
by (*simp add*: *case-prod-unfold*)
qed
qed

fun *add-distinguishing-set* :: (′*a* :: *linorder*, ′*b* :: *linorder*, ′*c* :: *linorder*) *fsm* ⇒ ((′*b*×′*c*) *list* × ′*a*) × ((′*b*×′*c*) *list* × ′*a*) ⇒ (′*b*×′*c*) *prefix-tree* ⇒ (′*b*×′*c*) *prefix-tree*
where
add-distinguishing-set M - t = distinguishing-set M

lemma *add-distinguishing-set-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in L\ M$
and $\beta \in L\ M$
and *after-initial M* $\alpha \neq$ *after-initial M* β
shows $\exists$ *io* $\in$ *set* (*add-distinguishing-set M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
using *distinguishing-set-distinguishes*[*OF assms*(*1*,*2*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*)]
by *force*

lemma *add-distinguishing-set-finite* :
finite-tree ((*add-distinguishing-set M*) *x t*)
unfolding *add-distinguishing-set.simps distinguishing-set.simps Let-def*
using *from-list-finite-tree*
by *simp*

21.7 Transition Sorting

definition *sort-unverified-transitions-by-state-cover-length* :: ('*a* :: *linorder*,'*b* :: *linorder*,'*c* :: *linorder*) *fsm* $\Rightarrow$ ('*a*,'*b*,'*c*) *state-cover-assignment* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition list* $\Rightarrow$ ('*a*,'*b*,'*c*) *transition list* **where**
sort-unverified-transitions-by-state-cover-length M V ts = (*let*
default-weight = *2* $*$ *size M*;
weights = *mapping-of* (*map* (λ*t* . (*t*, *length* (*V* (*t-source t*)) + *length* (*V* (*t-target t*)))) *ts*);
weight = (λ*q* . *case Mapping.lookup weights q of Some w* $\Rightarrow$ *w* | *None* $\Rightarrow$ *default-weight*)
in mergesort-by-rel (λ *t1 t2* . *weight t1* $\leq$ *weight t2*) *ts*)

lemma *sort-unverified-transitions-by-state-cover-length-retains-set* :
List.set xs = *List.set* (*sort-unverified-transitions-by-state-cover-length M1* (*get-state-cover M1*) *xs*)
unfolding *sort-unverified-transitions-by-state-cover-length-def Let-def*
unfolding *set-mergesort-by-rel*
by *simp*

end

22 Test Suites for Language Equivalence

This file introduces a type for test suites represented as a prefix tree in which each IO-pair is additionally labeled by a boolean value representing whether

the IO-pair should be exhibited by the SUT in order to pass the test suite.

theory *Test-Suite-Representations*
imports *../Minimisation ../Prefix-Tree*
begin

type-synonym $('b,'c)$ *test-suite* $= (('b \times 'c) \times bool)$ *prefix-tree*

function (*domintros*) *test-suite-from-io-tree* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('b \times 'c)$ *prefix-tree* $\Rightarrow$ $('b,'c)$ *test-suite* **where**
  test-suite-from-io-tree M q (PT m) = PT (λ ((x,y),b) . case m (x,y) of
    None ⇒ None |
    Some t ⇒ (case h-obs M q x y of
      None ⇒ (if b then None else Some empty) |
      Some q′ ⇒ (if b then Some (test-suite-from-io-tree M q′ t) else None)))
  by *pat-completeness auto*
termination
proof −
  {
    fix *M* :: $('a,'b,'c)$ *fsm*
    fix *q t*

    have *test-suite-from-io-tree-dom (M,q,t)*
    proof (*induction t arbitrary*: *M q*)
      case (*PT m*)
      then have $\bigwedge$*x y t q′. m (x, y) = Some t ⟹ test-suite-from-io-tree-dom (M, q′, t)*
        by *blast*
      then show *?case*
        using *test-suite-from-io-tree.domintros*[*of m q M*] **by** *auto*
    qed
  }
  then show *?thesis* **by** *auto*
qed

22.1 Transforming an IO-prefix-tree to a test suite

lemma *test-suite-from-io-tree-set* :
  assumes *observable M*
    and *q ∈ states M*
  shows (*set (test-suite-from-io-tree M q t)) = ((λ xs . map (λ x . (x,True)) xs) ‘ (set t ∩ LS M q))*
    ∪ ((λ xs . (map (λ x . (x,True)) (butlast xs))@[(last xs,False)]) ‘ {xs@[x] | xs x . xs ∈ set t ∩ LS M q ∧ xs@[x] ∈ set t − LS M q})
  (**is** *?S1 q t = ?S2 q t*)
proof
  show *?S1 q t ⊆ ?S2 q t*

```
proof
  fix xs assume xs ∈ ?S1 q t
  then have isin (test-suite-from-io-tree M q t) xs
    by auto
  then show xs ∈ ?S2 q t
    using ‹q ∈ states M›
  proof (induction xs arbitrary: q t)
    case Nil

    have [] ∈ set t
      using Nil.prems(1) set-Nil by auto
    moreover have [] ∈ LS M q
      using Nil.prems(2) by auto
    ultimately show ?case
      by auto
  next
    case (Cons x' xs)
    moreover obtain x y b where x' = ((x,y),b)
      by (metis surj-pair)
    moreover obtain m where t = PT m
      by (meson prefix-tree.exhaust)
    ultimately have isin (test-suite-from-io-tree M q (PT m)) (((x,y),b) # xs)
      by auto

    let ?fi = λ t . (case h-obs M q x y of
                 None ⇒ (if b then None else Some empty) |
                   Some q' ⇒ (if b then Some (test-suite-from-io-tree M q' t) else
None))
    let ?fo = case m (x,y) of
              None ⇒ None |
              Some t ⇒ ?fi t

    obtain tst where ?fo = Some tst
      using ‹isin (test-suite-from-io-tree M q (PT m)) (((x,y),b) # xs)›
      unfolding test-suite-from-io-tree.simps isin.simps by force
    then have isin tst xs
      using ‹isin (test-suite-from-io-tree M q (PT m)) (((x,y),b) # xs)›
      by auto

    obtain t' where m (x,y) = Some t'
                  and ?fi t' = Some tst
      using ‹?fo = Some tst›
    by (metis (no-types, lifting) option.case-eq-if option.collapse option.distinct(1))
    then consider h-obs M q x y = None ∧ ¬b ∧ tst = empty |
                 ∃ q' . h-obs M q x y = Some q' ∧ b ∧ tst = test-suite-from-io-tree
M q' t'
      unfolding option.case-eq-if
      using option.collapse[of h-obs M q x y]
      using option.distinct(1) option.inject
```

by *metis*
then show *?case* **proof** *cases*
case *1*
then have *h-obs M q x y = None* **and** *b = False* **and** *tst = empty*
by *auto*

have *isin empty xs*
using ‹*isin tst xs*› ‹*tst = empty*› **by** *auto*
then have *xs = []*
using *set-empty* **by** *auto*
then have ∗: *x′#xs = [((x,y),b)]*
using ‹*x′ = ((x,y),b)*› **by** *auto*

have *[] ∈ LS M q*
using ‹*q ∈ states M*› **by** *auto*
moreover have *[] ∈ set t*
using *set-Nil* **by** *auto*
moreover have *[(x,y)] ∉ LS M q*
using ‹*h-obs M q x y = None*› **unfolding** *h-obs-None*[*OF* ‹*observable M*›]
by *auto*
moreover have *isin t [(x,y)]*
unfolding ‹*t = PT m*› *isin.simps* **using** ‹*m (x,y) = Some t′*›
using *isin.elims*(*3*) **by** *auto*
ultimately have *[(x,y)] ∈ {xs @ [x] |xs x. xs ∈ Prefix-Tree.set t ∩ LS M q ∧ xs @ [x] ∈ Prefix-Tree.set t − LS M q}*
by *auto*
moreover have *(x′#xs) = ((λ xs . (map (λ x . (x,True)) (butlast xs))@[(last xs,False)]) [(x,y)])*
unfolding ∗ ‹*b = False*›
by *auto*
ultimately show *?thesis*
by *blast*
next
case *2*
then obtain *q′* **where** *h-obs M q x y = Some q′*
b = True
tst = test-suite-from-io-tree M q′ t′
by *blast*

have *p1*: *isin (test-suite-from-io-tree M q′ t′) xs*
using ‹*isin tst xs*› ‹*tst = test-suite-from-io-tree M q′ t′*› **by** *auto*
have *p2*: *q′ ∈ states M*
using ‹*h-obs M q x y = Some q′*› *fsm-transition-target* **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
by *fastforce*

have *xs ∈ ?S2 q′ t′*
using *Cons.IH*[*OF p1 p2*] **.**
then consider (*a*) *xs ∈ map (λx. (x, True)) ‘ (Prefix-Tree.set t′ ∩ LS M*

q') |
(b) $xs \in (\lambda xs.\ map\ (\lambda x.\ (x,\ True))\ (butlast\ xs)\ @\ [(last\ xs,\ False)])$
‘ $\{xs\ @\ [x]\ |xs\ x.\ xs \in Prefix\text{-}Tree.set\ t' \cap LS\ M\ q' \wedge xs\ @\ [x] \in Prefix\text{-}Tree.set\ t' - LS\ M\ q'\}$
by *blast*
then show *?thesis* **proof** *cases*
case *a*
then obtain xs' **where** $xs' \in set\ t'$ **and** $xs' \in LS\ M\ q'$
and $xs = map\ (\lambda x.\ (x,\ True))\ xs'$
by *auto*

have $(x,y)\#xs' \in set\ t$
using ‹$xs' \in set\ t'$› ‹$m\ (x,y) = Some\ t'$› **unfolding** ‹$t = PT\ m$› **by** *auto*
moreover have $(x,y)\#xs' \in LS\ M\ q$
using ‹$h\text{-}obs\ M\ q\ x\ y = Some\ q'$› ‹$xs' \in LS\ M\ q'$› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
using *LS-prepend-transition*[*of* $(q,x,y,q')\ M\ xs'$] **by** *auto*
moreover have $x'\#xs = map\ (\lambda x.\ (x,\ True))\ ((x,y)\#xs')$
unfolding ‹$x' = ((x,y),b)$› ‹$b = True$› ‹$xs = map\ (\lambda x.\ (x,\ True))\ xs'$› **by** *auto*
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*) *Int-iff UnI1 image-eqI*)
next
case *b*
then obtain xs' **where** $xs' \in \{xs\ @\ [x]\ |xs\ x.\ xs \in Prefix\text{-}Tree.set\ t' \cap LS\ M\ q' \wedge xs\ @\ [x] \in Prefix\text{-}Tree.set\ t' - LS\ M\ q'\}$
and $xs = (\lambda xs.\ map\ (\lambda x.\ (x,\ True))\ (butlast\ xs)\ @\ [(last\ xs,\ False)])\ xs'$
by *blast*
moreover obtain *bl l* **where** $xs' = bl\ @\ [l]$
using *calculation* **by** *blast*
ultimately have $bl \in set\ t'$ **and** $bl \in LS\ M\ q'$ **and** $bl\ @\ [l] \in set\ t'$ **and** $bl@[l] \notin LS\ M\ q'$
and $xs = (\lambda xs.\ map\ (\lambda x.\ (x,\ True))\ (butlast\ xs)\ @\ [(last\ xs,\ False)])\ (bl@[l])$
by *auto*

have $(x,y)\#bl \in set\ t$
using ‹$bl \in set\ t'$› ‹$m\ (x,y) = Some\ t'$› **unfolding** ‹$t = PT\ m$› **by** *auto*
moreover have $(x,y)\#bl \in LS\ M\ q$
using ‹$h\text{-}obs\ M\ q\ x\ y = Some\ q'$› ‹$bl \in LS\ M\ q'$› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
using *LS-prepend-transition*[*of* $(q,x,y,q')\ M\ bl$] **by** *auto*
moreover have $(x,y)\#(bl\ @\ [l]) \in set\ t$
using ‹$bl\ @\ [l] \in set\ t'$› ‹$m\ (x,y) = Some\ t'$› **unfolding** ‹$t = PT\ m$› **by** *auto*
moreover have $(x,y)\#(bl@[l]) \notin LS\ M\ q$
using ‹$h\text{-}obs\ M\ q\ x\ y = Some\ q'$› ‹$bl@[l] \notin LS\ M\ q'$› **unfolding**

```
h-obs-Some[OF ‹observable M›]
                using observable-language-transition-target[OF ‹observable M›, of
(q,x,y,q') bl@[l]]
            unfolding fst-conv snd-conv
            by blast
          ultimately have (x,y)#bl@[l] ∈ {xs @ [x] |xs x. xs ∈ Prefix-Tree.set t ∩
LS M q ∧ xs @ [x] ∈ Prefix-Tree.set t − LS M q}
            by fastforce
          moreover have x'#xs = (λxs. map (λx. (x, True)) (butlast xs) @ [(last
xs, False)]) ((x,y)#(bl@[l]))
            unfolding ‹x' = ((x,y),b)› ‹b = True› ‹xs = (λxs. map (λx. (x, True))
(butlast xs) @ [(last xs, False)]) (bl@[l])› by auto
          ultimately show ?thesis
            by fast
        qed
      qed
    qed
  qed

  show ?S2 q t ⊆ ?S1 q t
  proof
    fix xs assume xs ∈ ?S2 q t
    then consider xs ∈ map (λx. (x, True)) ` (set t ∩ LS M q) |
                  xs ∈ (λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)]) `
{xs @ [x] |xs x. xs ∈ Prefix-Tree.set t ∩ LS M q ∧ xs @ [x] ∈ Prefix-Tree.set t −
LS M q}
      by blast
    then show xs ∈ ?S1 q t proof cases
      case 1
      then show ?thesis
        using ‹q ∈ states M›
      proof (induction xs arbitrary: q t)
        case Nil
        then show ?case using set-Nil by auto
      next
        case (Cons x' xs)
        obtain xs'' where xs'' ∈ set t and xs'' ∈ LS M q
                      and x'#xs = map (λx. (x, True)) xs''
          using Cons.prems(1)
          by (meson IntD1 IntD2 imageE)

        then obtain x y xs' where (x,y)#xs' ∈ set t and (x,y)#xs' ∈ LS M q
                          and x'#xs = map (λx. (x, True)) ((x,y)#xs')
          by force
        then have x' = ((x,y),True) and xs = map (λx. (x, True)) xs'
          by auto

        obtain m where t = PT m
          by (meson prefix-tree.exhaust)
```

```
      have isin (PT m) ((x,y)#xs')
        using ‹(x,y)#xs' ∈ set t› unfolding ‹t = PT m› by auto
      then obtain t' where m (x,y) = Some t'
                    and isin t' xs'
        by (metis case-optionE isin.simps(2))

      have [(x,y)] ∈ LS M q
        using ‹(x,y)#xs' ∈ LS M q› language-prefix[of [(x,y)] xs' M q]
        by simp
      then obtain q' where h-obs M q x y = Some q'
     using h-obs-None[OF ‹observable M›, of q x y] unfolding LS-single-transition
by auto

      have isin (test-suite-from-io-tree M q (PT m)) (((x,y),True)#xs)
            = isin (test-suite-from-io-tree M q' t') (xs)
        using ‹m (x,y) = Some t'› ‹h-obs M q x y = Some q'› by auto
      then have *: x' # xs ∈ set (test-suite-from-io-tree M q t)
                  = (xs ∈ set (test-suite-from-io-tree M q' t'))
        unfolding ‹t = PT m› ‹x' = ((x,y),True)› by auto

      have xs' ∈ LS M q'
         using ‹h-obs M q x y = Some q'› unfolding h-obs-Some[OF ‹observable
M›, of q x y]
           using ‹(x,y)#xs' ∈ LS M q› observable-language-transition-target[OF
‹observable M›] by force
      moreover have xs' ∈ set t'
        using ‹isin t' xs'› by auto
      ultimately have p1: xs ∈ map (λx. (x, True)) ` (set t' ∩ LS M q')
        unfolding ‹xs = map (λx. (x, True)) xs'› by auto

      have p2: q' ∈ states M
           using ‹h-obs M q x y = Some q'› fsm-transition-target unfolding
h-obs-Some[OF ‹observable M›]
        by fastforce

      show ?case
        using Cons.IH[OF p1 p2] unfolding * .
    qed
  next
    case 2
    then show ?thesis
      using ‹q ∈ states M›
    proof (induction xs arbitrary: q t)
      case Nil
      then show ?case using set-Nil by auto
    next
      case (Cons x' xs)
      then obtain xsT where x'#xs = (λxs. map (λx. (x, True)) (butlast xs) @
```

[(*last xs*, *False*)]) *xsT*
and *xsT* ∈ {*xs* @ [*x*] |*xs x*. *xs* ∈ *Prefix-Tree.set t* ∩ *LS M q* ∧ *xs* @ [*x*] ∈ *Prefix-Tree.set t* − *LS M q*}
by *blast*
moreover obtain *bl l* **where** *xsT* = *bl* @ [*l*]
using *calculation* **by** *auto*
ultimately have *bl* ∈ *set t* **and** *bl* ∈ *LS M q* **and** *bl* @ [*l*] ∈ *set t* **and** *bl*@[*l*] ∉ *LS M q*
and *x′*#*xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl*@[*l*])
by *auto*

obtain *m* **where** *t* = *PT m*
by (*meson prefix-tree.exhaust*)

show *?case* **proof** (*cases xs*)
case *Nil*
then have *x′* = (*l*,*False*) **and** *bl* = []
using ‹*x′*#*xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl*@[*l*])›
by *auto*
moreover obtain *x y* **where** *l* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
ultimately have [(*x*,*y*)] ∈ *set t* **and** [(*x*,*y*)] ∉ *LS M q*
using ‹*bl* @ [*l*] ∈ *set t*› ‹*bl*@[*l*] ∉ *LS M q*› **by** *auto*

obtain *t′* **where** *m* (*x*,*y*) = *Some t′*
using ‹[(*x*,*y*)] ∈ *set t*› **unfolding** ‹*t* = *PT m*› **by** *force*
moreover have *h-obs M q x y* = *None*
using ‹[(*x*,*y*)] ∉ *LS M q*› **unfolding** *h-obs-None*[*OF* ‹*observable M*›] *LS-single-transition* **by** *auto*
ultimately have *isin* (*test-suite-from-io-tree M q* (*PT m*)) (*x′*#*xs*) = *isin empty* []
unfolding *Nil* ‹*x′* = (*l*,*False*)› *test-suite-from-io-tree.simps isin.simps* ‹*l* = (*x*,*y*)›
by (*simp add*: *Prefix-Tree.empty-def*)
then show *?thesis*
using *set-Nil* **unfolding** ‹*t* = *PT m*› **by** *auto*
next
case (*Cons x″ xs″*)
then obtain *x y bl′* **where** *bl* = (*x*,*y*)#*bl′*
using ‹*x′*#*xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl*@[*l*])›
by (*metis append.left-neutral butlast-snoc list.inject list.simps*(*8*) *neq-Nil-conv surj-pair*)
then have *x′* = ((*x*,*y*),*True*)
and *xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl′*@[*l*])
using ‹*x′*#*xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)])

(*bl*@[*l*])›
by *auto*

have *isin* (*PT m*) ((*x*,*y*)#(*bl*′@[*l*]))
using ‹*bl*@[*l*] ∈ *set t*› **unfolding** ‹*bl* = (*x*,*y*)#*bl*′› ‹*t* = *PT m*› **by** *auto*
then obtain *t*′ **where** *m* (*x*,*y*) = *Some t*′
and *isin t*′ (*bl*′@[*l*])
unfolding *isin.simps*
using *case-optionE* **by** *blast*

have [(*x*,*y*)] ∈ *LS M q*
using ‹*bl* ∈ *LS M q*› *language-prefix*[*of* [(*x*,*y*)] *bl*′ *M q*] **unfolding** ‹*bl* = (*x*,*y*)#*bl*′› **by** *auto*
then obtain *q*′ **where** *h-obs M q x y* = *Some q*′
using *h-obs-None*[*OF* ‹*observable M*›] **unfolding** *LS-single-transition* **by** *force*
then have *p2*: *q*′ ∈ *states M*
using *fsm-transition-target* **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
by *fastforce*

have *bl*′ ∈ *set t*′
using ‹*isin t*′ (*bl*′@[*l*])› *isin-prefix* **by** *auto*
moreover have *bl*′ ∈ *LS M q*′
using ‹*h-obs M q x y* = *Some q*′› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
using ‹*bl* ∈ *LS M q*› *observable-language-transition-target*[*OF* ‹*observable M*›]
unfolding ‹*bl* = (*x*,*y*)#*bl*′› **by** *force*
moreover have *bl*′@[*l*] ∈ *set t*′
using ‹*isin t*′ (*bl*′@[*l*])› **by** *auto*
moreover have *bl*′@[*l*] ∉ *LS M q*′
proof −
have (*x*, *y*) # (*bl*′ @ [*l*]) ∉ *LS M q*
using ‹*bl*@[*l*] ∉ *LS M q*› **unfolding** ‹*bl* = (*x*,*y*)#*bl*′› **by** *auto*
then show *?thesis*
using ‹*h-obs M q x y* = *Some q*′› **unfolding** *h-obs-Some*[*OF* ‹*observable M*›]
using *LS-prepend-transition*[*of* (*q*,*x*,*y*,*q*′) *M bl*′@[*l*]]
unfolding ‹*bl* = (*x*,*y*)#*bl*′› *fst-conv snd-conv* **by** *blast*
qed
ultimately have (*bl*′@[*l*]) ∈ {*xs* @ [*x*] |*xs x*. *xs* ∈ *Prefix-Tree.set t*′ ∩ *LS M q*′ ∧ *xs* @ [*x*] ∈ *Prefix-Tree.set t*′ − *LS M q*′}
by *blast*
moreover have *xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl*′@[*l*])
using ‹*x*′#*xs* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*bl*@[*l*])›
unfolding *Nil* ‹*bl* = (*x*,*y*)#*bl*′› **by** *auto*
ultimately have *xs* ∈ *Prefix-Tree.set* (*test-suite-from-io-tree M q*′ *t*′)

```
          using Cons.IH[of t', OF - p2] by blast
        then have isin (test-suite-from-io-tree M q t) (x'#xs)
          unfolding ‹x' = ((x,y),True)› ‹t = PT m›
          unfolding test-suite-from-io-tree.simps isin.simps
          using ‹m (x,y) = Some t'› ‹h-obs M q x y = Some q'› by auto
        then show ?thesis
          by auto
      qed
    qed
  qed
 qed
qed


function (domintros) passes-test-suite :: ('a,'b,'c) fsm ⇒ 'a ⇒ ('b,'c) test-suite ⇒
bool where
  passes-test-suite M q (PT m) = (∀ xyb ∈ dom m . case h-obs M q (fst (fst xyb))
(snd (fst xyb)) of
      None ⇒ ¬(snd xyb) |
      Some q' ⇒ snd xyb ∧ passes-test-suite M q' (case m xyb of Some t ⇒ t))
  by pat-completeness auto
termination
proof −
  {
    fix M :: ('a,'b,'c) fsm
    fix q t
    have passes-test-suite-dom (M,q,t)
    proof (induction t arbitrary: M q)
      case (PT m)
    then have ⋀ ab ba bb y x2 . m ((ab, ba), bb) = Some y ⟹ passes-test-suite-dom
(M, x2, y)
        by blast
      then show ?case
        using passes-test-suite.domintros[of q M m] by auto
    qed
  }
  then show ?thesis by auto
qed

lemma passes-test-suite-iff :
  assumes observable M
      and q ∈ states M
shows passes-test-suite M q t = (∀ iob ∈ set t . (map fst iob) ∈ LS M q ⟷
list-all snd iob)
proof
  show passes-test-suite M q t ⟹ ∀ iob∈Prefix-Tree.set t. (map fst iob ∈ LS M q)
= list-all snd iob
  proof
    fix iob assume passes-test-suite M q t
```

```
            and iob ∈ Prefix-Tree.set t
  then show (map fst iob ∈ LS M q) = list-all snd iob
    using ‹q ∈ states M›
  proof (induction iob arbitrary: q t)
    case Nil
    then show ?case by auto
  next
    case (Cons a iob)

    obtain m where t = PT m
      by (meson prefix-tree.exhaust)
    then have isin (PT m) (a#iob)
      using ‹a # iob ∈ Prefix-Tree.set t› by simp
    moreover obtain x y b where a = ((x,y),b)
      by (metis old.prod.exhaust)
    ultimately obtain t′ where m ((x,y),b) = Some t′
                        and isin t′ iob
      unfolding isin.simps
      using case-optionE by blast
    then have ((x,y),b) ∈ dom m
      by auto
    then have ∗: (case h-obs M q x y of
                  None ⇒ ¬b |
                  Some q′ ⇒ b ∧ passes-test-suite M q′ (case m ((x,y),b) of Some
t ⇒ t))
        using ‹passes-test-suite M q t› ‹((x,y),b) ∈ dom m› unfolding ‹t = PT
m› passes-test-suite.simps
        by (metis fst-conv snd-conv)

    show ?case proof (cases b)
      case False
      then have h-obs M q x y = None
        using ∗
        using case-optionE by blast
      then have [(x,y)] ∉ LS M q
        unfolding h-obs-None[OF ‹observable M›] by auto
      then have (map fst (a#iob)) ∉ LS M q
        unfolding ‹a = ((x,y),b)› using language-prefix[of [(x,y)] map fst iob M
q]
        by fastforce
      then show ?thesis
        unfolding ‹a = ((x,y),b)› using False by auto
    next
      case True
      then obtain q′ where h-obs M q x y = Some q′
        using ∗ case-optionE by blast
      then have ∗∗: ((map fst (a#iob)) ∈ LS M q) = ((map fst iob) ∈ LS M q′)
      using observable-language-transition-target[OF ‹observable M›, of (q,x,y,q′)
map fst iob]
```

unfolding ‹$a = ((x,y),b)$› *h-obs-Some*[*OF* ‹*observable M*›] *fst-conv snd-conv*
by (*metis* (*no-types*, *lifting*) *LS-prepend-transition fst-conv list.simps*(*9*) *mem-Collect-eq singletonI snd-conv*)
have ∗∗∗: *list-all snd* (a#iob) = *list-all snd iob*
unfolding ‹$a = ((x,y),b)$› **using** *True* **by** *auto*

have *passes-test-suite M q′ t′*
using ‹*passes-test-suite M q t*› ‹$((x,y),b) \in$ *dom m*› ‹*h-obs M q x y* = *Some q′*› *True*
unfolding ‹$t = PT\ m$› *passes-test-suite.simps*
using ∗ ‹$m\ ((x,\ y),\ b) = Some\ t'$› **by** *auto*
moreover have $iob \in set\ t'$
using ‹*isin t′ iob*› **by** *auto*
moreover have $q' \in states\ M$
using ‹*h-obs M q x y* = *Some q′*› *fsm-transition-target*
unfolding *h-obs-Some*[*OF* ‹*observable M*›]
by *fastforce*
ultimately show *?thesis*
using *Cons.IH* **unfolding** ∗∗ ∗∗∗ **by** *blast*
qed
qed
qed

show ∀ *iob*∈*Prefix-Tree.set t*. (*map fst iob* ∈ *LS M q*) = *list-all snd iob* ⟹ *passes-test-suite M q t*
proof (*induction t arbitrary*: *q*)
case (*PT m*)

have ⋀ *xyb* . *xyb* ∈ *dom m* ⟹ *case h-obs M q* (*fst* (*fst xyb*)) (*snd* (*fst xyb*)) *of None* ⇒ ¬ *snd xyb* | *Some q′* ⇒ *snd xyb* ∧ *passes-test-suite M q′* (*case m xyb of Some t* ⇒ *t*)
proof −
fix *xyb* **assume** *xyb* ∈ *dom m*
moreover obtain *x y b* **where** *xyb* = $((x,y),b)$
by (*metis old.prod.exhaust*)
ultimately obtain *t′* **where** $m\ ((x,y),b) = Some\ t'$
by *auto*
then have *isin* (*PT m*) $[((x,y),b)]$
by *auto*
then have $[((x,y),b)] \in set\ (PT\ m)$
by *auto*
then have (*map fst* $[((x,y),b)] \in LS\ M\ q$) = *list-all snd* $[((x,y),b)]$
using ‹∀ *iob*∈*Prefix-Tree.set* (*PT m*). (*map fst iob* ∈ *LS M q*) = *list-all snd iob*› **by** *blast*
then have ($[(x,y)] \in LS\ M\ q$) = b
by *auto*

show *case h-obs M q (fst (fst xyb)) (snd (fst xyb)) of None* ⇒ ¬ *snd xyb* | *Some q′* ⇒ *snd xyb* ∧ *passes-test-suite M q′ (case m xyb of Some t* ⇒ *t)*

proof (*cases h-obs M q x y*)
case *None*
then have [(*x*,*y*)] ∉ *LS M q*
unfolding *h-obs-None*[*OF* ‹*observable M*›] **by** *auto*
then have *b* = *False*
using ‹([(*x*,*y*)] ∈ *LS M q*) = *b*› **by** *blast*
then show *?thesis*
using *None* **unfolding** ‹*xyb* = ((*x*,*y*),*b*)› **by** *auto*
next
case (*Some q′*)
then have [(*x*,*y*)] ∈ *LS M q*
unfolding *h-obs-Some*[*OF* ‹*observable M*›] *LS-single-transition* **by** *force*
then have *b*
using ‹([(*x*,*y*)] ∈ *LS M q*) = *b*› **by** *blast*
moreover have *passes-test-suite M q′ t′*
proof −
have *Some t′* ∈ *range m*
using ‹*m* ((*x*,*y*),*b*) = *Some t′*›
by (*metis range-eqI*)
moreover have *t′* ∈ *set-option* (*Some t′*)
by *auto*
moreover have ∀ *iob*∈*Prefix-Tree.set t′*. (*map fst iob* ∈ *LS M q′*) = *list-all snd iob*
proof
fix *iob* **assume** *iob* ∈ *Prefix-Tree.set t′*
then have *isin t′ iob*
by *auto*
then have *isin* (*PT m*) (((*x*,*y*),*b*)#*iob*)
using ‹*m* ((*x*,*y*),*b*) = *Some t′*›
by *auto*
then have ((*x*,*y*),*b*)#*iob* ∈ *set* (*PT m*)
by *auto*
then have (*map fst* (((*x*,*y*),*b*)#*iob*) ∈ *LS M q*) = *list-all snd* (((*x*,*y*),*b*)#*iob*)
using *PT.prems* **by** *blast*
moreover have (*map fst* (((*x*,*y*),*b*)#*iob*) ∈ *LS M q*) = (*map fst iob* ∈ *LS M q′*)
using *observable-language-transition-target*[*OF* ‹*observable M*›, *of* (*q*,*x*,*y*,*q′*) *map fst iob*]
by (*metis* (*no-types*, *lifting*) *LS-prepend-transition Some h-obs-Some*[*OF* ‹*observable M*›] *fst-conv list.simps*(*9*) *mem-Collect-eq singletonI snd-conv*)
moreover have *list-all snd* (((*x*,*y*),*b*)#*iob*) = *list-all snd iob*
using ‹*b*› **by** *auto*
ultimately show (*map fst iob* ∈ *LS M q′*) = *list-all snd iob*
by *simp*
qed
ultimately show *?thesis*

```
          using PT.IH by blast
        qed
        ultimately show ?thesis
          using ‹m ((x,y),b) = Some t'› ‹xyb = ((x,y),b)› Some
          by simp
      qed
    qed
    then show ?case
      by auto
  qed
qed
```

```
lemma passes-test-suite-from-io-tree :
  assumes observable M
  and     observable I
  and     qM ∈ states M
  and     qI ∈ states I
shows passes-test-suite I qI (test-suite-from-io-tree M qM t) = ((set t ∩ LS M qM)
= (set t ∩ LS I qI))
proof −

  define ts where ts = test-suite-from-io-tree M qM t
  then have passes-test-suite I qI (test-suite-from-io-tree M qM t) = (∀ iob∈set ts.
(map fst iob ∈ LS I qI) = list-all snd iob)
    using passes-test-suite-iff[OF assms(2,4), of ts]
    by auto
  also have ... = ((set t ∩ LS M qM) = (set t ∩ LS I qI))
  proof
    have ts-set: set ts = map (λx. (x, True)) ‘ (set t ∩ LS M qM) ∪
                          (λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)]) ‘
                          {xs @ [x] |xs x. xs ∈ set t ∩ LS M qM ∧ xs @ [x] ∈ set t −
LS M qM}
     using test-suite-from-io-tree-set[OF assms(1,3), of t] ‹ts = test-suite-from-io-tree
M qM t›
      by auto

    show ∀ iob∈Prefix-Tree.set ts. (map fst iob ∈ LS I qI) = list-all snd iob ⟹ set
t ∩ LS M qM = set t ∩ LS I qI
    proof −
      assume ∀ iob∈Prefix-Tree.set ts. (map fst iob ∈ LS I qI) = list-all snd iob
      then have ts-assm: ⋀ iob . iob ∈ set ts ⟹ (map fst iob ∈ LS I qI) = list-all
snd iob
        by blast

      show set t ∩ LS M qM = set t ∩ LS I qI
      proof
```

show *set t* ∩ *LS M qM* ⊆ *set t* ∩ *LS I qI*
proof
  fix *io* **assume** *io* ∈ *set t* ∩ *LS M qM*
  then have *map* (λ*x*. (*x*, *True*)) *io* ∈ *set ts*
    unfolding *ts-set* **by** *auto*
  moreover have *list-all snd* (*map* (λ*x*. (*x*, *True*)) *io*)
    by (*induction io*; *auto*)
  moreover have *map fst* (*map* (λ*x*. (*x*, *True*)) *io*) = *io*
    by (*induction io*; *auto*)
  ultimately have *io* ∈ *LS I qI*
    using *ts-assm* **by** *force*
  then show *io* ∈ *set t* ∩ *LS I qI*
    using ‹*io* ∈ *set t* ∩ *LS M qM*› **by** *blast*
qed

show *set t* ∩ *LS I qI* ⊆ *set t* ∩ *LS M qM*
proof
  fix *io* **assume** *io* ∈ *set t* ∩ *LS I qI*
  show *io* ∈ *set t* ∩ *LS M qM*
  proof (*rule ccontr*)
    assume *io* ∉ *set t* ∩ *LS M qM*
    then have *io* ∈ *LS I qI* − *LS M qM*
      using ‹*io* ∈ *set t* ∩ *LS I qI*› **by** *blast*
    then obtain *io′ xy io″* **where** *io* = *io′* @ [*xy*] @ *io″*
                and *io′* ∈ *LS I qI* ∩ *LS M qM*
                and *io′* @ [*xy*] ∈ *LS I qI* − *LS M qM*
      using *minimal-failure-prefix-ob*[*OF assms*]
      by *blast*

    have *io′* ∈ *set t* ∩ *LS M qM*
      using ‹*io′* ∈ *LS I qI* ∩ *LS M qM*› ‹*io* ∈ *set t* ∩ *LS I qI*› *isin-prefix*[*of t io′* [*xy*] @ *io″*] *language-prefix*[*of io′* [*xy*] @ *io″*]
      unfolding ‹*io* = *io′* @ [*xy*] @ *io″*›
      by *auto*
    moreover have *io′* @ [*xy*] ∈ *set t* − *LS M qM*
        using ‹*io′* @ [*xy*] ∈ *LS I qI* − *LS M qM*› ‹*io* ∈ *set t* ∩ *LS I qI*› *isin-prefix*[*of t io′*@[*xy*] *io″*]
      unfolding ‹*io* = *io′* @ [*xy*] @ *io″*›
      by *auto*
    ultimately have *io′*@[*xy*] ∈ {*xs* @ [*x*] |*xs x*. *xs* ∈ *set t* ∩ *LS M qM* ∧ *xs* @ [*x*] ∈ *set t* − *LS M qM*}
      by *blast*
    then have (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*io′*@[*xy*]) ∈ *set ts*
      unfolding *ts-set* **by** *blast*
    then have (*map* (λ*x*. (*x*, *True*)) *io′* @ [(*xy*, *False*)]) ∈ *set ts*
      by *auto*
    moreover have (*map fst* (*map* (λ*x*. (*x*, *True*)) *io′* @ [(*xy*, *False*)]) ∈ *LS I qI*) ≠ *list-all snd* ((*map* (λ*x*. (*x*, *True*)) *io′* @ [(*xy*, *False*)]))

```
          proof −
            have (map fst (map (λx. (x, True)) io′ @ [(xy, False)])) = io′@[xy]
              by (induction io′; auto)
            then show ?thesis
                  using ‹io′ @ [xy] ∈ set t − LS M qM› ‹io ∈ set t ∩ LS I qI›
language-prefix[of io′@[xy] io′′ I qI]
              unfolding ‹io = io′ @ [xy] @ io′′›
              by auto
          qed
          ultimately show False
            using ts-assm by blast
        qed
      qed
    qed
  qed

  show set t ∩ LS M qM = Prefix-Tree.set t ∩ LS I qI ⟹ ∀ iob∈set ts. (map
fst iob ∈ LS I qI) = list-all snd iob
  proof
    fix iob assume set t ∩ LS M qM = set t ∩ LS I qI
              and iob ∈ set ts

    then consider (a) iob ∈ map (λx. (x, True)) ‘ (set t ∩ LS M qM) |
                  (b) iob ∈ (λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)])
‘
                        {xs @ [x] |xs x. xs ∈ set t ∩ LS M qM ∧ xs @ [x] ∈ set t −
LS M qM}
      using ts-set by blast

    then show (map fst iob ∈ LS I qI) = list-all snd iob
    proof cases
      case a
      then obtain io where iob = map (λx. (x, True)) io
                    and io ∈ set t ∩ LS M qM
        by blast

      then have map fst iob = io
        by auto
      then have map fst iob ∈ LS I qI
        using ‹io ∈ set t ∩ LS M qM› ‹set t ∩ LS M qM = set t ∩ LS I qI›
        by auto
      moreover have list-all snd iob
        unfolding ‹iob = map (λx. (x, True)) io› by (induction io; auto)
      ultimately show ?thesis
        by simp
    next
      case b

       then obtain ioxy where iob = (λxs. map (λx. (x, True)) (butlast xs) @
```

[(*last xs*, *False*)]) (*ioxy*)
and *ioxy* ∈ {*xs* @ [*x*] |*xs x*. *xs* ∈ *set t* ∩ *LS M qM* ∧ *xs* @ [*x*] ∈ *set t* − *LS M qM*}
by *blast*
then obtain *io xy* **where** *ioxy* = *io*@[*xy*]
and *io*@[*xy*] ∈ *set t* − *LS M qM*
by *blast*
then have ∗: *iob* = *map* (λ*x*. (*x*, *True*)) *io* @ [(*xy*, *False*)]
using ‹*iob* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*ioxy*)› **by** *auto*
then have ∗∗: *map fst iob* = *io*@[*xy*]
by (*induction io arbitrary*: *iob*; *auto*)

have ¬*map fst iob* ∈ *LS I qI*
unfolding ∗∗ **using** ‹*io*@[*xy*] ∈ *set t* − *LS M qM*› ‹*set t* ∩ *LS M qM* = *set t* ∩ *LS I qI*›
by *blast*
moreover have ¬ *list-all snd iob*
unfolding ∗ **by** *auto*
ultimately show *?thesis*
by *simp*
qed
qed
qed
finally show *?thesis* **.**
qed

22.2 Code Refinement

context includes *lifting-syntax*
begin

lemma *map-entries-parametric*:
((*A* ===> *B*) ===> (*A* ===> *C* ===> *rel-option D*) ===> (*B* ===> *rel-option C*) ===> *A* ===> *rel-option D*)
(λ*f g m x*. *case* (*m* ∘ *f*) *x of None* ⇒ *None* | *Some y* ⇒ *g x y*) (λ*f g m x*. *case* (*m* ∘ *f*) *x of None* ⇒ *None* | *Some y* ⇒ *g x y*)
by *transfer-prover*

end

lift-definition *map-entries* :: (′*c* ⇒ ′*a*) ⇒ (′*c* ⇒ ′*b* ⇒ ′*d option*) ⇒ (′*a*, ′*b*) *mapping* ⇒ (′*c*, ′*d*) *mapping*
is λ*f g m x*. *case* (*m* ∘ *f*) *x of None* ⇒ *None* | *Some y* ⇒ *g x y* **parametric** *map-entries-parametric* **.**

lemma *test-suite-from-io-tree-MPT*[*code*] :
test-suite-from-io-tree M q (*MPT m*) =

```
    MPT (map-entries
          fst
          (λ ((x,y),b) t . (case h-obs M q x y of
            None ⇒ (if b then None else Some empty) |
            Some q' ⇒ (if b then Some (test-suite-from-io-tree M q' t) else None)))
          m)
  (is ?t M q (MPT m) = MPT (?f M q m))
proof −
  have ⋀xyb . Mapping.lookup (?f M q m) xyb= (λ ((x,y),b) . case Mapping.lookup
m (x,y) of
    None ⇒ None |
    Some t ⇒ (case h-obs M q x y of
      None ⇒ (if b then None else Some empty) |
      Some q' ⇒ (if b then Some (test-suite-from-io-tree M q' t) else None))) xyb
  (is ⋀ xyb. ?f1 xyb = ?f2 xyb)
  proof −
    fix xyb

    show ?f1 xyb = ?f2 xyb
    proof −
      obtain x y b where ∗:xyb = ((x,y),b)
        by (metis prod.collapse)

      show ?thesis proof (cases Mapping.lookup m (fst xyb))
        case None

        have ?f1 xyb = None
            by (metis (no-types, lifting) None lookup.rep-eq map-entries.rep-eq op-
tion.simps(4))
        moreover have ?f2 xyb = None
          using None by (simp add: ∗)
        ultimately show ?thesis
          by simp
      next
        case (Some t)

        then have ∗∗:?f1 xyb = (λ ((x,y),b) t . (case h-obs M q x y of
            None ⇒ (if b then None else Some empty) |
            Some q' ⇒ (if b then Some (test-suite-from-io-tree M q' t) else None)))
xyb t
          by (simp add: lookup.rep-eq map-entries.rep-eq)

        show ?thesis
          unfolding ∗∗ using Some
          by (simp add: ∗)
      qed
    qed
  qed
  then show ?thesis
```

unfolding *MPT-def* **by** *auto*
qed

lemma *passes-test-suite-MPT*[*code*]:
passes-test-suite M q (*MPT m*) = (∀ *xyb* ∈ *Mapping.keys m* . *case h-obs M q* (*fst* (*fst xyb*)) (*snd* (*fst xyb*)) *of*
None ⇒ ¬(*snd xyb*) |
Some q′ ⇒ *snd xyb* ∧ *passes-test-suite M q′* (*case Mapping.lookup m xyb of Some t* ⇒ *t*))
by (*simp add*: *MPT-def keys-dom-lookup*)

22.3 Pass relations on list of lists representations of test suites

fun *passes-test-case* :: (′*a*,′*b*,′*c*) *fsm* ⇒ ′*a* ⇒ ((′*b* × ′*c*) × *bool*) *list* ⇒ *bool* **where**
passes-test-case M q [] = *True* |
passes-test-case M q (((*x*,*y*),*b*)#*io*) = (*if b*
then case h-obs M q x y of
Some q′ ⇒ *passes-test-case M q′ io* |
None ⇒ *False*
else h-obs M q x y = *None*)

lemma *passes-test-case-iff* :
assumes *observable M*
and *q* ∈ *states M*
shows *passes-test-case M q iob* = ((*map fst* (*takeWhile snd iob*) ∈ *LS M q*)
∧ (¬ (*list-all snd iob*) ⟶ *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) ∉ *LS M q*))
using *assms*(*2*) **proof** (*induction iob arbitrary*: *q*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a iob*)
obtain *x y b* **where** *a* = ((*x*,*y*),*b*)
by (*metis prod.collapse*)

show *?case* **proof** (*cases b*)
case *True*

show *?thesis* **proof** (*cases h-obs M q x y*)
case *None*
then have [(*x*,*y*)] ∉ *LS M q*
unfolding *h-obs-None*[*OF assms*(*1*)] *LS-single-transition* **by** *force*
then have (*map fst* (*takeWhile snd* (*a*#*iob*)) ∉ *LS M q*)
unfolding ‹*a* = ((*x*,*y*),*b*)› **using** *True*
by (*metis* (*mono-tags*, *opaque-lifting*) *append.simps*(*1*) *append.simps*(*2*) *fst-conv language-prefix list.simps*(*9*) *prod.sel*(*2*) *takeWhile.simps*(*2*))
moreover have *passes-test-case M q* (*a*#*iob*) = *False*

using *None* **unfolding** ‹$a = ((x,y),b)$› **using** *True* **by** *auto*
ultimately show *?thesis*
by *blast*
next
case (*Some q′*)
then have *passes-test-case M q* (*a*#*iob*) = *passes-test-case M q′ iob*
unfolding ‹$a = ((x,y),b)$› **using** *True* **by** *auto*
moreover have (*map fst* (*takeWhile snd* (*a*#*iob*)) ∈ *LS M q*) = (*map fst* (*takeWhile snd iob*) ∈ *LS M q′*)
proof −
have ∗: *map fst* (*takeWhile snd* (*a*#*iob*)) = (*x*,*y*)#(*map fst* (*takeWhile snd iob*))
using *True* **unfolding** ‹$a = ((x,y),b)$› **by** *auto*
show *?thesis*
using *Some*
unfolding ∗ *h-obs-Some*[*OF assms*(*1*)]
by (*metis LS-prepend-transition assms*(*1*) *fst-conv mem-Collect-eq observable-language-transition-target singletonI snd-conv*)
qed
moreover have (¬ *list-all snd* (*a*#*iob*) ⟶ *map fst* (*take* (*Suc* (*length* (*takeWhile snd* (*a*#*iob*)))) (*a*#*iob*)) ∉ *LS M q*)
= (¬ *list-all snd iob* ⟶ *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) ∉ *LS M q′*)
proof −
have ∗: *map fst* (*take* (*Suc* (*length* (*takeWhile snd* (*a*#*iob*)))) (*a*#*iob*)) = (*x*,*y*)#(*map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*))
using *True* **unfolding** ‹$a = ((x,y),b)$› **by** *auto*
have ∗∗: *list-all snd* (*a*#*iob*) = *list-all snd iob*
using *True* **unfolding** ‹$a = ((x,y),b)$› **by** *auto*
show *?thesis*
using *Some*
unfolding ∗ ∗∗ *h-obs-Some*[*OF assms*(*1*)]
by (*metis LS-prepend-transition assms*(*1*) *fst-conv mem-Collect-eq observable-language-transition-target prod.sel*(*2*) *singletonI*)
qed
ultimately show *?thesis*
unfolding *Cons.IH*[*OF h-obs-state*[*OF Some*]] **by** *simp*
qed
next
case *False*
show *?thesis* **proof** (*cases h-obs M q x y*)
case *None*
then have [(*x*,*y*)] ∉ *LS M q*
unfolding *h-obs-None*[*OF assms*(*1*)] *LS-single-transition* **by** *force*
then have (¬ *list-all snd* (*a*#*iob*) ⟶ *map fst* (*take* (*Suc* (*length* (*takeWhile snd* (*a*#*iob*)))) (*a*#*iob*)) ∉ *LS M q*)
unfolding ‹$a = ((x,y),b)$› **using** *False* **by** *auto*
moreover have (*map fst* (*takeWhile snd* (*a*#*iob*)) ∈ *LS M q*)
unfolding ‹$a = ((x,y),b)$› **using** *False Cons.prems* **by** *auto*

```
      moreover have passes-test-case M q (a#iob) = True
        unfolding ‹a = ((x,y),b)› using False None by auto
      ultimately show ?thesis
        by simp
    next
      case (Some q')
      then have [(x,y)] ∈ LS M q
        unfolding h-obs-Some[OF assms(1)] LS-single-transition by force
     then have ¬ (¬ list-all snd (a#iob) ⟶ map fst (take (Suc (length (takeWhile
snd (a#iob)))) (a#iob)) ∉ LS M q)
        unfolding ‹a = ((x,y),b)› using False by auto
      moreover have passes-test-case M q (a#iob) = False
        unfolding ‹a = ((x,y),b)› using False Some by auto
      ultimately show ?thesis
        by simp
    qed
  qed
qed


lemma test-suite-from-io-tree-finite-tree :
  assumes observable M
  and     qM ∈ states M
  and     finite-tree t
shows finite-tree (test-suite-from-io-tree M qM t)
proof −
  have finite (Prefix-Tree.set t ∩ LS M qM)
    using assms(3) unfolding finite-tree-iff by blast
  then have finite (map (λx. (x, True)) ‘ (set t ∩ LS M qM))
    by blast

  have ((λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)]) ‘
            {xs @ [x] |xs x. xs ∈ set t ∩ LS M qM ∧ xs @ [x] ∈ set t − LS M qM})
        ⊆ ((λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)]) ‘ (set t))
    by blast
  moreover have finite ((λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)])
‘ (set t))
    using assms(3) unfolding finite-tree-iff by blast
  ultimately have finite ((λxs. map (λx. (x, True)) (butlast xs) @ [(last xs, False)])
‘
            {xs @ [x] |xs x. xs ∈ set t ∩ LS M qM ∧ xs @ [x] ∈ set t − LS M qM})
    using finite-subset by blast
  then show ?thesis
    using ‹finite (map (λx. (x, True)) ‘ (set t ∩ LS M qM))›
    unfolding finite-tree-iff test-suite-from-io-tree-set[OF assms(1,2)]
    by blast
qed
```

```
lemma passes-test-case-prefix :
  assumes observable M
  and     passes-test-case M q (iob@iob')
shows passes-test-case M q iob
using assms(2) proof (induction iob arbitrary: q)
  case Nil
  then show ?case by auto
next
  case (Cons a iob)
  obtain x y b where a = ((x,y),b)
    by (metis prod.collapse)

  show ?case proof (cases b)
    case False
    then show ?thesis
      using Cons.prems unfolding ‹a = ((x,y),b)› by auto
  next
    case True

    show ?thesis proof (cases h-obs M q x y)
      case None
      then show ?thesis
        using Cons.prems unfolding ‹a = ((x,y),b)› by auto
    next
      case (Some q')
      then have passes-test-case M q' (iob @ iob')
        using True Cons.prems unfolding ‹a = ((x,y),b)› by auto
      then have passes-test-case M q' iob
        using Cons.IH by auto
      then show ?thesis
        using True Some unfolding ‹a = ((x,y),b)› by auto
    qed
  qed
qed


lemma passes-test-cases-of-test-suite :
  assumes observable M
  and     observable I
  and     qM ∈ states M
  and     qI ∈ states I
  and     finite-tree t
shows list-all (passes-test-case I qI) (sorted-list-of-maximal-sequences-in-tree (test-suite-from-io-tree
M qM t)) = passes-test-suite I qI (test-suite-from-io-tree M qM t)
  (is ?P1 = ?P2)
proof
```

have *list.set* (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*)) =
{$y \in$ *Prefix-Tree.set* (*test-suite-from-io-tree M qM t*). $\nexists y'$. $y' \neq [] \wedge y$ @ $y' \in$ *Prefix-Tree.set* (*test-suite-from-io-tree M qM t*)}
using *sorted-list-of-maximal-sequences-in-tree-set*[*OF test-suite-from-io-tree-finite-tree*[*OF assms*(*1*,*3*,*5*)]] **.**

show *?P1* $\Longrightarrow$ *?P2*
proof −
assume *?P1*
show *?P2*
unfolding *passes-test-suite-iff*[*OF assms*(*2*,*4*)]
proof
fix *iob* **assume** *iob* $\in$ *Prefix-Tree.set* (*test-suite-from-io-tree M qM t*)

then obtain *iob′* **where** *iob*@*iob′* $\in$ *list.set* (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*))
unfolding *sorted-list-of-maximal-sequences-in-tree-set*[*OF test-suite-from-io-tree-finite-tree*[*OF assms*(*1*,*3*,*5*)]]
using *test-suite-from-io-tree-finite-tree*[*OF assms*(*1*,*3*,*5*)] **unfolding** *finite-tree-iff*
using *prefix-free-set-maximal-list-ob*[*of set* (*test-suite-from-io-tree M qM t*)]
by *blast*

then have *passes-test-case I qI* (*iob*@*iob′*)
using ‹*?P1*› **by** (*simp add*: *list-all-iff*)
then have *passes-test-case I qI iob*
using *passes-test-case-prefix*[*OF assms*(*2*)] **by** *auto*
then have *map fst* (*takeWhile snd iob*) $\in$ *LS I qI*
and ($\neg$ *list-all snd iob* $\longrightarrow$ *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) $\notin$ *LS I qI*)
unfolding *passes-test-case-iff*[*OF assms*(*2*,*4*)]
by *auto*

have *list-all snd iob* $\Longrightarrow$ (*map fst iob* $\in$ *LS I qI*)
using ‹*map fst* (*takeWhile snd iob*) $\in$ *LS I qI*›
by (*simp only*: *list-all-iff flip*: *takeWhile-eq-all-conv*)
moreover have (*map fst iob* $\in$ *LS I qI*) $\Longrightarrow$ *list-all snd iob*
using ‹($\neg$ *list-all snd iob* $\longrightarrow$ *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) $\notin$ *LS I qI*)›
by (*metis append-take-drop-id language-prefix map-append*)
ultimately show (*map fst iob* $\in$ *LS I qI*) = *list-all snd iob*
by *blast*
qed
qed

show *?P2* $\Longrightarrow$ *?P1*
proof −
assume *?P2*

have $\bigwedge$ *iob* . *iob* ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*)) ⟹ *passes-test-case I qI iob*
proof −
fix *iob*
assume *iob* ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*))
then have *iob* ∈ *set* (*test-suite-from-io-tree M qM t*)
unfolding *sorted-list-of-maximal-sequences-in-tree-set*[*OF test-suite-from-io-tree-finite-tree*[*OF assms*(*1*,*3*,*5*)]]
by *blast*
then have ∗: (*map fst iob* ∈ *LS I qI*) = *list-all snd iob*
using ‹*?P2*› **unfolding** *passes-test-suite-iff*[*OF assms*(*2*,*4*)]
by *blast*

consider *iob* ∈ *map* (λ*x*. (*x*, *True*)) ‘ (*Prefix-Tree.set t* ∩ *LS M qM*) |
iob ∈ (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) ‘ {*xs* @ [*x*] |*xs x*. *xs* ∈ *Prefix-Tree.set t* ∩ *LS M qM* ∧ *xs* @ [*x*] ∈ *Prefix-Tree.set t* − *LS M qM*}
using ‹*iob* ∈ *set* (*test-suite-from-io-tree M qM t*)›
unfolding *test-suite-from-io-tree-set*[*OF assms*(*1*,*3*)]
by *blast*
then show *passes-test-case I qI iob* **proof** *cases*
case *1*
then obtain *io* **where** *iob* = *map* (λ*x*. (*x*, *True*)) *io*
by *blast*
have *list-all snd iob*
unfolding ‹*iob* = *map* (λ*x*. (*x*, *True*)) *io*› **by** (*induction io*; *auto*)
then have (*takeWhile snd iob*) = *iob*
by (*induction iob*; *auto*)

have *map fst* (*takeWhile snd iob*) ∈ *LS I qI*
using ∗ ‹*list-all snd iob*›
by (*simp add*: ‹*takeWhile snd iob* = *iob*›)
then show *?thesis*
unfolding *passes-test-case-iff*[*OF assms*(*2*,*4*)]
using ‹*list-all snd iob*›
by *auto*
next
case *2*
then obtain *xs x* **where** *iob* = (λ*xs*. *map* (λ*x*. (*x*, *True*)) (*butlast xs*) @ [(*last xs*, *False*)]) (*xs*@[*x*])
and *xs* ∈ *set t* ∩ *LS M qM*
and *xs* @ [*x*] ∈ *Prefix-Tree.set t* − *LS M qM*
by *blast*
then have ∗∗: *iob* = (*map* (λ*x*. (*x*, *True*)) *xs*) @ [(*x*,*False*)]
by *auto*

have *isin* (*test-suite-from-io-tree M qM t*) ((*takeWhile snd iob*)@(*dropWhile*

snd iob))
using ‹*iob* ∈ *set* (*test-suite-from-io-tree M qM t*)› **by** *auto*
then have (*takeWhile snd iob*) ∈ *set* (*test-suite-from-io-tree M qM t*)
using *isin-prefix*[*of test-suite-from-io-tree M qM t takeWhile snd iob dropWhile snd iob*] **by** *simp*
then have (*map fst* (*takeWhile snd iob*) ∈ *LS I qI*) = *list-all snd* (*takeWhile snd iob*)
using ‹*?P2*› **unfolding** *passes-test-suite-iff*[*OF assms*(*2*,*4*)]
by *blast*
moreover have *list-all snd* (*takeWhile snd iob*)
by (*induction iob*; *auto*)
ultimately have *map fst* (*takeWhile snd iob*) ∈ *LS I qI*
by *simp*

have ¬ *list-all snd iob*
using ∗∗ **by** *auto*
moreover have (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) = *iob*
unfolding ‹*iob* = (*map* (λ*x*. (*x*, *True*)) *xs*) @ [(*x*,*False*)]› **by** (*induction xs*; *auto*)
ultimately have *map fst* (*take* (*Suc* (*length* (*takeWhile snd iob*))) *iob*) ∉ *LS I qI*
using ∗ **by** *simp*

then show *?thesis*
using ‹*map fst* (*takeWhile snd iob*) ∈ *LS I qI*›
unfolding *passes-test-case-iff*[*OF assms*(*2*,*4*)]
by *simp*
qed
qed
then show *?P1*
using *Ball-set-list-all* **by** *blast*
qed
qed

lemma *passes-test-cases-from-io-tree* :
assumes *observable M*
and *observable I*
and *qM* ∈ *states M*
and *qI* ∈ *states I*
and *finite-tree t*
shows *list-all* (*passes-test-case I qI*) (*sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M qM t*)) = ((*set t* ∩ *LS M qM*) = (*set t* ∩ *LS I qI*))
unfolding *passes-test-cases-of-test-suite*[*OF assms*] *passes-test-suite-from-io-tree*[*OF assms*(*1*−*4*)]
by *blast*

22.4 Alternative Representations

22.4.1 Pass and Fail Traces

type-synonym $('b,'c)$ *pass-traces* $= ('b \times 'c)$ *list list*
type-synonym $('b,'c)$ *fail-traces* $= ('b \times 'c)$ *list list*
type-synonym $('b,'c)$ *trace-test-suite* $= ('b,'c)$ *pass-traces* $\times$ $('b,'c)$ *fail-traces*

fun *trace-test-suite-from-tree* :: $('a::linorder,'b::linorder,'c::linorder)$ *fsm* $\Rightarrow$ $('b \times 'c)$ *prefix-tree* $\Rightarrow$ $('b,'c)$ *trace-test-suite* **where**
  trace-test-suite-from-tree M T = (*let*
    (*passes'*,*fails*) = *separate-by* (*is-in-language M* (*initial M*)) (*sorted-list-of-sequences-in-tree T*);
      passes = *sorted-list-of-maximal-sequences-in-tree* (*from-list passes'*)
    in (*passes*, *fails*))

lemma *trace-test-suite-from-tree-language-equivalence* :
  assumes *observable M* **and** *finite-tree T*
  shows $(L\ M \cap set\ T = L\ M' \cap set\ T) = (list.set\ (fst\ (trace\text{-}test\text{-}suite\text{-}from\text{-}tree\ M\ T)) \subseteq L\ M' \wedge L\ M' \cap list.set\ (snd\ (trace\text{-}test\text{-}suite\text{-}from\text{-}tree\ M\ T)) = \{\})$
proof −

  obtain *passes' fails* **where** $*$: (*passes'*,*fails*) = *separate-by* (*is-in-language M* (*initial M*)) (*sorted-list-of-sequences-in-tree T*)
    by *auto*

  define *passes* **where** *passes* = *sorted-list-of-maximal-sequences-in-tree* (*from-list passes'*)

  have *fst* (*trace-test-suite-from-tree M T*) = *passes*
    using $*$ *passes-def* **by** *auto*
  have *snd* (*trace-test-suite-from-tree M T*) = *fails*
    using $*$ *passes-def* **by** *auto*

  have $list.set\ passes' = L\ M \cap set\ T$
    using $*$ *sorted-list-of-sequences-in-tree-set*[*OF assms*(*2*)]
    unfolding *separate-by.simps*
    unfolding *is-in-language-iff*[*OF assms*(*1*) *fsm-initial*]
    by (*metis inter-set-filter old.prod.inject*)
  moreover have $list.set\ passes' \subseteq L\ M' = (list.set\ passes \subseteq L\ M')$
  proof −

    have $\bigwedge io\ .\ io \in list.set\ passes - \{[]\} \Longrightarrow \exists\ io'\ .\ io@io' \in list.set\ passes'$
      unfolding *passes-def*
     unfolding *sorted-list-of-maximal-sequences-in-tree-set*[*OF from-list-finite-tree*]
      unfolding *from-list-set* **by** *force*
    moreover have $[] \in list.set\ passes'$
      unfolding ‹$list.set\ passes' = L\ M \cap set\ T$› **by** *auto*
    ultimately have $\bigwedge io\ .\ io \in list.set\ passes \Longrightarrow \exists\ io'\ .\ io@io' \in list.set\ passes'$
      by *force*

moreover have $\bigwedge$ *io* . *io* $\in$ *list.set passes'* $\Longrightarrow$ $\exists$ *io'* . *io@io'* $\in$ *list.set passes*
proof −
have $\bigwedge$ *io* . *io* $\in$ *list.set passes'* $\Longrightarrow$ *io* $\in$ *set* (*from-list passes'*)
unfolding *from-list-set* **by** *auto*
moreover have $\bigwedge$ *io*. *io* $\in$ *set* (*from-list passes'*) $\Longrightarrow$ $\exists$ *io'* . *io@io'* $\in$ *list.set passes*
unfolding *passes-def*
unfolding *sorted-list-of-maximal-sequences-in-tree-set*[*OF from-list-finite-tree*]
using *from-list-finite-tree sorted-list-of-maximal-sequences-in-tree-ob sorted-list-of-maximal-sequences-in-t* **by** *fastforce*
ultimately show $\bigwedge$ *io* . *io* $\in$ *list.set passes'* $\Longrightarrow$ $\exists$ *io'* . *io@io'* $\in$ *list.set passes*
by *blast*
qed
ultimately show *?thesis*
using *language-prefix*[*of - - M' initial M'*]
by (*meson subset-iff*)
qed
moreover have *list.set fails* = *set T* − *L M*
using ∗ *sorted-list-of-sequences-in-tree-set*[*OF assms*(*2*)]
unfolding *separate-by.simps*
unfolding *is-in-language-iff*[*OF assms*(*1*) *fsm-initial*]
by (*simp add*: *set-diff-eq*)
ultimately show *?thesis*
unfolding ‹*fst* (*trace-test-suite-from-tree M T*) = *passes*›
unfolding ‹*snd* (*trace-test-suite-from-tree M T*) = *fails*›
by *blast*
qed

22.4.2 Input Sequences

fun *test-suite-to-input-sequences* :: ('*b*::*linorder*×'*c*::*linorder*) *prefix-tree* ⇒ '*b list list* **where**
test-suite-to-input-sequences T = *sorted-list-of-maximal-sequences-in-tree* (*from-list* (*map input-portion* (*sorted-list-of-maximal-sequences-in-tree T*)))

lemma *test-suite-to-input-sequences-pass* :
fixes *T* :: ('*b*::*linorder* × '*c*::*linorder*) *prefix-tree*
assumes *finite-tree T*
and (*L M* = *L M'*) ⟷ (*L M* ∩ *set T* = *L M'* ∩ *set T*)
shows (*L M* = *L M'*) ⟷ ({*io* ∈ *L M* . (∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs'* ∈ *list.set* (*prefixes xs*) . *input-portion io* = *xs'*)}
= {*io* ∈ *L M'* . (∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs'* ∈ *list.set* (*prefixes xs*) . *input-portion io* = *xs'*)})
proof −

have ∗: $\bigwedge$ *io* :: ('*b*::*linorder* × '*c*::*linorder*) *list* .
(∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs'* ∈ *list.set*

(*prefixes xs*) . *input-portion io* = *xs'*) = (∃ *io'*∈*set T*. *map fst io* = *map fst io'*)
proof −
fix *io* :: (*'b*::*linorder* × *'c*::*linorder*) *list*

have (∃ *io'*∈*set T*. *map fst io* = *map fst io'*) = (∃ α ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree T*) . ∃ α' ∈ *list.set* (*prefixes* α) . *map fst io* = *map fst* α')
proof
have ∗: ⋀ *io'* . *io'*∈*set T* ⟷ (∃ *io''*. *io'*@*io''* ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree T*))
using *sorted-list-of-maximal-sequences-in-tree-set*[*OF assms*(*1*)]
using *assms*(*1*) *set-prefix sorted-list-of-maximal-sequences-in-tree-ob* **by** *fastforce*

show (∃ *io'*∈*set T*. *map fst io* = *map fst io'*) ⟹ (∃ α ∈ *list.set* (*sorted-list-of-maximal-sequences-in-tree T*) . ∃ α' ∈ *list.set* (*prefixes* α) . *map fst io* = *map fst* α')
by (*metis append-Nil2 assms*(*1*) *prefixes-prepend prefixes-set-Nil sorted-list-of-maximal-sequences-in-tree-o*

show ∃α∈*list.set* (*sorted-list-of-maximal-sequences-in-tree T*). ∃α'∈*list.set* (*prefixes* α). *map fst io* = *map fst* α' ⟹ ∃ *io'*∈*Prefix-Tree.set T*. *map fst io* = *map fst io'*
by (*metis* ∗ *prefixes-set-ob*)
qed
also have . . . = (∃ *xs* ∈ *list.set* (*map input-portion* (*sorted-list-of-maximal-sequences-in-tree T*)) . ∃ *xs'* ∈ *list.set* (*prefixes xs*) . *map fst io* = *xs'*)
proof −
have ∗: *list.set* (*map input-portion* (*sorted-list-of-maximal-sequences-in-tree T*)) = *input-portion* ‘ (*list.set* (*sorted-list-of-maximal-sequences-in-tree T*))
by *auto*
have ∗∗: ⋀ (α :: (*'b*::*linorder* × *'c*::*linorder*) *list*) . (∃ α' ∈ *list.set* (*prefixes* α) . *map fst io* = *map fst* α') = (∃ *xs'* ∈ *list.set* (*prefixes* (*input-portion* α)) . *map fst io* = *xs'*)
proof
fix α :: (*'b*::*linorder* × *'c*::*linorder*) *list*
show ∃α'∈*list.set* (*prefixes* α). *map fst io* = *map fst* α' ⟹ ∃*xs'*∈*list.set* (*prefixes* (*map fst* α)). *map fst io* = *xs'*
proof −
assume ∃α'∈*list.set* (*prefixes* α). *map fst io* = *map fst* α'
then obtain α' α'' **where** α'@α'' = α **and** *map fst io* = *map fst* α'
unfolding *prefixes-set* **by** *blast*
then show ∃*xs'*∈*list.set* (*prefixes* (*map fst* α)). *map fst io* = *xs'*
unfolding *prefixes-set*
by *auto*
qed
show ∃*xs'*∈*list.set* (*prefixes* (*map fst* α)). *map fst io* = *xs'* ⟹ ∃α'∈*list.set* (*prefixes* α). *map fst io* = *map fst* α'
proof −
assume ∃*xs'*∈*list.set* (*prefixes* (*map fst* α)). *map fst io* = *xs'*
then obtain *xs' xs''* **where** *xs'*@*xs''* = (*map fst* α) **and** *map fst io* = *xs'*
unfolding *prefixes-set* **by** *blast*

then have *map fst* (*take* (*length xs′*) *α*) = *map fst io*
by (*metis* ‹∃ *xs′*∈*list.set* (*prefixes* (*map fst α*)). *map fst io* = *xs′*› *prefixes-take-iff take-map*)
moreover have (*take* (*length xs′*) *α*) ∈ *list.set* (*prefixes α*)
by (*metis* ‹*map fst io* = *xs′*› *calculation length-map prefixes-take-iff*)
ultimately show *?thesis*
by *metis*
qed
qed
show *?thesis*
unfolding ∗∗ ∗
by *blast*
qed
also have . . . = (∃ *xs* ∈ *list.set* (*test-suite-to-input-sequences T*) . ∃ *xs′* ∈ *list.set* (*prefixes xs*) . *input-portion io* = *xs′*)
proof
show ∃ *xs*∈*list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*)). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′* ⟹ ∃ *xs*∈*list.set* (*test-suite-to-input-sequences T*). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′*
proof −
assume ∃ *xs*∈*list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*)). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′*
then obtain *xs′ xs″* **where** *xs′*@*xs″* ∈ *list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*))
and *map fst io* = *xs′*
unfolding *prefixes-set* **by** *blast*
then have ∗:*xs′*@*xs″* ∈ *set* (*from-list* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*)))
unfolding *from-list-set* **by** *blast*

show *?thesis*
using *sorted-list-of-maximal-sequences-in-tree-ob*[*OF from-list-finite-tree* ∗] ‹*map fst io* = *xs′*›
unfolding *test-suite-to-input-sequences.simps*
by (*metis append.assoc append-Nil2 prefixes-prepend prefixes-set-Nil*)
qed
show ∃ *xs*∈*list.set* (*test-suite-to-input-sequences T*). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′* ⟹ ∃ *xs*∈*list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*)). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′*
proof −
assume ∃ *xs*∈*list.set* (*test-suite-to-input-sequences T*). ∃ *xs′*∈*list.set* (*prefixes xs*). *map fst io* = *xs′*
then obtain *xs′ xs″* **where** *xs′*@*xs″* ∈ *list.set* (*test-suite-to-input-sequences T*)
and *map fst io* = *xs′*
unfolding *prefixes-set* **by** *blast*
then have *xs′*@*xs″* = [] ∨ (∃ *xs‴* . (*xs′*@*xs″*)@*xs‴*∈*list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*)))
unfolding *test-suite-to-input-sequences.simps*

unfolding *sorted-list-of-maximal-sequences-in-tree-set*[*OF from-list-finite-tree*]

unfolding *from-list-set*
by *blast*
then obtain xs''' **where** $(xs'@xs'')@xs''' \in$ *list.set* (*map* (*map fst*) (*sorted-list-of-maximal-sequences-in-tree T*))
by (*metis Nil-is-append-conv* ‹*map fst io* = *xs'*› *append.left-neutral calculation list.simps*(*8*) *set-Nil*)
then show *?thesis*
using ‹*map fst io* = *xs'*›
by (*metis append.assoc append.right-neutral prefixes-prepend prefixes-set-Nil*)

qed
qed
finally show ($\exists$ *xs* $\in$ *list.set* (*test-suite-to-input-sequences T*) . $\exists$ *xs'* $\in$ *list.set* (*prefixes xs*) . *input-portion io* = *xs'*) = ($\exists$ *io'*$\in$*set T*. *map fst io* = *map fst io'*)
by *presburger*
qed

show *?thesis*
unfolding $*$
using *equivalence-io-relaxation*[*OF assms*(*2*)] **.**
qed

lemma *test-suite-to-input-sequences-pass-alt-def* :
fixes *T* :: (*'b::linorder* $\times$ *'c::linorder*) *prefix-tree*
assumes *finite-tree T*
and (*L M* = *L M'*) $\longleftrightarrow$ (*L M* $\cap$ *set T* = *L M'* $\cap$ *set T*)
shows (*L M* = *L M'*) $\longleftrightarrow$ ($\forall$ *xs* $\in$ *list.set* (*test-suite-to-input-sequences T*) . $\forall$ *xs'* $\in$ *list.set* (*prefixes xs*) . {*io* $\in$ *L M* . *input-portion io* = *xs'*} = {*io* $\in$ *L M'* . *input-portion io* = *xs'*})
unfolding *test-suite-to-input-sequences-pass*[*OF assms*]
by *blast*

end

23 Simple Convergence Graphs

This theory introduces a very simple implementation of convergence graphs that consists of a list of convergent classes represented as sets of traces.

theory *Simple-Convergence-Graph*
imports *Convergence-Graph*
begin

23.1 Basic Definitions

type-synonym *'a simple-cg* = *'a list fset list*

definition *simple-cg-empty* :: *'a simple-cg* **where**
simple-cg-empty = []

fun *simple-cg-lookup* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a list list* **where**
simple-cg-lookup xs ys = *sorted-list-of-fset* (*finsert ys* (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *ys* |∈| *x*) *xs*)))

fun *simple-cg-lookup-with-conv* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a list list* **where**
simple-cg-lookup-with-conv g ys = (*let*
lookup-for-prefix = (λ*i* . *let*
pref = *take i ys*;
suff = *drop i ys*;
pref-conv = (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *pref* |∈| *x*) *g*))
in fimage (λ *pref'* . *pref'*@*suff*) *pref-conv*)
in sorted-list-of-fset (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty* [*0*..<*Suc* (*length ys*)])))

fun *simple-cg-insert'* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a simple-cg* **where**
simple-cg-insert' xs ys = (*case find* (λ*x* . *ys* |∈| *x*) *xs*
of Some x ⇒ *xs* |
None ⇒ {|*ys*|}#*xs*)

fun *simple-cg-insert* :: (*'a::linorder*) *simple-cg* ⇒ *'a list* ⇒ *'a simple-cg* **where**
simple-cg-insert xs ys = *foldl* (λ *xs' ys'* . *simple-cg-insert' xs' ys'*) *xs* (*prefixes ys*)

fun *simple-cg-initial* :: (*'a,'b::linorder,'c::linorder*) *fsm* ⇒ (*'b*×*'c*) *prefix-tree* ⇒ (*'b*×*'c*) *simple-cg* **where**
simple-cg-initial M1 T = *foldl* (λ *xs' ys'* . *simple-cg-insert' xs' ys'*) *simple-cg-empty* (*filter* (*is-in-language M1* (*initial M1*)) (*sorted-list-of-sequences-in-tree T*))

23.2 Merging by Closure

The following implementation of the merge operation follows the closure operation described by Simão et al. in Simão, A., Petrenko, A. and Yevtushenko, N. (2012), On reducing test length for FSMs with extra states. Softw. Test. Verif. Reliab., 22: 435-454. https://doi.org/10.1002/stvr.452. That is, two traces u and v are merged by adding u,v to the list of convergent classes followed by computing the closure of the graph based on two operations: (1) classes A and B can be merged if there exists some class C such that C contains some w1, w2 and there exists some w such that A contains w1.w and B contains w2.w. (2) classes A and B can be merged if one is a subset of the other.

fun *can-merge-by-suffix* :: *'a list fset* ⇒ *'a list fset* ⇒ *'a list fset* ⇒ *bool* **where**

can-merge-by-suffix x x1 x2 = (∃ α β γ . α |∈| x ∧ β |∈| x ∧ α@γ |∈| x1 ∧ β@γ |∈| x2)

lemma *can-merge-by-suffix-code*[*code*] :
can-merge-by-suffix x x1 x2 =
(∃ *ys* ∈ *fset x* .
∃ *ys1* ∈ *fset x1* .
is-prefix ys ys1 ∧
(∃ *ys'* ∈ *fset x* . *ys'*@(*drop* (*length ys*) *ys1*) |∈| *x2*))
(**is** *?P1* = *?P2*)
proof
show *?P1* ⟹ *?P2*
by (*metis append-eq-conv-conj can-merge-by-suffix.elims*(*2*) *is-prefix-prefix*)
show *?P2* ⟹ *?P1*
by (*metis append-eq-conv-conj can-merge-by-suffix.elims*(*3*) *is-prefix-prefix*)
qed

fun *prefixes-in-list-helper* :: *'a* ⇒ *'a list list* ⇒ (*bool* × *'a list list*) ⇒ *bool* × *'a list list* **where**
prefixes-in-list-helper x [] *res* = *res* |
prefixes-in-list-helper x ([]#*yss*) *res* = *prefixes-in-list-helper x yss* (*True*, *snd res*) |
prefixes-in-list-helper x ((*y*#*ys*)#*yss*) *res* =
(*if x* = *y then prefixes-in-list-helper x yss* (*fst res*, *ys* # *snd res*)
else prefixes-in-list-helper x yss res)

fun *prefixes-in-list* :: *'a list* ⇒ *'a list* ⇒ *'a list list* ⇒ *'a list list* ⇒ *'a list list* **where**
prefixes-in-list [] *prev yss res* = (*if List.member yss* [] *then prev*#*res else res*) |
prefixes-in-list (*x*#*xs*) *prev yss res* = (*let*
(*b*,*yss'*) = *prefixes-in-list-helper x yss* (*False*,[])
in if b then prefixes-in-list xs (*prev*@[*x*]) *yss'* (*prev* # *res*)
else prefixes-in-list xs (*prev*@[*x*]) *yss'* *res*)

fun *prefixes-in-set* :: (*'a*::*linorder*) *list* ⇒ *'a list fset* ⇒ *'a list list* **where**
prefixes-in-set xs yss = *prefixes-in-list xs* [] (*sorted-list-of-fset yss*) []

value *prefixes-in-list* [*1*::*nat*,*2*,*3*,*4*,*5*] []
[[*1*,*2*,*3*], [*1*,*2*,*4*], [*1*,*3*], [], [*1*], [*1*,*5*,*3*], [*2*,*5*]] []

value *prefixes-in-list-helper* (*1*::*nat*)
[[*1*,*2*,*3*], [*1*,*2*,*4*], [*1*,*3*], [], [*1*], [*1*,*5*,*3*], [*2*,*5*]]
(*False*,[])

lemma *prefixes-in-list-helper-prop* :
shows *fst* (*prefixes-in-list-helper x yss res*) = (*fst res* ∨ [] ∈ *list.set yss*) (**is** *?P1*)
and *list.set* (*snd* (*prefixes-in-list-helper x yss res*)) = *list.set* (*snd res*) ∪ {*ys* . *x*#*ys* ∈ *list.set yss*} (**is** *?P2*)

```
proof −
  have ?P1 ∧ ?P2
  proof (induction yss arbitrary: res)
    case Nil
    then show ?case by auto
  next
    case (Cons ys yss)
    show ?case proof (cases ys)
      case Nil
      then show ?thesis
        using Cons.IH by auto
    next
      case (Cons y ys')
      show ?thesis proof (cases x = y)
        case True
        have ∗: prefixes-in-list-helper x (ys # yss) res = prefixes-in-list-helper y yss
(fst res, ys' # snd res)
          unfolding Cons True by auto
        show ?thesis
          using Cons.IH[of (fst res, ys' # snd res)]
          unfolding ∗
          unfolding Cons
          unfolding True
          by auto
      next
        case False
        then have ∗: prefixes-in-list-helper x (ys # yss) res = prefixes-in-list-helper
x yss res
          unfolding Cons by auto

        show ?thesis
          unfolding ∗
          unfolding Cons
          using Cons.IH[of res] False
          by force
      qed
    qed
  qed
  then show ?P1 and ?P2 by blast+
qed

lemma prefixes-in-list-prop :
shows list.set (prefixes-in-list xs prev yss res) = list.set res ∪ {prev@ys | ys . ys ∈
list.set (prefixes xs) ∧ ys ∈ list.set yss}
proof (induction xs arbitrary: prev yss res)
  case Nil
  show ?case
    unfolding prefixes-in-list.simps prefixes-set by auto
next
```

```
  case (Cons x xs)

  obtain b yss' where prefixes-in-list-helper x yss (False,[]) = (b,yss')
    using prod.exhaust by metis
  then have b = ([] ∈ list.set yss)
        and list.set yss' = {ys . x#ys ∈ list.set yss}
    using prefixes-in-list-helper-prop[of x yss (False,[])]
    by auto

  show ?case proof (cases b)
    case True
   then have *: prefixes-in-list (x#xs) prev yss res = prefixes-in-list xs (prev@[x])
yss' (prev # res)
      using ‹prefixes-in-list-helper x yss (False,[]) = (b,yss')› by auto
    show ?thesis
      unfolding *
      unfolding Cons ‹list.set yss' = {ys . x#ys ∈ list.set yss}›
      using True unfolding ‹b = ([] ∈ list.set yss)›
      by auto
  next
    case False
   then have *: prefixes-in-list (x#xs) prev yss res = prefixes-in-list xs (prev@[x])
yss' res
      using ‹prefixes-in-list-helper x yss (False,[]) = (b,yss')› by auto

    show ?thesis
      unfolding *
      unfolding Cons ‹list.set yss' = {ys . x#ys ∈ list.set yss}›
      using False unfolding ‹b = ([] ∈ list.set yss)›
      by auto
  qed
qed

lemma prefixes-in-set-prop :
  list.set (prefixes-in-set xs yss) = list.set (prefixes xs) ∩ fset yss
  unfolding prefixes-in-set.simps
  unfolding prefixes-in-list-prop
  by auto
```

```
lemma can-merge-by-suffix-validity :
  assumes observable M1 and observable M2
  and     ⋀ u v . u |∈| x ⟹ v |∈| x ⟹ u ∈ L M1 ⟹ u ∈ L M2 ⟹ converge
M1 u v ∧ converge M2 u v
  and     ⋀ u v . u |∈| x1 ⟹ v |∈| x1 ⟹ u ∈ L M1 ⟹ u ∈ L M2 ⟹ converge
```

M1 u v ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *can-merge-by-suffix x x1 x2*
and *u* |∈| (*x1* |∪| *x2*)
and *v* |∈| (*x1* |∪| *x2*)
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
proof −
obtain α β γ **where** α |∈| *x* **and** β |∈| *x* **and** α@γ |∈| *x1* **and** β@γ |∈| *x2*
using ‹*can-merge-by-suffix x x1 x2*› **by** *auto*

consider *u* |∈| *x1* | *u* |∈| *x2*
using ‹*u* |∈| (*x1* |∪| *x2*)› **by** *blast*
then show *?thesis* **proof** *cases*
case *1*

then have *converge M1 u* (α@γ) **and** *converge M2 u* (α@γ)
using ‹*u* |∈| (*x1* |∪| *x2*)› *assms*(*4*)[*OF* - ‹α@γ |∈| *x1*› *assms*(*9*,*10*)]
by *blast*+
then have (α@γ) ∈ *L M1* **and** (α@γ) ∈ *L M2*
by *auto*
then have α ∈ *L M1* **and** α ∈ *L M2*
using *language-prefix* **by** *metis*+
then have *converge M1* α β **and** *converge M2* α β
using *assms*(*3*) ‹α |∈| *x*› ‹β |∈| *x*›
by *blast*+
have *converge M1* (α@γ) (β@γ)
using ‹*converge M1* α β›
by (*meson* ‹α @ γ ∈ *L M1*› *assms*(*1*) *converge.simps converge-append*)
then have β@γ ∈ *L M1*
by *auto*
have *converge M2* (α@γ) (β@γ)
using ‹*converge M2* α β›
by (*meson* ‹α @ γ ∈ *L M2*› *assms*(*2*) *converge.simps converge-append*)
then have β@γ ∈ *L M2*
by *auto*

consider (*11*) *v* |∈| *x1* | (*12*) *v* |∈| *x2*
using ‹*v* |∈| (*x1* |∪| *x2*)› **by** *blast*
then show *?thesis* **proof** *cases*
case *11*
show *?thesis*
using *1 11 assms*(*10*) *assms*(*4*) *assms*(*9*) **by** *blast*
next
case *12*
then have *converge M1 v* (β@γ) **and** *converge M2 v* (β@γ)
using *assms*(*5*)[*OF* ‹β@γ |∈| *x2*› - ‹β@γ ∈ *L M1*› ‹β@γ ∈ *L M2*›]
by *auto*

```
    then show ?thesis
      using ‹converge M1 (α@γ) (β@γ)› ‹converge M2 (α@γ) (β@γ)› ‹converge
M1 u (α@γ)› ‹converge M2 u (α@γ)›
      by auto
  qed
 next
  case 2

  then have converge M1 u (β@γ) and converge M2 u (β@γ)
    using ‹u |∈| (x1 |∪| x2)› assms(5)[OF - ‹β@γ |∈| x2› assms(9,10)]
    by blast+
  then have (β@γ) ∈ L M1 and (β@γ) ∈ L M2
    by auto
  then have β ∈ L M1 and β ∈ L M2
    using language-prefix by metis+
  then have converge M1 α β and converge M2 α β
    using assms(3)[OF ‹β |∈| x› ‹α |∈| x›]
    by auto
  have converge M1 (α@γ) (β@γ)
      using ‹converge M1 α β›
        using ‹β @ γ ∈ L M1› ‹β ∈ L M1› assms(1) converge-append con-
verge-append-language-iff by blast
  then have α@γ ∈ L M1
    by auto
  have converge M2 (α@γ) (β@γ)
    using ‹converge M2 α β›
   using ‹β @ γ ∈ L M2› ‹β ∈ L M2› assms(2) converge-append converge-append-language-iff
by blast
  then have α@γ ∈ L M2
    by auto


  consider (21) v |∈| x1 | (22) v |∈| x2
    using ‹v |∈| (x1 |∪| x2)› by blast
  then show ?thesis proof cases
    case 22
    show ?thesis
      using 2 22 assms(10) assms(5) assms(9) by blast
  next
    case 21
    then have converge M1 v (α@γ) and converge M2 v (α@γ)
      using assms(4)[OF ‹α@γ |∈| x1› - ‹α@γ ∈ L M1› ‹α@γ ∈ L M2›]
      by auto
    then show ?thesis
      using ‹converge M1 (α@γ) (β@γ)› ‹converge M2 (α@γ) (β@γ)›  ‹converge
M1 u (β@γ)› ‹converge M2 u (β@γ)›
      by auto
  qed
 qed
```

qed

fun *simple-cg-closure-phase-1-helper′* :: *′a list fset* ⇒ *′a list fset* ⇒ *′a simple-cg* ⇒ (*bool* × *′a list fset* × *′a simple-cg*) **where**
simple-cg-closure-phase-1-helper′ x x1 xs =
(*let* (*x2s*,*others*) = *separate-by* (*can-merge-by-suffix x x1*) *xs*;
x1Union = *foldl* (|∪|) *x1 x2s*
in (*x2s* ≠ [],*x1Union*,*others*))

lemma *simple-cg-closure-phase-1-helper′-False* :
¬*fst* (*simple-cg-closure-phase-1-helper′ x x1 xs*) ⟹ *simple-cg-closure-phase-1-helper′ x x1 xs* = (*False*,*x1*,*xs*)
unfolding *simple-cg-closure-phase-1-helper′.simps Let-def separate-by.simps*
by (*simp add*: *filter-empty-conv*)

lemma *simple-cg-closure-phase-1-helper′-True* :
assumes *fst* (*simple-cg-closure-phase-1-helper′ x x1 xs*)
shows *length* (*snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))) < *length xs*
proof −
have *snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*)) = *filter* (λ*x2* . ¬ (*can-merge-by-suffix x x1 x2*)) *xs*
by *auto*
moreover have *filter* (λ*x2* . (*can-merge-by-suffix x x1 x2*)) *xs* ≠ []
using *assms* **unfolding** *simple-cg-closure-phase-1-helper′.simps Let-def separate-by.simps*
by *fastforce*
ultimately show *?thesis*
using *filter-not-all-length*[*of can-merge-by-suffix x x1 xs*]
by *metis*
qed

lemma *simple-cg-closure-phase-1-helper′-length* :
length (*snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))) ≤ *length xs*
by *auto*

lemma *simple-cg-closure-phase-1-helper′-validity-fst* :
assumes *observable M1* **and** *observable M2*
and ⋀ *u v* . *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *u* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))
and *v* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*

proof −

have ∗:⋀ *w* . *w* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*)) ⟹ *w* |∈| *x1* ∨ (∃ *x2* . *x2* ∈ *list.set xs* ∧ *w* |∈| *x2* ∧ *can-merge-by-suffix x x1 x2*)
proof −
fix *w* **assume** *w* |∈| *fst* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))
then have *w* |∈| *ffUnion* (*fset-of-list* (*x1*#(*filter* (*can-merge-by-suffix x x1*) *xs*)))
using *foldl-funion-fsingleton*[**where** *xs*=(*filter* (*can-merge-by-suffix x x1*) *xs*)]
by *auto*

then obtain *x2* **where** *w* |∈| *x2*
and *x2* |∈| *fset-of-list* (*x1*#(*filter* (*can-merge-by-suffix x x1*) *xs*))
using *ffUnion-fmember-ob*
by *metis*
then consider *x2*=*x1* | *x2* ∈ *list.set* (*filter* (*can-merge-by-suffix x x1*) *xs*)
by (*meson fset-of-list-elem set-ConsD*)
then show *w* |∈| *x1* ∨ (∃ *x2* . *x2* ∈ *list.set xs* ∧ *w* |∈| *x2* ∧ *can-merge-by-suffix x x1 x2*)
using ‹*w* |∈| *x2*› **by** (*cases*; *auto*)
qed

consider *u* |∈| *x1* | (∃ *x2* . *x2* ∈ *list.set xs* ∧ *u* |∈| *x2* ∧ *can-merge-by-suffix x x1 x2*)
using ∗[*OF assms*(*6*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *1*

consider (*a*) *v* |∈| *x1* | (*b*) (∃ *x2* . *x2* ∈ *list.set xs* ∧ *v* |∈| *x2* ∧ *can-merge-by-suffix x x1 x2*)
using ∗[*OF assms*(*7*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *a*
then show *?thesis* **using** *assms*(*4*)[*OF 1 - assms*(*8*,*9*)] **by** *auto*
next
case *b*
then obtain *x2v* **where** *x2v* ∈ *list.set xs* **and** *v* |∈| *x2v* **and** *can-merge-by-suffix x x1 x2v*
using ∗[*OF assms*(*6*)]
by *blast*

then have *u* |∈| *x1* |∪| *x2v* **and** *v* |∈| *x1* |∪| *x2v*
using *1* **by** *auto*

show *?thesis*
using *can-merge-by-suffix-validity*[*OF assms*(*1*,*2*), *of x x1 x2v*, *OF assms*(*3*,*4*) *assms*(*5*)[*OF* ‹*x2v* ∈ *list.set xs*›] ‹*can-merge-by-suffix x x1 x2v*› ‹*u* |∈| *x1* |∪| *x2v*› ‹*v* |∈| *x1* |∪| *x2v*› *assms*(*8*,*9*)]
by *blast*

qed
next
case *2*
then obtain *x2u* **where** *x2u* ∈ *list.set xs* **and** *u* |∈| *x2u* **and** *can-merge-by-suffix x x1 x2u*
using ∗[*OF assms*(*6*)]
by *blast*
then have *u* |∈| *x1* |∪| *x2u*
by *auto*

consider (*a*) *v* |∈| *x1* | (*b*) (∃ *x2* . *x2* ∈ *list.set xs* ∧ *v* |∈| *x2* ∧ *can-merge-by-suffix x x1 x2*)
using ∗[*OF assms*(*7*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *a*
then have *v* |∈| *x1* |∪| *x2u*
by *auto*
show *?thesis*
using *can-merge-by-suffix-validity*[*OF assms*(*1*,*2*), *of x x1 x2u*, *OF assms*(*3*,*4*) *assms*(*5*)[*OF* ‹*x2u* ∈ *list.set xs*›] ‹*can-merge-by-suffix x x1 x2u*› ‹*u* |∈| *x1* |∪| *x2u*› ‹*v* |∈| *x1* |∪| *x2u*› *assms*(*8*,*9*)]
by *blast*
next
case *b*

then obtain *x2v* **where** *x2v* ∈ *list.set xs* **and** *v* |∈| *x2v* **and** *can-merge-by-suffix x x1 x2v*
using ∗[*OF assms*(*6*)]
by *blast*
then have *v* |∈| *x1* |∪| *x2v*
by *auto*

have ⋀ *v* . *v* |∈| *x1* |∪| *x2u* ⟹ *converge M1 u v* ∧ *converge M2 u v*
using *can-merge-by-suffix-validity*[*OF assms*(*1*,*2*), *of x x1 x2u*, *OF assms*(*3*,*4*) *assms*(*5*)[*OF* ‹*x2u* ∈ *list.set xs*›] ‹*can-merge-by-suffix x x1 x2u*› ‹*u* |∈| *x1* |∪| *x2u*› - *assms*(*8*,*9*)]
by *blast*
have ⋀ *u* . *u* |∈| *x1* |∪| *x2v* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
using *can-merge-by-suffix-validity*[*OF assms*(*1*,*2*), *of x x1 x2v*, *OF assms*(*3*,*4*) *assms*(*5*)[*OF* ‹*x2v* ∈ *list.set xs*›] ‹*can-merge-by-suffix x x1 x2v*› - ‹*v* |∈| *x1* |∪| *x2v*›]
by *blast*

obtain *αv βv γv* **where** *αv* |∈| *x* **and** *βv* |∈| *x* **and** *αv*@*γv* |∈| *x1* **and** *βv*@*γv* |∈| *x2v*
using ‹*can-merge-by-suffix x x1 x2v*› **by** *auto*

show *?thesis*
using ‹⋀*u*. ⟦*u* |∈| *x1* |∪| *x2v*; *u* ∈ *L M1*; *u* ∈ *L M2*⟧ ⟹ *converge M1 u v*

$\wedge$ *converge M2 u v*› ‹$\bigwedge$*v*. *v* |∈| *x1* |∪| *x2u* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*› ‹α*v* @ γ*v* |∈| *x1*› **by** *fastforce*
  qed
 qed
qed

lemma *simple-cg-closure-phase-1-helper′-validity-snd* :
 assumes $\bigwedge$ *x2 u v* . *x2* ∈ *list.set xs* $\Longrightarrow$ *u* |∈| *x2* $\Longrightarrow$ *v* |∈| *x2* $\Longrightarrow$ *u* ∈ *L M1* $\Longrightarrow$ *u* ∈ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
 and *x2* ∈ *list.set* (*snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*)))
 and *u* |∈| *x2*
 and *v* |∈| *x2*
 and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* $\wedge$ *converge M2 u v*
proof −
 have *list.set* (*snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))) ⊆ *list.set xs*
  by *auto*
 then show *?thesis*
  using *assms* **by** *blast*
qed

fun *simple-cg-closure-phase-1-helper* :: *′a list fset* ⇒ *′a simple-cg* ⇒ (*bool* × *′a simple-cg*) ⇒ (*bool* × *′a simple-cg*) **where**
 simple-cg-closure-phase-1-helper x [] (*b*,*done*) = (*b*,*done*) |
 simple-cg-closure-phase-1-helper x (*x1*#*xs*) (*b*,*done*) = (*let* (*hasChanged*,*x1′*,*xs′*) = *simple-cg-closure-phase-1-helper′ x x1 xs*
 in simple-cg-closure-phase-1-helper x xs′ (*b* ∨ *hasChanged*, *x1′* # *done*))

lemma *simple-cg-closure-phase-1-helper-validity* :
 assumes *observable M1* **and** *observable M2*
 and $\bigwedge$ *u v* . *u* |∈| *x* $\Longrightarrow$ *v* |∈| *x* $\Longrightarrow$ *u* ∈ *L M1* $\Longrightarrow$ *u* ∈ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
 and $\bigwedge$ *x2 u v* . *x2* ∈ *list.set don* $\Longrightarrow$ *u* |∈| *x2* $\Longrightarrow$ *v* |∈| *x2* $\Longrightarrow$ *u* ∈ *L M1* $\Longrightarrow$ *u* ∈ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
 and $\bigwedge$ *x2 u v* . *x2* ∈ *list.set xss* $\Longrightarrow$ *u* |∈| *x2* $\Longrightarrow$ *v* |∈| *x2* $\Longrightarrow$ *u* ∈ *L M1* $\Longrightarrow$ *u* ∈ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
 and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-1-helper x xss* (*b*,*don*)))
 and *u* |∈| *x2*
 and *v* |∈| *x2*
 and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* $\wedge$ *converge M2 u v*
 using *assms*(*4*,*5*,*6*)
proof (*induction length xss arbitrary*: *xss don b rule*: *less-induct*)
 case *less*
 show *?case* **proof** (*cases xss*)

case *Nil*
then have *x2* ∈ *list.set don*
using *less.prems*(*3*) **by** *auto*
then show *?thesis*
using *less.prems*(*1*) *assms*(*7*,*8*,*9*,*10*)
by *blast*
next
case (*Cons x1 xs*)
obtain *b′ x1′ xs′* **where** *simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)
using *prod.exhaust* **by** *metis*
then have *simple-cg-closure-phase-1-helper x xss* (*b*,*don*) = *simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′* # *don*)
unfolding *Cons* **by** *auto*

have ∗:⋀ *u v* . *u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
using *less.prems*(*2*)[*of x1*] **unfolding** *Cons*
by (*meson list.set-intros*(*1*))

have ∗∗:⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
using *less.prems*(*2*) **unfolding** *Cons*
by (*meson list.set-intros*(*2*))

have ∗∗∗:⋀ *u v*. *u* |∈| *x1′* ⟹ *v* |∈| *x1′* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
using *simple-cg-closure-phase-1-helper′-validity-fst*[*of M1 M2 x x1 xs - -*, *OF assms*(*1*,*2*,*3*) ∗ ∗∗, *of* λ *a b c* . *a*]
unfolding ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *fst-conv snd-conv*
by *blast*

have *length xs′* < *length xss*
using *simple-cg-closure-phase-1-helper′-length*[*of x x1 xs*]
unfolding ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *Cons* **by** *auto*

have (⋀*x2 u v*. *x2* ∈ *list.set* (*x1′* # *don*) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using ∗∗∗ *less.prems*(*1*)
by (*metis set-ConsD*)

have *xs′* = *snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))
using ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› **by** *auto*
have (⋀*x2 u v*. *x2* ∈ *list.set xs′* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *simple-cg-closure-phase-1-helper′-validity-snd*[*of xs′ M1*]
unfolding ‹*xs′* = *snd* (*snd* (*simple-cg-closure-phase-1-helper′ x x1 xs*))›

using ** *simple-cg-closure-phase-1-helper′-validity-snd* **by** *blast*

have *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′* # *don*)))
using *less.prems*(*3*) **unfolding** ‹*simple-cg-closure-phase-1-helper x xss* (*b*,*don*) = *simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′* # *don*)› **.**
then show *?thesis*
using *less.hyps*[*OF* ‹*length xs′* < *length xss*› ‹(⋀*x2 u v*. *x2* ∈ *list.set* (*x1′* # *don*) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)› ‹(⋀*x2 u v*. *x2* ∈ *list.set xs′* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)›, *of x1′*#*don* λ *a b c* . *a* λ *a b c* . *a*]
by *force*
qed
qed

lemma *simple-cg-closure-phase-1-helper-length* :
length (*snd* (*simple-cg-closure-phase-1-helper x xss* (*b*,*don*))) ≤ *length xss* + *length don*
proof (*induction length xss arbitrary*: *xss b don rule*: *less-induct*)
case *less*
show *?case* **proof** (*cases xss*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*Cons x1 xs*)
obtain *b′ x1′ xs′* **where** *simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)
using *prod.exhaust* **by** *metis*
then have *simple-cg-closure-phase-1-helper x xss* (*b*,*don*) = *simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′* # *don*)
unfolding *Cons* **by** *auto*

have *length xs′* < *length xss*
using *simple-cg-closure-phase-1-helper′-length*[*of x x1 xs*]
unfolding ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *Cons* **by** *auto*
then have *length* (*snd* (*simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′*#*don*))) ≤ *length xs′* + *length* (*x1′*#*don*)
using *less*[*of xs′*] **unfolding** *Cons* **by** *blast*
moreover have *length xs′* + *length* (*x1′*#*don*) ≤ *length xss* + *length don*
using *simple-cg-closure-phase-1-helper′-length*[*of x x1 xs*]
unfolding ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *snd-conv Cons* **by** *auto*
ultimately show *?thesis*
unfolding ‹*simple-cg-closure-phase-1-helper x xss* (*b*,*don*) = *simple-cg-closure-phase-1-helper x xs′* (*b* ∨ *b′*, *x1′* # *don*)›
by *presburger*

```
  qed
qed


lemma simple-cg-closure-phase-1-helper-True :
  assumes fst (simple-cg-closure-phase-1-helper x xss (False,don))
  and     xss ≠ []
shows length (snd (simple-cg-closure-phase-1-helper x xss (False,don))) < length
xss + length don
  using assms
proof (induction length xss arbitrary: xss don rule: less-induct)
  case less
  show ?case proof (cases xss)
    case Nil
    then show ?thesis using less.prems(2) by auto
  next
    case (Cons x1 xs)
   obtain b' x1' xs' where simple-cg-closure-phase-1-helper' x x1 xs = (b',x1',xs')
      using prod.exhaust by metis
   then have simple-cg-closure-phase-1-helper x xss (False,don) = simple-cg-closure-phase-1-helper
x xs' (b', x1' # don)
      unfolding Cons by auto

    show ?thesis proof (cases b')
      case True
      then have length xs' < length xs
        using simple-cg-closure-phase-1-helper'-True[of x x1 xs]
        unfolding ‹simple-cg-closure-phase-1-helper' x x1 xs = (b',x1',xs')› fst-conv
snd-conv
        by blast
     then have length (snd (simple-cg-closure-phase-1-helper x xs' (b', x1' # don)))
< length xss + length don
        using simple-cg-closure-phase-1-helper-length[of x xs' b' x1'#don]
        unfolding Cons
        by auto
      then show ?thesis
            unfolding ‹simple-cg-closure-phase-1-helper x xss (False,don) = sim-
ple-cg-closure-phase-1-helper x xs' (b', x1' # don)› .
    next
      case False
    then have simple-cg-closure-phase-1-helper x xss (False,don) = simple-cg-closure-phase-1-helper
x xs' (False, x1' # don)
      using ‹simple-cg-closure-phase-1-helper x xss (False,don) = simple-cg-closure-phase-1-helper
x xs' (b', x1' # don)›
        by auto
      then have fst (simple-cg-closure-phase-1-helper x xs' (False, x1' # don))
        using less.prems(1) by auto

      have length xs' < length xss
```

using *simple-cg-closure-phase-1-helper′-length*[*of x x1 xs*]
unfolding ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *Cons*
by *auto*

have *xs′* ≠ []
using ‹*simple-cg-closure-phase-1-helper′ x x1 xs* = (*b′*,*x1′*,*xs′*)› *False*
by (*metis* ‹*fst* (*simple-cg-closure-phase-1-helper x xs′* (*False*, *x1′* # *don*))› *simple-cg-closure-phase-1-helper.simps*(*1*) *fst-eqD*)

show *?thesis*
using *less.hyps*[*OF* ‹*length xs′* < *length xss*› ‹*fst* (*simple-cg-closure-phase-1-helper x xs′* (*False*, *x1′* # *don*))› ‹*xs′* ≠ []›] ‹*length xs′* < *length xss*›
unfolding ‹*simple-cg-closure-phase-1-helper x xss* (*False*,*don*) = *simple-cg-closure-phase-1-helper x xs′* (*False*, *x1′* # *don*)›
unfolding *Cons*
by *auto*
qed
qed
qed

fun *simple-cg-closure-phase-1* :: *′a simple-cg* ⇒ (*bool* × *′a simple-cg*) **where**
simple-cg-closure-phase-1 xs = *foldl* (λ (*b*,*xs*) *x*. *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) *xs*

lemma *simple-cg-closure-phase-1-validity* :
assumes *observable M1* **and** *observable M2*
and ⋀ *x2 u v* . *x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-1 xs*))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1* **and** *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
proof −

have ⋀ *xss x2 u v* . (⋀ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*) ⟹ *x2* ∈ *list.set* (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) *xss*)) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
proof −
fix *xss x2 u v*
assume ⋀ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) *xss*))

and u |∈| $x2$
and v |∈| $x2$
and $u \in L\ M1$
and $u \in L\ M2$

then show *converge M1 u v* ∧ *converge M2 u v*
proof (*induction xss arbitrary*: *x2 u v rule*: *rev-induct*)
case *Nil*
then have $x2 \in$ *list.set xs*
by *auto*
then show *?case*
using *Nil.prems*(*3*,*4*,*5*,*6*) *assms*(*3*) **by** *blast*
next
case (*snoc x xss*)

obtain *b xss′* **where** (*foldl* (λ (*b*,*xs*) *x* . *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) *xss*) = (*b*,*xss′*)
using *prod.exhaust* **by** *metis*
moreover obtain *b′ xss″* **where** *simple-cg-closure-phase-1-helper x xss′* (*False*,[]) = (*b′*,*xss″*)
using *prod.exhaust* **by** *metis*
ultimately have ∗:(*foldl* (λ (*b*,*xs*) *x* . *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) (*xss*@[*x*])) = (*b*∨*b′*,*xss″*)
by *auto*

have (⋀*u v*. *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *snoc.prems*(*1*)
by (*metis append-Cons list.set-intros*(*1*) *list-set-sym*)
moreover have (⋀*x2 u v*. *x2* ∈ *list.set* [] ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
by *auto*
moreover have (⋀*x2 u v*. *x2* ∈ *list.set xss′* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
proof −
have (⋀*x2 u v*. *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *snoc.prems*(*1*)
by (*metis* (*no-types*, *lifting*) *append-Cons append-Nil2 insertCI list.simps*(*15*) *list-set-sym*)
then show (⋀*x2 u v*. *x2* ∈ *list.set xss′* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *snoc.IH*
unfolding ‹(*foldl* (λ (*b*,*xs*) *x* . *let* (*b′*,*xs′*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*∨*b′*,*xs′*)) (*False*,*xs*) *xss*) = (*b*,*xss′*)› *snd-conv*
by *blast*
qed
ultimately have (⋀*x2 u v*. *x2* ∈ *list.set xss″* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)

using *simple-cg-closure-phase-1-helper-validity*[*OF assms*(*1*,*2*), *of x* [] *xss'* -
False]
unfolding ‹*simple-cg-closure-phase-1-helper x xss'* (*False*,[]) = (*b'*,*xss''*)›
snd-conv
by *blast*
then show *?case*
using *snoc.prems*(*2*,*3*,*4*,*5*,*6*)
unfolding ∗ *snd-conv*
by *blast*
qed
qed

then show *?thesis*
using *assms*(*3*,*4*,*5*,*6*,*7*,*8*)
unfolding *simple-cg-closure-phase-1.simps*
by *blast*
qed

lemma *simple-cg-closure-phase-1-length-helper* :
length (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs*
(*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) *xss*)) ≤ *length xs*
proof (*induction xss rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xss*)

obtain *b xss'* **where** (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper*
x xs (*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) *xss*) = (*b*,*xss'*)
using *prod.exhaust* **by** *metis*
moreover obtain *b' xss''* **where** *simple-cg-closure-phase-1-helper x xss'* (*False*,[])
= (*b'*,*xss''*)
using *prod.exhaust* **by** *metis*
ultimately have ∗:(*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper*
x xs (*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) (*xss*@[*x*])) = (*b*∨*b'*,*xss''*)
by *auto*

have *length xss'* ≤ *length xs*
using *snoc.IH*
unfolding ‹(*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x*
xs (*False*,[]) *in* (*b*∨*b'*,*xs'*)) (*False*,*xs*) *xss*) = (*b*,*xss'*)›
by *auto*
moreover have *length xss''* ≤ *length xss'*
using *simple-cg-closure-phase-1-helper-length*[*of x xss' False* []]
unfolding ‹*simple-cg-closure-phase-1-helper x xss'* (*False*,[]) = (*b'*,*xss''*)›
by *auto*
ultimately show *?case*
unfolding ∗ *snd-conv*
by *simp*

qed

lemma *simple-cg-closure-phase-1-length* :
length (*snd* (*simple-cg-closure-phase-1 xs*)) $\leq$ *length xs*
using *simple-cg-closure-phase-1-length-helper* **by** *auto*

lemma *simple-cg-closure-phase-1-True* :
assumes *fst* (*simple-cg-closure-phase-1 xs*)
shows *length* (*snd* (*simple-cg-closure-phase-1 xs*)) $<$ *length xs*
proof −
have $\bigwedge$ *xss* . *fst* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*) $\Longrightarrow$ *length* (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*)) $<$ *length xs*
proof −
fix *xss*
assume *fst* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*)
then show *length* (*snd* (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*)) $<$ *length xs*
proof (*induction xss rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xss*)

obtain *b xss'* **where** (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*) = (*b*,*xss'*)
using *prod.exhaust* **by** *metis*
moreover obtain *b' xss''* **where** *simple-cg-closure-phase-1-helper x xss'* (*False*,[]) = (*b'*,*xss''*)
using *prod.exhaust* **by** *metis*
ultimately have (*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) (*xss*@[*x*])) = (*b*$\vee$*b'*,*xss''*)
by *auto*

consider *b* | *b'*
using *snoc.prems*
unfolding ‹(*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) (*xss*@[*x*])) = (*b*$\vee$*b'*,*xss''*)› *fst-conv*
by *blast*
then show *?case* **proof** *cases*
case *1*
then have *length xss'* $<$ *length xs*
using *snoc.IH*
unfolding ‹(*foldl* (λ (*b*,*xs*) *x* . *let* (*b'*,*xs'*) = *simple-cg-closure-phase-1-helper x xs* (*False*,[]) *in* (*b*$\vee$*b'*,*xs'*)) (*False*,*xs*) *xss*) = (*b*,*xss'*)› *fst-conv snd-conv*
by *auto*
moreover have *length xss''* $\leq$ *length xss'*

```
        using simple-cg-closure-phase-1-helper-length[of x xss' False []]
        unfolding ‹simple-cg-closure-phase-1-helper x xss' (False,[]) = (b',xss'')›
        by auto
      ultimately show ?thesis
      unfolding ‹(foldl (λ (b,xs) x . let (b',xs') = simple-cg-closure-phase-1-helper
x xs (False,[]) in (b∨b',xs')) (False,xs) (xss@[x])) = (b∨b',xss'')› snd-conv
        by simp
    next
      case 2
      have length xss' ≤ length xs
        using simple-cg-closure-phase-1-length-helper[of xss xs]
      by (metis ‹foldl (λ(b, xs) x. let (b', xs') = simple-cg-closure-phase-1-helper x
xs (False, []) in (b ∨ b', xs')) (False, xs) xss = (b, xss')› simple-cg-closure-phase-1-length-helper
snd-conv)
      moreover have length xss'' < length xss'
      proof −
        have xss' ≠ []
          using 2 ‹simple-cg-closure-phase-1-helper x xss' (False, []) = (b', xss'')›
by auto
        then show ?thesis
          using simple-cg-closure-phase-1-helper-True[of x xss' []] 2
         unfolding ‹simple-cg-closure-phase-1-helper x xss' (False,[]) = (b',xss'')›
fst-conv snd-conv
          by auto
      qed
      ultimately show ?thesis
      unfolding ‹(foldl (λ (b,xs) x . let (b',xs') = simple-cg-closure-phase-1-helper
x xs (False,[]) in (b∨b',xs')) (False,xs) (xss@[x])) = (b∨b',xss'')› snd-conv
        by simp
    qed
  qed
qed
then show ?thesis
  using assms by auto
qed
```

```
fun can-merge-by-intersection :: 'a list fset ⇒ 'a list fset ⇒ bool where
  can-merge-by-intersection x1 x2 = (∃ α . α |∈| x1 ∧ α |∈| x2)

lemma can-merge-by-intersection-code[code] :
  can-merge-by-intersection x1 x2 = (∃ α ∈ fset x1 . α |∈| x2)
  unfolding can-merge-by-intersection.simps
  by metis


lemma can-merge-by-intersection-validity :
```

assumes $\bigwedge$ *u v . u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and $\bigwedge$ *u v . u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *can-merge-by-intersection x1 x2*
and *u* |∈| (*x1* |∪| *x2*)
and *v* |∈| (*x1* |∪| *x2*)
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
proof −
obtain α **where** α |∈| *x1* **and** α |∈| *x2*
using *assms*(*3*) **by** *auto*

have *converge M1 u* α ∧ *converge M2 u* α
using ‹α |∈| *x1*› ‹α |∈| *x2*› *assms*(*1*,*2*,*4*,*6*,*7*) **by** *blast*
moreover have *converge M1 v* α ∧ *converge M2 v* α
by (*metis* (*no-types*, *opaque-lifting*) ‹α |∈| *x1*› ‹α |∈| *x2*› *assms*(*1*) *assms*(*2*) *assms*(*5*) *calculation converge.simps funion-iff*)
ultimately show *?thesis*
by *simp*
qed

fun *simple-cg-closure-phase-2-helper* :: *'a list fset* ⇒ *'a simple-cg* ⇒ (*bool* × *'a list fset* × *'a simple-cg*) **where**
simple-cg-closure-phase-2-helper x1 xs =
(*let* (*x2s*,*others*) = *separate-by* (*can-merge-by-intersection x1*) *xs*;
x1Union = *foldl* (|∪|) *x1 x2s*
in (*x2s* ≠ [],*x1Union*,*others*))

lemma *simple-cg-closure-phase-2-helper-length* :
length (*snd* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))) ≤ *length xs*
by *auto*

lemma *simple-cg-closure-phase-2-helper-validity-fst* :
assumes $\bigwedge$ *u v . u* |∈| *x1* ⟹ *v* |∈| *x1* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and $\bigwedge$ *x2 u v . x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *u* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))
and *v* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
proof −

have ∗:$\bigwedge$ *w . w* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*)) ⟹ *w* |∈| *x1* ∨ (∃ *x2 . x2* ∈ *list.set xs* ∧ *w* |∈| *x2* ∧ *can-merge-by-intersection x1 x2*)

proof −
fix *w* **assume** *w* |∈| *fst* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))
then have *w* |∈| *ffUnion* (*fset-of-list* (*x1*#(*filter* (*can-merge-by-intersection x1*) *xs*)))
using *foldl-funion-fsingleton*[**where** *xs*=(*filter* (*can-merge-by-intersection x1*) *xs*)]
by *auto*

then obtain *x2* **where** *w* |∈| *x2*
and *x2* |∈| *fset-of-list* (*x1*#(*filter* (*can-merge-by-intersection x1*) *xs*))
using *ffUnion-fmember-ob*
by *metis*
then consider *x2*=*x1* | *x2* ∈ *list.set* (*filter* (*can-merge-by-intersection x1*) *xs*)
by (*meson fset-of-list-elem set-ConsD*)
then show *w* |∈| *x1* ∨ (∃ *x2* . *x2* ∈ *list.set xs* ∧ *w* |∈| *x2* ∧ *can-merge-by-intersection x1 x2*)
using ‹*w* |∈| *x2*› **by** (*cases*; *auto*)
qed

consider *u* |∈| *x1* | (∃ *x2* . *x2* ∈ *list.set xs* ∧ *u* |∈| *x2* ∧ *can-merge-by-intersection x1 x2*)
using ∗[*OF assms*(*3*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *1*

consider (*a*) *v* |∈| *x1* | (*b*) (∃ *x2* . *x2* ∈ *list.set xs* ∧ *v* |∈| *x2* ∧ *can-merge-by-intersection x1 x2*)
using ∗[*OF assms*(*4*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *a*
then show *?thesis* **using** *assms*(*1*)[*OF 1 - assms*(*5*,*6*)] **by** *auto*
next
case *b*
then obtain *x2v* **where** *x2v* ∈ *list.set xs* **and** *v* |∈| *x2v* **and** *can-merge-by-intersection x1 x2v*
using ∗[*OF assms*(*3*)]
by *blast*
show *?thesis*
using *can-merge-by-intersection-validity*[*of x1 M1 M2 x2v*, *OF assms*(*1*) *assms*(*2*)[*OF* ‹*x2v* ∈ *list.set xs*›] ‹*can-merge-by-intersection x1 x2v*›]
using *1* ‹*v* |∈| *x2v*› *assms*(*5*,*6*)
by *blast*
qed
next
case *2*
then obtain *x2u* **where** *x2u* ∈ *list.set xs* **and** *u* |∈| *x2u* **and** *can-merge-by-intersection x1 x2u*
using ∗[*OF assms*(*3*)]

by *blast*
obtain αu **where** αu $|\in|$ *x1* **and** αu $|\in|$ *x2u*
using ‹*can-merge-by-intersection x1 x2u*› **by** *auto*

consider (*a*) *v* $|\in|$ *x1* | (*b*) ($\exists$ *x2* . *x2* $\in$ *list.set xs* $\wedge$ *v* $|\in|$ *x2* $\wedge$ *can-merge-by-intersection x1 x2*)
using $*$[*OF assms*(*4*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *a*

show *?thesis*
using *can-merge-by-intersection-validity*[*of x1 M1 M2 x2u*, *OF assms*(*1*) *assms*(*2*)[*OF* ‹*x2u* $\in$ *list.set xs*›] ‹*can-merge-by-intersection x1 x2u*›]
using ‹*u* $|\in|$ *x2u*› *a assms*(*5*,*6*)
by *blast*
next
case *b*

then obtain *x2v* **where** *x2v* $\in$ *list.set xs* **and** *v* $|\in|$ *x2v* **and** *can-merge-by-intersection x1 x2v*
using $*$[*OF assms*(*4*)]
by *blast*
obtain αv **where** αv $|\in|$ *x1* **and** αv $|\in|$ *x2v*
using ‹*can-merge-by-intersection x1 x2v*› **by** *auto*

have $\bigwedge$ *v* . *v* $|\in|$ *x1* $|\cup|$ *x2u* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
using *can-merge-by-intersection-validity*[*of x1 M1 M2 x2u*, *OF assms*(*1*) *assms*(*2*)[*OF* ‹*x2u* $\in$ *list.set xs*›] ‹*can-merge-by-intersection x1 x2u*› - - *assms*(*5*,*6*)] ‹*u* $|\in|$ *x2u*›
by *blast*
have $\bigwedge$ *u* . *u* $|\in|$ *x1* $|\cup|$ *x2v* $\Longrightarrow$ *u* $\in$ *L M1* $\Longrightarrow$ *u* $\in$ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*
using *can-merge-by-intersection-validity*[*of x1 M1 M2 x2v*, *OF assms*(*1*) *assms*(*2*)[*OF* ‹*x2v* $\in$ *list.set xs*›] ‹*can-merge-by-intersection x1 x2v*›] ‹*v* $|\in|$ *x2v*›
by *blast*

show *?thesis*
using ‹$\bigwedge$*u*. $[\![$*u* $|\in|$ *x1* $|\cup|$ *x2v*; *u* $\in$ *L M1*; *u* $\in$ *L M2*$]\!]$ $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*› ‹$\bigwedge$*v*. *v* $|\in|$ *x1* $|\cup|$ *x2u* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*› ‹αu $|\in|$ *x1*› **by** *fastforce*
qed
qed
qed

lemma *simple-cg-closure-phase-2-helper-validity-snd* :
assumes $\bigwedge$ *x2 u v* . *x2* $\in$ *list.set xs* $\Longrightarrow$ *u* $|\in|$ *x2* $\Longrightarrow$ *v* $|\in|$ *x2* $\Longrightarrow$ *u* $\in$ *L M1* $\Longrightarrow$ *u* $\in$ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*

and $x2 \in$ *list.set* (*snd* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*)))
and u |∈| $x2$
and v |∈| $x2$
and $u \in L\ M1$
and $u \in L\ M2$
shows *converge M1 u v* ∧ *converge M2 u v*
proof −
have *list.set* (*snd* (*snd* (*simple-cg-closure-phase-2-helper x1 xs*))) ⊆ *list.set xs*
by *auto*
then show *?thesis*
using *assms* **by** *blast*
qed

lemma *simple-cg-closure-phase-2-helper-True* :
assumes *fst* (*simple-cg-closure-phase-2-helper x xs*)
shows *length* (*snd* (*snd* (*simple-cg-closure-phase-2-helper x xs*))) < *length xs*
proof −
have *snd* (*snd* (*simple-cg-closure-phase-2-helper x xs*)) = *filter* (λ*x2* . ¬ (*can-merge-by-intersection x x2*)) *xs*
by *auto*
moreover have *filter* (λ*x2* . (*can-merge-by-intersection x x2*)) *xs* ≠ []
using *assms* **unfolding** *simple-cg-closure-phase-1-helper′.simps Let-def separate-by.simps*
by *fastforce*
ultimately show *?thesis*
using *filter-not-all-length*[*of can-merge-by-intersection x xs*]
by *metis*
qed

function *simple-cg-closure-phase-2′* :: *′a simple-cg* ⇒ (*bool* × *′a simple-cg*) ⇒ (*bool* × *′a simple-cg*) **where**
simple-cg-closure-phase-2′ [] (*b*,*done*) = (*b*,*done*) |
simple-cg-closure-phase-2′ (*x*#*xs*) (*b*,*done*) = (*let* (*hasChanged*,*x′*,*xs′*) = *simple-cg-closure-phase-2-helper x xs*
in if hasChanged then simple-cg-closure-phase-2′ xs′ (*True*,*x′*#*done*)
else simple-cg-closure-phase-2′ xs (*b*,*x*#*done*))
by *pat-completeness auto*
termination
proof −
{
fix *xa* :: (*bool* × *′a list fset* × *′a simple-cg*)
fix *x xs b don xb y xaa ya*
assume *xa* = *simple-cg-closure-phase-2-helper x xs*
and (*xb*, *y*) = *xa*
and (*xaa*, *ya*) = *y*
and *xb*

have *length ya* < *Suc* (*length xs*)

using *simple-cg-closure-phase-2-helper-True*[*of x xs*] ‹*xb*›
unfolding ‹*xa* = *simple-cg-closure-phase-2-helper x xs*›[*symmetric*]
unfolding ‹(*xb*, *y*) = *xa*›[*symmetric*] ‹(*xaa*, *ya*) = *y*›[*symmetric*]
unfolding *fst-conv snd-conv*
by *auto*

then have ((*ya*, *True*, *xaa* # *don*), *x* # *xs*, *b*, *don*) ∈ *measure* (λ(*xs*, *bd*). *length xs*)
by *auto*
}
then show *?thesis*
apply (*relation measure* (λ (*xs*,*bd*) . *length xs*))
by *force*+
qed

lemma *simple-cg-closure-phase-2′-validity* :
assumes ⋀ *x2 u v* . *x2* ∈ *list.set don* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and ⋀ *x2 u v* . *x2* ∈ *list.set xss* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-2′ xss* (*b*,*don*)))
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
using *assms*(*1*,*2*,*3*)
proof (*induction length xss arbitrary*: *xss b don rule*: *less-induct*)
case *less*
show *?case* **proof** (*cases xss*)
case *Nil*
show *?thesis* **using** *less.prems*(*3*) *less.prems*(*1*)[*OF - assms*(*4*,*5*,*6*,*7*)] **unfolding** *Nil*
by *auto*
next
case (*Cons x xs*)
obtain *hasChanged x′ xs′* **where** *simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)
using *prod.exhaust* **by** *metis*

show *?thesis* **proof** (*cases hasChanged*)
case *True*
then have *simple-cg-closure-phase-2′ xss* (*b*,*don*) = *simple-cg-closure-phase-2′ xs′* (*True*,*x′*#*don*)
using ‹*simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)›
unfolding *Cons*
by *auto*

have ∗:(⋀*u v*. *u* |∈| *x* ⟹ *v* |∈| *x* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge*

M1 u v ∧ *converge M2 u v*) **and**
**:(⋀*x2 u v. x2* ∈ *list.set xs* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *less.prems*(*2*) **unfolding** *Cons*
by (*meson list.set-intros*)+

have *length xs′* < *length xss*
unfolding *Cons*
using *simple-cg-closure-phase-2-helper-True*[*of x xs*] *True*
unfolding ‹*simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)› *fst-conv snd-conv*
by *auto*
moreover have (⋀*x2 u v. x2* ∈ *list.set* (*x′* # *don*) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *simple-cg-closure-phase-2-helper-validity-fst*[*of x M1 M2 xs*, *OF* ∗ **, *of* λ *a b c . a*]
using *less.prems*(*1*)
unfolding ‹*simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)› *fst-conv snd-conv*
using *set-ConsD*[*of - x′ don*]
by *blast*
moreover have (⋀*x2 u v. x2* ∈ *list.set xs′* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *simple-cg-closure-phase-2-helper-validity-snd*[*of xs M1 M2 - x*, *OF* **, *of* λ *a b c . a*]
unfolding ‹*simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)› *fst-conv snd-conv*
by *blast*
moreover have *x2* ∈ *list.set* (*snd* (*simple-cg-closure-phase-2′ xs′* (*True*, *x′* # *don*)))
using *less.prems*(*3*) **unfolding** ‹*simple-cg-closure-phase-2′ xss* (*b*,*don*) = *simple-cg-closure-phase-2′ xs′* (*True*,*x′*#*don*)› **.**
ultimately show *?thesis*
using *less.hyps*[*of xs′ x′*#*don*]
by *blast*
next
case *False*
then have *simple-cg-closure-phase-2′ xss* (*b*,*don*) = *simple-cg-closure-phase-2′ xs* (*b*,*x*#*don*)
using ‹*simple-cg-closure-phase-2-helper x xs* = (*hasChanged*,*x′*,*xs′*)›
unfolding *Cons*
by *auto*

have *length xs* < *length xss*
unfolding *Cons* **by** *auto*
moreover have (⋀*x2 u v. x2* ∈ *list.set* (*x* # *don*) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *less.prems*(*1*,*2*) **unfolding** *Cons*
by (*metis list.set-intros*(*1*) *set-ConsD*)

```
      moreover have (⋀x2 u v. x2 ∈ list.set xs ⟹ u |∈| x2 ⟹ v |∈| x2 ⟹ u
∈ L M1 ⟹ u ∈ L M2 ⟹ converge M1 u v ∧ converge M2 u v)
        using less.prems(2) unfolding Cons
        by (metis list.set-intros(2))
       moreover have x2 ∈ list.set (snd (simple-cg-closure-phase-2' xs (b, x #
don)))
        using less.prems(3)
      unfolding ‹simple-cg-closure-phase-2' xss (b,don) = simple-cg-closure-phase-2'
xs (b,x#don)›
        unfolding Cons .
      ultimately show ?thesis
        using less.hyps[of xs x#don b]
        by blast
    qed
  qed
qed


lemma simple-cg-closure-phase-2'-length :
  length (snd (simple-cg-closure-phase-2' xss (b,don))) ≤ length xss + length don
proof (induction length xss arbitrary: xss b don rule: less-induct)
  case less
  show ?case proof (cases xss)
    case Nil
    then show ?thesis by auto
  next
    case (Cons x xs)
   obtain hasChanged x' xs' where simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')
      using prod.exhaust by metis

    show ?thesis proof (cases hasChanged)
      case True
     then have simple-cg-closure-phase-2' xss (b,don) = simple-cg-closure-phase-2'
xs' (True,x'#don)
        using ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
        unfolding Cons
        by auto
      have length xs' < length xss
        using simple-cg-closure-phase-2-helper-True[of x xs] True
          unfolding ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
snd-conv fst-conv
        unfolding Cons
        by auto

      then show ?thesis
        using less.hyps[of xs' True x'#don]
      unfolding ‹simple-cg-closure-phase-2' xss (b,don) = simple-cg-closure-phase-2'
xs' (True,x'#don)›
        unfolding Cons by auto
```

```
    next
      case False
     then have simple-cg-closure-phase-2' xss (b,don) = simple-cg-closure-phase-2'
xs (b,x#don)
        using ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
        unfolding Cons
        by auto

      show ?thesis
        using less.hyps[of xs b x#don]
      unfolding ‹simple-cg-closure-phase-2' xss (b,don) = simple-cg-closure-phase-2'
xs (b,x#don)›
        unfolding Cons
        by auto
    qed
  qed
qed


lemma simple-cg-closure-phase-2'-True :
  assumes fst (simple-cg-closure-phase-2' xss (False,don))
  and     xss ≠ []
shows length (snd (simple-cg-closure-phase-2' xss (False,don))) < length xss +
length don
  using assms
proof (induction length xss arbitrary: xss don rule: less-induct)
  case less
  show ?case proof (cases xss)
    case Nil
    then show ?thesis
      using less.prems(2) by auto
  next
    case (Cons x xs)
   obtain hasChanged x' xs' where simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')
      using prod.exhaust by metis

    show ?thesis proof (cases hasChanged)
      case True
     then have simple-cg-closure-phase-2' xss (False,don) = simple-cg-closure-phase-2'
xs' (True,x'#don)
        using ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
        unfolding Cons
        by auto
      have length xs' < length xs
        using simple-cg-closure-phase-2-helper-True[of x xs] True
          unfolding ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
snd-conv fst-conv
        unfolding Cons
        by auto
```

```
      moreover have length (snd (simple-cg-closure-phase-2' xs' (True,x'#don)))
≤ length xs' + length (x'#don)
        using simple-cg-closure-phase-2'-length by metis
      ultimately show ?thesis
      unfolding ‹simple-cg-closure-phase-2' xss (False,don) = simple-cg-closure-phase-2'
xs' (True,x'#don)›
        unfolding Cons
        by auto
    next
      case False
    then have simple-cg-closure-phase-2' xss (False,don) = simple-cg-closure-phase-2'
xs (False,x#don)
        using ‹simple-cg-closure-phase-2-helper x xs = (hasChanged,x',xs')›
        unfolding Cons
        by auto

      have xs ≠ []
      using ‹simple-cg-closure-phase-2' xss (False, don) = simple-cg-closure-phase-2'
xs (False, x # don)› less.prems(1) by auto

      show ?thesis
        using less.hyps[of xs x#don, OF - - ‹xs ≠ []›]
        using less.prems(1)
      unfolding ‹simple-cg-closure-phase-2' xss (False,don) = simple-cg-closure-phase-2'
xs (False,x#don)›
        unfolding Cons
        by auto
    qed
  qed
qed



fun simple-cg-closure-phase-2 :: 'a simple-cg ⇒ (bool × 'a simple-cg) where
  simple-cg-closure-phase-2 xs = simple-cg-closure-phase-2' xs (False,[])


lemma simple-cg-closure-phase-2-validity :
  assumes ⋀ x2 u v . x2 ∈ list.set xss ⟹ u |∈| x2 ⟹ v |∈| x2 ⟹ u ∈ L M1
⟹ u ∈ L M2 ⟹ converge M1 u v ∧ converge M2 u v
  and     x2 ∈ list.set (snd (simple-cg-closure-phase-2 xss))
  and     u |∈| x2
  and     v |∈| x2
  and     u ∈ L M1
  and     u ∈ L M2
shows converge M1 u v ∧ converge M2 u v
  using assms(2)
  unfolding simple-cg-closure-phase-2.simps
  using simple-cg-closure-phase-2'-validity[OF - assms(1) - assms(3,4,5,6), of []
```

```
xss λ a b c . a False]
  by auto

lemma simple-cg-closure-phase-2-length :
  length (snd (simple-cg-closure-phase-2 xss)) ≤ length xss
  unfolding simple-cg-closure-phase-2.simps
  using simple-cg-closure-phase-2'-length[of xss False []]
  by auto

lemma simple-cg-closure-phase-2-True :
  assumes fst (simple-cg-closure-phase-2 xss)
shows length (snd (simple-cg-closure-phase-2 xss)) < length xss
proof −
  have xss ≠ []
    using assms by auto
  then show ?thesis
    using simple-cg-closure-phase-2'-True[of xss []] assms by auto
qed



function simple-cg-closure :: 'a simple-cg ⇒ 'a simple-cg where
  simple-cg-closure g = (let (hasChanged1,g1) = simple-cg-closure-phase-1 g;
                         (hasChanged2,g2) = simple-cg-closure-phase-2 g1
     in if hasChanged1 ∨ hasChanged2
        then simple-cg-closure g2
        else g2)
  by pat-completeness auto
termination
proof −
  {
    fix g :: 'a simple-cg
    fix x hasChanged1 g1 xb hasChanged2 g2
    assume x = simple-cg-closure-phase-1 g
           (hasChanged1, g1) = x
           xb = simple-cg-closure-phase-2 g1
           (hasChanged2, g2) = xb
           hasChanged1 ∨ hasChanged2

    then have simple-cg-closure-phase-1 g = (hasChanged1, g1)
          and simple-cg-closure-phase-2 g1 = (hasChanged2, g2)
      by auto

    have length g1 ≤ length g
      using ‹simple-cg-closure-phase-1 g = (hasChanged1, g1)›
      using simple-cg-closure-phase-1-length[of g]
      by auto
    have length g2 ≤ length g1
      using ‹simple-cg-closure-phase-2 g1 = (hasChanged2, g2)›
```

```
      using simple-cg-closure-phase-2-length[of g1]
      by auto


    consider hasChanged1 | hasChanged2
      using ‹hasChanged1 ∨ hasChanged2› by blast
    then have length g2 < length g
    proof cases
      case 1
      then have length g1 < length g
        using ‹simple-cg-closure-phase-1 g = (hasChanged1, g1)›
        using simple-cg-closure-phase-1-True[of g]
        by auto
      then show ?thesis
        using ‹length g2 ≤ length g1›
        by linarith
    next
      case 2
      then have length g2 < length g1
        using ‹simple-cg-closure-phase-2 g1 = (hasChanged2, g2)›
        using simple-cg-closure-phase-2-True[of g1]
        by auto
      then show ?thesis
        using ‹length g1 ≤ length g›
        by linarith
    qed
    then have (g2, g) ∈ measure length
      by auto
  }
  then show ?thesis by (relation measure length; force)
qed


lemma simple-cg-closure-validity :
  assumes observable M1 and observable M2
  and     ⋀ x2 u v . x2 ∈ list.set g ⟹ u |∈| x2 ⟹ v |∈| x2 ⟹ u ∈ L M1 ⟹
u ∈ L M2 ⟹ converge M1 u v ∧ converge M2 u v
  and     x2 ∈ list.set (simple-cg-closure g)
  and     u |∈| x2
  and     v |∈| x2
  and     u ∈ L M1
  and     u ∈ L M2
shows converge M1 u v ∧ converge M2 u v
  using assms(3,4)
proof (induction length g arbitrary: g rule: less-induct)
  case less

  obtain hasChanged1 hasChanged2 g1 g2 where simple-cg-closure-phase-1 g =
(hasChanged1, g1)
```

and *simple-cg-closure-phase-2 g1* = (*hasChanged2*, *g2*)
using *prod.exhaust* **by** *metis*

have *length g1* ≤ *length g*
using ‹*simple-cg-closure-phase-1 g* = (*hasChanged1*, *g1*)›
using *simple-cg-closure-phase-1-length*[*of g*]
by *auto*
have *length g2* ≤ *length g1*
using ‹*simple-cg-closure-phase-2 g1* = (*hasChanged2*, *g2*)›
using *simple-cg-closure-phase-2-length*[*of g1*]
by *auto*

have (⋀*x2 u v*. *x2* ∈ *list.set g2* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
proof −
have (⋀*x2 u v*. *x2* ∈ *list.set g1* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *simple-cg-closure-phase-1-validity*[*OF assms*(*1*,*2*), *of g*]
using *less.prems*(*1*)
unfolding ‹*simple-cg-closure-phase-1 g* = (*hasChanged1*, *g1*)› *snd-conv*
by *blast*
then show (⋀*x2 u v*. *x2* ∈ *list.set g2* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
using *simple-cg-closure-phase-2-validity*[*of g1*]
unfolding ‹*simple-cg-closure-phase-2 g1* = (*hasChanged2*, *g2*)› *snd-conv*
by *blast*
qed

show *?thesis* **proof** (*cases hasChanged1* ∨ *hasChanged2*)
case *True*
then consider *hasChanged1* | *hasChanged2*
by *blast*
then have *length g2* < *length g*
proof *cases*
case *1*
then have *length g1* < *length g*
using ‹*simple-cg-closure-phase-1 g* = (*hasChanged1*, *g1*)›
using *simple-cg-closure-phase-1-True*[*of g*]
by *auto*
then show *?thesis*
using ‹*length g2* ≤ *length g1*›
by *linarith*
next
case *2*
then have *length g2* < *length g1*
using ‹*simple-cg-closure-phase-2 g1* = (*hasChanged2*, *g2*)›
using *simple-cg-closure-phase-2-True*[*of g1*]
by *auto*

```
      then show ?thesis
        using ‹length g1 ≤ length g›
        by linarith
    qed
    moreover have x2 ∈ list.set (simple-cg-closure g2)
      using less.prems(2)
     using ‹simple-cg-closure-phase-1 g = (hasChanged1, g1)› ‹simple-cg-closure-phase-2
g1 = (hasChanged2, g2)› True
      by auto
    moreover note ‹(⋀x2 u v. x2 ∈ list.set g2 ⟹ u |∈| x2 ⟹ v |∈| x2 ⟹ u
∈ L M1 ⟹ u ∈ L M2 ⟹ converge M1 u v ∧ converge M2 u v)›
    ultimately show ?thesis
      using less.hyps[of g2]
      by blast
  next
    case False
    then have (simple-cg-closure g) = g2
     using ‹simple-cg-closure-phase-1 g = (hasChanged1, g1)› ‹simple-cg-closure-phase-2
g1 = (hasChanged2, g2)›
      by auto
    show ?thesis
      using less.prems(2)
      using ‹(⋀x2 u v. x2 ∈ list.set g2 ⟹ u |∈| x2 ⟹ v |∈| x2 ⟹ u ∈ L M1
⟹ u ∈ L M2 ⟹ converge M1 u v ∧ converge M2 u v)› assms(5,6,7,8)
      unfolding ‹(simple-cg-closure g) = g2›
      by blast
  qed
qed


fun simple-cg-insert-with-conv :: ('a::linorder) simple-cg ⇒ 'a list ⇒ 'a simple-cg
where
  simple-cg-insert-with-conv g ys = (let
      insert-for-prefix = (λ g i . let
                                pref = take i ys;
                                suff = drop i ys;
                                pref-conv = simple-cg-lookup g pref
                                 in foldl (λ g' ys' . simple-cg-insert' g' (ys'@suff)) g
pref-conv);
      g' = simple-cg-insert g ys;
      g'' = foldl insert-for-prefix g' [0..<length ys]
  in simple-cg-closure g'')


fun simple-cg-merge :: 'a simple-cg ⇒ 'a list ⇒ 'a list ⇒ 'a simple-cg where
  simple-cg-merge g ys1 ys2 = simple-cg-closure (⦃|ys1,ys2|⦄#g)

lemma simple-cg-merge-validity :
```

assumes *observable M1* **and** *observable M2*
and *converge M1 u′ v′* ∧ *converge M2 u′ v′*
and ⋀ *x2 u v* . *x2* ∈ *list.set g* ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*
and *x2* ∈ *list.set* (*simple-cg-merge g u′ v′*)
and *u* |∈| *x2*
and *v* |∈| *x2*
and *u* ∈ *L M1*
and *u* ∈ *L M2*
shows *converge M1 u v* ∧ *converge M2 u v*
proof −
have (⋀*x2 u v*. *x2* ∈ *list.set* ({|*u′*,*v′*|}#*g*) ⟹ *u* |∈| *x2* ⟹ *v* |∈| *x2* ⟹ *u* ∈ *L M1* ⟹ *u* ∈ *L M2* ⟹ *converge M1 u v* ∧ *converge M2 u v*)
proof −
fix *x2 u v* **assume** *x2* ∈ *list.set* ({|*u′*,*v′*|}#*g*) **and** *u* |∈| *x2* **and** *v* |∈| *x2* **and** *u* ∈ *L M1* **and** *u* ∈ *L M2*
then consider *x2* = {|*u′*,*v′*|} | *x2* ∈ *list.set g*
by *auto*
then show *converge M1 u v* ∧ *converge M2 u v* **proof** *cases*
case *1*
then have *u* ∈ {*u′*,*v′*} **and** *v* ∈ {*u′*,*v′*}
using ‹*u* |∈| *x2*› ‹*v* |∈| *x2*› **by** *auto*
then show *?thesis*
using *assms*(*3*)
by (*cases u* = *u′*; *cases v* = *v′*; *auto*)
next
case *2*
then show *?thesis*
using *assms*(*4*) ‹*u* |∈| *x2*› ‹*v* |∈| *x2*› ‹*u* ∈ *L M1*› ‹*u* ∈ *L M2*›
by *blast*
qed
qed
moreover have *x2* ∈ *list.set* (*simple-cg-closure* ({|*u′*,*v′*|}#*g*))
using *assms*(*5*) **by** *auto*
ultimately show *?thesis*
using *simple-cg-closure-validity*[*OF assms*(*1*,*2*) - - *assms*(*6*,*7*,*8*,*9*)]
by *blast*
qed

23.3 Invariants

lemma *simple-cg-lookup-iff* :
β ∈ *list.set* (*simple-cg-lookup G α*) ⟷ (*β* = *α* ∨ (∃ *x* . *x* ∈ *list.set G* ∧ *α* |∈| *x* ∧ *β* |∈| *x*))
proof (*induction G rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x G*)

```
  show ?case proof (cases α |∈| x ∧ β |∈| x)
    case True
    then have β ∈ list.set (simple-cg-lookup (G@[x]) α)
      unfolding simple-cg-lookup.simps
      unfolding sorted-list-of-set-set
      by simp
    then show ?thesis
      using True by auto
  next
    case False
     have β ∈ list.set (simple-cg-lookup (G@[x]) α) = (β = α ∨ (β ∈ list.set
(simple-cg-lookup G α)))
    proof −
      consider α |∉| x | β |∉| x
        using False by blast
     then show β ∈ list.set (simple-cg-lookup (G@[x]) α) = (β = α ∨ (β ∈ list.set
(simple-cg-lookup G α)))
      proof cases
        case 1
        then show ?thesis
          unfolding simple-cg-lookup.simps
          unfolding sorted-list-of-set-set
          by auto
      next
        case 2
        then have β ∉ list.set (sorted-list-of-fset x)
          by simp
        then have (β ∈ list.set (simple-cg-lookup (G@[x]) α)) = (β ∈ Set.insert α
(list.set (simple-cg-lookup G α)))
          unfolding simple-cg-lookup.simps
          unfolding sorted-list-of-set-set
          by auto
        then show ?thesis
          by (induction G; auto)
      qed
    qed
    moreover have (∃ x′ . x′ ∈ list.set (G@[x]) ∧ α |∈| x′ ∧ β |∈| x′) = (∃ x . x
∈ list.set G ∧ α |∈| x ∧ β |∈| x)
      using False by auto
    ultimately show ?thesis
      using snoc.IH
      by blast
  qed
qed


lemma simple-cg-insert′-invar :
  convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert′
proof −
```

have $\bigwedge$ G γ α β . $\gamma \in L$ $M1 \Longrightarrow$
$\gamma \in L$ $M2 \Longrightarrow$
($\bigwedge\alpha$. $\alpha \in L$ $M1 \Longrightarrow \alpha \in L$ $M2 \Longrightarrow \alpha \in$ *list.set* (*simple-cg-lookup* G α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* G α) $\longrightarrow$ *converge* $M1$ α β $\wedge$ *converge* $M2$ α β)) $\Longrightarrow$
$\alpha \in L$ $M1 \Longrightarrow \alpha \in L$ $M2 \Longrightarrow \alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α) $\longrightarrow$ *converge* $M1$ α β $\wedge$ *converge* $M2$ α β)
proof
fix G γ α
assume $\gamma \in L$ $M1$
and $\gamma \in L$ $M2$
and $*$:($\bigwedge\alpha$. $\alpha \in L$ $M1 \Longrightarrow \alpha \in L$ $M2 \Longrightarrow \alpha \in$ *list.set* (*simple-cg-lookup* G α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* G α) $\longrightarrow$ *converge* $M1$ α β $\wedge$ *converge* $M2$ α β))
and $\alpha \in L$ $M1$
and $\alpha \in L$ $M2$

show $\alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α)
unfolding *simple-cg-lookup.simps*
unfolding *sorted-list-of-set-set*
by *auto*

have $\bigwedge$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α) $\Longrightarrow$ *converge* $M1$ α β $\wedge$ *converge* $M2$ α β
proof −
fix β
assume $**$: $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α)
show *converge* $M1$ α β $\wedge$ *converge* $M2$ α β

proof (*cases* $\beta \in$ *list.set* (*simple-cg-lookup* G α))
case *True*
then show *?thesis*
using $*$[*OF* ‹$\alpha \in L$ $M1$› ‹$\alpha \in L$ $M2$›]
by *presburger*
next
case *False*

show *?thesis* **proof** (*cases find* ((|$\in$|) γ) G)
case *None*
then have (*simple-cg-insert'* G γ) = {|γ|}#G
by *auto*

have $\alpha = \gamma \wedge \beta = \gamma$
using *False* ‹$\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* G γ) α)›
unfolding ‹(*simple-cg-insert'* G γ) = {|γ|}#G›
by (*metis fsingleton-iff set-ConsD simple-cg-lookup-iff*)
then show *?thesis*

using ‹$\gamma \in L\ M1$› ‹$\gamma \in L\ M2$› **by** *auto*
next
case (*Some x*)
then have (*simple-cg-insert′ G γ*) = *G*
by *auto*
then show *?thesis*
using ∗[*OF* ‹$\alpha \in L\ M1$› ‹$\alpha \in L\ M2$›] ∗∗
by *presburger*
qed
qed
qed
then show (∀ *β* . *β* ∈ *list.set* (*simple-cg-lookup* (*simple-cg-insert′ G γ*) *α*) ⟶ *converge M1 α β* ∧ *converge M2 α β*)
by *blast*
qed
then show *?thesis*
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
by *blast*
qed

lemma *simple-cg-insert′-foldl-helper*:
assumes *list.set xss* ⊆ *L M1* ∩ *L M2*
and (⋀*α β*. *β* ∈ *list.set* (*simple-cg-lookup G α*) ⟹ *α* ∈ *L M1* ⟹ *α* ∈ *L M2* ⟹ *converge M1 α β* ∧ *converge M2 α β*)
shows (⋀*α β*. *β* ∈ *list.set* (*simple-cg-lookup* (*foldl* (λ *xs′ ys′* . *simple-cg-insert′ xs′ ys′*) *G xss*) *α*) ⟹ *α* ∈ *L M1* ⟹ *α* ∈ *L M2* ⟹ *converge M1 α β* ∧ *converge M2 α β*)
using ‹*list.set xss* ⊆ *L M1* ∩ *L M2*›
proof (*induction xss rule*: *rev-induct*)
case *Nil*
then show *?case*
using ‹(⋀*α β*. *β* ∈ *list.set* (*simple-cg-lookup G α*) ⟹ *α* ∈ *L M1* ⟹ *α* ∈ *L M2* ⟹ *converge M1 α β* ∧ *converge M2 α β*)›
by *auto*
next
case (*snoc x xs*)

have *x* ∈ *L M1* **and** *x* ∈ *L M2*
using *snoc.prems* **by** *auto*

have *list.set xs* ⊆ *L M1* ∩ *L M2*
using *snoc.prems* **by** *auto*
then have ∗:(⋀*α β*. *β* ∈ *list.set* (*simple-cg-lookup* (*foldl* (λ *xs′ ys′*. *simple-cg-insert′ xs′ ys′*) *G xs*) *α*) ⟹ *α* ∈ *L M1* ⟹ *α* ∈ *L M2* ⟹ *converge M1 α β* ∧ *converge M2 α β*)
using *snoc.IH*
by *blast*

have $**$:(*foldl* (λ *xs′ ys′*. *simple-cg-insert′ xs′ ys′*) *G* (*xs*@[*x*])) = *simple-cg-insert′* (*foldl* (λ *xs′ ys′* . *simple-cg-insert′ xs′ ys′*) *G xs*) *x*
by *auto*

show *?case*
using *snoc.prems*(*1*,*2*,*3*) $*$ ‹*x* $\in$ *L M1*› ‹*x* $\in$ *L M2*›
unfolding $**$
using *simple-cg-insert′-invar*[*of M1 M2*]
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
using *simple-cg-lookup-iff*
by *blast*
qed

lemma *simple-cg-insert-invar* :
convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert
proof −

have $\bigwedge$ *G* γ α β . $\gamma \in$ *L M1* $\Longrightarrow$
$\gamma \in$ *L M2* $\Longrightarrow$
($\bigwedge\alpha$. $\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup G* α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β)) $\Longrightarrow$
$\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert G* γ) α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert G* γ) α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β)
proof
fix *G* γ α
assume $\gamma \in$ *L M1*
and $\gamma \in$ *L M2*
and $*$:($\bigwedge\alpha$. $\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup G* α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β))
and $\alpha \in$ *L M1*
and $\alpha \in$ *L M2*

show $\alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert G* γ) α)
unfolding *simple-cg-lookup.simps*
unfolding *sorted-list-of-set-set*
by *auto*

note *simple-cg-insert′-foldl-helper*[*of prefixes* γ *M1 M2*]
moreover have *list.set* (*prefixes* γ) $\subseteq$ *L M1* $\cap$ *L M2*
by (*metis* (*no-types*, *lifting*) *IntI* ‹$\gamma \in$ *L M1*› ‹$\gamma \in$ *L M2*› *language-prefix prefixes-set-ob subsetI*)
ultimately show ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert G* γ) α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β)
using ‹$\alpha \in$ *L M1*› ‹$\alpha \in$ *L M2*›
by (*metis* $*$ *simple-cg-insert.simps*)

qed
then show *?thesis*
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
by *blast*
qed

lemma *simple-cg-closure-invar-helper* :
assumes *observable M1* **and** *observable M2*
and $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
and $\beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (simple\text{-}cg\text{-}closure\ G)\ \alpha)$
and $\alpha \in L\ M1$ **and** $\alpha \in L\ M2$
shows $converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta$
proof $(cases\ \beta = \alpha)$
case *True*
then show *?thesis* **using** *assms*(*5*,*6*) **by** *auto*
next
case *False*
show *?thesis*
proof

obtain x **where** $x \in list.set\ (simple\text{-}cg\text{-}closure\ G)$ **and** $\alpha\ |\in|\ x$ **and** $\beta\ |\in|\ x$
using *False* ‹$\beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (simple\text{-}cg\text{-}closure\ G)\ \alpha)$› **unfolding** *simple-cg-lookup-iff*
by *blast*

have $\bigwedge\ x2\ u\ v\ .\ x2 \in list.set\ G \Longrightarrow u\ |\in|\ x2 \Longrightarrow v\ |\in|\ x2 \Longrightarrow u \in L\ M1 \Longrightarrow u \in L\ M2 \Longrightarrow converge\ M1\ u\ v \wedge converge\ M2\ u\ v$
using ‹$(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$›
unfolding *simple-cg-lookup-iff*
by *blast*

have $(\bigwedge x2\ u\ v.\ x2 \in list.set\ G \Longrightarrow u\ |\in|\ x2 \Longrightarrow v\ |\in|\ x2 \Longrightarrow u \in L\ M1 \Longrightarrow u \in L\ M2 \Longrightarrow converge\ M1\ u\ v \wedge converge\ M2\ u\ v)$
using ‹$(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$›
unfolding *simple-cg-lookup-iff* **by** *blast*
then show $converge\ M1\ \alpha\ \beta$
using ‹$\alpha\ |\in|\ x$› ‹$\beta\ |\in|\ x$› ‹$x \in list.set\ (simple\text{-}cg\text{-}closure\ G)$› *assms*(*1*) *assms*(*2*) *assms*(*5*) *assms*(*6*) *simple-cg-closure-validity* **by** *blast*

have $(\bigwedge x2\ u\ v.\ x2 \in list.set\ G \Longrightarrow u\ |\in|\ x2 \Longrightarrow v\ |\in|\ x2 \Longrightarrow u \in L\ M1 \Longrightarrow u \in L\ M2 \Longrightarrow converge\ M1\ u\ v \wedge converge\ M2\ u\ v)$
using ‹$(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ G\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$›

unfolding *simple-cg-lookup-iff* **by** *blast*
then show *converge M2* α β
using ‹α |∈| x› ‹β |∈| x› ‹$x \in$ *list.set* (*simple-cg-closure G*)› *assms*(*1*) *assms*(*2*) *assms*(*5*) *assms*(*6*) *simple-cg-closure-validity* **by** *blast*
qed
qed

lemma *simple-cg-merge-invar* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-merge-invar M1 M2 simple-cg-lookup simple-cg-merge*
proof −

have $\bigwedge$ G γ γ' α β.
converge M1 γ $\gamma' \Longrightarrow$
converge M2 γ $\gamma' \Longrightarrow$
($\bigwedge\alpha$ β. $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\Longrightarrow \alpha \in$ *L M1* $\Longrightarrow \alpha \in$ *L M2* $\Longrightarrow$ *converge M1* α $\beta \wedge$ *converge M2* α β) $\Longrightarrow$
$\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-merge G* γ γ') α) $\Longrightarrow \alpha \in$ *L M1* $\Longrightarrow \alpha \in$ *L M2* $\Longrightarrow$ *converge M1* α $\beta \wedge$ *converge M2* α β
proof −
fix G γ γ' α β
assume *converge M1* γ γ'
converge M2 γ γ'
($\bigwedge\alpha$ β. $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\Longrightarrow \alpha \in$ *L M1* $\Longrightarrow \alpha \in$ *L M2* $\Longrightarrow$ *converge M1* α $\beta \wedge$ *converge M2* α β)
$\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-merge G* γ γ') α)
$\alpha \in$ *L M1*
$\alpha \in$ *L M2*
show *converge M1* α $\beta \wedge$ *converge M2* α β
proof (*cases* $\beta = \alpha$)
case *True*
then show *?thesis* **using** ‹$\alpha \in$ *L M1*› ‹$\alpha \in$ *L M2*› **by** *auto*
next
case *False*
then obtain x **where** $x \in$ *list.set* (*simple-cg-merge G* γ γ') **and** α |∈| x **and** β |∈| x
using ‹$\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-merge G* γ γ') α)› **unfolding** *simple-cg-lookup-iff*
by *blast*

have ($\bigwedge$*x2 u v*. *x2* $\in$ *list.set G* $\Longrightarrow$ *u* |∈| *x2* $\Longrightarrow$ *v* |∈| *x2* $\Longrightarrow$ *u* $\in$ *L M1* $\Longrightarrow$ *u* $\in$ *L M2* $\Longrightarrow$ *converge M1 u v* $\wedge$ *converge M2 u v*)
using ‹($\bigwedge\alpha$ β. $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\Longrightarrow \alpha \in$ *L M1* $\Longrightarrow \alpha \in$ *L M2* $\Longrightarrow$ *converge M1* α $\beta \wedge$ *converge M2* α β)›
unfolding *simple-cg-lookup-iff* **by** *blast*
then show *?thesis*
using *simple-cg-merge-validity*[*OF assms*(*1*,*2*) - - ‹$x \in$ *list.set* (*simple-cg-merge*

$G\ \gamma\ \gamma'$)› ‹α |∈| x› ‹β |∈| x› ‹$\alpha \in L\ M1$› ‹$\alpha \in L\ M2$›]
‹*converge M1* $\gamma\ \gamma'$› ‹*converge M2* $\gamma\ \gamma'$›
by *blast*
qed
qed
then show *?thesis*
unfolding *convergence-graph-merge-invar-def convergence-graph-lookup-invar-def*
unfolding *simple-cg-lookup-iff*
by *metis*
qed

lemma *simple-cg-empty-invar* :
convergence-graph-lookup-invar M1 M2 simple-cg-lookup simple-cg-empty
unfolding *convergence-graph-lookup-invar-def simple-cg-empty-def*
by *auto*

lemma *simple-cg-initial-invar* :
assumes *observable M1*
shows *convergence-graph-initial-invar M1 M2 simple-cg-lookup simple-cg-initial*
proof −

have $\bigwedge$ T . ($L\ M1 \cap set\ T = (L\ M2 \cap set\ T)$) $\Longrightarrow$ *finite-tree* $T \Longrightarrow (\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (simple\text{-}cg\text{-}initial\ M1\ T)\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
proof −
fix T **assume** ($L\ M1 \cap set\ T = (L\ M2 \cap set\ T)$) **and** *finite-tree* T
then have $list.set\ (filter\ (is\text{-}in\text{-}language\ M1\ (initial\ M1))\ (sorted\text{-}list\text{-}of\text{-}sequences\text{-}in\text{-}tree\ T)) \subseteq L\ M1 \cap L\ M2$
unfolding *is-in-language-iff*[*OF assms fsm-initial*]
using *sorted-list-of-sequences-in-tree-set*[*OF* ‹*finite-tree T*›]
by *auto*
moreover have $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ simple\text{-}cg\text{-}empty\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
using *simple-cg-empty-invar*
unfolding *convergence-graph-lookup-invar-def*
by *blast*
ultimately show $(\bigwedge\alpha\ \beta.\ \beta \in list.set\ (simple\text{-}cg\text{-}lookup\ (simple\text{-}cg\text{-}initial\ M1\ T)\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow converge\ M1\ \alpha\ \beta \wedge converge\ M2\ \alpha\ \beta)$
using *simple-cg-insert′-foldl-helper*[*of* (*filter* (*is-in-language M1* (*initial M1*)) (*sorted-list-of-sequences-in-tree T*)) *M1 M2*]
unfolding *simple-cg-initial.simps*
by *blast*
qed
then show *?thesis*
unfolding *convergence-graph-initial-invar-def convergence-graph-lookup-invar-def*
using *simple-cg-lookup-iff* **by** *blast*
qed

lemma *simple-cg-insert-with-conv-invar* :
assumes *observable M1*
assumes *observable M2*
shows *convergence-graph-insert-invar M1 M2 simple-cg-lookup simple-cg-insert-with-conv*
proof −
have $\bigwedge$ *G* γ α β . $\gamma \in$ *L M1* $\Longrightarrow$
$\gamma \in$ *L M2* $\Longrightarrow$
($\bigwedge\alpha$. $\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup G* α)
$\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2*
α β)) $\Longrightarrow$
$\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert-with-conv*
G γ) α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert-with-conv G* γ)
α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β)
proof
fix *G ys* α
assume *ys* $\in$ *L M1*
and *ys* $\in$ *L M2*
and $*$:($\bigwedge\alpha$. $\alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ $\alpha \in$ *list.set* (*simple-cg-lookup G*
α) $\wedge$ ($\forall$ β . $\beta \in$ *list.set* (*simple-cg-lookup G* α) $\longrightarrow$ *converge M1* α β $\wedge$ *converge*
M2 α β))
and $\alpha \in$ *L M1*
and $\alpha \in$ *L M2*

show $\alpha \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert-with-conv G ys*) α)
using *simple-cg-lookup-iff* **by** *blast*

have $\bigwedge$ β . $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert-with-conv G ys*) α)
$\Longrightarrow$ *converge M1* α β $\wedge$ *converge M2* α β
proof −
fix β
assume $\beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert-with-conv G ys*) α)

define *insert-for-prefix* **where** *insert-for-prefix*:
insert-for-prefix = (λ *g i* . *let*
pref = *take i ys*;
suff = *drop i ys*;
pref-conv = *simple-cg-lookup g pref*
in foldl (λ *g′ ys′* . *simple-cg-insert′ g′* (*ys′@suff*)) *g*
pref-conv)
define *g′* **where** *g′*: *g′* = *simple-cg-insert G ys*
define *g″* **where** *g″*: *g″* = *foldl insert-for-prefix g′* [*0..<length ys*]

have *simple-cg-insert-with-conv G ys* = *simple-cg-closure g″*
unfolding *simple-cg-insert-with-conv.simps g″ g′ insert-for-prefix Let-def*
by *force*

have *g′-invar*: $(\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup* $g'\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta)$
using g' $*$
using *simple-cg-insert-invar* ‹*ys* $\in L\ M1$› ‹*ys* $\in L\ M2$›
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
by *blast*

have *insert-for-prefix-invar*: $\bigwedge\ i\ g\ .\ (\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup* $g\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta) \Longrightarrow (\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup* (*insert-for-prefix* $g\ i$) $\alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta)$
proof −
fix $i\ g$ **assume** $(\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup* $g\ \alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta)$

define *pref* **where** *pref*: *pref* = *take i ys*
define *suff* **where** *suff*: *suff* = *drop i ys*
let *?pref-conv* = *simple-cg-lookup g pref*

have *insert-for-prefix g i* = *foldl* (λ g' ys' . *simple-cg-insert′* g' (ys'@*suff*)) *g ?pref-conv*
unfolding *insert-for-prefix pref suff Let-def* **by** *force*

have *ys* = *pref* @ *suff*
unfolding *pref suff* **by** *auto*
then have *pref* $\in L\ M1$ **and** *pref* $\in L\ M2$
using ‹*ys* $\in L\ M1$› ‹*ys* $\in L\ M2$› *language-prefix* **by** *metis+*

have *insert-step-invar*: $\bigwedge$ *ys′ pc G* . *list.set pc* $\subseteq$ *list.set* (*simple-cg-lookup g pref*) $\Longrightarrow$ *ys′* $\in$ *list.set pc* $\Longrightarrow$
$(\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup G* $\alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta) \Longrightarrow$
$(\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup* (*simple-cg-insert′ G* (ys'@*suff*)) $\alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta)$
proof −
fix *ys′ pc G*
assume *list.set pc* $\subseteq$ *list.set* (*simple-cg-lookup g pref*)
and *ys′* $\in$ *list.set pc*
and $(\bigwedge\alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup G* $\alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta)$
then have *converge M1 pref ys′* **and** *converge M2 pref ys′*
using ‹$\bigwedge\beta\ \alpha.\ \beta \in$ *list.set* (*simple-cg-lookup g* $\alpha) \Longrightarrow \alpha \in L\ M1 \Longrightarrow \alpha \in L\ M2 \Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta$›
using ‹*pref* $\in L\ M1$› ‹*pref* $\in L\ M2$›
by *blast+*

have (ys'@*suff*) $\in L\ M1$
using ‹*converge M1 pref ys′*›

using ‹*ys* = *pref* @ *suff*› ‹*ys* ∈ *L M1*› *assms*(*1*) *converge-append-language-iff* **by** *blast*
moreover have (*ys'*@*suff*) ∈ *L M2*
using ‹*converge M2 pref ys'*›
using ‹*ys* = *pref* @ *suff*› ‹*ys* ∈ *L M2*› *assms*(*2*) *converge-append-language-iff* **by** *blast*
ultimately show (⋀α β. β ∈ *list.set* (*simple-cg-lookup* (*simple-cg-insert'* *G* (*ys'*@*suff*)) α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)
using ‹(⋀α β. β ∈ *list.set* (*simple-cg-lookup G* α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)›
using *simple-cg-insert'-invar*[*of M1 M2*]
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
using *simple-cg-lookup-iff* **by** *blast*
qed

have *insert-foldl-invar*: ⋀ *pc G* . *list.set pc* ⊆ *list.set* (*simple-cg-lookup g pref*) ⟹
(⋀α β. β ∈ *list.set* (*simple-cg-lookup G* α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β) ⟹
(⋀α β. β ∈ *list.set* (*simple-cg-lookup* (*foldl* (λ *g' ys'* . *simple-cg-insert'* *g'* (*ys'*@*suff*)) *G pc*) α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)
proof −
fix *pc G* **assume** *list.set pc* ⊆ *list.set* (*simple-cg-lookup g pref*)
and (⋀α β. β ∈ *list.set* (*simple-cg-lookup G* α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)

then show (⋀α β. β ∈ *list.set* (*simple-cg-lookup* (*foldl* (λ *g' ys'* . *simple-cg-insert'* *g'* (*ys'*@*suff*)) *G pc*) α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)
proof (*induction pc rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc a pc*)

have ∗∗:(*foldl* (λ*g' ys'*. *simple-cg-insert'* *g'* (*ys'* @ *suff*)) *G* (*pc* @ [*a*]))
= *simple-cg-insert'* (*foldl* (λ*g' ys'*. *simple-cg-insert'* *g'* (*ys'* @ *suff*)) *G pc*) (*a*@*suff*)
unfolding *foldl-append* **by** *auto*
have *list.set pc* ⊆ *list.set* (*simple-cg-lookup g pref*)
using *snoc.prems*(*4*) **by** *auto*
then have ∗: (⋀α β. β ∈ *list.set* (*simple-cg-lookup* (*foldl* (λ *g' ys'* . *simple-cg-insert'* *g'* (*ys'*@*suff*)) *G pc*) α) ⟹ α ∈ *L M1* ⟹ α ∈ *L M2* ⟹ *converge M1* α β ∧ *converge M2* α β)
using *snoc.IH*
using *snoc.prems*(*5*) **by** *blast*

```
          have a ∈ list.set (pc @ [a]) by auto
          then show ?case
            using snoc.prems(1,2,3)
            unfolding **
                using insert-step-invar[OF snoc.prems(4), of a (foldl (λ g' ys' .
simple-cg-insert' g' (ys'@suff)) G pc), OF - *]
            by blast
        qed
      qed

      show (⋀α β. β ∈ list.set (simple-cg-lookup (insert-for-prefix g i) α) ⟹ α
∈ L M1 ⟹ α ∈ L M2 ⟹ converge M1 α β ∧ converge M2 α β)
            using insert-foldl-invar[of ?pref-conv g, OF - ‹(⋀α β. β ∈ list.set
(simple-cg-lookup g α) ⟹ α ∈ L M1 ⟹ α ∈ L M2 ⟹ converge M1 α β ∧
converge M2 α β)›]
          unfolding ‹insert-for-prefix g i = foldl (λ g' ys' . simple-cg-insert' g'
(ys'@suff)) g ?pref-conv›
        by blast
    qed

    have insert-for-prefix-foldl-invar: ⋀ ns . (⋀α β. β ∈ list.set (simple-cg-lookup
(foldl insert-for-prefix g' ns) α) ⟹ α ∈ L M1 ⟹ α ∈ L M2 ⟹ converge M1 α
β ∧ converge M2 α β)
    proof −
      fix ns show (⋀α β. β ∈ list.set (simple-cg-lookup (foldl insert-for-prefix g'
ns) α) ⟹ α ∈ L M1 ⟹ α ∈ L M2 ⟹ converge M1 α β ∧ converge M2 α β)
      proof (induction ns rule: rev-induct)
        case Nil
        then show ?case using g'-invar by auto
      next
        case (snoc a ns)
        show ?case
          using snoc.prems
          using insert-for-prefix-invar [OF snoc.IH]
          by auto
      qed
    qed

    show ‹converge M1 α β ∧ converge M2 α β›
      using ‹β ∈ list.set (simple-cg-lookup (simple-cg-insert-with-conv G ys) α)›
      unfolding ‹simple-cg-insert-with-conv G ys = simple-cg-closure g''› g''
      using insert-for-prefix-foldl-invar[of - [0..<length ys] -]
      using simple-cg-closure-invar-helper[OF assms, of (foldl insert-for-prefix g'
[0..<length ys]), OF insert-for-prefix-foldl-invar[of - [0..<length ys] -]]
      using ‹α ∈ L M1› ‹α ∈ L M2› by blast
  qed
  then show (∀ β . β ∈ list.set (simple-cg-lookup (simple-cg-insert-with-conv G
ys) α) ⟶ converge M1 α β ∧ converge M2 α β)
    by blast
```

qed

then show *?thesis*
unfolding *convergence-graph-insert-invar-def convergence-graph-lookup-invar-def*
by *blast*
qed

lemma *simple-cg-lookup-with-conv-from-lookup-invar*:
assumes *observable M1* **and** *observable M2*
and *convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*
shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G*
proof −

have $(\bigwedge \alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup-with-conv G* α) $\Longrightarrow \alpha \in$ *L M1* $\Longrightarrow$ $\alpha \in$ *L M2* $\Longrightarrow$ *converge M1* $\alpha\ \beta \wedge$ *converge M2* $\alpha\ \beta$)
proof −
fix *ys* β **assume** $\beta \in$ *list.set* (*simple-cg-lookup-with-conv G ys*) **and** *ys* $\in$ *L M1* **and** *ys* $\in$ *L M2*

define *lookup-for-prefix* **where** *lookup-for-prefix*:
lookup-for-prefix = (λ*i* . *let*
pref = *take i ys*;
suff = *drop i ys*;
pref-conv = (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *pref* |∈| *x*) *G*))
in fimage (λ *pref′* . *pref′*@*suff*) *pref-conv*)

have $\bigwedge$ *ns* . $\beta \in$ *list.set* (*sorted-list-of-fset* (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty ns*))) $\Longrightarrow$ *converge M1 ys* $\beta \wedge$ *converge M2 ys* β
proof −
fix *ns* **assume** $\beta \in$ *list.set* (*sorted-list-of-fset* (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty ns*)))
then show *converge M1 ys* $\beta \wedge$ *converge M2 ys* β
proof (*induction ns rule*: *rev-induct*)
case *Nil*
then show *?case* **using** ‹*ys* $\in$ *L M1*› ‹*ys* $\in$ *L M2*› **by** *auto*
next
case (*snoc a ns*)

have *list.set* (*sorted-list-of-fset* (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty* (*ns*@[*a*]))))) =
(*fset* (*lookup-for-prefix a*) ∪ *list.set* (*sorted-list-of-fset* (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty ns*))))
by *auto*
then consider $\beta \in$ *fset* (*lookup-for-prefix a*) | $\beta \in$ *list.set* (*sorted-list-of-fset* (*finsert ys* (*foldl* (λ *cs i* . *lookup-for-prefix i* |∪| *cs*) *fempty ns*)))

using *snoc.prems* **by** *auto*
then show *?case* **proof** *cases*
case *1*
define *pref* **where** *pref*: *pref* = *take a ys*
define *suff* **where** *suff*: *suff* = *drop a ys*
define *pref-conv* **where** *pref-conv*: *pref-conv* = (*foldl* (|∪|) *fempty* (*filter* (λ*x* . *pref* |∈| *x*) *G*))

have *lookup-for-prefix a* = *fimage* (λ *pref′* . *pref′*@*suff*) *pref-conv*
unfolding *lookup-for-prefix pref suff pref-conv*
by *metis*
then have β ∈ *list.set* (*map* (λ *pref′* . *pref′*@*suff*) (*sorted-list-of-fset* (*finsert pref* (*foldl* (|∪|) {||} (*filter* ((|∈|) *pref*) *G*)))))
using *1* **unfolding** *pref-conv* **by** *auto*
then obtain γ **where** γ ∈ *list.set* (*simple-cg-lookup G pref*)
and β = γ@*suff*
unfolding *simple-cg-lookup.simps*
by (*meson set-map-elem*)
then have *converge M1* γ *pref* **and** *converge M2* γ *pref*
using ‹*convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*›
unfolding *convergence-graph-lookup-invar-def*
by (*metis* ‹*ys* ∈ *L M1*› ‹*ys* ∈ *L M2*› *append-take-drop-id converge-sym language-prefix pref*)+
then show *?thesis*
by (*metis* ‹⋀*thesis*. (⋀γ. ⟦γ ∈ *list.set* (*simple-cg-lookup G pref*); β = γ @ *suff*⟧ ⟹ *thesis*) ⟹ *thesis*› ‹*ys* ∈ *L M1*› ‹*ys* ∈ *L M2*› *append-take-drop-id assms*(*1*) *assms*(*2*) *assms*(*3*) *converge-append converge-append-language-iff convergence-graph-lookup-invar-def language-prefix pref suff*)
next
case *2*
then show *?thesis* **using** *snoc.IH* **by** *blast*
qed
qed
qed

then show *converge M1 ys* β ∧ *converge M2 ys* β
using ‹β ∈ *list.set* (*simple-cg-lookup-with-conv G ys*)›
unfolding *simple-cg-lookup-with-conv.simps Let-def lookup-for-prefix sorted-list-of-set-set*
by *blast*
qed
moreover have ⋀ α . α ∈ *list.set* (*simple-cg-lookup-with-conv G* α)
unfolding *simple-cg-lookup-with-conv.simps* **by** *auto*
ultimately show *?thesis*
unfolding *convergence-graph-lookup-invar-def*
by *blast*
qed

lemma *simple-cg-lookup-from-lookup-invar-with-conv*:
assumes *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G*

shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*
proof −

have $\bigwedge \alpha\ \beta.\ \beta \in$ *list.set* (*simple-cg-lookup G* α) $\Longrightarrow \beta \in$ *list.set* (*simple-cg-lookup-with-conv G* α)
proof −

fix $\alpha\ \beta$ **assume** $\beta \in$ *list.set* (*simple-cg-lookup G* α)

define *lookup-for-prefix* **where** *lookup-for-prefix*:
lookup-for-prefix = (λi . *let*
pref = *take i* α;
suff = *drop i* α;
pref-conv = *simple-cg-lookup G pref*
in map (λ *pref′* . *pref′*@*suff*) *pref-conv*)

have *lookup-for-prefix* (*length* α) = *simple-cg-lookup G* α
unfolding *lookup-for-prefix* **by** *auto*
moreover have *list.set* (*lookup-for-prefix* (*length* α)) $\subseteq$ *list.set* (*simple-cg-lookup-with-conv G* α)
unfolding *simple-cg-lookup-with-conv.simps lookup-for-prefix Let-def sorted-list-of-set-set* **by** *auto*
ultimately show $\beta \in$ *list.set* (*simple-cg-lookup-with-conv G* α)
using ‹$\beta \in$ *list.set* (*simple-cg-lookup G* α)›
by (*metis subsetD*)
qed

then show *?thesis*
using *assms*
unfolding *convergence-graph-lookup-invar-def*
using *simple-cg-lookup-iff* **by** *blast*
qed

lemma *simple-cg-lookup-invar-with-conv-eq* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-lookup-invar M1 M2 simple-cg-lookup-with-conv G* = *convergence-graph-lookup-invar M1 M2 simple-cg-lookup G*
using *simple-cg-lookup-with-conv-from-lookup-invar*[*OF assms*] *simple-cg-lookup-from-lookup-invar-with-conv M1 M2*]
by *blast*

lemma *simple-cg-insert-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-insert-invar M1 M2 simple-cg-lookup-with-conv simple-cg-insert*
using *simple-cg-insert-invar*[*of M1 M2*]
unfolding *convergence-graph-insert-invar-def*
unfolding *simple-cg-lookup-invar-with-conv-eq*[*OF assms*]

.

lemma *simple-cg-merge-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-merge-invar M1 M2 simple-cg-lookup-with-conv simple-cg-merge*
using *simple-cg-merge-invar*[*OF assms*]
unfolding *convergence-graph-merge-invar-def*
unfolding *simple-cg-lookup-invar-with-conv-eq*[*OF assms*]
.

lemma *simple-cg-initial-invar-with-conv* :
assumes *observable M1* **and** *observable M2*
shows *convergence-graph-initial-invar M1 M2 simple-cg-lookup-with-conv simple-cg-initial*
using *simple-cg-initial-invar*[*OF assms*(*1*), *of M2*]
unfolding *convergence-graph-initial-invar-def*
unfolding *simple-cg-lookup-invar-with-conv-eq*[*OF assms*]
.

end

24 Intermediate Frameworks

This theory provides partial applications of the H, SPY, and Pair-Frameworks.

theory *Intermediate-Frameworks*
imports *Intermediate-Implementations Test-Suite-Representations ../OFSM-Tables-Refined Simple-Convergence-Graph Empty-Convergence-Graph*
begin

24.1 Partial Applications of the SPY-Framework

definition *spy-framework-static-with-simple-graph* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒
nat ⇒
('*b*×'*c*) *prefix-tree*
where
spy-framework-static-with-simple-graph M1
dist-fun
m
= *spy-framework M1*
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*establish-convergence-static dist-fun*)
(*handle-io-pair False True*)
simple-cg-initial

simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *spy-framework-static-with-simple-graph-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M1* $\Longrightarrow$ *q2* $\in$ *states M1* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ $\exists$ *io* . $\forall$ *k1 k2* . *io* $\in$ *set* (*dist-fun k1 q1*) $\cap$ *set* (*dist-fun k2 q2*) $\wedge$ *distinguishes M1 q1 q2 io*
and $\bigwedge$ *q k* . *q* $\in$ *states M1* $\Longrightarrow$ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*spy-framework-static-with-simple-graph M1 dist-fun m*)) = (*L M2* $\cap$ *set* (*spy-framework-static-with-simple-graph M1 dist-fun m*)))
and *finite-tree* (*spy-framework-static-with-simple-graph M1 dist-fun m*)
using *spy-framework-completeness-and-finiteness*[*OF assms*(*1*−*8*),
of get-state-cover-assignment, *OF get-state-cover-assignment-is-state-cover-assignment*,
of (λ *M V ts* . *ts*),
OF - simple-cg-initial-invar-with-conv[*OF assms*(*1*,*2*)],
OF - simple-cg-insert-invar-with-conv[*OF assms*(*1*,*2*)],
OF - simple-cg-merge-invar-with-conv[*OF assms*(*1*,*2*)],
of handle-state-cover-static dist-fun
establish-convergence-static dist-fun
handle-io-pair False True
]
using *handle-state-cover-static-separates-state-cover*[*OF assms*(*9*,*10*)]
using *establish-convergence-static-verifies-transition*[*of M1 dist-fun M2 get-state-cover-assignment M1 simple-cg-initial simple-cg-insert simple-cg-lookup-with-conv*, *OF assms*(*9*,*10*)]
using *handle-io-pair-verifies-io-pair*[*of False True M1 M2 simple-cg-insert simple-cg-lookup-with-conv*]
unfolding *spy-framework-static-with-simple-graph-def*
by *blast*+

definition *spy-framework-static-with-empty-graph* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒
(*nat* ⇒ 'a ⇒ ('b×'c) *prefix-tree*) ⇒
nat ⇒
('b×'c) *prefix-tree*
where
spy-framework-static-with-empty-graph M1
dist-fun
m
= *spy-framework M1*
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*establish-convergence-static dist-fun*)
(*handle-io-pair False True*)
empty-cg-initial
empty-cg-insert
empty-cg-lookup
empty-cg-merge
m

lemma *spy-framework-static-with-empty-graph-completeness-and-finiteness* :
fixes *M1* :: ('a::linorder,'b::linorder,'c::linorder) *fsm*
fixes *M2* :: ('d,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-framework-static-with-empty-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*spy-framework-static-with-empty-graph M1 dist-fun m*)))
and *finite-tree* (*spy-framework-static-with-empty-graph M1 dist-fun m*)
using *spy-framework-completeness-and-finiteness*[*OF assms*(*1*−*8*),
of get-state-cover-assignment, *OF get-state-cover-assignment-is-state-cover-assignment*,
of (λ *M V ts* . *ts*),
OF - *empty-graph-initial-invar*,
OF - *empty-graph-insert-invar*,
OF - *empty-graph-merge-invar*,

of handle-state-cover-static dist-fun
establish-convergence-static dist-fun
handle-io-pair False True
]
using *handle-state-cover-static-separates-state-cover*[*OF assms*(*9*,*10*)]
using *establish-convergence-static-verifies-transition*[*of M1 dist-fun M2 get-state-cover-assignment M1 empty-cg-initial empty-cg-insert empty-cg-lookup*, *OF assms*(*9*,*10*)]
using *handle-io-pair-verifies-io-pair*[*of False True M1 M2 empty-cg-insert empty-cg-lookup*]
unfolding *spy-framework-static-with-empty-graph-def*
by *blast*+

24.2 Partial Applications of the H-Framework

definition *h-framework-static-with-simple-graph* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒
nat ⇒
('*b*×'*c*) *prefix-tree*
where
h-framework-static-with-simple-graph M1 dist-fun m =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*handleUT-static dist-fun*)
(*handle-io-pair False False*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *h-framework-static-with-simple-graph-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework-static-with-simple-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*h-framework-static-with-simple-graph M1 dist-fun*

m)))
and *finite-tree* (*h-framework-static-with-simple-graph M1 dist-fun m*)
using *h-framework-completeness-and-finiteness*[*OF assms*(*1*−*8*),
of get-state-cover-assignment
(λ *M V ts* . *ts*) ,
OF get-state-cover-assignment-is-state-cover-assignment
-
simple-cg-initial-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-insert-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-merge-invar-with-conv[*OF assms*(*1*,*2*)]
handle-state-cover-static-separates-state-cover[*OF assms*(*9*,*10*)]
handleUT-static-handles-transition[*OF assms*(*9*,*10*)]
verifies-io-pair-handled[*OF handle-io-pair-verifies-io-pair*[*of False False M1 M2 simple-cg-insert simple-cg-lookup-with-conv*]]
]
unfolding *h-framework-static-with-simple-graph-def*[*symmetric*]
by *presburger*+

definition *h-framework-static-with-simple-graph-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ (*nat* ⇒ *'a* ⇒ (*'b*×*'c*) *prefix-tree*) ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
h-framework-static-with-simple-graph-lists M dist-fun m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-static-with-simple-graph M dist-fun m*))

lemma *h-framework-static-with-simple-graph-lists-completeness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-static-with-simple-graph-li*
M1 dist-fun m)
unfolding *h-framework-static-with-simple-graph-lists-def*
using *h-framework-static-with-simple-graph-completeness-and-finiteness*(*1*)[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*)
using *passes-test-cases-from-io-tree*[*OF assms*(*1*,*2*) *fsm-initial fsm-initial h-framework-static-with-simple-grap*

assms]]
by *blast*

definition *h-framework-static-with-empty-graph* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
(*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒
nat ⇒
('*b*×'*c*) *prefix-tree*
where
h-framework-static-with-empty-graph M1 dist-fun m =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-static dist-fun*)
(λ *M V ts* . *ts*)
(*handleUT-static dist-fun*)
(*handle-io-pair False False*)
empty-cg-initial
empty-cg-insert
empty-cg-lookup
empty-cg-merge
m

lemma *h-framework-static-with-empty-graph-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-framework-static-with-empty-graph M1 dist-fun m*)) = (*L M2* ∩ *set* (*h-framework-static-with-empty-graph M1 dist-fun m*)))
and *finite-tree* (*h-framework-static-with-empty-graph M1 dist-fun m*)
using *h-framework-completeness-and-finiteness*[*OF assms*(1−8),
of get-state-cover-assignment
(λ *M V ts* . *ts*) ,
OF get-state-cover-assignment-is-state-cover-assignment
-
empty-graph-initial-invar
empty-graph-insert-invar

empty-graph-merge-invar
handle-state-cover-static-separates-state-cover[*OF assms*(*9*,*10*)]
handleUT-static-handles-transition[*OF assms*(*9*,*10*)]
verifies-io-pair-handled[*OF handle-io-pair-verifies-io-pair*[*of False False M1 M2 empty-cg-insert empty-cg-lookup*]]
]
unfolding *h-framework-static-with-empty-graph-def*[*symmetric*]
by *presburger+*

definition *h-framework-static-with-empty-graph-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ (*nat* ⇒ '*a* ⇒ ('*b*×'*c*) *prefix-tree*) ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-framework-static-with-empty-graph-lists M dist-fun m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-static-with-empty-graph M dist-fun m*))

lemma *h-framework-static-with-empty-graph-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ *q1 q2* . *q1* ∈ *states M1* ⟹ *q2* ∈ *states M1* ⟹ *q1* ≠ *q2* ⟹ ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*dist-fun k1 q1*) ∩ *set* (*dist-fun k2 q2*) ∧ *distinguishes M1 q1 q2 io*
and ⋀ *q k* . *q* ∈ *states M1* ⟹ *finite-tree* (*dist-fun k q*)
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-static-with-empty-graph-lis M1 dist-fun m*)
unfolding *h-framework-static-with-empty-graph-lists-def*
using *h-framework-static-with-empty-graph-completeness-and-finiteness*(*1*)[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*)
using *passes-test-cases-from-io-tree*[*OF assms*(*1*,*2*) *fsm-initial fsm-initial h-framework-static-with-empty-grap assms*]]
by *blast*

definition *h-framework-dynamic* ::
(('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒
('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒
nat ⇒
bool ⇒
bool ⇒
('*b*×'*c*) *prefix-tree*

where
h-framework-dynamic convergence-decisision M1 m completeInputTraces useInputHeuristic =
h-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
sort-unverified-transitions-by-state-cover-length
(*handleUT-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*) *convergence-decisision*)
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *h-framework-dynamic-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)) = (*L M2* $\cap$ *set* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*h-framework-dynamic convergenceDecision M1 m completeInputTraces useInputHeuristic*)
using *h-framework-completeness-and-finiteness*[*OF assms*,
of get-state-cover-assignment
sort-unverified-transitions-by-state-cover-length
,
OF get-state-cover-assignment-is-state-cover-assignment
sort-unverified-transitions-by-state-cover-length-retains-set[*of - M1 get-state-cover-assignment*]
simple-cg-initial-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-insert-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-merge-invar-with-conv[*OF assms*(*1*,*2*)]
handle-state-cover-dynamic-separates-state-cover[*OF get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)], *of com-*

pleteInputTraces useInputHeuristic M2 simple-cg-initial simple-cg-insert simple-cg-lookup-with-conv]
handleUT-dynamic-handles-transition[*of M1* (*get-distinguishing-sequence-from-ofsm-tables M1*) *completeInputTraces useInputHeuristic convergenceDecision M2* - - *simple-cg-insert simple-cg-lookup-with-conv simple-cg-merge*, *OF get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]]
verifies-io-pair-handled[*OF handle-io-pair-verifies-io-pair*[*of completeInputTraces useInputHeuristic M1 M2 simple-cg-insert simple-cg-lookup-with-conv*]]
]
unfolding *h-framework-dynamic-def*[*symmetric*]
by *presburger*+

definition *h-framework-dynamic-lists* :: (('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *state-cover-assignment* ⇒ ('*a*,'*b*,'*c*) *transition* ⇒ ('*a*,'*b*,'*c*) *transition list* ⇒ *nat* ⇒ *bool*) ⇒ ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-framework-dynamic-lists convergenceDecision M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-framework-dynamic convergenceDecision M m completeInputTraces useInputHeuristic*))

lemma *h-framework-dynamic-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-framework-dynamic-lists convergenceDecision M1 m completeInputTraces useInputHeuristic*)
unfolding *h-framework-dynamic-lists-def*
h-framework-dynamic-completeness-and-finiteness(*1*)[*OF assms*, *of convergenceDecision completeInputTraces useInputHeuristic*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial h-framework-dynamic-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

24.3 Partial Applications of the Pair-Framework

definition *pair-framework-h-components* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ (('*b* × '*c*) *list* × '*a*) × ('*b* × '*c*) *list* × '*a* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ ('*b* × '*c*) *prefix-tree*) ⇒
('*b*×'*c*) *prefix-tree*

where
pair-framework-h-components M m get-separating-traces = (*let*
V = *get-state-cover-assignment M*
in pair-framework M m (*get-initial-test-suite-H V*) (*get-pairs-H V*) *get-separating-traces*)

lemma *pair-framework-h-components-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ α β *t* . α ∈ *L M1* ⟹ β ∈ *L M1* ⟹ *after-initial M1* α ≠ *after-initial M1* β ⟹ ∃ *io* ∈ *set* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) *io*
and ⋀ α β *t* . α ∈ *L M1* ⟹ β ∈ *L M1* ⟹ *after-initial M1* α ≠ *after-initial M1* β ⟹ *finite-tree* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*pair-framework-h-components M1 m get-separating-traces*)) = (*L M2* ∩ *set* (*pair-framework-h-components M1 m get-separating-traces*)))
and *finite-tree* (*pair-framework-h-components M1 m get-separating-traces*)
proof −
show (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*pair-framework-h-components M1 m get-separating-traces*)) = (*L M2* ∩ *set* (*pair-framework-h-components M1 m get-separating-traces*)))
using *pair-framework-completeness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*) *get-state-cover-assignment-is-state-cover-ass*
, *of get-initial-test-suite-H* (*get-state-cover-assignment M1*) *get-pairs-H* (*get-state-cover-assignment M1*) *get-separating-traces*
, *OF get-initial-test-suite-H-set-and-finite*(*1*)[*of get-state-cover-assignment M1 M1 m*]
, *OF get-pairs-H-set*(*1*)[*OF assms*(*1*) *get-state-cover-assignment-is-state-cover-assignment*, **where** *m*=*m*] *assms*(*8*)
]
unfolding *pair-framework-h-components-def Let-def*
using *get-pairs-H-set*(*1*)[*OF assms*(*1*) *get-state-cover-assignment-is-state-cover-assignment*, **where** *m*=*m*]
using *assms*(*8*)
unfolding *pair-framework-h-components-def Let-def*
by *presburger*

show *finite-tree* (*pair-framework-h-components M1 m get-separating-traces*)

using *pair-framework-finiteness*[*of M1 get-separating-traces get-initial-test-suite-H (get-state-cover-assignment M1) m get-pairs-H (get-state-cover-assignment M1),*
OF assms(*9*) *get-initial-test-suite-H-set-and-finite*(*2*)[*of get-state-cover-assignment M1 M1 m*] *get-pairs-H-set*(*2*)[*OF assms*(*1*) *get-state-cover-assignment-is-state-cover*
]
unfolding *pair-framework-h-components-def Let-def*
by *auto*
qed

definition *pair-framework-h-components-2* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒
(('*a*,'*b*,'*c*) *fsm* ⇒ (('*b* × '*c*) *list* × '*a*) × ('*b* × '*c*) *list* × '*a* ⇒ ('*b* × '*c*) *prefix-tree* ⇒ ('*b* × '*c*) *prefix-tree*) ⇒
bool ⇒
('*b*×'*c*) *prefix-tree*
where
pair-framework-h-components-2 M m get-separating-traces c = (*let*
V = *get-state-cover-assignment M*
in pair-framework M m (*get-initial-test-suite-H-2 c V*) (*get-pairs-H V*) *get-separating-traces*)

lemma *pair-framework-h-components-2-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
and ⋀ α β *t* . α ∈ *L M1* ⟹ β ∈ *L M1* ⟹ *after-initial M1* α ≠ *after-initial M1* β ⟹ ∃ *io* ∈ *set* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M1* (*after-initial M1* α) (*after-initial M1* β) *io*
and ⋀ α β *t* . α ∈ *L M1* ⟹ β ∈ *L M1* ⟹ *after-initial M1* α ≠ *after-initial M1* β ⟹ *finite-tree* (*get-separating-traces M1* ((α,*after-initial M1* α),(β,*after-initial M1* β)) *t*)
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)) = (*L M2* ∩ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)))
and *finite-tree* (*pair-framework-h-components-2 M1 m get-separating-traces c*)
proof −
show (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)) = (*L M2* ∩ *set* (*pair-framework-h-components-2 M1 m get-separating-traces c*)))
using *pair-framework-completeness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*) *get-state-cover-assignment-is-state-cover-ass*
, *of get-initial-test-suite-H-2 c* (*get-state-cover-assignment*

M1) *get-pairs-H* (*get-state-cover-assignment M1*) *get-separating-traces*
, *OF get-initial-test-suite-H-2-set-and-finite*(*1*)[*of*
get-state-cover-assignment M1 M1 m]
, *OF get-pairs-H-set*(*1*)[*OF assms*(*1*)
get-state-cover-assignment-is-state-cover-assignment, **where** *m*=*m*] *assms*(*8*)
]
unfolding *pair-framework-h-components-2-def Let-def*
using *get-pairs-H-set*(*1*)[*OF assms*(*1*) *get-state-cover-assignment-is-state-cover-assignment*,
where *m*=*m*]
using *assms*(*8*)
unfolding *pair-framework-h-components-def Let-def*
by *presburger*

show *finite-tree* (*pair-framework-h-components-2 M1 m get-separating-traces c*)
using *pair-framework-finiteness*[*of M1 get-separating-traces get-initial-test-suite-H-2*
c (*get-state-cover-assignment M1*) *m get-pairs-H* (*get-state-cover-assignment M1*),
OF assms(*9*) *get-initial-test-suite-H-2-set-and-finite*(*2*)[*of c*
get-state-cover-assignment M1 M1 m] *get-pairs-H-set*(*2*)[*OF assms*(*1*) *get-state-cover-assignment-is-state-cover*
]
unfolding *pair-framework-h-components-2-def Let-def*
by *auto*
qed

24.4 Code Generation

lemma *h-framework-dynamic-code*[*code*] :
h-framework-dynamic convergence-decisision M1 m completeInputTraces useInputHeuristic = (*let*
tables = (*compute-ofsm-tables M1* (*size M1* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M1 q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M1*)
(*states-as-list M1*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M1* ∧ *q2* ∈ *states M1* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M1 q1 q2)
in
h-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic*
distHelper)
sort-unverified-transitions-by-state-cover-length
(*handleUT-dynamic completeInputTraces useInputHeuristic distHelper*
convergence-decisision)
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv

simple-cg-merge
m)
unfolding *h-framework-dynamic-def*
apply (*subst* (*1 2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M1*])
unfolding *Let-def*
by *presburger*

end

25 Implementations of the H-Method

theory *H-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

25.1 Using the H-Framework

definition *h-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ('*b*×'*c*) *prefix-tree* **where**
h-method-via-h-framework = *h-framework-dynamic* (λ *M V t X l* . *False*)

definition *h-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-method-via-h-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-method-via-h-framework M m completeInputTraces useInputHeuristic*))

lemma *h-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*h-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-completeness-and-finiteness*[*OF assms*]
unfolding *h-method-via-h-framework-def*
by *blast+*

lemma *h-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm*
fixes *M2* :: ('d,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*h-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-lists-completeness*[*OF assms*]
unfolding *h-method-via-h-framework-lists-def h-framework-dynamic-lists-def h-method-via-h-framework-def*
by *blast*

25.2 Using the Pair-Framework

25.2.1 Selection of Distinguishing Traces

fun *add-distinguishing-sequence-if-required* :: ('a $\Rightarrow$ 'a $\Rightarrow$ ('b $\times$ 'c) *list*) $\Rightarrow$ ('a,'b::*linorder*,'c::*linorder*) *fsm* $\Rightarrow$ (('b$\times$'c) *list* $\times$ 'a) $\times$ (('b$\times$'c) *list* $\times$ 'a) $\Rightarrow$ ('b$\times$'c) *prefix-tree* $\Rightarrow$ ('b$\times$'c) *prefix-tree* **where**
add-distinguishing-sequence-if-required dist-fun M (α,*q1*), (β,*q2*)) *t* = (*if intersection-is-distinguishing M* (*after t* α) *q1* (*after t* β) *q2*
then empty
else insert empty (*dist-fun q1 q2*))

lemma *add-distinguishing-sequence-if-required-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in$ *L M*
and $\beta \in$ *L M*
and *after-initial M* $\alpha \neq$ *after-initial M* β
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M* $\Longrightarrow$ *q2* $\in$ *states M* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows $\exists$ *io* $\in$ *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
proof (*cases intersection-is-distinguishing M* (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β))
case *True*
then have (*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t* = *empty*
by *auto*
then have *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) = (*set* (*after t* α) $\cap$ *set* (*after t* β))
using *Prefix-Tree.set-empty*

by (*metis Int-insert-right inf.absorb-iff2 inf-bot-right insert-is-Un set-Nil sup-absorb2*)

moreover have $\exists$ *io* $\in$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
using *True* **unfolding** *intersection-is-distinguishing-correctness*[*OF assms*(*1*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)]]
by *auto*
ultimately show *?thesis*
by *blast*
next
case *False*
then have *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = *set* (*insert empty* (*dist-fun* (*after-initial M* α) (*after-initial M* β)))
by *auto*
then have *dist-fun* (*after-initial M* α) (*after-initial M* β) $\in$ *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β))
unfolding *insert-set* **by** *auto*
then show *?thesis*
using *assms*(*6*)[*OF after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*)] **by** *blast*
qed

lemma *add-distinguishing-sequence-if-required-finite* :
finite-tree ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
proof (*cases intersection-is-distinguishing M* (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β))
case *True*
then have ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = *empty*
by *auto*
then show *?thesis*
using *empty-finite-tree* **by** *simp*
next
case *False*
then have ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = (*insert empty* (*dist-fun* (*after-initial M* α) (*after-initial M* β)))
by *auto*
then show *?thesis*
using *insert-finite-tree*[*OF empty-finite-tree*] **by** *metis*
qed

fun *add-distinguishing-sequence-and-complete-if-required* :: ('*a* $\Rightarrow$ '*a* $\Rightarrow$ ('*b* $\times$ '*c*) *list*) $\Rightarrow$ *bool* $\Rightarrow$ ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ (('*b*$\times$'*c*) *list* $\times$ '*a*) $\times$ (('*b*$\times$'*c*) *list* $\times$ '*a*) $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
add-distinguishing-sequence-and-complete-if-required distFun completeInputTraces

M ((α,*q1*), (β,*q2*)) *t* =
(*if intersection-is-distinguishing M* (*after t* α) *q1* (*after t* β) *q2*
then empty
else let w = *distFun q1 q2*;
T = *insert empty w*
in if completeInputTraces
then let T1 = *from-list* (*language-for-input M q1* (*map fst w*));
T2 = *from-list* (*language-for-input M q2* (*map fst w*))
in Prefix-Tree.combine T (*Prefix-Tree.combine T1 T2*)
else T)

lemma *add-distinguishing-sequence-and-complete-if-required-distinguishes* :
assumes *observable M*
and *minimal M*
and α ∈ *L M*
and β ∈ *L M*
and *after-initial M* α ≠ *after-initial M* β
and ⋀ *q1 q2* . *q1* ∈ *states M* ⟹ *q2* ∈ *states M* ⟹ *q1* ≠ *q2* ⟹ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows ∃ *io* ∈ *set* ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
proof (*cases intersection-is-distinguishing M* (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β))
case *True*
then have (*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t* = *empty*
by *auto*
then have *set* ((*add-distinguishing-sequence-if-required dist-fun M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) = (*set* (*after t* α) ∩ *set* (*after t* β))
using *Prefix-Tree.set-empty*
by (*metis Int-insert-right inf.absorb-iff2 inf-bot-right insert-is-Un set-Nil sup-absorb2*)

moreover have ∃ *io* ∈ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
using *True* **unfolding** *intersection-is-distinguishing-correctness*[*OF assms*(*1*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)]]
by *auto*
ultimately show *?thesis*
by *blast*
next
case *False*
then have *set* (*insert empty* (*dist-fun* (*after-initial M* α) (*after-initial M* β))) ⊆ *set* ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
using *combine-set*[*of insert empty* (*dist-fun* (*after-initial M* α) (*after-initial M* β))]
unfolding *add-distinguishing-sequence-and-complete-if-required.simps Let-def*

by (*cases c*; *fastforce*)
moreover have *dist-fun* (*after-initial M* α) (*after-initial M* β) $\in$ *set* (*insert empty* (*dist-fun* (*after-initial M* α) (*after-initial M* β)))
unfolding *insert-set* **by** *auto*
ultimately have *dist-fun* (*after-initial M* α) (*after-initial M* β) $\in$ *set* ((*add-distinguishing-sequence-and-comp*
dist-fun c M) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β))
by *blast*
then show *?thesis*
using *assms*(*6*)[*OF after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*)]
by (*meson distinguishes-def*)
qed

lemma *add-distinguishing-sequence-and-complete-if-required-finite* :
finite-tree ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
proof (*cases intersection-is-distinguishing M* (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β))
case *True*
then have ((*add-distinguishing-sequence-and-complete-if-required dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = *empty*
by *auto*
then show *?thesis*
using *empty-finite-tree* **by** *simp*
next
case *False*

define *w* **where** *w*: *w* = *dist-fun* (*after-initial M* α) (*after-initial M* β)
define *T* **where** *T*: *T* = *insert empty w*
define *T1* **where** *T1*: *T1* = *from-list* (*language-for-input M* (*after-initial M* α) (*map fst w*))
define *T2* **where** *T2*: *T2* = *from-list* (*language-for-input M* (*after-initial M* β) (*map fst w*))

have *finite-tree T*
using *insert-finite-tree*[*OF empty-finite-tree*]
unfolding *T* **by** *auto*
moreover have *finite-tree* (*Prefix-Tree.combine T* (*Prefix-Tree.combine T1 T2*))
using *combine-finite-tree*[*OF* ‹*finite-tree T*› *combine-finite-tree*[*OF from-list-finite-tree from-list-finite-tree*]]
unfolding *T1 T2*
by *auto*
ultimately show *?thesis*
using *False*
unfolding *add-distinguishing-sequence-and-complete-if-required.simps w T T1 T2 Let-def*
by *presburger*
qed

function *find-cheapest-distinguishing-trace* :: ('a,'b::linorder,'c::linorder) *fsm* ⇒ ('a ⇒ 'a ⇒ ('b × 'c) *list*) ⇒ ('b×'c) *list* ⇒ ('b×'c) *prefix-tree* ⇒ 'a ⇒ ('b×'c) *prefix-tree* ⇒ 'a ⇒ (('b×'c) *list* × *nat* × *nat*) **where**

```
  find-cheapest-distinguishing-trace M distFun ios (PT m1) q1 (PT m2) q2 =
    (let
      f = (λ (ω,l,w) (x,y) . if (x,y) ∉ list.set ios then (ω,l,w) else
          (let
            w1L = if (PT m1) = empty then 0 else 1;
            w1C = if (x,y) ∈ dom m1 then 0 else 1;
            w1 = min w1L w1C;
            w2L = if (PT m2) = empty then 0 else 1;
            w2C = if (x,y) ∈ dom m2 then 0 else 1;
            w2 = min w2L w2C;
            w' = w1 + w2
          in
            case h-obs M q1 x y of
              None ⇒ (case h-obs M q2 x y of
                None ⇒ (ω,l,w) |
                  Some - ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else
(ω,l,w)) |
              Some q1' ⇒ (case h-obs M q2 x y of
              None ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else (ω,l,w)
|
                Some q2' ⇒ (if q1' = q2'
                  then (ω,l,w)
                  else (case m1 (x,y) of
                    None ⇒ (case m2 (x,y) of
                      None ⇒ let ω' = distFun q1' q2';
                                l' = 2 + 2 ∗ length ω'
                            in if (w' < w) ∨ (w' = w ∧ l' < l) then ((x,y)#ω',l',w')
else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2'
                                in if (w'' + w1 < w) ∨ (w'' + w1 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w1) else (ω,l,w)) |
                    Some t1' ⇒ (case m2 (x,y) of
                      None ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace M
distFun ios t1' q1' empty q2'
                                in if (w'' + w2 < w) ∨ (w'' + w2 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w2) else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios t1' q1' t2' q2'
                                in if (w'' < w) ∨ (w'' = w ∧ l'' < l) then
((x,y)#ω'',l'',w'') else (ω,l,w)))))))
    in
```

foldl f (*distFun q1 q2*, *0*, *3*) *ios*)
by *pat-completeness auto*
termination
proof −

let *?f* = (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)

have ⋀(*M*::(′*a*,′*b*::*linorder*,′*c*::*linorder*) *fsm*)
(*distFun* :: (′*a* ⇒ ′*a* ⇒ (′*b* × ′*c*) *list*))
(*ios* :: (′*b*×′*c*) *list*)
m1 (*q1*::′*a*) *m2* (*q2*::′*a*) *x y t2*′ *q1*′ *q2*′.
¬ (*x*, *y*) ∉ *list.set ios* ⟹
m1 (*x*, *y*) = *None* ⟹
m2 (*x*, *y*) = *Some t2*′ ⟹
((*M*, *distFun*, *ios*, *Prefix-Tree.empty*, *q1*′, *t2*′, *q2*′), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
proof −
fix *M*::(′*a*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fix *distFun* :: (′*a* ⇒ ′*a* ⇒ (′*b* × ′*c*) *list*)
fix *ios* :: (′*b*×′*c*) *list*
fix *m1 m2* :: (′*b*×′*c*) ⇀ (′*b*×′*c*) *prefix-tree*
fix *t2*′
fix *q1 q2 q1*′ *q2*′ :: ′*a*
fix *x*
fix *y*

assume *m1* (*x*, *y*) = *None*
assume *m2* (*x*, *y*) = *Some t2*′
assume ¬ (*x*, *y*) ∉ *list.set ios*

define *pre* **where** *pre* = (*M*, *distFun*, *ios*, *PT m1*, *q1*, *PT m2*, *q2*)
define *post* **where** *post* = (*M*, *distFun*, *ios*, *Prefix-Tree.empty*::(′*b*×′*c*) *prefix-tree*, *q1*′, *t2*′, *q2*′)

have *height-over ios empty* ≤ *height-over ios* (*PT m1*)
unfolding *height-over.simps height-over-empty* **by** *auto*
then have *?f post* < *?f pre*
unfolding *pre-def post-def case-prod-conv*
by (*meson* ‹¬ (*x*, *y*) ∉ *list.set ios*› ‹*m2* (*x*, *y*) = *Some t2*′› *add-le-less-mono height-over-subtree-less*)
then show ((*M*, *distFun*, *ios*, *Prefix-Tree.empty*, *q1*′, *t2*′, *q2*′), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
unfolding *pre-def*[*symmetric*] *post-def*[*symmetric*]
by *simp*

qed

moreover have $\bigwedge$(*M*::('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*)
(*distFun* :: ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*))
(*ios* :: ('*b*×'*c*) *list*)
m1 (*q1*::'*a*) *m2* (*q2*::'*a*) *x y t1'* *q1'* *q2'*.
¬ (*x*, *y*) ∉ *list.set ios* ⟹
m1 (*x*, *y*) = *Some t1'* ⟹
m2 (*x*, *y*) = *None* ⟹
((*M*, *distFun*, *ios*, *t1'*, *q1'*, *empty*, *q2'*), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
proof −
fix *M*::('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fix *distFun* :: ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*)
fix *ios* :: ('*b*×'*c*) *list*
fix *m1 m2* :: ('*b*×'*c*) ⇀ ('*b*×'*c*) *prefix-tree*
fix *t1'*
fix *q1 q2 q1' q2'* :: '*a*
fix *x* :: '*b*
fix *y* :: '*c*

assume *m1* (*x*, *y*) = *Some t1'*
assume *m2* (*x*, *y*) = *None*
assume ¬ (*x*, *y*) ∉ *list.set ios*

define *pre* **where** *pre* = (*M*, *distFun*, *ios*, *PT m1*, *q1*, *PT m2*, *q2*)
define *post* **where** *post* = (*M*, *distFun*, *ios*, *t1'*, *q1'*, *Prefix-Tree.empty*::('*b*×'*c*) *prefix-tree*, *q2'*)

have *height-over ios empty* ≤ *height-over ios* (*PT m2*)
unfolding *height-over.simps height-over-empty* **by** *auto*
then have *?f post* < *?f pre*
unfolding *pre-def post-def case-prod-conv*
by (*meson* ‹¬ (*x*, *y*) ∉ *list.set ios*› ‹*m1* (*x*, *y*) = *Some t1'*› *add-mono-thms-linordered-field*(*3*) *height-over-subtree-less*)
then show ((*M*, *distFun*, *ios*, *t1'*, *q1'*, *Prefix-Tree.empty*, *q2'*), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
unfolding *pre-def*[*symmetric*] *post-def*[*symmetric*]
by *simp*
qed

moreover have $\bigwedge$(*M*::('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*)

(*distFun* :: ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*))
(*ios* :: ('*b*×'*c*) *list*)
m1 (*q1*::'*a*) *m2* (*q2*::'*a*) *x y t1' t2' q1' q2'*.
¬ (*x*, *y*) ∉ *list.set ios* ⟹
m1 (*x*, *y*) = *Some t1'* ⟹
m2 (*x*, *y*) = *Some t2'* ⟹
((*M*, *distFun*, *ios*, *t1'*, *q1'*, *t2'*, *q2'*), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
proof −
fix *M*::('*a*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fix *distFun* :: ('*a* ⇒ '*a* ⇒ ('*b* × '*c*) *list*)
fix *ios* :: ('*b*×'*c*) *list*
fix *m1 m2* :: ('*b*×'*c*) ⇀ ('*b*×'*c*) *prefix-tree*
fix *t1' t2'* :: ('*b*×'*c*) *prefix-tree*
fix *q1 q2 q1' q2'* :: '*a*
fix *x* :: '*b*
fix *y* :: '*c*

define *pre* **where** *pre* = (*M*, *distFun*, *ios*, *PT m1*, *q1*, *PT m2*, *q2*)
define *post* **where** *post* = (*M*, *distFun*, *ios*, *t1'*, *q1'*, *t2'*, *q2'*)

assume *m1* (*x*, *y*) = *Some t1'*
moreover assume *m2* (*x*, *y*) = *Some t2'*
moreover assume ¬ (*x*, *y*) ∉ *list.set ios*
ultimately have *?f post* < *?f pre*
unfolding *pre-def post-def case-prod-conv*
by (*meson add-less-mono height-over-subtree-less*)
then show ((*M*, *distFun*, *ios*, *t1'*, *q1'*,*t2'*, *q2'*), *M*, *distFun*, *ios*,
PT m1, *q1*, *PT m2*, *q2*)
∈ *measure* (λ(*M*, *dF*, *ios*, *t1*, *q1*, *t2*, *q2*). *height-over ios t1* + *height-over ios t2*)
unfolding *pre-def*[*symmetric*] *post-def*[*symmetric*]
by *simp*
qed

ultimately show *?thesis*
by (*relation measure* (λ (*M*,*dF*,*ios*,*t1*,*q1*,*t2*,*q2*) . *height-over ios t1* + *height-over ios t2*); *simp*)
qed

lemma *find-cheapest-distinguishing-trace-alt-def* :
find-cheapest-distinguishing-trace M distFun ios (*PT m1*) *q1* (*PT m2*) *q2* =
(*let*

$f = (\lambda\ (\omega,l,w)\ (x,y).$
$(let$
$w1L = if\ (PT\ m1) = empty\ then\ 0\ else\ 1;$
$w1C = if\ (x,y) \in dom\ m1\ then\ 0\ else\ 1;$
$w1 = min\ w1L\ w1C;$
$w2L = if\ (PT\ m2) = empty\ then\ 0\ else\ 1;$
$w2C = if\ (x,y) \in dom\ m2\ then\ 0\ else\ 1;$
$w2 = min\ w2L\ w2C;$
$w' = w1 + w2$
in
$case\ h\text{-}obs\ M\ q1\ x\ y\ of$
$None \Rightarrow (case\ h\text{-}obs\ M\ q2\ x\ y\ of$
$None \Rightarrow (\omega,l,w)\ |$
$Some\ \text{-} \Rightarrow if\ w' = 0 \lor w' \leq w\ then\ ([(x,y)],w1C+w2C,w')\ else$
$(\omega,l,w))\ |$
$Some\ q1' \Rightarrow (case\ h\text{-}obs\ M\ q2\ x\ y\ of$
$None \Rightarrow if\ w' = 0 \lor w' \leq w\ then\ ([(x,y)],w1C+w2C,w')\ else\ (\omega,l,w)$
$|$
$Some\ q2' \Rightarrow (if\ q1' = q2'$
$then\ (\omega,l,w)$
$else\ (case\ m1\ (x,y)\ of$
$None \Rightarrow (case\ m2\ (x,y)\ of$
$None \Rightarrow let\ \omega' = distFun\ q1'\ q2';$
$l' = 2 + 2 * length\ \omega'$
$in\ if\ (w' < w) \lor (w' = w \land l' < l)\ then\ ((x,y)\#\omega',l',w')$
$else\ (\omega,l,w)\ |$
$Some\ t2' \Rightarrow let\ (\omega'',l'',w'') = find\text{-}cheapest\text{-}distinguishing\text{-}trace$
$M\ distFun\ ios\ empty\ q1'\ t2'\ q2'$
$in\ if\ (w'' + w1 < w) \lor (w'' + w1 = w \land l''+1 < l)$
$then\ ((x,y)\#\omega'',l''+1,w''+w1)\ else\ (\omega,l,w))\ |$
$Some\ t1' \Rightarrow (case\ m2\ (x,y)\ of$
$None \Rightarrow let\ (\omega'',l'',w'') = find\text{-}cheapest\text{-}distinguishing\text{-}trace\ M$
$distFun\ ios\ t1'\ q1'\ empty\ q2'$
$in\ if\ (w'' + w2 < w) \lor (w'' + w2 = w \land l''+1 < l)$
$then\ ((x,y)\#\omega'',l''+1,w''+w2)\ else\ (\omega,l,w)\ |$
$Some\ t2' \Rightarrow let\ (\omega'',l'',w'') = find\text{-}cheapest\text{-}distinguishing\text{-}trace$
$M\ distFun\ ios\ t1'\ q1'\ t2'\ q2'$
$in\ if\ (w'' < w) \lor (w'' = w \land l'' < l)\ then$
$((x,y)\#\omega'',l'',w'')\ else\ (\omega,l,w)))))))$
in
$foldl\ f\ (distFun\ q1\ q2,\ 0,\ 3)\ ios)$
(**is** *find-cheapest-distinguishing-trace M distFun ios (PT m1) q1 (PT m2) q2 = ?find-cheapest-distinguishing-trace*)

proof −
define f' **where** $f' = (\lambda\ (\omega,l,w)\ (x,y)\ .$
$(let$
$w1L = if\ (PT\ m1) = empty\ then\ 0\ else\ 1;$
$w1C = if\ (x,y) \in dom\ m1\ then\ 0\ else\ 1;$

$w1 = min\ w1L\ w1C;$
$w2L = if\ (PT\ m2) = empty\ then\ 0\ else\ 1;$
$w2C = if\ (x,y) \in dom\ m2\ then\ 0\ else\ 1;$
$w2 = min\ w2L\ w2C;$
$w' = w1 + w2$
in
case h-obs M q1 x y of
None $\Rightarrow$ (*case h-obs M q2 x y of*
None $\Rightarrow (\omega,l,w)$ |
Some - $\Rightarrow$ *if* $w' = 0 \vee w' \leq w$ *then* $([(x,y)],w1C+w2C,w')$ *else* (ω,l,w)) |
Some q1' $\Rightarrow$ (*case h-obs M q2 x y of*
None $\Rightarrow$ *if* $w' = 0 \vee w' \leq w$ *then* $([(x,y)],w1C+w2C,w')$ *else* (ω,l,w) |
Some q2' $\Rightarrow$ (*if q1' = q2'*
then (ω,l,w)
else (*case m1* (x,y) *of*
None $\Rightarrow$ (*case m2* (x,y) *of*
None $\Rightarrow$ *let* $\omega' =$ *distFun q1' q2';*
$l' = 2 + 2 * length\ \omega'$
in if $(w' < w) \vee (w' = w \wedge l' < l)$ *then* $((x,y)\#\omega',l',w')$ *else* (ω,l,w) |
Some t2' $\Rightarrow$ *let* (ω'',l'',w'') = *find-cheapest-distinguishing-trace M distFun ios empty q1' t2' q2'*
in if $(w'' + w1 < w) \vee (w'' + w1 = w \wedge l''+1 < l)$ *then* $((x,y)\#\omega'',l''+1,w''+w1)$ *else* (ω,l,w)) |
Some t1' $\Rightarrow$ (*case m2* (x,y) *of*
None $\Rightarrow$ *let* (ω'',l'',w'') = *find-cheapest-distinguishing-trace M distFun ios t1' q1' empty q2'*
in if $(w'' + w2 < w) \vee (w'' + w2 = w \wedge l''+1 < l)$ *then* $((x,y)\#\omega'',l''+1,w''+w2)$ *else* (ω,l,w) |
Some t2' $\Rightarrow$ *let* (ω'',l'',w'') = *find-cheapest-distinguishing-trace M distFun ios t1' q1' t2' q2'*
in if $(w'' < w) \vee (w'' = w \wedge l'' < l)$ *then* $((x,y)\#\omega'',l'',w'')$ *else* (ω,l,w))))))))

define f **where** $f = (\lambda\ (\omega,l,w)\ (x,y)\ .\ if\ (x,y) \notin list.set\ ios\ then\ (\omega,l,w)\ else$
(*let*
$w1L = if\ (PT\ m1) = empty\ then\ 0\ else\ 1;$
$w1C = if\ (x,y) \in dom\ m1\ then\ 0\ else\ 1;$
$w1 = min\ w1L\ w1C;$
$w2L = if\ (PT\ m2) = empty\ then\ 0\ else\ 1;$
$w2C = if\ (x,y) \in dom\ m2\ then\ 0\ else\ 1;$
$w2 = min\ w2L\ w2C;$
$w' = w1 + w2$
in
case h-obs M q1 x y of
None $\Rightarrow$ (*case h-obs M q2 x y of*
None $\Rightarrow (\omega,l,w)$ |

Some - ⇒ *if* $w' = 0 \lor w' \leq w$ *then* ([(x,y)],w1C+w2C,w′) *else* (ω,l,w)) |
Some q1′ ⇒ (*case h-obs M q2 x y of*
None ⇒ *if* $w' = 0 \lor w' \leq w$ *then* ([(x,y)],w1C+w2C,w′) *else* (ω,l,w) |
Some q2′ ⇒ (*if q1′ = q2′*
then (ω,l,w)
else (*case m1* (x,y) *of*
None ⇒ (*case m2* (x,y) *of*
None ⇒ *let* ω′ = *distFun q1′ q2′*;
l′ = 2 + 2 ∗ *length* ω′
in if (w′ < w) ∨ (w′ = w ∧ l′ < l) *then* ((x,y)#ω′,l′,w′) *else* (ω,l,w) |
Some t2′ ⇒ *let* (ω″,l″,w″) = *find-cheapest-distinguishing-trace M distFun ios empty q1′ t2′ q2′*
in if (w″ + w1 < w) ∨ (w″ + w1 = w ∧ l″+1 < l) *then* ((x,y)#ω″,l″+1,w″+w1) *else* (ω,l,w)) |
Some t1′ ⇒ (*case m2* (x,y) *of*
None ⇒ *let* (ω″,l″,w″) = *find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′*
in if (w″ + w2 < w) ∨ (w″ + w2 = w ∧ l″+1 < l) *then* ((x,y)#ω″,l″+1,w″+w2) *else* (ω,l,w) |
Some t2′ ⇒ *let* (ω″,l″,w″) = *find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′*
in if (w″ < w) ∨ (w″ = w ∧ l″ < l) *then* ((x,y)#ω″,l″,w″) *else* (ω,l,w)))))))
then have *f* = (λ *y x* . *if x* ∉ *list.set ios then y else f′ y x*)
unfolding *f′-def* **by** *fast*
moreover have *find-cheapest-distinguishing-trace M distFun ios* (*PT m1*) *q1* (*PT m2*) *q2* = *foldl f* (*distFun q1 q2*, *0*, *3*) *ios*
unfolding *find-cheapest-distinguishing-trace.simps f-def*[*symmetric*] **by** *auto*
ultimately have *find-cheapest-distinguishing-trace M distFun ios* (*PT m1*) *q1* (*PT m2*) *q2* = *foldl* (λ *y x* . *if x* ∉ *list.set ios then y else f′ y x*) (*distFun q1 q2*, *0*, *3*) *ios*
by *auto*
then show *?thesis*
unfolding *f′-def*[*symmetric*]
using *foldl-elem-check*[*of ios list.set ios*]
by *auto*
qed

lemma *find-cheapest-distinguishing-trace-code*[*code*] :
find-cheapest-distinguishing-trace M distFun ios (*MPT m1*) *q1* (*MPT m2*) *q2* =
(*let*
f = (λ (ω,l,w) (x,y) .
(*let*
w1L = *if is-leaf* (*MPT m1*) *then 0 else 1*;
w1C = *if* (x,y) ∈ *Mapping.keys m1 then 0 else 1*;

```
            w1 = min w1L w1C;
            w2L = if is-leaf (MPT m2) then 0 else 1;
            w2C = if(x,y) ∈ Mapping.keys m2 then 0 else 1;
            w2 = min w2L w2C;
            w' = w1 + w2
          in
            case h-obs M q1 x y of
              None ⇒ (case h-obs M q2 x y of
                None ⇒ (ω,l,w) |
                  Some - ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else
(ω,l,w)) |
              Some q1' ⇒ (case h-obs M q2 x y of
              None ⇒ if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w') else (ω,l,w)
|
                Some q2' ⇒ (if q1' = q2'
                  then (ω,l,w)
                  else (case Mapping.lookup m1 (x,y) of
                    None ⇒ (case Mapping.lookup m2 (x,y) of
                      None ⇒ let ω' = distFun q1' q2';
                                l' = 2 + 2 ∗ length ω'
                            in if (w' < w) ∨ (w' = w ∧ l' < l) then ((x,y)#ω',l',w')
else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2'
                                in if (w'' + w1 < w) ∨ (w'' + w1 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w1) else (ω,l,w)) |
                    Some t1' ⇒ (case Mapping.lookup m2 (x,y) of
                      None ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace M
distFun ios t1' q1' empty q2'
                              in if (w'' + w2 < w) ∨ (w'' + w2 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w2) else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios t1' q1' t2' q2'
                                  in if (w'' < w) ∨ (w'' = w ∧ l'' < l) then
((x,y)#ω'',l'',w'') else (ω,l,w)))))))
    in
      foldl f (distFun q1 q2, 0, 3) ios)
  unfolding find-cheapest-distinguishing-trace-alt-def MPT-def
  by (simp add: keys-dom-lookup)
```

lemma *find-cheapest-distinguishing-trace-is-distinguishing-trace* :

```
  assumes observable M
  and     minimal M
  and     q1 ∈ states M
  and     q2 ∈ states M
  and     q1 ≠ q2
  and     ⋀ q1 q2 . q1 ∈ states M ⟹ q2 ∈ states M ⟹ q1 ≠ q2 ⟹ distinguishes
```

M q1 q2 (*distFun q1 q2*)
shows *distinguishes M q1 q2* (*fst* (*find-cheapest-distinguishing-trace M distFun ios t1 q1 t2 q2*))
using *assms*(*3*,*4*,*5*)
proof (*induction height-over ios t1 + height-over ios t2 arbitrary*: *t1 q1 t2 q2 rule*: *less-induct*)
case *less*

obtain *m1* **where** *t1* = *PT m1*
using *prefix-tree.exhaust* **by** *blast*
obtain *m2* **where** *t2* = *PT m2*
using *prefix-tree.exhaust* **by** *blast*

define *f* **where** *f* = (λ (ω,*l*,*w*) (*x*,*y*) .
(*let*
w1L = *if* (*PT m1*) = *empty then 0 else 1*;
w1C = *if* (*x*,*y*) ∈ *dom m1 then 0 else 1*;
w1 = *min w1L w1C*;
w2L = *if* (*PT m2*) = *empty then 0 else 1*;
w2C = *if* (*x*,*y*) ∈ *dom m2 then 0 else 1*;
w2 = *min w2L w2C*;
w' = *w1* + *w2*
in
case h-obs M q1 x y of
None ⇒ (*case h-obs M q2 x y of*
None ⇒ (ω,*l*,*w*) |
Some - ⇒ *if w'* = *0* ∨ *w'* ≤ *w then* ([(*x*,*y*)],*w1C*+*w2C*,*w'*) *else* (ω,*l*,*w*)) |
Some q1' ⇒ (*case h-obs M q2 x y of*
None ⇒ *if w'* = *0* ∨ *w'* ≤ *w then* ([(*x*,*y*)],*w1C*+*w2C*,*w'*) *else* (ω,*l*,*w*) |
Some q2' ⇒ (*if q1'* = *q2'*
then (ω,*l*,*w*)
else (*case m1* (*x*,*y*) *of*
None ⇒ (*case m2* (*x*,*y*) *of*
None ⇒ *let* ω' = *distFun q1' q2'*;
l' = *2* + *2* ∗ *length* ω'
in if (*w'* < *w*) ∨ (*w'* = *w* ∧ *l'* < *l*) *then* ((*x*,*y*)#ω',*l'*,*w'*) *else* (ω,*l*,*w*) |
Some t2' ⇒ *let* (ω'',*l''*,*w''*) = *find-cheapest-distinguishing-trace M distFun ios empty q1' t2' q2'*
in if (*w''* + *w1* < *w*) ∨ (*w''* + *w1* = *w* ∧ *l''*+*1* < *l*) *then* ((*x*,*y*)#ω'',*l''*+*1*,*w''*+*w1*) *else* (ω,*l*,*w*)) |
Some t1' ⇒ (*case m2* (*x*,*y*) *of*
None ⇒ *let* (ω'',*l''*,*w''*) = *find-cheapest-distinguishing-trace M distFun ios t1' q1' empty q2'*
in if (*w''* + *w2* < *w*) ∨ (*w''* + *w2* = *w* ∧ *l''*+*1* < *l*) *then* ((*x*,*y*)#ω'',*l''*+*1*,*w''*+*w2*) *else* (ω,*l*,*w*) |

Some t2′ ⇒ let (ω″,l″,w″) = find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′
in if (w″ < w) ∨ (w″ = w ∧ l″ < l) then ((x,y)#ω″,l″,w″) else (ω,l,w)))))))

then have *find-cheapest-distinguishing-trace M distFun ios t1 q1 t2 q2 = foldl f (distFun q1 q2, 0, 3) ios*
unfolding ‹*t1 = PT m1*› ‹*t2 = PT m2*›
unfolding *find-cheapest-distinguishing-trace-alt-def Let-def*
by *fast*

define *ios′* **where** *ios′=ios*

have *list.set ios′ ⊆ list.set ios ⟹ distinguishes M q1 q2 (fst (foldl f (distFun q1 q2, 0, 3) ios′))*
proof (*induction ios′ rule*: *rev-induct*)
case *Nil*
then show *?case* **using** *assms*(*6*)[*OF less.prems*] **by** *auto*
next
case (*snoc xy ios′*)

obtain *x y* **where** *xy* = (*x*,*y*)
using *prod.exhaust* **by** *metis*
moreover obtain *ω l w* **where** (*foldl f* (*distFun q1 q2*, *0*, *3*) *ios′*) = (*ω*,*l*,*w*)
using *prod.exhaust* **by** *metis*
ultimately have *foldl f* (*distFun q1 q2*, *0*, *3*) (*ios′*@[*xy*]) = *f* (*ω*,*l*,*w*) (*x*,*y*)
by *auto*

have *distinguishes M q1 q2 ω*
using ‹(*foldl f* (*distFun q1 q2*, *0*, *3*) *ios′*) = (*ω*,*l*,*w*)› *snoc* **by** *auto*

have (*x*,*y*) ∈ *list.set ios*
using *snoc.prems* **unfolding** ‹*xy* = (*x*,*y*)› **by** *auto*

define *w1L* **where** *w1L* = (*if* (*PT m1*) = *empty then 0 else 1*::*nat*)
define *w1C* **where** *w1C* = (*if* (*x*,*y*) ∈ *dom m1 then 0 else 1*::*nat*)
define *w1* **where** *w1* = *min w1L w1C*
define *w2L* **where** *w2L* = (*if* (*PT m2*) = *empty then 0 else 1*::*nat*)
define *w2C* **where** *w2C* = (*if* (*x*,*y*) ∈ *dom m2 then 0 else 1*::*nat*)
define *w2* **where** *w2* = *min w2L w2C*
define *w′* **where** *w′* = *w1* + *w2*

have ∗:*f* (*ω*,*l*,*w*) (*x*,*y*) = (*case h-obs M q1 x y of*
None ⇒ (*case h-obs M q2 x y of*
None ⇒ (*ω*,*l*,*w*) |
Some - ⇒ *if w′* = *0* ∨ *w′* ≤ *w then* ([(*x*,*y*)],*w1C*+*w2C*,*w′*) *else* (*ω*,*l*,*w*)) |
Some q1′ ⇒ (*case h-obs M q2 x y of*
None ⇒ *if w′* = *0* ∨ *w′* ≤ *w then* ([(*x*,*y*)],*w1C*+*w2C*,*w′*) *else* (*ω*,*l*,*w*)

```
|
                Some q2' ⇒ (if q1' = q2'
                  then (ω,l,w)
                  else (case m1 (x,y) of
                    None ⇒ (case m2 (x,y) of
                      None ⇒ let ω' = distFun q1' q2';
                                l' = 2 + 2 ∗ length ω'
                            in if (w' < w) ∨ (w' = w ∧ l' < l) then ((x,y)#ω',l',w')
else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2'
                                in if (w'' + w1 < w) ∨ (w'' + w1 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w1) else (ω,l,w)) |
                    Some t1' ⇒ (case m2 (x,y) of
                      None ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace M
distFun ios t1' q1' empty q2'
                                in if (w'' + w2 < w) ∨ (w'' + w2 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w2) else (ω,l,w) |
                      Some t2' ⇒ let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios t1' q1' t2' q2'
                                  in if (w'' < w) ∨ (w'' = w ∧ l'' < l) then
((x,y)#ω'',l'',w'') else (ω,l,w))))))
      unfolding w1-def w2-def w'-def w1L-def w1C-def w2L-def w2C-def
      unfolding f-def case-prod-conv Let-def
      by fast

    have distinguishes M q1 q2 (fst (f (ω,l,w) (x,y)))
    proof (cases h-obs M q1 x y)
      case None
      then show ?thesis proof (cases h-obs M q2 x y)
        case None
        have f (ω,l,w) (x,y) = (ω,l,w)
          unfolding ∗
          unfolding ‹h-obs M q1 x y = None› None by auto
        then show ?thesis
          using ‹distinguishes M q1 q2 ω› by auto
      next
        case (Some a)
         have f (ω,l,w) (x,y) = (if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w')
else (ω,l,w))
          unfolding ∗ None Some by auto
        moreover have distinguishes M q1 q2 [(x,y)]
         using distinguishes-sym[OF h-obs-distinguishes[OF assms(1) Some None]]
.
        ultimately show ?thesis
          using ‹distinguishes M q1 q2 ω› by auto
      qed
    next
      case (Some q1')
```

```
    then have q1' ∈ states M
      by (meson h-obs-state)

    show ?thesis proof (cases h-obs M q2 x y)
      case None
       have f (ω,l,w) (x,y) = (if w' = 0 ∨ w' ≤ w then ([(x,y)],w1C+w2C,w')
else (ω,l,w))
        unfolding ∗ None Some by auto
      moreover have distinguishes M q1 q2 [(x,y)]
        using h-obs-distinguishes[OF assms(1) Some None] .
      ultimately show ?thesis
        using ‹distinguishes M q1 q2 ω› by auto
    next
      case (Some q2')
      then have q2' ∈ states M
        by (meson h-obs-state)

      show ?thesis proof (cases q1' = q2')
        case True
        have f (ω,l,w) (x,y) = (ω,l,w)
          unfolding ∗
          unfolding ‹h-obs M q1 x y = Some q1'› Some True by auto
        then show ?thesis
          using ‹distinguishes M q1 q2 ω› by auto
      next
        case False

         have dist': ⋀ ω . distinguishes M q1' q2' ω ⟹ distinguishes M q1 q2
((x,y)#ω)
          using distinguishes-after-prepend[OF assms(1), of q1 x y q2]
          using ‹h-obs M q1 x y = Some q1'› ‹h-obs M q2 x y = Some q2'›
          unfolding after-h-obs[OF assms(1) ‹h-obs M q1 x y = Some q1'›]
          unfolding after-h-obs[OF assms(1) ‹h-obs M q2 x y = Some q2'›]
          by auto

        show ?thesis proof (cases m1 (x,y))
          case None
          show ?thesis proof (cases m2 (x,y))
            case None

             have ∗∗: f (ω,l,w) (x,y) = (let ω' = distFun q1' q2';  l' = 2 + 2 ∗
length ω'
                                          in if (w' < w) ∨ (w' = w ∧ l' < l) then
((x,y)#ω',l',w') else (ω,l,w))
              unfolding ∗
               unfolding ‹h-obs M q1 x y = Some q1'› ‹h-obs M q2 x y = Some
q2'› ‹m1 (x, y) = None› ‹m2 (x, y) = None›
              using False
              by auto
```

```
        have distinguishes M q1' q2' (distFun q1' q2')
          using ‹q1' ∈ states M› ‹q2' ∈ states M› assms(6) False by blast
        then have distinguishes M q1 q2 ((x,y)#(distFun q1' q2'))
          using dist' by auto
        then  show ?thesis
          using ‹distinguishes M q1 q2 ω›
          unfolding ** Let-def by auto
      next
        case (Some t2')

      have **: f (ω,l,w) (x,y) = (let (ω'',l'',w'') = find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2'
                                  in if (w'' + w1 < w) ∨ (w'' + w1 = w ∧ l''+1 < l)
then ((x,y)#ω'',l''+1,w''+w1) else (ω,l,w))
          unfolding *
           unfolding ‹h-obs M q1 x y = Some q1'› ‹h-obs M q2 x y = Some
q2'› ‹m1 (x, y) = None› ‹m2 (x, y) = Some t2'›
          using False
          by auto

      obtain ω'' l'' w'' where ***:find-cheapest-distinguishing-trace M distFun
ios Prefix-Tree.empty q1' t2' q2' = (ω'', l'', w'')
          using prod.exhaust by metis

        have distinguishes M q1' q2' (fst (find-cheapest-distinguishing-trace M
distFun ios empty q1' t2' q2'))
        proof −

          have height-over ios empty + height-over ios t2' < height-over ios t1
+ height-over ios t2
           using height-over-subtree-less[of m2 (x,y), OF ‹m2 (x,y) = Some
t2'› ‹(x,y) ∈ list.set ios› ]
            unfolding height-over-empty ‹t2 = PT m2›[symmetric]
            by (simp add: ‹t1 = PT m1›)
          then show ?thesis
            using less.hyps[OF - ‹q1' ∈ states M› ‹q2' ∈ states M› False]
            by blast
        qed
      then have distinguishes M q1 q2 ((x,y)#(fst (find-cheapest-distinguishing-trace
M distFun ios empty q1' t2' q2')))
          using dist' by blast
        then  show ?thesis
          using ‹distinguishes M q1 q2 ω›
          unfolding ** *** Let-def fst-conv case-prod-conv by auto
      qed
    next
      case (Some t1')
      show ?thesis proof (cases m2 (x,y))
```

case *None*

have $**$: *f* (ω,*l*,*w*) (*x*,*y*) = (*let* (ω′′,*l*′′,*w*′′) = *find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′*
in if (*w*′′ + *w2* < *w*) ∨ (*w*′′ + *w2* = *w* ∧ *l*′′+*1* < *l*)
then ((*x*,*y*)#ω′′,*l*′′+*1*,*w*′′+*w2*) *else* (ω,*l*,*w*))
unfolding $*$
unfolding ‹*h-obs M q1 x y* = *Some q1′*› ‹*h-obs M q2 x y* = *Some q2′*› ‹*m1* (*x*, *y*) = *Some t1′*› ‹*m2* (*x*, *y*) = *None*›
using *False*
by *auto*

obtain ω′′ *l*′′ *w*′′ **where** $***$:*find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′* = (ω′′, *l*′′, *w*′′)
using *prod.exhaust* **by** *metis*

have *distinguishes M q1′ q2′* (*fst* (*find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′*))
proof −

have *height-over ios t1′* + *height-over ios empty* < *height-over ios t1* + *height-over ios t2*
using *height-over-subtree-less*[*of m1* (*x*,*y*), *OF* ‹*m1* (*x*,*y*) = *Some t1′*› ‹(*x*,*y*) ∈ *list.set ios*›]
unfolding *height-over-empty* ‹*t1* = *PT m1*›[*symmetric*]
by (*simp add*: ‹*t2* = *PT m2*›)
then show *?thesis*
using *less.hyps*[*OF* - ‹*q1′* ∈ *states M*› ‹*q2′* ∈ *states M*› *False*]
by *blast*
qed
then have *distinguishes M q1 q2* ((*x*,*y*)#(*fst* (*find-cheapest-distinguishing-trace M distFun ios t1′ q1′ empty q2′*)))
using *dist′* **by** *blast*
then show *?thesis*
using ‹*distinguishes M q1 q2* ω›
unfolding $**$ $***$ *Let-def fst-conv case-prod-conv* **by** *auto*
next
case (*Some t2′*)

have $**$: *f* (ω,*l*,*w*) (*x*,*y*) = (*let* (ω′′,*l*′′,*w*′′) = *find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′*
in if (*w*′′ < *w*) ∨ (*w*′′ = *w* ∧ *l*′′ < *l*) *then* ((*x*,*y*)#ω′′,*l*′′,*w*′′) *else* (ω,*l*,*w*))
unfolding $*$
unfolding ‹*h-obs M q1 x y* = *Some q1′*› ‹*h-obs M q2 x y* = *Some q2′*› ‹*m1* (*x*, *y*) = *Some t1′*› ‹*m2* (*x*, *y*) = *Some t2′*›
using *False*
by *auto*
obtain ω′′ *l*′′ *w*′′ **where** $***$:*find-cheapest-distinguishing-trace M distFun*

ios t1′ q1′ t2′ q2′ = (*ω″*, *l″*, *w″*)
using *prod.exhaust* **by** *metis*

have *distinguishes M q1′ q2′* (*fst* (*find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′*))
proof −

have *height-over ios t1′* + *height-over ios t2′* < *height-over ios t1* + *height-over ios t2*
using *height-over-subtree-less*[*of m1* (*x*,*y*), *OF* ‹*m1* (*x*,*y*) = *Some t1′*› ‹(*x*,*y*) ∈ *list.set ios*›]
using *height-over-subtree-less*[*of m2* (*x*,*y*), *OF* ‹*m2* (*x*,*y*) = *Some t2′*› ‹(*x*,*y*) ∈ *list.set ios*›]
unfolding ‹*t1* = *PT m1*›[*symmetric*] ‹*t2* = *PT m2*›[*symmetric*]
by *auto*
then show *?thesis*
using *less.hyps*[*OF* - ‹*q1′* ∈ *states M*› ‹*q2′* ∈ *states M*› *False*]
by *blast*
qed
then have *distinguishes M q1 q2* ((*x*,*y*)#(*fst* (*find-cheapest-distinguishing-trace M distFun ios t1′ q1′ t2′ q2′*)))
using *dist′* **by** *blast*
then show *?thesis*
using ‹*distinguishes M q1 q2 ω*›
unfolding ** *** *Let-def fst-conv case-prod-conv* **by** *auto*
qed
qed
qed
qed
qed
then show *?case*
unfolding ‹*foldl f* (*distFun q1 q2*, *0*, *3*) (*ios′*@[*xy*]) = *f* (*ω*,*l*,*w*) (*x*,*y*)› **.**
qed

then show *?case*
unfolding ‹*find-cheapest-distinguishing-trace M distFun ios t1 q1 t2 q2* = *foldl f* (*distFun q1 q2*, *0*, *3*) *ios*›
‹*ios′* = *ios*›
by *blast*
qed

fun *add-cheapest-distinguishing-trace* :: (*′a* ⇒ *′a* ⇒ (*′b* × *′c*) *list*) ⇒ *bool* ⇒ (*′a*::*linorder*,*′b*::*linorder*,*′c*::*linorder*) *fsm* ⇒ ((*′b*×*′c*) *list* × *′a*) × ((*′b*×*′c*) *list* × *′a*) ⇒ (*′b*×*′c*) *prefix-tree* ⇒ (*′b*×*′c*) *prefix-tree* **where**
add-cheapest-distinguishing-trace distFun completeInputTraces M ((*α*,*q1*), (*β*,*q2*)) *t* =
(*let w* = (*fst* (*find-cheapest-distinguishing-trace M distFun* (*List.product* (*inputs-as-list*

M) (*outputs-as-list M*)) (*after t* α) *q1* (*after t* β) *q2*));
T = *insert empty w*
in if completeInputTraces
then let T1 = *complete-inputs-to-tree M q1* (*outputs-as-list M*) (*map fst w*);
T2 = *complete-inputs-to-tree M q2* (*outputs-as-list M*) (*map fst w*)
in Prefix-Tree.combine T (*Prefix-Tree.combine T1 T2*)
else T)

lemma *add-cheapest-distinguishing-trace-distinguishes* :
assumes *observable M*
and *minimal M*
and $\alpha \in$ *L M*
and $\beta \in$ *L M*
and *after-initial M* $\alpha \neq$ *after-initial M* β
and $\bigwedge$ *q1 q2* . *q1* $\in$ *states M* $\Longrightarrow$ *q2* $\in$ *states M* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *distinguishes M q1 q2* (*dist-fun q1 q2*)
shows $\exists$ *io* $\in$ *set* ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
proof −
define *w* **where** *w* = (*fst* (*find-cheapest-distinguishing-trace M dist-fun* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*)) (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β)))

have *set* (*insert empty w*) $\subseteq$ *set* ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
using *combine-set*[*of insert empty w*] *w-def*
unfolding *add-cheapest-distinguishing-trace.simps Let-def*
by (*cases c*; *fastforce*)
moreover have *w* $\in$ *set* (*insert empty w*)
unfolding *insert-set* **by** *auto*
ultimately have *w* $\in$ *set* ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β))
by *blast*
moreover have *distinguishes M* (*after-initial M* α) (*after-initial M* β) *w*
using *find-cheapest-distinguishing-trace-is-distinguishing-trace*[*OF assms*(*1*,*2*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*,*6*)]
unfolding *w-def*
by *blast*
ultimately show *?thesis*
by *blast*
qed

lemma *add-cheapest-distinguishing-trace-finite* :
finite-tree ((*add-cheapest-distinguishing-trace dist-fun c M*) ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
proof −

define *w* **where** *w*: *w* = (*fst* (*find-cheapest-distinguishing-trace M dist-fun* (*List.product* (*inputs-as-list M*) (*outputs-as-list M*)) (*after t* α) (*after-initial M* α) (*after t* β) (*after-initial M* β)))
define *T* **where** *T*: *T* = *insert empty w*
define *T1* **where** *T1*: *T1* = *complete-inputs-to-tree M* (*after-initial M* α) (*outputs-as-list M*) (*map fst w*)
define *T2* **where** *T2*: *T2* = *complete-inputs-to-tree M* (*after-initial M* β) (*outputs-as-list M*) (*map fst w*)

have *finite-tree T*
using *insert-finite-tree*[*OF empty-finite-tree*]
unfolding *T* **by** *auto*
moreover have *finite-tree* (*Prefix-Tree.combine T* (*Prefix-Tree.combine T1 T2*))
using *combine-finite-tree*[*OF* ‹*finite-tree T*› *combine-finite-tree*[*OF complete-inputs-to-tree-finite-tree complete-inputs-to-tree-finite-tree*]]
unfolding *T1 T2*
by *auto*
ultimately show *?thesis*
unfolding *add-cheapest-distinguishing-trace.simps w T T1 T2 Let-def*
by *presburger*
qed

25.2.2 Implementation

definition *h-method-via-pair-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
h-method-via-pair-framework M m = *pair-framework-h-components M m* (*add-distinguishing-sequence-if-requi* (*get-distinguishing-sequence-from-ofsm-tables M*))

lemma *h-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework M m*) = *L I* ∩ *set* (*h-method-via-pair-framework M m*))
and *finite-tree* (*h-method-via-pair-framework M m*)
using *pair-framework-h-components-completeness-and-finiteness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*), **where** *get-separating-traces*=(*add-distinguishing-sequence-if-required* (*get-distinguishing-sequence-from-ofsm-t M*)), *OF add-distinguishing-sequence-if-required-distinguishes*[*OF assms*(*1*,*3*), **where** *dist-fun*=(*get-distinguishing-sequence-from-ofsm-tables M*)] *add-distinguishing-sequence-if-required-finite*[**wher** *dist-fun*=(*get-distinguishing-sequence-from-ofsm-tables M*)]]
using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *h-method-via-pair-framework-def*[*symmetric*]
by *blast+*

definition *h-method-via-pair-framework-2* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒ *nat* ⇒ *bool* ⇒ ('b×'c) *prefix-tree* **where**
h-method-via-pair-framework-2 M m c = *pair-framework-h-components M m* (*add-distinguishing-sequence-and*
(*get-distinguishing-sequence-from-ofsm-tables M*) *c*)

lemma *h-method-via-pair-framework-2-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework-2 M m c*) =
L I ∩ *set* (*h-method-via-pair-framework-2 M m c*))
and *finite-tree* (*h-method-via-pair-framework-2 M m c*)
using *pair-framework-h-components-completeness-and-finiteness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*),
where *get-separating-traces*=(*add-distinguishing-sequence-and-complete-if-required*
(*get-distinguishing-sequence-from-ofsm-tables M*) *c*), *OF add-distinguishing-sequence-and-complete-if-required-*
assms(*1*,*3*), **where** *dist-fun*=(*get-distinguishing-sequence-from-ofsm-tables M*)] *add-distinguishing-sequence-an*
dist-fun=(*get-distinguishing-sequence-from-ofsm-tables M*)]]
using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *h-method-via-pair-framework-2-def*[*symmetric*]
by *blast+*

definition *h-method-via-pair-framework-3* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ('b×'c) *prefix-tree* **where**
h-method-via-pair-framework-3 M m c1 c2 = *pair-framework-h-components-2 M*
m (*add-cheapest-distinguishing-trace* (*get-distinguishing-sequence-from-ofsm-tables*
M) *c2*) *c1*

lemma *h-method-via-pair-framework-3-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ (*L M* ∩ *set* (*h-method-via-pair-framework-3 M m c1 c2*)
= *L I* ∩ *set* (*h-method-via-pair-framework-3 M m c1 c2*))
and *finite-tree* (*h-method-via-pair-framework-3 M m c1 c2*)
using *pair-framework-h-components-2-completeness-and-finiteness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*),
where *get-separating-traces*=(*add-cheapest-distinguishing-trace* (*get-distinguishing-sequence-from-ofsm-tables*
M) *c2*), *OF add-cheapest-distinguishing-trace-distinguishes*[*OF assms*(*1*,*3*), **where**
dist-fun=(*get-distinguishing-sequence-from-ofsm-tables M*)] *add-cheapest-distinguishing-trace-finite*[**where**
dist-fun=(*get-distinguishing-sequence-from-ofsm-tables M*)]]
using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *h-method-via-pair-framework-3-def*[*symmetric*]

by *blast+*

definition *h-method-via-pair-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
h-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*h-method-via-pair-framework M m*))

lemma *h-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* ≤ *m*
and *m* ≥ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) ⟷ *list-all* (*passes-test-case I* (*initial I*)) (*h-method-via-pair-framework-lists M m*)
unfolding *h-method-via-pair-framework-lists-def*
h-method-via-pair-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial h-method-via-pair-framework-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

25.2.3 Code Equations

lemma *h-method-via-pair-framework-code*[*code*] :
h-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
distFun = *add-distinguishing-sequence-if-required distHelper*
in pair-framework-h-components M m distFun)
unfolding *h-method-via-pair-framework-def*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

lemma *h-method-via-pair-framework-2-code*[*code*] :
h-method-via-pair-framework-2 M m c = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)

(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
distFun = *add-distinguishing-sequence-and-complete-if-required distHelper c*
in pair-framework-h-components M m distFun)
unfolding *h-method-via-pair-framework-2-def*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

lemma *h-method-via-pair-framework-3-code*[*code*] :
h-method-via-pair-framework-3 M m c1 c2 = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
distFun = *add-cheapest-distinguishing-trace distHelper c2*
in pair-framework-h-components-2 M m distFun c1)
unfolding *h-method-via-pair-framework-3-def*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

end

26 Implementations of the HSI-Method

theory *HSI-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

26.1 Using the H-Framework

definition *hsi-method-via-h-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ (*'b*×*'c*) *prefix-tree* **where**
hsi-method-via-h-framework M m = *h-framework-static-with-empty-graph M* (λ *k q* . *get-HSI M q*) *m*

definition *hsi-method-via-h-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
hsi-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-h-framework M m*))

lemma *hsi-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*hsi-method-via-h-framework M1 m*))
= (*L M2* $\cap$ *set* (*hsi-method-via-h-framework M1 m*)))
and *finite-tree* (*hsi-method-via-h-framework M1 m*)
using *h-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*,
where *dist-fun*=(λ *k q* . *get-HSI M1 q*)]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
using *get-HSI-finite*
unfolding *hsi-method-via-h-framework-def*
by *blast*+

lemma *hsi-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*hsi-method-via-h-framework-lists M1 m*)
using *h-framework-static-with-empty-graph-lists-completeness*[*OF assms*, **where** *dist-fun*=(λ *k q* . *get-HSI M1 q*), *OF - get-HSI-finite*]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *hsi-method-via-h-framework-lists-def h-framework-static-with-empty-graph-lists-def hsi-method-via-h-framework-def*
by *blast*

26.2 Using the SPY-Framework

definition *hsi-method-via-spy-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
hsi-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (λ *k q* . *get-HSI M q*) *m*

lemma *hsi-method-via-spy-framework-completeness-and-finiteness* :

fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*hsi-method-via-spy-framework M1 m*))
= (*L M2* $\cap$ *set* (*hsi-method-via-spy-framework M1 m*)))
and *finite-tree* (*hsi-method-via-spy-framework M1 m*)
unfolding *hsi-method-via-spy-framework-def*
using *spy-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*,
of (λ *k q* . *get-HSI M1 q*)]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
using *get-HSI-finite*[*of M1*]
by *blast*+

definition *hsi-method-via-spy-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*)
fsm $\Rightarrow$ *nat* $\Rightarrow$ (('*b*$\times$'*c*) $\times$ *bool*) *list list* **where**
hsi-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree*
(*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-spy-framework M m*))

lemma *hsi-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*hsi-method-via-spy-framework-lists*
M1 m)
unfolding *hsi-method-via-spy-framework-lists-def*
hsi-method-via-spy-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial*
hsi-method-via-spy-framework-completeness-and-finiteness(*2*)[*OF assms*]]
by *blast*

26.3 Using the Pair-Framework

definition *hsi-method-via-pair-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*)
fsm $\Rightarrow$ *nat* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
hsi-method-via-pair-framework M m = *pair-framework-h-components M m* (*add-distinguishing-sequence*)

lemma *hsi-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) $\longleftrightarrow$ (*L M* $\cap$ *set* (*hsi-method-via-pair-framework M m*) = *L I* $\cap$ *set* (*hsi-method-via-pair-framework M m*))
and *finite-tree* (*hsi-method-via-pair-framework M m*)
using *pair-framework-h-components-completeness-and-finiteness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*),
where *get-separating-traces*=*add-distinguishing-sequence*, *OF add-distinguishing-sequence-distinguishes*[*OF assms*(*1*,*3*)] *add-distinguishing-sequence-finite*]
using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *hsi-method-via-pair-framework-def*[*symmetric*]
by *blast+*

definition *hsi-method-via-pair-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ((*'b*$\times$*'c*) $\times$ *bool*) *list list* **where**
hsi-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*hsi-method-via-pair-framework M m*))

lemma *hsi-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) $\longleftrightarrow$ *list-all* (*passes-test-case I* (*initial I*)) (*hsi-method-via-pair-framework-lists M m*)
unfolding *hsi-method-via-pair-framework-lists-def*
hsi-method-via-pair-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial hsi-method-via-pair-framework-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

26.4 Code Generation

lemma *hsi-method-via-pair-framework-code*[*code*] :
hsi-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* $\neq$ *snd qq*) (*List.product* (*states-as-list M*)

(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
distFun = (λ *M* ((*io1*,*q1*),(*io2*,*q2*)) *t* . *insert empty* (*distHelper q1 q2*))
in pair-framework-h-components M m distFun)
unfolding *hsi-method-via-pair-framework-def pair-framework-h-components-def pair-framework-def*
unfolding *add-distinguishing-sequence.simps*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def case-prod-conv*
by *presburger*

lemma *hsi-method-via-spy-framework-code*[*code*] :
hsi-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *distHelper q q*′) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else get-HSI M q*)
in spy-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
proof −

define *hsiMap*′ **where** *hsiMap*′ = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun*′ **where** *distFun*′ = (λ *M q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap*′ *q*) *else get-HSI M q*)

have ∗: *?f2* = *spy-framework-static-with-empty-graph M* (λ *k q* . *distFun*′ *M q*) *m*
unfolding *distFun*′*-def hsiMap*′*-def Let-def*
apply (*subst* (*2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* ((≠) *q*) (*states-as-list*

M))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *M q* . *if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap*′ = *hsiMap*
unfolding *hsiMap-def hsiMap*′*-def*
using *mapping-of-map-of*
by *blast*
then have *distFun*′ = *distFun*
unfolding *distFun-def distFun*′*-def* **by** *meson*

have ∗∗:*distFun M* = *get-HSI M*
proof
fix *q* **show** *distFun M q* = *get-HSI M q*
proof (*cases q* ∈ *states M*)
case *True*
then have *q* ∈ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *blast*
then show *?thesis*
unfolding *distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps*
using *True*
by *fastforce*
next
case *False*
then show *?thesis* **using** *distFun-def* **by** *auto*
qed
qed

show *?thesis*
unfolding ∗ ∗∗ ‹*distFun*′ = *distFun*› *hsi-method-via-spy-framework-def* **by** *simp*
qed

lemma *hsi-method-via-h-framework-code*[*code*] :
hsi-method-via-h-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *distHelper q q*′) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));

distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else get-HSI M q*)
in h-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
proof −

define *hsiMap′* **where** *hsiMap′* = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun′* **where** *distFun′* = (λ *M q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap′ q*) *else get-HSI M q*)

have ∗: *?f2* = *h-framework-static-with-empty-graph M* (λ *k q* . *distFun′ M q*) *m*
unfolding *distFun′-def hsiMap′-def Let-def*
apply (*subst* (*2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *M q* . *if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap′* = *hsiMap*
unfolding *hsiMap-def hsiMap′-def*
using *mapping-of-map-of*
by *blast*
then have *distFun′* = *distFun*
unfolding *distFun-def distFun′-def* **by** *meson*

have ∗∗:*distFun M* = *get-HSI M*
proof
fix *q* **show** *distFun M q* = *get-HSI M q*
proof (*cases q* ∈ *states M*)
case *True*
then have *q* ∈ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *blast*
then show *?thesis*
unfolding *distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps*
using *True*
by *fastforce*

next
case *False*
then show *?thesis* **using** *distFun-def* **by** *auto*
qed
qed

show *?thesis*
unfolding $*$ $**$ ‹*distFun′* = *distFun*› *hsi-method-via-h-framework-def* **by** *simp*
qed

end

27 Implementations of the Partial-S-Method

theory *Partial-S-Method-Implementations*
imports *Intermediate-Frameworks*
begin

27.1 Using the H-Framework

fun *distance-at-most* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ *bool* **where**
distance-at-most M q1 q2 0 = (*q1* = *q2*) |
distance-at-most M q1 q2 (*Suc k*) = ((*q1* = *q2*) ∨ (∃ *x* ∈ *inputs M* . ∃ (*y*,*q1*′) ∈ *h M* (*q1*,*x*) . *distance-at-most M q1*′ *q2 k*))

definition *do-establish-convergence* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ (′*a*,′*b*,′*c*) *state-cover-assignment* ⇒ (′*a*,′*b*,′*c*) *transition* ⇒ (′*a*,′*b*,′*c*) *transition list* ⇒ *nat* ⇒ *bool* **where**
do-establish-convergence M V t X l = (*find* (λ *t*′ . *distance-at-most M* (*t-target t*) (*t-source t*′) *l*) *X* ≠ *None*)

definition *partial-s-method-via-h-framework* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (′*b*×′*c*) *prefix-tree* **where**
partial-s-method-via-h-framework = *h-framework-dynamic do-establish-convergence*

definition *partial-s-method-via-h-framework-lists* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ((′*b*×′*c*) × *bool*) *list list* **where**
partial-s-method-via-h-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*partial-s-method-via-h-framework M m completeInputTraces useInputHeuristic*))

lemma *partial-s-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*::*linorder*) *fsm*
fixes *M2* :: (′*e*,′*b*,′*c*) *fsm*
assumes *observable M1*

and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* $\cap$ *set* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*partial-s-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-completeness-and-finiteness*[*OF assms*]
unfolding *partial-s-method-via-h-framework-def*
by *blast+*

lemma *partial-s-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*partial-s-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-lists-completeness*[*OF assms*]
unfolding *partial-s-method-via-h-framework-lists-def h-framework-dynamic-lists-def partial-s-method-via-h-framework-def*
by *blast*

end

28 Implementations of the SPY-Method

theory *SPY-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

28.1 Using the H-Framework

definition *spy-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
spy-method-via-h-framework M m = *h-framework-static-with-simple-graph M* (λ

k q . get-HSI M q) m

definition *spy-method-via-h-framework-lists* :: (*'a::linorder*,*'b::linorder*,*'c::linorder*) *fsm* ⇒ *nat* ⇒ ((*'b*×*'c*) × *bool*) *list list* **where**
spy-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spy-method-via-h-framework M m*))

lemma *spy-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a::linorder*,*'b::linorder*,*'c::linorder*) *fsm*
fixes *M2* :: (*'e*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*spy-method-via-h-framework M1 m*)))
and *finite-tree* (*spy-method-via-h-framework M1 m*)
using *h-framework-static-with-simple-graph-completeness-and-finiteness*[*OF assms*, **where** *dist-fun*=(λ *k q . get-HSI M1 q*)]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
using *get-HSI-finite*
unfolding *spy-method-via-h-framework-def*
by *blast+*

lemma *spy-method-via-h-framework-lists-completeness* :
fixes *M1* :: (*'a::linorder*,*'b::linorder*,*'c::linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spy-method-via-h-framework-lists M1 m*)
using *h-framework-static-with-simple-graph-lists-completeness*[*OF assms*, **where** *dist-fun*=(λ *k q . get-HSI M1 q*), *OF - get-HSI-finite*]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *spy-method-via-h-framework-lists-def h-framework-static-with-simple-graph-lists-def spy-method-via-h-framework-def*
by *blast*

28.2 Using the SPY-Framework

definition *spy-method-via-spy-framework* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒ *nat* ⇒ ('b×'c) *prefix-tree* **where**
spy-method-via-spy-framework M m = *spy-framework-static-with-simple-graph M* (λ *k q* . *get-HSI M q*) *m*

lemma *spy-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: ('a::linorder,'b::linorder,'c::linorder) *fsm*
fixes *M2* :: ('d,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spy-method-via-spy-framework M1 m*)) = (*L M2* ∩ *set* (*spy-method-via-spy-framework M1 m*)))
and *finite-tree* (*spy-method-via-spy-framework M1 m*)
unfolding *spy-method-via-spy-framework-def*
using *spy-framework-static-with-simple-graph-completeness-and-finiteness*[*OF assms*, *of* (λ *k q* . *get-HSI M1 q*)]
using *get-HSI-distinguishes*[*OF assms*(*1*,*3*)]
using *get-HSI-finite*[*of M1*]
by *blast*+

definition *spy-method-via-spy-framework-lists* :: ('a::linorder,'b::linorder,'c::linorder) *fsm* ⇒ *nat* ⇒ (('b×'c) × *bool*) *list list* **where**
spy-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spy-method-via-spy-framework M m*))

lemma *spy-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('a::linorder,'b::linorder,'c::linorder) *fsm*
fixes *M2* :: ('d,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spy-method-via-spy-framework-lists M1 m*)
unfolding *spy-method-via-spy-framework-lists-def*
spy-method-via-spy-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial spy-method-via-spy-framework-completeness-and-finiteness*(*2*)[*OF assms*]]

by *blast*

28.3 Code Generation

lemma *spy-method-via-spy-framework-code*[*code*] :
spy-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provide*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);

hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *distHelper q q*′) (*filter*
((≠) *q*) (*states-as-list M*))))) (*states-as-list M*));
distFun = (λ *k q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap q*) *else*
get-HSI M q)
in spy-framework-static-with-simple-graph M distFun m)
(**is** *?f1* = *?f2*)
proof −

define *hsiMap*′ **where** *hsiMap*′ = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′
. *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* ((≠) *q*) (*states-as-list*
M))))) (*states-as-list M*))
define *distFun*′ **where** *distFun*′ = (λ *M q* . *if q* ∈ *states M then the* (*Mapping.lookup*
hsiMap′ *q*) *else get-HSI M q*)

have ∗: *?f2* = *spy-framework-static-with-simple-graph M* (λ *k q* . *distFun*′ *M q*)
m
unfolding *distFun*′*-def hsiMap*′*-def Let-def*
apply (*subst* (*2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of*
M])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ .
get-distinguishing-sequence-from-ofsm-tables M q q′) (*filter* ((≠) *q*) (*states-as-list*
M))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *M q* . *if q* ∈ *states M then the* (*hsiMap q*)
else get-HSI M q)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables*
M q q′) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap*′ = *hsiMap*

```
    unfolding hsiMap-def hsiMap′-def
    using mapping-of-map-of
    by blast
  then have distFun′ = distFun
    unfolding distFun-def distFun′-def by meson

  have **:distFun M = get-HSI M
  proof
    fix q show distFun M q = get-HSI M q
    proof (cases q ∈ states M)
      case True
      then have q ∈ list.set (states-as-list M)
        using states-as-list-set by blast
      then show ?thesis
        unfolding distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps
        using True
        by fastforce
    next
      case False
      then show ?thesis using distFun-def by auto
    qed
  qed

  show ?thesis
     unfolding * ** ‹distFun′ = distFun› spy-method-via-spy-framework-def by
simp
qed

lemma spy-method-via-h-framework-code[code] :
  spy-method-via-h-framework M m = (let
      tables = (compute-ofsm-tables M (size M − 1));
    distMap = mapping-of (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables-with-provid
tables M q1 q2))
                    (filter (λ qq . fst qq ≠ snd qq) (List.product (states-as-list M)
(states-as-list M))));
     distHelper = (λ q1 q2 . if q1 ∈ states M ∧ q2 ∈ states M ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M q1 q2);

      hsiMap = mapping-of (map (λ q . (q,from-list (map (λq′ . distHelper q q′)
(filter ((≠) q) (states-as-list M))))) (states-as-list M));
      distFun = (λ k q . if q ∈ states M then the (Mapping.lookup hsiMap q) else
get-HSI M q)
    in h-framework-static-with-simple-graph M distFun m)
(is ?f1 = ?f2)
proof −

  define hsiMap′ where hsiMap′ = mapping-of (map (λ q . (q,from-list (map (λq′
. get-distinguishing-sequence-from-ofsm-tables M q q′) (filter ((≠) q) (states-as-list
```

M))))) (*states-as-list M*))
define *distFun′* **where** *distFun′* = (λ *M q* . *if q* ∈ *states M then the* (*Mapping.lookup hsiMap′ q*) *else get-HSI M q*)

have ∗: *?f2* = *h-framework-static-with-simple-graph M* (λ *k q* . *distFun′ M q*) *m*
unfolding *distFun′-def hsiMap′-def Let-def*
apply (*subst* (*2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *M q* . *if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap′* = *hsiMap*
unfolding *hsiMap-def hsiMap′-def*
using *mapping-of-map-of*
by *blast*
then have *distFun′* = *distFun*
unfolding *distFun-def distFun′-def* **by** *meson*

have ∗∗:*distFun M* = *get-HSI M*
proof
fix *q* **show** *distFun M q* = *get-HSI M q*
proof (*cases q* ∈ *states M*)
case *True*
then have *q* ∈ *list.set* (*states-as-list M*)
using *states-as-list-set* **by** *blast*
then show *?thesis*
unfolding *distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps*
using *True*
by *fastforce*
next
case *False*
then show *?thesis* **using** *distFun-def* **by** *auto*
qed
qed

show *?thesis*
unfolding ∗ ∗∗ ‹*distFun′* = *distFun*› *spy-method-via-h-framework-def* **by** *simp*
qed

end

29 Implementations of the SPYH-Method

theory *SPYH-Method-Implementations*
imports *Intermediate-Frameworks*
begin

29.1 Using the H-Framework

definition *spyh-method-via-h-framework* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ ('b×'c) *prefix-tree* **where**
spyh-method-via-h-framework = *h-framework-dynamic* (λ *M V t X l* . *True*)

definition *spyh-method-via-h-framework-lists* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (('b×'c) × *bool*) *list list* **where**
spyh-method-via-h-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spyh-method-via-h-framework M m completeInputTraces useInputHeuristic*))

lemma *spyh-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm*
fixes *M2* :: ('e,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* ∩ *set* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*spyh-method-via-h-framework M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-completeness-and-finiteness*[*OF assms*]
unfolding *spyh-method-via-h-framework-def*
by *blast+*

lemma *spyh-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('a::*linorder*,'b::*linorder*,'c::*linorder*) *fsm*
fixes *M2* :: ('d,'b,'c) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*

and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*spyh-method-via-h-framework-lists M1 m completeInputTraces useInputHeuristic*)
using *h-framework-dynamic-lists-completeness*[*OF assms*]
unfolding *spyh-method-via-h-framework-lists-def h-framework-dynamic-lists-def spyh-method-via-h-framework-def*
by *blast*

29.2 Using the SPY-Framework

definition *spyh-method-via-spy-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic =
spy-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
sort-unverified-transitions-by-state-cover-length
(*establish-convergence-dynamic completeInputTraces useInputHeuristic* (*get-distinguishing-sequence-from-ofsm-tables M1*))
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m

lemma *spyh-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)) = (*L M2* $\cap$ *set* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)))
and *finite-tree* (*spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic*)
using *spy-framework-completeness-and-finiteness*[*OF assms*,
of get-state-cover-assignment

sort-unverified-transitions-by-state-cover-length
,
OF get-state-cover-assignment-is-state-cover-assignment
sort-unverified-transitions-by-state-cover-length-retains-set[*of - M1 get-state-cover-assignment*]
simple-cg-initial-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-insert-invar-with-conv[*OF assms*(*1*,*2*)]
simple-cg-merge-invar-with-conv[*OF assms*(*1*,*2*)]
handle-state-cover-dynamic-separates-state-cover[*OF get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)], *of completeInputTraces useInputHeuristic M2 simple-cg-initial simple-cg-insert simple-cg-lookup-with-conv*]
establish-convergence-dynamic-verifies-transition[*of M1* (*get-distinguishing-sequence-from-ofsm-tables M1*) *completeInputTraces useInputHeuristic M2 - - simple-cg-insert simple-cg-lookup-with-conv*, *OF get-distinguishing-sequence-from-ofsm-tabl assms*(*1*,*3*)]]
handle-io-pair-verifies-io-pair[*of completeInputTraces useInputHeuristic M1 M2 simple-cg-insert simple-cg-lookup-with-conv*]
]
unfolding *spyh-method-via-spy-framework-def*[*symmetric*]
by *presburger+*

definition *spyh-method-via-spy-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ *bool* ⇒ *bool* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
spyh-method-via-spy-framework-lists M m completeInputTraces useInputHeuristic = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*spyh-method-via-spy-framework M m completeInputTraces useInputHeuristic*))

lemma *spyh-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*initial M2*)) (*spyh-method-via-spy-framework-lists M1 m completeInputTraces useInputHeuristic*)
unfolding *spyh-method-via-spy-framework-lists-def*
spyh-method-via-spy-framework-completeness-and-finiteness(*1*)[*OF assms*, *of completeInputTraces useInputHeuristic*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial*

spyh-method-via-spy-framework-completeness-and-finiteness(*2*)[*OF assms*]]
by *blast*

29.3 Code Generation

lemma *spyh-method-via-spy-framework-code*[*code*] :
spyh-method-via-spy-framework M1 m completeInputTraces useInputHeuristic =
(*let*
tables = (*compute-ofsm-tables M1* (*size M1* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M1 q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M1*)
(*states-as-list M1*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M1* ∧ *q2* ∈ *states M1* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M1 q1 q2)
in
spy-framework M1
get-state-cover-assignment
(*handle-state-cover-dynamic completeInputTraces useInputHeuristic*
distHelper)
sort-unverified-transitions-by-state-cover-length
(*establish-convergence-dynamic completeInputTraces useInputHeuris-*
tic distHelper)
(*handle-io-pair completeInputTraces useInputHeuristic*)
simple-cg-initial
simple-cg-insert
simple-cg-lookup-with-conv
simple-cg-merge
m)
unfolding *spyh-method-via-spy-framework-def*
apply (*subst* (*1 2*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of*
M1])
unfolding *Let-def*
by *presburger*

end

30 Refined Code Generation for Test Suites

This theory provides alternative code equations for selected functions on test suites. Currently only Mapping via RBT is supported.

theory *Test-Suite-Representations-Refined*
imports *Test-Suite-Representations ../Prefix-Tree-Refined ../Util-Refined*
begin

lemma *test-suite-from-io-tree-refined*[*code*] :
fixes *M* :: (*'a*,*'b* :: *ccompare*, *'c* :: *ccompare*) *fsm*

and *m* :: (('*b*×'*c*), ('*b*×'*c*) *prefix-tree*) *mapping-rbt*
shows *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*))
= (*case ID CCOMPARE*(('*b* × '*c*)) *of*
None ⇒ *Code.abort* (*STR* ''*test-suite-from-io-tree RBT-set*: *ccompare* = *None*'') (λ- . *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*))) |
Some - ⇒ *MPT* (*Mapping.tabulate* (*map* (λ((*x*,*y*),*t*) . ((*x*,*y*),*h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*)) (λ ((*x*,*y*),*b*) . *case h-obs M q x y of None* ⇒ *Prefix-Tree.empty* | *Some q*′ ⇒ *test-suite-from-io-tree M q*′ (*case RBT-Mapping2.lookup m* (*x*,*y*) *of Some t*′ ⇒ *t*′))))
proof (*cases ID CCOMPARE*(('*b* × '*c*)))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)
then have *ID CCOMPARE*(('*b* × '*c*)) ≠ *None*
using *Some* **by** *auto*

have *distinct* (*map fst* (*RBT-Mapping2.entries m*))
apply *transfer*
using *linorder.distinct-entries*[*OF ID-ccompare*[*OF Some*]]
ord.is-rbt-rbt-sorted
Some
by *auto*

have ⋀ *a b* . (*RBT-Mapping2.lookup m a* = *Some b*) = ((*a*,*b*) ∈ *List.set* (*RBT-Mapping2.entries m*))
using *map-of-entries*[*OF* ‹*ID CCOMPARE*(('*b* × '*c*)) ≠ *None*›, *of m*]
using *map-of-eq-Some-iff*[*OF* ‹*distinct* (*map fst* (*RBT-Mapping2.entries m*))›]
by *auto*

let *?f2* = *Mapping.tabulate* (*map* (λ((*x*,*y*),*t*) . ((*x*,*y*),*h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*)) (λ ((*x*,*y*),*b*) . *case h-obs M q x y of None* ⇒ *Prefix-Tree.empty* | *Some q*′ ⇒ *test-suite-from-io-tree M q*′ (*case RBT-Mapping2.lookup m* (*x*,*y*) *of Some t*′ ⇒ *t*′))
let *?f1* = λ *xs* . λ ((*x*,*y*),*b*) . *case map-of xs* (*x*,*y*) *of*
None ⇒ *None* |
Some t ⇒ (*case h-obs M q x y of*
None ⇒ (*if b then None else Some empty*) |
Some q′ ⇒ (*if b then Some* (*test-suite-from-io-tree M q*′ *t*) *else None*))

have *Mapping.lookup ?f2* = *?f1* (*RBT-Mapping2.entries m*)
proof
fix *k*
show *Mapping.lookup ?f2 k* = *?f1* (*RBT-Mapping2.entries m*) *k*
proof −
obtain *x y b* **where** *k* = ((*x*,*y*),*b*)
by (*metis prod.exhaust-sel*)

show *?thesis* **proof** (*cases RBT-Mapping2.lookup m* (*x*,*y*))

case *None*

then have $((x,y),b) \notin$ *List.set* (*map* ($\lambda((x,y),t)$. $((x,y)$,*h-obs M q x y* $\neq$ *None*)) (*RBT-Mapping2.entries m*))
using ‹$\bigwedge$ *a b* . (*RBT-Mapping2.lookup m a* = *Some b*) = ((*a*,*b*) $\in$ *List.set* (*RBT-Mapping2.entries m*))›[*of* (*x*,*y*)]
by *auto*
then have *Mapping.lookup ?f2* ((*x*,*y*),*b*) = *None*
by (*metis* (*mono-tags*, *lifting*) *Mapping.lookup.rep-eq map-of-map-Pair-key tabulate.rep-eq*)
moreover have *?f1* (*RBT-Mapping2.entries m*) ((*x*,*y*),*b*) = *None*
using ‹$\bigwedge$ *a b* . (*RBT-Mapping2.lookup m a* = *Some b*) = ((*a*,*b*) $\in$ *List.set* (*RBT-Mapping2.entries m*))›[*of* (*x*,*y*)]
None
by (*metis* (*no-types*, *lifting*) *map-of-SomeD not-None-eq old.prod.case option.simps*(*4*))
ultimately show *?thesis*
unfolding ‹*k* = ((*x*,*y*),*b*)› **by** *simp*
next
case (*Some t′*)
then have ((*x*,*y*),*t′*) $\in$ *List.set* (*RBT-Mapping2.entries m*)
using ‹$\bigwedge$ *a b* . (*RBT-Mapping2.lookup m a* = *Some b*) = ((*a*,*b*) $\in$ *List.set* (*RBT-Mapping2.entries m*))› **by** *auto*

show *?thesis* **proof** (*cases h-obs M q x y*)
case *None*

show *?thesis* **proof** (*cases b*)
case *True*

then have $((x,y),b) \notin$ *List.set* (*map* ($\lambda((x,y),t)$. $((x,y)$,*h-obs M q x y* $\neq$ *None*)) (*RBT-Mapping2.entries m*))
using *None* **by** *auto*
then have *Mapping.lookup ?f2* ((*x*,*y*),*b*) = *None*
by (*metis* (*mono-tags*, *lifting*) *Mapping.lookup.rep-eq map-of-map-Pair-key tabulate.rep-eq*)
moreover have *?f1* (*RBT-Mapping2.entries m*) ((*x*,*y*),*b*) = *None*
using *Some None True*
using ‹((*x*, *y*), *t′*) $\in$ *list.set* (*RBT-Mapping2.entries m*)› ‹*distinct* (*map fst* (*RBT-Mapping2.entries m*))›
by *auto*
ultimately show *?thesis*
unfolding ‹*k* = ((*x*,*y*),*b*)› **by** *simp*
next
case *False*

then have ((*x*,*y*),*b*) $\in$ *List.set* (*map* ($\lambda((x,y),t)$. $((x,y)$,*h-obs M q x y* $\neq$ *None*)) (*RBT-Mapping2.entries m*))
using *None* ‹((*x*,*y*),*t′*) $\in$ *List.set* (*RBT-Mapping2.entries m*)› **by** *force*

then have *Mapping.lookup ?f2* ((*x*,*y*),*b*) = *Some empty*
proof −
have ⋀*p ps f*. (*p*::(′*b* × ′*c*) × *bool*) ∉ *list.set ps* ∨ *Mapping.lookup* (*Mapping.tabulate ps f*) *p* = *Some* (*f p*::((′*b* × ′*c*) × *bool*) *prefix-tree*)
by (*simp add*: *Mapping.lookup.rep-eq map-of-map-Pair-key tabulate.rep-eq*)
then show *?thesis*
using *None* ‹((*x*, *y*), *b*) ∈ *list.set* (*map* (λ((*x*, *y*), *t*). ((*x*, *y*), *h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*))› **by** *auto*
qed
moreover have *?f1* (*RBT-Mapping2.entries m*) ((*x*,*y*),*b*) = *Some empty*
using *Some None False*
using ‹((*x*, *y*), *t*′) ∈ *list.set* (*RBT-Mapping2.entries m*)›
using ‹*distinct* (*map fst* (*RBT-Mapping2.entries m*))› **by** *auto*
ultimately show *?thesis*
unfolding ‹*k* = ((*x*,*y*),*b*)› **by** *simp*
qed
next
case (*Some q*′)
show *?thesis* **proof** (*cases b*)
case *True*
then have ((*x*,*y*),*b*) ∈ *List.set* (*map* (λ((*x*,*y*),*t*) . ((*x*,*y*),*h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*))
using *Some* ‹((*x*,*y*),*t*′) ∈ *List.set* (*RBT-Mapping2.entries m*)› **by** *force*
then have *Mapping.lookup ?f2* ((*x*,*y*),*b*) = *Some* (*test-suite-from-io-tree M q*′ *t*′)
proof −
have ⋀*p ps f*. (*p*::(′*b* × ′*c*) × *bool*) ∉ *list.set ps* ∨ *Mapping.lookup* (*Mapping.tabulate ps f*) *p* = *Some* (*f p*::((′*b* × ′*c*) × *bool*) *prefix-tree*)
by (*simp add*: *Mapping.lookup.rep-eq map-of-map-Pair-key tabulate.rep-eq*)
then show *?thesis*
using *Some* ‹*RBT-Mapping2.lookup m* (*x*, *y*) = *Some t*′› ‹((*x*, *y*), *b*) ∈ *list.set* (*map* (λ((*x*, *y*), *t*). ((*x*, *y*), *h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*))› **by** *auto*
qed
moreover have *?f1* (*RBT-Mapping2.entries m*) ((*x*,*y*),*b*) = *Some* (*test-suite-from-io-tree M q*′ *t*′)
using *Some* ‹*RBT-Mapping2.lookup m* (*x*, *y*) = *Some t*′› *True*
using ‹((*x*, *y*), *t*′) ∈ *list.set* (*RBT-Mapping2.entries m*)›
using ‹*distinct* (*map fst* (*RBT-Mapping2.entries m*))› **by** *auto*
ultimately show *?thesis*
unfolding ‹*k* = ((*x*,*y*),*b*)› **by** *simp*
next
case *False*
then have ((*x*,*y*),*b*) ∉ *List.set* (*map* (λ((*x*,*y*),*t*) . ((*x*,*y*),*h-obs M q x y* ≠ *None*)) (*RBT-Mapping2.entries m*))
using *Some* **by** *auto*
then have *Mapping.lookup ?f2* ((*x*,*y*),*b*) = *None*

by (*metis* (*mono-tags*, *lifting*) *Mapping.lookup.rep-eq map-of-map-Pair-key tabulate.rep-eq*)
moreover have *?f1* (*RBT-Mapping2.entries m*) ((*x*,*y*),*b*) = *None*
using *Some* ‹*RBT-Mapping2.lookup m* (*x*, *y*) = *Some t′*› *False*
using ‹((*x*, *y*), *t′*) ∈ *list.set* (*RBT-Mapping2.entries m*)› ‹*distinct* (*map fst* (*RBT-Mapping2.entries m*))›
by *auto*
ultimately show *?thesis*
unfolding ‹*k* = ((*x*,*y*),*b*)› **by** *simp*
qed
qed
qed
qed
qed

obtain *m′* **where** *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*)) = *MPT m′*
by (*simp add*: *test-suite-from-io-tree-MPT*)
then have *test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*)) = *PT* (*Mapping.lookup m′*)
using *MPT-def* **by** *simp*
have *Mapping.lookup m′* = (λ ((*x*,*y*),*b*) . *case RBT-Mapping2.lookup m* (*x*,*y*) *of*
None ⇒ *None* |
Some t ⇒ (*case h-obs M q x y of*
None ⇒ (*if b then None else Some empty*) |
Some q′ ⇒ (*if b then Some* (*test-suite-from-io-tree M q′ t*) *else None*)))
proof −
have *test-suite-from-io-tree M q* (*PT* (*Mapping.lookup* (*RBT-Mapping m*))) = *PT* (*Mapping.lookup m′*)
by (*metis MPT-def* ‹*test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*)) = *PT* (*Mapping.lookup m′*)›)
then have *Mapping.lookup m′* = (λ((*b*, *c*), *ba*). *case Mapping.lookup* (*RBT-Mapping m*) (*b*, *c*) *of None* ⇒ *None* | *Some p* ⇒ (*case h-obs M q b c of None* ⇒ *if ba then None else Some Prefix-Tree.empty* | *Some a* ⇒ *if ba then Some* (*test-suite-from-io-tree M a p*) *else None*))
by *auto*
then show *?thesis*
by (*simp add*: *Mapping.lookup.abs-eq*)
qed
then have *Mapping.lookup m′* = *?f1* (*RBT-Mapping2.entries m*)
unfolding *map-of-entries*[*OF* ‹*ID CCOMPARE*(('*b* × '*c*)) ≠ *None*›, *of m*] **by** *simp*
then have *Mapping.lookup ?f2* = *Mapping.lookup m′*
using ‹*Mapping.lookup ?f2* = *?f1* (*RBT-Mapping2.entries m*)› **by** *simp*
then show *?thesis*
using *Some* **unfolding** ‹*test-suite-from-io-tree M q* (*MPT* (*RBT-Mapping m*)) = *MPT m′*›
by (*simp add*: *MPT-def*)
qed

end

31 Implementations of the W-Method

theory *W-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

31.1 Using the H-Framework

definition *w-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
w-method-via-h-framework M m = *h-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set M*) *m*

definition *w-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
w-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-h-framework M m*))

lemma *w-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*w-method-via-h-framework M1 m*)) = (*L M2* ∩ *set* (*w-method-via-h-framework M1 m*)))
and *finite-tree* (*w-method-via-h-framework M1 m*)
using *h-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*,
where *dist-fun*=(λ *k q* . *distinguishing-set M1*)]
using *distinguishing-set-distinguishes*[*OF assms*(*1*,*3*)]
using *distinguishing-set-finite*
unfolding *w-method-via-h-framework-def*
by *blast*+

lemma *w-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*

and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-h-framework-lists M1 m*)
using *h-framework-static-with-empty-graph-lists-completeness*[*OF assms*, **where** *dist-fun*=(λ *k q* . *distinguishing-set M1*), *OF - distinguishing-set-finite*]
using *distinguishing-set-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *w-method-via-h-framework-lists-def h-framework-static-with-empty-graph-lists-def w-method-via-h-framework-def*
by *blast*

definition *w-method-via-h-framework-2* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
w-method-via-h-framework-2 M m = *h-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set-reduced M*) *m*

definition *w-method-via-h-framework-2-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ (('*b*$\times$'*c*) $\times$ *bool*) *list list* **where**
w-method-via-h-framework-2-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*w-method-via-h-framework-2 M m*))

lemma *w-method-via-h-framework-2-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*w-method-via-h-framework-2 M1 m*)) = (*L M2* $\cap$ *set* (*w-method-via-h-framework-2 M1 m*)))
and *finite-tree* (*w-method-via-h-framework-2 M1 m*)
using *h-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*, **where** *dist-fun*=(λ *k q* . *distinguishing-set-reduced M1*)]
using *distinguishing-set-reduced-distinguishes*[*OF assms*(*1*,*3*)]
using *distinguishing-set-reduced-finite*
unfolding *w-method-via-h-framework-2-def*
by *blast+*

lemma *w-method-via-h-framework-lists-2-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*

fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-h-framework-2-lists M1 m*)
using *h-framework-static-with-empty-graph-lists-completeness*[*OF assms*, **where** *dist-fun*=(λ *k q* . *distinguishing-set-reduced M1*), *OF - distinguishing-set-reduced-finite*]
using *distinguishing-set-reduced-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *w-method-via-h-framework-2-lists-def h-framework-static-with-empty-graph-lists-def w-method-via-h-framework-2-def*
by *blast*

31.2 Using the SPY-Framework

definition *w-method-via-spy-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ (*'b*$\times$*'c*) *prefix-tree* **where**
w-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (λ *k q* . *distinguishing-set M*) *m*

lemma *w-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*w-method-via-spy-framework M1 m*)) = (*L M2* $\cap$ *set* (*w-method-via-spy-framework M1 m*)))
and *finite-tree* (*w-method-via-spy-framework M1 m*)
unfolding *w-method-via-spy-framework-def*
using *spy-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*, *of* (λ *k q* . *distinguishing-set M1*)]
using *distinguishing-set-distinguishes*[*OF assms*(*1*,*3*)]
using *distinguishing-set-finite*[*of M1*]
by (*metis IntI*)+

definition *w-method-via-spy-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ((*'b*$\times$*'c*) $\times$ *bool*) *list list* **where**
w-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree*

(*test-suite-from-io-tree M* (*initial M*) (*w-method-via-spy-framework M m*))

lemma *w-method-via-spy-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*w-method-via-spy-framework-lists M1 m*)
unfolding *w-method-via-spy-framework-lists-def*
w-method-via-spy-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial w-method-via-spy-framework-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

31.3 Using the Pair-Framework

definition *w-method-via-pair-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ('*b*$\times$'*c*) *prefix-tree* **where**
w-method-via-pair-framework M m = *pair-framework-h-components M m add-distinguishing-set*

lemma *w-method-via-pair-framework-completeness-and-finiteness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) $\longleftrightarrow$ (*L M* $\cap$ *set* (*w-method-via-pair-framework M m*) = *L I* $\cap$ *set* (*w-method-via-pair-framework M m*))
and *finite-tree* (*w-method-via-pair-framework M m*)
using *pair-framework-h-components-completeness-and-finiteness*[*OF assms*(*1*,*2*,*3*,*5*,*4*,*6*,*7*), **where** *get-separating-traces*=*add-distinguishing-set*, *OF add-distinguishing-set-distinguishes*[*OF assms*(*1*,*3*)] *add-distinguishing-set-finite*]
using *get-distinguishing-sequence-from-ofsm-tables-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *w-method-via-pair-framework-def*[*symmetric*]
by *blast+*

definition *w-method-via-pair-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ (('*b*$\times$'*c*) $\times$ *bool*) *list list* **where**
w-method-via-pair-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree*

(*test-suite-from-io-tree M* (*initial M*) (*w-method-via-pair-framework M m*))

lemma *w-method-implementation-lists-completeness* :
assumes *observable M*
and *observable I*
and *minimal M*
and *size I* $\leq$ *m*
and *m* $\geq$ *size-r M*
and *inputs I* = *inputs M*
and *outputs I* = *outputs M*
shows (*L M* = *L I*) $\longleftrightarrow$ *list-all* (*passes-test-case I* (*initial I*)) (*w-method-via-pair-framework-lists M m*)
unfolding *w-method-via-pair-framework-lists-def*
w-method-via-pair-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial w-method-via-pair-framework-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

31.4 Code Generation

lemma *w-method-via-pair-framework-code*[*code*] :
w-method-via-pair-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
distFun = (λ *M x t* . *distSet*)
in pair-framework-h-components M m distFun)
unfolding *w-method-via-pair-framework-def pair-framework-h-components-def pair-framework-def*
unfolding *add-distinguishing-set.simps*
unfolding *distinguishing-set.simps*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

lemma *w-method-via-spy-framework-code*[*code*] :
w-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*

(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
distFun = (λ *k q* . *distSet*)
in spy-framework-static-with-empty-graph M distFun m)
unfolding *w-method-via-spy-framework-def*
unfolding *add-distinguishing-set.simps*
unfolding *distinguishing-set.simps*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

lemma *w-method-via-h-framework-code*[*code*] :
w-method-via-h-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
distFun = (λ *k q* . *distSet*)
in h-framework-static-with-empty-graph M distFun m)
unfolding *w-method-via-h-framework-def*
unfolding *distinguishing-set.simps*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

lemma *w-method-via-h-framework-2-code*[*code*] :
w-method-via-h-framework-2 M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid*
tables M q1 q2))
(*filter* (λ *qq* . *fst qq* ≠ *snd qq*) (*List.product* (*states-as-list M*)
(*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* ∈ *states M* ∧ *q2* ∈ *states M* ∧ *q1* ≠ *q2 then the*
(*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables*
M q1 q2);
pairs = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*));
handlePair = (λ *W* (*q*,*q*′) . *if contains-distinguishing-trace M W q q*′
then W
else insert W (*distHelper q q*′));
distSet = *foldl handlePair empty pairs*;

distFun = (λ *k q* . *distSet*)
in h-framework-static-with-empty-graph M distFun m)
unfolding *w-method-via-h-framework-2-def*
unfolding *distinguishing-set-reduced.simps*
apply (*subst get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

end

32 Implementations of the Wp-Method

theory *Wp-Method-Implementations*
imports *Intermediate-Frameworks Pair-Framework ../Distinguishability Test-Suite-Representations ../OFSM-Tables-Refined HOL−Library.List-Lexorder*
begin

32.1 Distinguishing Sets

fun *add-distinguishing-set-or-state-identifier* :: *nat* ⇒ (′*a* :: *linorder*, ′*b* :: *linorder*, ′*c* :: *linorder*) *fsm* ⇒ ((′*b* × ′*c*) *list* × ′*a*) × (′*b* × ′*c*) *list* × ′*a* ⇒ (′*b* × ′*c*) *prefix-tree* ⇒ (′*b*×′*c*) *prefix-tree* **where**
add-distinguishing-set-or-state-identifier k M ((*io1*,*q1*),(*io2*,*q2*)) *t* = (*if length io1* = *k* ∨ *length io2* = *k*
then insert empty (*get-distinguishing-sequence-from-ofsm-tables M q1 q2*)
else distinguishing-set M)

lemma *add-distinguishing-set-or-state-identifier-distinguishes* :
assumes *observable M*
and *minimal M*
and α ∈ *L M*
and β ∈ *L M*
and *after-initial M* α ≠ *after-initial M* β
shows ∃ *io* ∈ *set* (*add-distinguishing-set-or-state-identifier k M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) ∪ (*set* (*after t* α) ∩ *set* (*after t* β)) . *distinguishes M* (*after-initial M* α) (*after-initial M* β) *io*
proof (*cases length* α = *k* ∨ *length* β = *k*)
case *False*
then show *?thesis*
using *distinguishing-set-distinguishes*[*OF assms*(*1*,*2*) *after-is-state*[*OF assms*(*1*,*3*)] *after-is-state*[*OF assms*(*1*,*4*)] *assms*(*5*)]
by *auto*
next
case *True*
then have *set* ((*add-distinguishing-set-or-state-identifier k*) *M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = *set* (*insert empty* (*get-distinguishing-sequence-from-ofsm-tables M* (*after-initial M* α) (*after-initial M* β)))
by *auto*
then have *get-distinguishing-sequence-from-ofsm-tables M* (*after-initial M* α)

(*after-initial M* β) $\in$ *set* ((*add-distinguishing-set-or-state-identifier k*) *M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) $\cup$ (*set* (*after t* α) $\cap$ *set* (*after t* β))
  unfolding *insert-set* **by** *auto*
 then show *?thesis*
   by (*meson after-is-state assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *get-distinguishing-sequence-from-ofsm-tables-distinguishes*)
qed

lemma *add-distinguishing-set-or-state-identifier-finite* :
 finite-tree ((*add-distinguishing-set-or-state-identifier k*) *M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*)
proof (*cases length* α = *k* $\vee$ *length* β = *k*)
 case *False*
 then show *?thesis*
  unfolding *add-distinguishing-set.simps distinguishing-set.simps Let-def*
  using *from-list-finite-tree*
  by *simp*
next
 case *True*
 then have ((*add-distinguishing-set-or-state-identifier k*) *M* ((α,*after-initial M* α),(β,*after-initial M* β)) *t*) = (*insert empty* (*get-distinguishing-sequence-from-ofsm-tables M* (*after-initial M* α) (*after-initial M* β)))
  by *auto*
 then show *?thesis*
  using *insert-finite-tree*[*OF empty-finite-tree*] **by** *metis*
qed

fun *distinguishing-set-or-state-identifier* :: *nat* $\Rightarrow$ (*'a* :: *linorder*, *'b* :: *linorder*, *'c* :: *linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ *'a* $\Rightarrow$ (*'b*$\times$*'c*) *prefix-tree* **where**
 distinguishing-set-or-state-identifier l M k q = (*if k* = *l*
  then get-HSI M q
  else distinguishing-set M)

lemma *get-HSI-subset* :
 assumes *observable M*
 and *minimal M*
 and *q* $\in$ *states M*
shows *set* (*get-HSI M q*) $\subseteq$ *set* (*distinguishing-set M*)
proof
 fix *io* **assume** *io* $\in$ *set* (*get-HSI M q*)

 show *io* $\in$ *set* (*distinguishing-set M*)
 proof (*cases io* = [])
  case *True*
  then show *?thesis* **by** *auto*
 next

case *False*

then obtain *io′* **where** ∗:*io@io′* ∈ *list.set* (*map* (*get-distinguishing-sequence-from-ofsm-tables M q*) (*filter* ((≠) *q*) (*states-as-list M*)))
using ‹*io* ∈ *set* (*get-HSI M q*)›
unfolding *get-HSI.simps from-list-set*
by *blast*
obtain *q′* **where** *q* ≠ *q′* **and** *q′* ∈ *states M* **and** *io@io′* = *get-distinguishing-sequence-from-ofsm-tables M q q′*
using *states-as-list-set*[*of M*] *filter-map-elem*[*OF* ∗]
by *blast*

have (*q*,*q′*) ∈ *list.set* (*filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*)))
∨ (*q′*,*q*) ∈ *list.set* (*filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*)))
using *list-ordered-pairs-set-containment*[*of q states-as-list M q′*] ‹*q* ∈ *states M*› ‹*q′* ∈ *states M*› ‹*q* ≠ *q′*›
unfolding *states-as-list-set*
by *force*
moreover define *pairs* **where** *pairs*: *pairs* = *filter* (λ (*x*,*y*) . *x* ≠ *y*) (*list-ordered-pairs* (*states-as-list M*))
ultimately have (*q*,*q′*) ∈ *list.set pairs* ∨ (*q′*,*q*) ∈ *list.set pairs*
by *auto*
then have *get-distinguishing-sequence-from-ofsm-tables M q q′* ∈ *list.set* (*map* (*case-prod* (*get-distinguishing-sequence-from-ofsm-tables M*)) *pairs*)
using *get-distinguishing-sequence-from-ofsm-tables-sym*[*OF assms* ‹*q′* ∈ *states M*› ‹*q* ≠ *q′*›, *symmetric*]
by (*metis case-prod-conv map-set*)
then have *io@io′* ∈ *set* (*distinguishing-set M*)
unfolding ‹*io@io′* = *get-distinguishing-sequence-from-ofsm-tables M q q′*› *distinguishing-set.simps Let-def pairs*
using *from-list-set-elem*
by *blast*
then show *?thesis*
using *set-prefix* **by** *metis*
qed
qed

lemma *distinguishing-set-or-state-identifier-distinguishes* :
assumes *observable M*
and *minimal M*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows ∃ *io* . ∀ *k1 k2* . *io* ∈ *set* (*distinguishing-set-or-state-identifier l M k1 q1*) ∩ *set* (*distinguishing-set-or-state-identifier l M k2 q2*) ∧ *distinguishes M q1 q2 io*
using *get-HSI-distinguishes*[*OF assms*]
using *distinguishing-set-distinguishes*[*OF assms*]
using *get-HSI-subset*[*OF assms*(*1*,*2*,*3*)]
using *get-HSI-subset*[*OF assms*(*1*,*2*,*4*)]

unfolding *distinguishing-set-or-state-identifier.simps*
by *auto*

lemma *distinguishing-set-or-state-identifier-finite* :
finite-tree (*distinguishing-set-or-state-identifier l M k q*)
using *get-HSI-finite*[*of M q*]
using *distinguishing-set-finite*[*of M*]
unfolding *distinguishing-set-or-state-identifier.simps*
by (*cases k = l*; *force*)

32.2 Using the H-Framework

definition *wp-method-via-h-framework* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
⇒ *nat* ⇒ ('*b*×'*c*) *prefix-tree* **where**
wp-method-via-h-framework M m = *h-framework-static-with-empty-graph M* (*distinguishing-set-or-state-identi*
(*Suc* (*m* − *size-r M*)) *M*) *m*

definition *wp-method-via-h-framework-lists* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*)
fsm ⇒ *nat* ⇒ (('*b*×'*c*) × *bool*) *list list* **where**
wp-method-via-h-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree*
(*test-suite-from-io-tree M* (*initial M*) (*wp-method-via-h-framework M m*))

lemma *wp-method-via-h-framework-completeness-and-finiteness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*e*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* ≤ *m*
and *size M2* ≤ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*wp-method-via-h-framework M1 m*))
= (*L M2* ∩ *set* (*wp-method-via-h-framework M1 m*)))
and *finite-tree* (*wp-method-via-h-framework M1 m*)
using *h-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*,
where *dist-fun*=*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M1*)) *M1*]
using *distinguishing-set-or-state-identifier-distinguishes*[*OF assms*(*1*,*3*)]
using *distinguishing-set-or-state-identifier-finite*
unfolding *wp-method-via-h-framework-def*
by *blast*+

lemma *wp-method-via-h-framework-lists-completeness* :
fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*

and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*wp-method-via-h-framework-lists M1 m*)
using *h-framework-static-with-empty-graph-lists-completeness*[*OF assms*, **where** *dist-fun*=*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M1*)) *M1*, *OF - distinguishing-set-or-state-identifier-finite*]
using *distinguishing-set-or-state-identifier-distinguishes*[*OF assms*(*1*,*3*)]
unfolding *wp-method-via-h-framework-lists-def h-framework-static-with-empty-graph-lists-def wp-method-via-h-framework-def*
by *blast*

32.3 Using the SPY-Framework

definition *wp-method-via-spy-framework* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ (*'b*×*'c*) *prefix-tree* **where**
wp-method-via-spy-framework M m = *spy-framework-static-with-empty-graph M* (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *m*

lemma *wp-method-via-spy-framework-completeness-and-finiteness* :
fixes *M1* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm*
fixes *M2* :: (*'d*,*'b*,*'c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*wp-method-via-spy-framework M1 m*)) = (*L M2* $\cap$ *set* (*wp-method-via-spy-framework M1 m*)))
and *finite-tree* (*wp-method-via-spy-framework M1 m*)
unfolding *wp-method-via-spy-framework-def*
using *spy-framework-static-with-empty-graph-completeness-and-finiteness*[*OF assms*, *of distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M1*)) *M1*]
using *distinguishing-set-or-state-identifier-distinguishes*[*OF assms*(*1*,*3*)]
using *distinguishing-set-or-state-identifier-finite*
by *metis+*

definition *wp-method-via-spy-framework-lists* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*::*linorder*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ ((*'b*×*'c*) × *bool*) *list list* **where**
wp-method-via-spy-framework-lists M m = *sorted-list-of-maximal-sequences-in-tree* (*test-suite-from-io-tree M* (*initial M*) (*wp-method-via-spy-framework M m*))

lemma *wp-method-via-spy-framework-lists-completeness* :

fixes *M1* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*::*linorder*) *fsm*
fixes *M2* :: ('*d*,'*b*,'*c*) *fsm*
assumes *observable M1*
and *observable M2*
and *minimal M1*
and *minimal M2*
and *size-r M1* $\leq$ *m*
and *size M2* $\leq$ *m*
and *inputs M2* = *inputs M1*
and *outputs M2* = *outputs M1*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*initial M2*)) (*wp-method-via-spy-framework-lists M1 m*)
unfolding *wp-method-via-spy-framework-lists-def*
wp-method-via-spy-framework-completeness-and-finiteness(*1*)[*OF assms*]
passes-test-cases-from-io-tree[*OF assms*(*1*,*2*) *fsm-initial fsm-initial wp-method-via-spy-framework-completeness-and-finiteness*(*2*)[*OF assms*]]
by *blast*

32.4 Code Generation

lemma *wp-method-via-spy-framework-code*[*code*] :
wp-method-via-spy-framework M m = (*let*
tables = (*compute-ofsm-tables M* (*size M* − *1*));
distMap = *mapping-of* (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*), *get-distinguishing-sequence-from-ofsm-tables-with-provid tables M q1 q2*))
(*filter* (λ *qq* . *fst qq* $\neq$ *snd qq*) (*List.product* (*states-as-list M*) (*states-as-list M*))));
distHelper = (λ *q1 q2* . *if q1* $\in$ *states M* $\wedge$ *q2* $\in$ *states M* $\wedge$ *q1* $\neq$ *q2 then the* (*Mapping.lookup distMap* (*q1*,*q2*)) *else get-distinguishing-sequence-from-ofsm-tables M q1 q2*);
pairs = *filter* (λ (*x*,*y*) . *x* $\neq$ *y*) (*list-ordered-pairs* (*states-as-list M*));
distSet = *from-list* (*map* (*case-prod distHelper*) *pairs*);
hsiMap = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *distHelper q q*′) (*filter* (($\neq$) *q*) (*states-as-list M*))))) (*states-as-list M*));
l = (*Suc* (*m* − *size-r M*));
distFun = (λ *k q* . *if k* = *l*
then (*if q* $\in$ *states M then the* (*Mapping.lookup hsiMap q*) *else get-HSI M q*)
else distSet)
in spy-framework-static-with-empty-graph M distFun m)
(**is** *?f1* = *?f2*)
proof −
define *hsiMap*′ **where** *hsiMap*′ = *mapping-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q*′ . *get-distinguishing-sequence-from-ofsm-tables M q q*′) (*filter* (($\neq$) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun*′ **where** *distFun*′ = (λ *k q* . *if k* = (*Suc* (*m* − *size-r M*))
then (*if q* $\in$ *states M then the* (*Mapping.lookup hsiMap*′ *q*) *else get-HSI M q*)
else distinguishing-set M)

have ∗: *?f2* = *spy-framework-static-with-empty-graph M distFun′ m*
unfolding *distFun′-def hsiMap′-def distinguishing-set.simps Let-def*
apply (*subst* (*3 4*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *k q* . *if k* = (*Suc* (*m* − *size-r M*))
then (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)
else distinguishing-set M)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap′* = *hsiMap*
unfolding *hsiMap-def hsiMap′-def*
using *mapping-of-map-of*
by *blast*
then have *distFun′* = *distFun*
unfolding *distFun-def distFun′-def* **by** *meson*

have ∗∗:*distFun′* = (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*)
proof
fix *k* **show** *distFun′ k* = (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *k*
proof (*cases k* = (*Suc* (*m* − *size-r M*)))
case *False*
then show *?thesis*
unfolding *distFun-def distinguishing-set-or-state-identifier.simps* ‹*distFun′* = *distFun*› **by** *auto*
next
case *True*
then have *distFun k* = (λ *q* . (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*))
and (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *k* = (λ *q* . *get-HSI M q*)
unfolding *distFun-def distinguishing-set-or-state-identifier.simps* **by** *auto*
moreover have (λ *q* . (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)) = (λ *q* . *get-HSI M q*)
proof
fix *q* **show** (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*) = *get-HSI M q*
proof (*cases q* ∈ *states M*)
case *True*

```
        then have q ∈ list.set (states-as-list M)
          using states-as-list-set by blast
        then show ?thesis
          unfolding distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps
          using True
          by fastforce
      next
        case False
        then show ?thesis using distFun-def by auto
      qed
    qed
    ultimately show ?thesis unfolding ‹distFun′ = distFun› by simp
  qed
qed

 show ?thesis
  unfolding ∗ ∗∗ wp-method-via-spy-framework-def by simp
qed

lemma wp-method-via-h-framework-code[code] :
 wp-method-via-h-framework M m = (let
    tables = (compute-ofsm-tables M (size M − 1));
   distMap = mapping-of (map (λ (q1,q2) . ((q1,q2), get-distinguishing-sequence-from-ofsm-tables-with-provid
tables M q1 q2))
                    (filter (λ qq . fst qq ≠ snd qq) (List.product (states-as-list M)
(states-as-list M))));
    distHelper = (λ q1 q2 . if q1 ∈ states M ∧ q2 ∈ states M ∧ q1 ≠ q2 then the
(Mapping.lookup distMap (q1,q2)) else get-distinguishing-sequence-from-ofsm-tables
M q1 q2);
    pairs = filter (λ (x,y) . x ≠ y) (list-ordered-pairs (states-as-list M));
    distSet = from-list (map (case-prod distHelper) pairs);
     hsiMap = mapping-of (map (λ q . (q,from-list (map (λq′ . distHelper q q′)
(filter ((≠) q) (states-as-list M))))) (states-as-list M));
    l = (Suc (m − size-r M));
    distFun = (λ k q . if k = l
                then (if q ∈ states M then the (Mapping.lookup hsiMap q) else
get-HSI M q)
               else distSet)
   in h-framework-static-with-empty-graph M distFun m)
(is ?f1 = ?f2)
proof −
 define hsiMap′ where hsiMap′ = mapping-of (map (λ q . (q,from-list (map (λq′
. get-distinguishing-sequence-from-ofsm-tables M q q′) (filter ((≠) q) (states-as-list
M))))) (states-as-list M))
 define distFun′ where distFun′ = (λ k q . if k = (Suc (m − size-r M))
                then (if q ∈ states M then the (Mapping.lookup hsiMap′ q) else
get-HSI M q)
               else distinguishing-set M)
 have ∗: ?f2 = h-framework-static-with-empty-graph M distFun′ m
```

unfolding *distFun′-def hsiMap′-def distinguishing-set.simps Let-def*
apply (*subst* (*3 4*) *get-distinguishing-sequence-from-ofsm-tables-precomputed*[*of M*])
unfolding *Let-def*
by *presburger*

define *hsiMap* **where** *hsiMap* = *map-of* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*))
define *distFun* **where** *distFun* = (λ *k q* . *if k* = (*Suc* (*m* − *size-r M*))
then (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)
else distinguishing-set M)

have *distinct* (*map fst* (*map* (λ *q* . (*q*,*from-list* (*map* (λ*q′* . *get-distinguishing-sequence-from-ofsm-tables M q q′*) (*filter* ((≠) *q*) (*states-as-list M*))))) (*states-as-list M*)))
using *states-as-list-distinct*
by (*metis map-pair-fst*)
then have *Mapping.lookup hsiMap′* = *hsiMap*
unfolding *hsiMap-def hsiMap′-def*
using *mapping-of-map-of*
by *blast*
then have *distFun′* = *distFun*
unfolding *distFun-def distFun′-def* **by** *meson*

have ∗∗:*distFun′* = (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*)
proof
fix *k* **show** *distFun′ k* = (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *k*
proof (*cases k* = (*Suc* (*m* − *size-r M*)))
case *False*
then show *?thesis*
unfolding *distFun-def distinguishing-set-or-state-identifier.simps* ‹*distFun′* = *distFun*› **by** *auto*
next
case *True*
then have *distFun k* = (λ *q* . (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*))
and (*distinguishing-set-or-state-identifier* (*Suc* (*m* − *size-r M*)) *M*) *k* = (λ *q* . *get-HSI M q*)
unfolding *distFun-def distinguishing-set-or-state-identifier.simps* **by** *auto*
moreover have (λ *q* . (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*)) = (λ *q* . *get-HSI M q*)
proof
fix *q* **show** (*if q* ∈ *states M then the* (*hsiMap q*) *else get-HSI M q*) = *get-HSI M q*
proof (*cases q* ∈ *states M*)
case *True*
then have *q* ∈ *list.set* (*states-as-list M*)

```
        using states-as-list-set by blast
      then show ?thesis
        unfolding distFun-def hsiMap-def map-of-map-pair-entry get-HSI.simps
        using True
        by fastforce
    next
      case False
      then show ?thesis using distFun-def by auto
    qed
  qed
  ultimately show ?thesis unfolding ‹distFun' = distFun› by simp
 qed
qed

 show ?thesis
  unfolding * ** wp-method-via-h-framework-def by simp
qed


end
```

33 Backwards Reachability Analysis

This theory introduces function *select-inputs* which is used for the calculation of both state preambles and state separators.

```
theory Backwards-Reachability-Analysis
imports ../FSM
begin
```

Function *select-inputs* calculates an associative list that maps states to a single input each such that the FSM induced by this input selection is acyclic, single input and whose only deadlock states (if any) are contained in *stateSet*. The following parameters are used: 1) transition function f (typically $(h\ M)$ for some FSM M) 2) a source state *q0* (selection terminates as soon as this states is assigned some input) 3) a list of inputs that may be assigned to states 4) a list of states not yet taken (these are considered when searching for the next possible assignment) 5) a set *stateSet* of all states that already have an input assigned to them by m 6) an associative list m containing previously chosen assignments

```
function select-inputs :: (('a × 'b) ⇒ ('c × 'a) set) ⇒ 'a ⇒ 'b list ⇒ 'a list ⇒ 'a
set ⇒ ('a × 'b) list ⇒ ('a × 'b) list where
  select-inputs f q0 inputList [] stateSet m = (case find (λ x . f (q0,x) ≠ {} ∧ (∀
(y,q'') ∈ f (q0,x) . (q'' ∈ stateSet))) inputList of
      Some x ⇒ m@[(q0,x)] |
      None   ⇒ m) |
  select-inputs f q0 inputList (n#nL) stateSet m =
```

(*case find* (λ *x* . *f* (*q0*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q0*,*x*) . (*q''* $\in$ *stateSet*))) *inputList of*
Some x $\Rightarrow$ *m*@[(*q0*,*x*)] |
None $\Rightarrow$ (*case find-remove-2* (λ *q' x* . *f* (*q'*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q'*,*x*) . (*q''* $\in$ *stateSet*))) (*n*#*nL*) *inputList*
of None $\Rightarrow$ *m* |
Some (*q'*,*x*,*stateList'*) $\Rightarrow$ *select-inputs f q0 inputList stateList'* (*insert q' stateSet*) (*m*@[(*q'*,*x*)])))
by *pat-completeness auto*
termination
proof −
{
fix *f* :: (('*a* × '*b*) $\Rightarrow$ ('*c* × '*a*) *set*)
fix *q0* :: '*a*
fix *inputList* :: '*b list*
fix *n* :: '*a*
fix *nL* :: '*a list*
fix *stateSet* :: '*a set*
fix *m* :: ('*a* × '*b*) *list*
fix *qynL' q ynL' x nL'*
assume *find* (λ*x*. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *stateSet*)) *inputList* = *None*
and *find-remove-2* (λ*q' x*. *f* (*q'*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, *q''*)$\in$*f* (*q'*, *x*). *q''* $\in$ *stateSet*)) (*n* # *nL*) *inputList* = *Some qynL'*
and (*q*, *ynL'*) = *qynL'*
and (*x*, *nL'*) = *ynL'*

then have $*$: *find-remove-2* (λ*q' x*. *f* (*q'*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, *q''*)$\in$*f* (*q'*, *x*). *q''* $\in$ *stateSet*)) (*n* # *nL*) *inputList* = *Some* (*q*,*x*,*nL'*)
by *auto*

have *q* $\in$ *set* (*n* # *nL*)
and *nL'* = *remove1 q* (*n* # *nL*)
using *find-remove-2-set*(*2*,*6*)[*OF* $*$] **by** *simp+*

then have *length nL'* < *length* (*n* # *nL*)
using *remove1-length* **by** *metis*

then have ((*f*, *q0*, *inputList*, *nL'*, *insert q stateSet*, *m* @ [(*q*, *x*)]), *f*, *q0*, *inputList*, *n* # *nL*, *stateSet*, *m*)
$\in$ *measure* (λ(*f*, *q0*, *iL*, *nL*, *nS*, *m*). *length nL*)
by *auto*
}
then show *?thesis*
by (*relation measure* (λ (*f*,*q0*,*iL*,*nL*,*nS*,*m*) . *length nL*); *simp*)
qed

```
lemma select-inputs-length :
  length (select-inputs f q0 inputList stateList stateSet m) ≤ (length m) + Suc
(length stateList)
proof (induction length stateList arbitrary: stateList stateSet m)
  case 0
  then show ?case
    by (cases find (λx. f (q0, x) ≠ {} ∧ (∀(y, q'')∈f (q0, x). q'' ∈ stateSet))
inputList; auto)
next
  case (Suc k)
  then obtain n nL where stateList = n # nL
    by (meson Suc-length-conv)

  show ?case
  proof (cases find (λ x . f (q0,x) ≠ {} ∧ (∀ (y,q'') ∈ f (q0,x) . (q'' ∈ stateSet)))
inputList)
    case None
    then show ?thesis
    proof (cases find-remove-2 (λ q' x . f (q',x) ≠ {} ∧ (∀ (y,q'') ∈ f (q',x) . (q''
∈ stateSet))) stateList inputList)
      case None
      then show ?thesis
        using ‹find (λx. f (q0, x) ≠ {} ∧ (∀(y, q'')∈f (q0, x). q'' ∈ stateSet))
inputList = None›
        unfolding ‹stateList = n # nL› by auto
    next
      case (Some a)
      then obtain q' x stateList' where ∗: find-remove-2 (λ q' x . f (q',x) ≠ {} ∧
(∀ (y,q'') ∈ f (q',x) . (q'' ∈ stateSet))) (n#nL) inputList
                                        = Some (q',x,stateList')
        unfolding ‹stateList = n # nL› by (metis prod-cases3)
      have k = length stateList'
        using find-remove-2-length[OF ∗] ‹Suc k = length stateList›
        unfolding ‹stateList = n # nL›
        by simp
      show ?thesis
       using Suc.hyps(1)[of stateList' insert q' stateSet m@[(q',x)], OF ‹k = length
stateList'›]
      unfolding ‹stateList = n # nL› select-inputs.simps None ∗ find-remove-2-length[OF
∗]
        by simp
    qed
  next
    case (Some a)
    then show ?thesis
      unfolding ‹stateList = n # nL› by auto
  qed
qed
```

lemma *select-inputs-length-min* :
length (*select-inputs f q0 inputList stateList stateSet m*) $\geq$ (*length m*)
proof (*induction length stateList arbitrary*: *stateList stateSet m*)
case *0*
then show *?case*
by (*cases find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$(*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *stateSet*)) *inputList*; *auto*)
next
case (*Suc k*)
then obtain *n nL* **where** *stateList* = *n* # *nL*
by (*meson Suc-length-conv*)

show *?case*
proof (*cases find* (λ *x* . *f* (*q0*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q0*,*x*) . (*q''* $\in$ *stateSet*))) *inputList*)
case *None*
then show *?thesis*
proof (*cases find-remove-2* (λ *q' x* . *f* (*q'*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q'*,*x*) . (*q''* $\in$ *stateSet*))) *stateList inputList*)
case *None*
then show *?thesis* **using** ‹*find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$(*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *stateSet*)) *inputList* = *None*›
unfolding ‹*stateList* = *n* # *nL*› **by** *auto*
next
case (*Some a*)
then obtain *q' x stateList'* **where** $*$: *find-remove-2* (λ *q' x* . *f* (*q'*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q'*,*x*) . (*q''* $\in$ *stateSet*))) (*n*#*nL*) *inputList*
= *Some* (*q'*,*x*,*stateList'*)
unfolding ‹*stateList* = *n* # *nL*› **by** (*metis prod-cases3*)
have *k* = *length stateList'*
using *find-remove-2-length*[*OF* $*$] ‹*Suc k* = *length stateList*›
unfolding ‹*stateList* = *n* # *nL*›
by *simp*
show *?thesis*
unfolding ‹*stateList* = *n* # *nL*› *select-inputs.simps None* $*$ *find-remove-2-length*[*OF* $*$]
using *Suc.hyps*(*1*)[*of stateList' m*@[(*q'*,*x*)] *insert q' stateSet* , *OF* ‹*k* = *length stateList'*›]
by *simp*
qed
next
case (*Some a*)
then show *?thesis* **unfolding** ‹*stateList* = *n* # *nL*› **by** *auto*
qed
qed

lemma *select-inputs-helper1* :

find (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$(*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *nS*)) *iL* = *Some x*
$\Longrightarrow$ (*select-inputs f q0 iL nL nS m*) = *m*@[(*q0*,*x*)]
by (*cases nL*; *auto*)

lemma *select-inputs-take* :
take (*length m*) (*select-inputs f q0 inputList stateList stateSet m*) = *m*
proof (*induction length stateList arbitrary*: *stateList stateSet m*)
case *0*
then show *?case*
by (*cases find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$(*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *stateSet*)) *inputList*; *auto*)
next
case (*Suc k*)
then obtain *n nL* **where** *stateList* = *n* # *nL*
by (*meson Suc-length-conv*)

show *?case* **proof** (*cases find* (λ *x* . *f* (*q0*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q0*,*x*) . (*q''* $\in$ *stateSet*))) *inputList*)
case *None*
then show *?thesis*
proof (*cases find-remove-2* (λ *q' x* . *f* (*q'*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q'*,*x*) . (*q''* $\in$ *stateSet*))) *stateList inputList*)
case *None*
then show *?thesis* **using** ‹*find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$(*y*, *q''*)$\in$*f* (*q0*, *x*). *q''* $\in$ *stateSet*)) *inputList* = *None*›
unfolding ‹*stateList* = *n* # *nL*› **by** *auto*
next
case (*Some a*)
then obtain *q' x stateList'* **where** $*$: *find-remove-2* (λ *q' x* . *f* (*q'*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *f* (*q'*,*x*) . (*q''* $\in$ *stateSet*))) (*n*#*nL*) *inputList*
= *Some* (*q'*,*x*,*stateList'*)
unfolding ‹*stateList* = *n* # *nL*›
by (*metis prod-cases3*)
have *k* = *length stateList'*
using *find-remove-2-length*[*OF* $*$] ‹*Suc k* = *length stateList*›
unfolding ‹*stateList* = *n* # *nL*›
by *simp*

have $**$: (*select-inputs f q0 inputList stateList stateSet m*)
= *select-inputs f q0 inputList stateList'* (*insert q' stateSet*) (*m* @ [(*q'*, *x*)])
unfolding ‹*stateList* = *n* # *nL*› *select-inputs.simps None* $*$
by *simp*
show *?thesis*
unfolding $**$
using *Suc.hyps*(*1*)[*of stateList' m*@[(*q'*,*x*)] *insert q' stateSet* , *OF* ‹*k* = *length stateList'*›]
by (*metis butlast-snoc butlast-take diff-Suc-1 length-append-singleton se-*

lect-inputs-length-min)
  qed
 next
  case (*Some a*)
  then show *?thesis* **unfolding** ‹*stateList* = *n* # *nL*› **by** *auto*
 qed
qed

lemma *select-inputs-take′* :
 take (*length m*) (*select-inputs f q0 iL nL nS* (*m*@*m′*)) = *m*
 using *select-inputs-take*
 by (*metis* (*no-types*, *lifting*) *add-leE append-eq-append-conv select-inputs-length-min length-append*
  length-take min-absorb2 take-add)

lemma *select-inputs-distinct* :
 assumes *distinct* (*map fst m*)
 and *set* (*map fst m*) ⊆ *nS*
 and *q0* ∉ *nS*
 and *distinct nL*
 and *q0* ∉ *set nL*
 and *set nL* ∩ *nS* = {}
 shows *distinct* (*map fst* (*select-inputs f q0 iL nL nS m*))
using *assms* **proof** (*induction length nL arbitrary*: *nL nS m*)
 case *0*
 then show *?case*
  by (*cases find* (λ*x*. *f* (*q0*, *x*) ≠ {} ∧ (∀(*y*, *q′′*)∈*f* (*q0*, *x*). *q′′* ∈ *nS*)) *iL*; *auto*)
next
 case (*Suc k*)
 then obtain *n nL′′* **where** *nL* = *n* # *nL′′*
  by (*meson Suc-length-conv*)

 show *?case* **proof** (*cases find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q′′*) ∈ *f* (*q0*,*x*) . (*q′′* ∈ *nS*))) *iL*)
  case *None*
  then show *?thesis*
  proof (*cases find-remove-2* (λ *q′ x* . *f* (*q′*,*x*) ≠ {} ∧ (∀ (*y*,*q′′*) ∈ *f* (*q′*,*x*) . (*q′′* ∈ *nS*))) *nL iL*)
   case *None*
   then have (*select-inputs f q0 iL nL nS m*) = *m*
    using ‹*find* (λ*x*. *f* (*q0*, *x*) ≠ {} ∧ (∀(*y*, *q′′*)∈*f* (*q0*, *x*). *q′′* ∈ *nS*)) *iL* = *None*›
    unfolding ‹*nL* = *n* # *nL′′*› **by** *auto*
   then show *?thesis*
    using *Suc.prems* **by** *auto*
  next
   case (*Some a*)

then obtain q' x nL' **where** $*$: *find-remove-2* (λ q' x . f $(q',x) \neq \{\} \wedge (\forall (y,q'') \in f\ (q',x)\ .\ (q'' \in nS)))$ nL iL
$= Some\ (q',x,nL')$
by (*metis prod-cases3*)

have $k = length\ nL'$
using *find-remove-2-length*[*OF* $*$] ‹*Suc k = length nL*›
by *simp*

have *select-inputs f q0 iL nL nS m = select-inputs f q0 iL nL'* (*insert q' nS*) (*m* @ [(*q'*, *x*)])
using $*$
unfolding ‹*nL = n # nL''*› *select-inputs.simps None*
by *auto*

have $q' \in set\ nL$
and $set\ nL' = set\ nL - \{q'\}$
and $distinct\ nL'$
using *find-remove-2-set*[*OF* $*$] ‹*distinct nL*› **by** *auto*

have *distinct* (*map fst* (*m*@[(*q'*,*x*)]))
using ‹(*set* (*map fst m*)) $\subseteq$ *nS*› ‹*set nL* $\cap$ *nS* = {}› ‹*q'* $\in$ *set nL*› ‹*distinct* (*map fst m*)›
by *auto*
have $q0 \notin insert\ q'\ nS$
using *Suc.prems*(*3*) *Suc.prems*(*5*) ‹*q'* $\in$ *set nL*› **by** *auto*
have *set* (*map fst* (*m*@[(*q'*,*x*)])) $\subseteq$ *insert q' nS*
using ‹(*set* (*map fst m*)) $\subseteq$ *nS*› **by** *auto*
have (*set* (*map fst* (*m*@[(*q'*,*x*)]))) $\subseteq$ *insert q' nS*
using‹(*set* (*map fst m*)) $\subseteq$ *nS*› **by** *auto*
have $q0 \notin set\ nL'$
by (*simp add*: *Suc.prems*(*5*) ‹*set nL'* = *set nL* − {*q'*}›)
have *set nL'* $\cap$ *insert q' nS* = {}
using *Suc.prems*(*6*) ‹*set nL'* = *set nL* − {*q'*}› **by** *auto*

show *?thesis*
unfolding *select-inputs.simps None* $*$
using *Suc.hyps*(*1*)[*OF* ‹*k = length nL'*› ‹*distinct* (*map fst* (*m*@[(*q'*,*x*)]))›
‹*set* (*map fst* (*m*@[(*q'*,*x*)])) $\subseteq$ *insert q' nS*›
‹*q0* $\notin$ *insert q' nS*›
‹*distinct nL'*›
‹*q0* $\notin$ *set nL'*›
‹*set nL'* $\cap$ *insert q' nS* = {}›]
unfolding ‹*select-inputs f q0 iL nL nS m = select-inputs f q0 iL nL'* (*insert q' nS*) (*m* @ [(*q'*, *x*)])›
by *assumption*
qed
next
case (*Some a*)

then show *?thesis*
using *Suc* ‹$nL = n \# nL''$› **by** *auto*
qed
qed

lemma *select-inputs-index-properties* :
assumes $i < length\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m)$
and $i \geq length\ m$
and $distinct\ (map\ fst\ m)$
and $nS = nS0 \cup set\ (map\ fst\ m)$
and $q0 \notin nS$
and $distinct\ nL$
and $q0 \notin set\ nL$
and $set\ nL \cap nS = \{\}$
shows $fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \in (insert\ q0\ (set\ nL))$
$fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \notin nS0$
$snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \in set\ iL$
$(\forall\ qx' \in set\ (take\ i\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m))\ .\ fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \neq fst\ qx')$
$(\exists\ t \in transitions\ M\ .\ t\text{-}source\ t = fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \wedge t\text{-}input\ t = snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i))$
$(\forall\ t \in transitions\ M\ .\ (t\text{-}source\ t = fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \wedge t\text{-}input\ t = snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i)) \longrightarrow (t\text{-}target\ t \in nS0 \vee (\exists\ qx' \in set\ (take\ i\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m))\ .\ fst\ qx' = (t\text{-}target\ t))))$
proof −
have *combined-props* :
$fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \in (insert\ q0\ (set\ nL))$
$\wedge\ snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \in set\ iL$
$\wedge\ fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \notin nS0$
$\wedge\ (\exists\ t \in transitions\ M\ .\ t\text{-}source\ t = fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \wedge t\text{-}input\ t = snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i))$
$\wedge\ (\forall\ t \in transitions\ M\ .\ (t\text{-}source\ t = fst\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i) \wedge t\text{-}input\ t = snd\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m\ !\ i)) \longrightarrow (t\text{-}target\ t \in nS0 \vee (\exists\ qx' \in set\ (take\ i\ (select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m))\ .\ fst\ qx' = (t\text{-}target\ t))))$
using *assms* **proof** (*induction length nL arbitrary*: *nL nS m*)
case *0*
show *?case* **proof** ($cases\ find\ (\lambda\ x\ .\ (h\ M)\ (q0,x) \neq \{\} \wedge (\forall\ (y,q'') \in (h\ M)\ (q0,x)\ .\ (q'' \in nS)))\ iL$)
case *None*
then have $(select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m) = m$ **using** *0* **by** *auto*
then have *False* **using** *0.prems*(*1*,*2*) **by** *auto*
then show *?thesis* **by** *simp*
next
case (*Some x*)
have $(select\text{-}inputs\ (h\ M)\ q0\ iL\ nL\ nS\ m) = m@[(q0,x)]$ **using** *select-inputs-helper1*[*OF Some*] **by** *assumption*

then have (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) = (*q0*,*x*) **using** *0.prems*(*1*,*2*)
using *antisym* **by** *fastforce*

have *fst* (*q0*, *x*) ∈ *insert q0* (*set nL*) **by** *auto*
moreover have *snd* (*q0*, *x*) ∈ *set iL* **using** *find-set*[*OF Some*] **by** *auto*
moreover have *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∉ *nS0*
using ‹*select-inputs* (*h M*) *q0 iL nL nS m* ! *i* = (*q0*, *x*)› *assms*(*4*) *assms*(*5*) **by** *auto*

moreover have (∃ *t*∈*FSM.transitions M*. *t-source t* = *fst* (*q0*, *x*) ∧ *t-input t* = *snd* (*q0*, *x*))
using *find-condition*[*OF Some*] **unfolding** *fst-conv snd-conv h.simps*
by *fastforce*
moreover have ⋀ *t* . *t* ∈ *FSM.transitions M* ⟹
t-source t = *fst* (*q0*, *x*) ⟹ *t-input t* = *snd* (*q0*, *x*) ⟹
t-target t ∈ *nS0* ∨ (∃ *qx'*∈*set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)). *fst qx'* = *t-target t*)
proof −
fix *t* **assume** *t* ∈ *FSM.transitions M t-source t* = *fst* (*q0*, *x*) *t-input t* = *snd* (*q0*, *x*)
then have *t-target t* ∈ *nS*
using *find-condition*[*OF Some*] **unfolding** *h.simps fst-conv snd-conv*
by (*metis* (*no-types*, *lifting*) *case-prod-beta' h-simps mem-Collect-eq prod.collapse*)
then show *t-target t* ∈ *nS0* ∨ (∃ *qx'*∈*set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)). *fst qx'* = *t-target t*)
using ‹*nS* = *nS0* ∪ *set* (*map fst m*)›
using *0.prems*(*1*) *0.prems*(*2*) ‹*select-inputs* (*h M*) *q0 iL nL nS m* = *m* @ [(*q0*, *x*)]› **by** *fastforce*
qed

ultimately show *?thesis*
unfolding ‹(*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) = (*q0*,*x*)› **by** *blast*
qed
next
case (*Suc k*)
then obtain *n nL''* **where** *nL* = *n* # *nL''*
by (*meson Suc-length-conv*)

show *?case* **proof** (*cases find* (λ *x* . (*h M*) (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ (*h M*) (*q0*,*x*) . (*q''* ∈ *nS*))) *iL*)
case *None*
show *?thesis* **proof** (*cases find-remove-2* (λ *q'* *x* . (*h M*) (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ (*h M*) (*q'*,*x*) . (*q''* ∈ *nS*))) *nL iL*)
case *None*
then have (*select-inputs* (*h M*) *q0 iL nL nS m*) = *m* **using** ‹*find* (λ*x*. *h M* (*q0*, *x*) ≠ {} ∧ (∀ (*y*, *q''*)∈*h M* (*q0*, *x*). *q''* ∈ *nS*)) *iL* = *None*› ‹*nL* = *n* # *nL''*› **by** *auto*
then have *False* **using** *Suc.prems*(*1*,*2*) **by** *auto*

then show *?thesis* **by** *simp*
next
case (*Some a*)
then obtain q' x nL' **where** $**$: *find-remove-2* (λ q' x . (h M) (q',x) $\neq$ {} $\wedge$ ($\forall$ (y,q'') $\in$ (h M) (q',x) . (q'' $\in$ nS))) nL iL = *Some* (q',x,nL')
by (*metis prod-cases3*)

have k = *length* nL'
using *find-remove-2-length*[*OF* $**$] ‹*Suc k = length nL*› **by** *simp*

have *select-inputs* (h M) *q0 iL nL nS m* = *select-inputs* (h M) *q0 iL nL'* (*insert* q' *nS*) (m @ [(q', x)])
using $**$
unfolding ‹*nL = n # nL''*› *select-inputs.simps None* **by** *auto*
then have i < *length* (*select-inputs* (h M) *q0 iL nL'* (*insert* q' *nS*) (m@[(q',x)]))
using *Suc.prems*(*1*) **by** *auto*

have (*set* (*map fst* (m @ [(q', x)]))) $\subseteq$ *insert* q' *nS*
using *Suc.prems*(*4*) **by** *auto*

have q' $\in$ *set nL*
and *set* nL' = *set nL* − {q'}
and *distinct* nL'
using *find-remove-2-set*[*OF* $**$] ‹*distinct nL*› **by** *auto*
have *set* nL' $\subseteq$ *set nL*
using *find-remove-2-set*(*4*)[*OF* $**$ ‹*distinct nL*›] **by** *blast*

have *distinct* (*map fst* (m @ [(q', x)]))
using *Suc.prems*(*4*) ‹*set nL* $\cap$ *nS* = {}› ‹q' $\in$ *set nL*› ‹*distinct* (*map fst* m)› **by** *auto*
have *distinct* (*map fst* (m@[(q',x)]))
using *Suc.prems*(*4*) ‹*set nL* $\cap$ *nS* = {}› ‹q' $\in$ *set nL*› ‹*distinct* (*map fst* m)› **by** *auto*
have *q0* $\notin$ *insert* q' *nS*
using *Suc.prems*(*7*) *Suc.prems*(*5*) ‹q' $\in$ *set nL*› **by** *auto*
have *insert* q' *nS* = *nS0* $\cup$ *set* (*map fst* (m@[(q',x)]))
using *Suc.prems*(*4*) **by** *auto*
have *q0* $\notin$ *set* nL'
by (*metis Suc.prems*(*7*) ‹*set* nL' $\subseteq$ *set nL*› *subset-code*(*1*))
have *set* nL' $\cap$ *insert* q' *nS* = {}
using *Suc.prems*(*8*) ‹*set* nL' = *set nL* − {q'}› **by** *auto*

show *?thesis* **proof** (*cases length* (m @ [(q', x)]) $\leq$ i)
case *True*

show *?thesis*
using *Suc.hyps*(*1*)[*OF* ‹k = *length* nL'› ‹i < *length* (*select-inputs* (h M)

q0 iL nL′ (insert q′ nS) (m@[(q′,x)])›
True
‹*distinct (map fst (m @ [(q′, x)]))*›
‹*insert q′ nS = nS0 ∪ set (map fst (m@[(q′,x)]))*›
‹*q0 ∉ insert q′ nS*›
‹*distinct nL′*›
‹*q0 ∉ set nL′*›
‹*set nL′ ∩ insert q′ nS = {}*›]
unfolding ‹*select-inputs (h M) q0 iL nL nS m = select-inputs (h M) q0 iL nL′ (insert q′ nS) (m@[(q′,x)])*›
using ‹*set nL′ ⊆ set nL*› **by** *blast*
next
case *False*
then have *i = length m*
using *Suc.prems(2)* **by** *auto*
then have ∗∗∗: *select-inputs (h M) q0 iL nL nS m ! i = (q′,x)*
unfolding ‹*select-inputs (h M) q0 iL nL nS m = select-inputs (h M) q0 iL nL′ (insert q′ nS) (m@[(q′,x)])*›
using *select-inputs-take*
by (*metis length-append-singleton lessI nth-append-length nth-take*)

have *q′ ∈ insert q0 (set nL)*
by (*simp add*: ‹*q′ ∈ set nL*›)
moreover have *x ∈ set iL*
using *find-remove-2-set(3)*[*OF* ∗∗] **by** *auto*
moreover have *q′ ∉ nS0*
using *Suc.prems(4) Suc.prems(8)* ‹*q′ ∈ set nL*› **by** *blast*
moreover have (∃ *t∈FSM.transitions M. t-source t = q′ ∧ t-input t = x*)

using *find-remove-2-set(1)*[*OF* ∗∗] **unfolding** *h.simps* **by** *force*
moreover have (∀ *t∈FSM.transitions M. t-source t = q′ ∧ t-input t = x ⟶ t-target t ∈ nS0 ∨* (∃ *qx′∈set (take i (select-inputs (h M) q0 iL nL nS m)). fst qx′ = t-target t*))
unfolding ‹*i = length m*› *select-inputs-take*
using *find-remove-2-set(1)*[*OF* ∗∗] **unfolding** *h.simps* ‹*nS = nS0 ∪ (set (map fst m))*› **by** *force*
ultimately show *?thesis*
unfolding ∗∗∗ *fst-conv snd-conv* **by** *blast*
qed
qed
next
case (*Some x*)
have (*select-inputs (h M) q0 iL nL nS m) = m@[(q0,x)]* **using** *select-inputs-helper1*[*OF Some*] **by** *assumption*
then have (*select-inputs (h M) q0 iL nL nS m ! i) = (q0,x)* **using** *Suc.prems(1,2)*
using *antisym* **by** *fastforce*

have *fst (q0, x) ∈ insert q0 (set nL)* **by** *auto*

moreover have *snd* (*q0*, *x*) ∈ *set iL* **using** *find-set*[*OF Some*] **by** *auto*
moreover have *fst* (*q0*, *x*) ∉ *nS0*
using *assms*(*4*) *assms*(*5*) **by** *auto*
moreover have ⋀ *qx′* . *qx′* ∈ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) − *set* (*take* (*length m*) (*select-inputs* (*h M*) *q0 iL nL nS m*)) ⟹ *fst* (*q0*, *x*) ≠ *fst qx′*
using *Suc.prems*(*1*,*2*) ‹*select-inputs* (*h M*) *q0 iL nL nS m* = *m* @ [(*q0*, *x*)]› **by** *auto*
moreover have (∃ *t*∈*FSM.transitions M*. *t-source t* = *fst* (*q0*, *x*) ∧ *t-input t* = *snd* (*q0*, *x*))
using *find-condition*[*OF Some*] **unfolding** *fst-conv snd-conv h.simps*
by *fastforce*
moreover have ⋀ *t* . *t* ∈ *FSM.transitions M* ⟹
t-source t = *fst* (*q0*, *x*) ⟹ *t-input t* = *snd* (*q0*, *x*) ⟹
t-target t ∈ *nS0* ∨ (∃ *qx′*∈*set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)). *fst qx′* = *t-target t*)
proof −
fix *t* **assume** *t* ∈ *FSM.transitions M t-source t* = *fst* (*q0*, *x*) *t-input t* = *snd* (*q0*, *x*)
then have *t-target t* ∈ *nS*
using *find-condition*[*OF Some*] **unfolding** *h.simps fst-conv snd-conv*
by (*metis* (*no-types*, *lifting*) *case-prod-beta′ h-simps mem-Collect-eq prod.collapse*)
then show *t-target t* ∈ *nS0* ∨ (∃ *qx′*∈*set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)). *fst qx′* = *t-target t*)
using ‹*nS* = *nS0* ∪ (*set* (*map fst m*))›
using *Suc.prems*(*1*,*2*) ‹*select-inputs* (*h M*) *q0 iL nL nS m* = *m* @ [(*q0*, *x*)]› **by** *fastforce*
qed

ultimately show *?thesis*
unfolding ‹(*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) = (*q0*,*x*)› **by** *blast*
qed
qed

then show *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∈ (*insert q0* (*set nL*))
snd (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∈ *set iL*
fst (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∉ *nS0*
(∃ *t* ∈ *transitions M* . *t-source t* = *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∧ *t-input t* = *snd* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*))
(∀ *t* ∈ *transitions M* . (*t-source t* = *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ∧ *t-input t* = *snd* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*)) ⟶ (*t-target t* ∈ *nS0* ∨ (∃ *qx′* ∈ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) . *fst qx′* = (*t-target t*))))
by *blast*+

show (∀ *qx′* ∈ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) . *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) ≠ *fst qx′*)
proof

fix qx' **assume** $qx' \in$ *set* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*))
then obtain *j* **where** (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*)) ! *j* = qx' **and** *j* < *length* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*))
by (*meson in-set-conv-nth*)
then have *fst* qx' = (*map fst* (*select-inputs* (*h M*) *q0 iL nL nS m*)) ! *j* **and** *j* < *length* (*select-inputs* (*h M*) *q0 iL nL nS m*) **by** *auto*

moreover have *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) = (*map fst* (*select-inputs* (*h M*) *q0 iL nL nS m*)) ! *i*
using *assms*(*1*) **by** *auto*

moreover have $j \neq i$
using ‹*j* < *length* (*take i* (*select-inputs* (*h M*) *q0 iL nL nS m*))› **by** *auto*

moreover have *set* (*map fst m*) $\subseteq$ *nS*
using ‹*nS* = *nS0* $\cup$ *set* (*map fst m*)› **by** *blast*

ultimately show *fst* (*select-inputs* (*h M*) *q0 iL nL nS m* ! *i*) $\neq$ *fst* qx'
using *assms*(*1*)
using *select-inputs-distinct*[*OF* ‹*distinct* (*map fst m*)› - ‹*q0* $\notin$ *nS*› ‹*distinct nL*› ‹*q0* $\notin$ *set nL*› ‹*set nL* $\cap$ *nS* = {}›]
by (*metis distinct-Ex1 in-set-conv-nth length-map*)
qed
qed

lemma *select-inputs-initial* :
assumes *qx* $\in$ *set* (*select-inputs f q0 iL nL nS m*) − *set m*
and *fst qx* = *q0*
shows (*last* (*select-inputs f q0 iL nL nS m*)) = *qx*
using *assms*(*1*) **proof** (*induction length nL arbitrary*: *nS nL m*)
case *0*
then have *nL* = [] **by** *auto*

have *find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, q'')$\in$*f* (*q0*, *x*). $q'' \in$ *nS*)) *iL* $\neq$ *None*
using *0* **unfolding** ‹*nL* = []› *select-inputs.simps*
by (*metis Diff-cancel empty-iff option.simps*(*4*))
then obtain *x* **where** $*$: *find* (λx. *f* (*q0*, *x*) $\neq$ {} $\wedge$ ($\forall$ (*y*, q'')$\in$*f* (*q0*, *x*). $q'' \in$ *nS*)) *iL* = *Some x*
by *auto*

have *set* (*select-inputs f q0 iL nL nS m*) − *set m* = {*qx*}
using *0.prems*(*1*) **unfolding** *select-inputs-helper1*[*OF* $*$]
by *auto*

then show *?case* **unfolding** *select-inputs-helper1*[*OF* $*$]
by (*metis DiffD1 DiffD2 UnE empty-iff empty-set insert-iff last-snoc list.simps*(*15*) *set-append*)
next

case (*Suc k*)
then obtain *n nL''* **where** *nL* = *n* # *nL''*
by (*meson Suc-length-conv*)

show *?case* **proof** (*cases find* (λ*x*. *f* (*q0*, *x*) ≠ {} ∧ (∀(*y*, *q''*)∈*f* (*q0*, *x*). *q''* ∈ *nS*)) *iL*)
case *None*
show *?thesis* **proof** (*cases find-remove-2* (λ *q' x* . *f* (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q'*,*x*) . (*q''* ∈ *nS*))) *nL iL*)
case *None*
have (*select-inputs f q0 iL nL nS m*) = *m*
using ‹*find* (λ*x*. *f* (*q0*, *x*) ≠ {} ∧ (∀(*y*, *q''*)∈*f* (*q0*, *x*). *q''* ∈ *nS*)) *iL* = *None*› *None* ‹*nL* = *n* # *nL''*› **by** *auto*
then have *False*
using *Suc.prems*(*1*)
by *simp*
then show *?thesis* **by** *simp*
next
case (*Some a*)
then obtain *q' x nL'* **where** **: *find-remove-2* (λ *q' x* . *f* (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q'*,*x*) . (*q''* ∈ *nS*))) *nL iL* = *Some* (*q'*,*x*,*nL'*)
by (*metis prod-cases3*)

have *k* = *length nL'*
using *find-remove-2-length*[*OF* **] ‹*Suc k* = *length nL*› **by** *simp*

have *select-inputs f q0 iL nL nS m* = *select-inputs f q0 iL nL'* (*insert q' nS*) (*m* @ [(*q'*, *x*)])
using **
unfolding ‹*nL* = *n* # *nL''*› *select-inputs.simps None* **by** *auto*
then have *qx* ∈ *set* (*select-inputs f q0 iL nL'* (*insert q' nS*) (*m*@[(*q'*,*x*)])) − *set m*
using *Suc.prems* **by** *auto*
moreover have *q0* ≠ *q'*
using *None* **unfolding** *find-None-iff*
using *find-remove-2-set*(*1*,*2*,*3*)[*OF* **]
by *blast*
ultimately have *qx* ∈ *set* (*select-inputs f q0 iL nL'* (*insert q' nS*) (*m*@[(*q'*,*x*)])) − *set* (*m*@[(*q'*,*x*)])
using ‹*fst qx* = *q0*› **by** *auto*
then show *?thesis*
using *Suc.hyps* **unfolding** ‹(*select-inputs f q0 iL nL nS m*) = (*select-inputs f q0 iL nL'* (*insert q' nS*) (*m*@[(*q'*,*x*)]))›
using ‹*k* = *length nL'*› **by** *blast*
qed
next
case (*Some a*)

have *set* (*select-inputs f q0 iL nL nS m*) − *set m* = {*qx*}

using *Suc.prems*(*1*) **unfolding** *select-inputs-helper1*[*OF Some*]
by *auto*

then show *?thesis* **unfolding** *select-inputs-helper1*[*OF Some*]
by (*metis DiffD1 DiffD2 UnE empty-iff empty-set insert-iff last-snoc list.simps*(*15*) *set-append*)
qed
qed

lemma *select-inputs-max-length* :
assumes *distinct nL*
shows *length* (*select-inputs f q0 iL nL nS m*) ≤ *length m* + *Suc* (*length nL*)
using *assms* **proof** (*induction length nL arbitrary*: *nL nS m*)
case *0*
then show *?case* **by** (*cases find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *nS*))) *iL*; *auto*)
next
case (*Suc k*)
then obtain *n nL''* **where** *nL* = *n* # *nL''*
by (*meson Suc-length-conv*)

show *?case* **proof** (*cases find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *nS*))) *iL*)
case *None*
show *?thesis* **proof** (*cases find-remove-2* (λ *q' x* . *f* (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q'*,*x*) . (*q''* ∈ *nS*))) *nL iL*)
case *None*
show *?thesis* **unfolding** ‹*nL* = *n* # *nL''*› *select-inputs.simps None* ‹*find* (λ*x*. *f* (*q0*, *x*) ≠ {} ∧ (∀(*y*, *q''*)∈*f* (*q0*, *x*). *q''* ∈ *nS*)) *iL* = *None*›
using *None* ‹*nL* = *n* # *nL''*› **by** *auto*
next
case (*Some a*)
then obtain *q' x nL'* **where** ∗∗: *find-remove-2* (λ *q' x* . *f* (*q'*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q'*,*x*) . (*q''* ∈ *nS*))) *nL iL* = *Some* (*q'*,*x*,*nL'*)
by (*metis prod-cases3*)
have *k* = *length nL'*
using *find-remove-2-length*[*OF* ∗∗] ‹*Suc k* = *length nL*› **by** *simp*

have *select-inputs f q0 iL nL nS m* = *select-inputs f q0 iL nL'* (*insert q' nS*) (*m* @ [(*q'*, *x*)])
using ∗∗
unfolding ‹*nL* = *n* # *nL''*› *select-inputs.simps None* **by** *auto*

have *length nL* = *Suc* (*length nL'*) ∧ *distinct nL'*
using *find-remove-2-set*(*2*,*4*,*5*)[*OF* ∗∗] ‹*distinct nL*›
by (*metis One-nat-def Suc-pred distinct-card distinct-remove1 equals0D length-greater-0-conv length-remove1 set-empty2 set-remove1-eq*)
then have *length* (*select-inputs f q0 iL nL'* (*insert q' nS*) (*m*@[(*q'*,*x*)])) ≤

length m + *Suc* (*length nL*)
using *Suc.hyps*(*1*)[*OF* ‹*k* = *length nL'*›]
by (*metis add-Suc-shift length-append-singleton*)
then show *?thesis*
using ‹(*select-inputs f q0 iL nL nS m*) = *select-inputs f q0 iL nL'* (*insert q' nS*) (*m*@[(*q'*,*x*)])› **by** *simp*
qed
next
case (*Some a*)
show *?thesis* **unfolding** *select-inputs-helper1*[*OF Some*] **by** *auto*
qed
qed

lemma *select-inputs-q0-containment* :
assumes *f* (*q0*,*x*) ≠ {}
and (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *nS*))
and *x* ∈ *set iL*
shows (∃ *qx* ∈ *set* (*select-inputs f q0 iL nL nS m*) . *fst qx* = *q0*)
proof −
have *find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *nS*))) *iL* ≠ *None*
using *assms* **unfolding** *find-None-iff* **by** *blast*
then obtain *x'* **where** ∗: *find* (λ *x* . *f* (*q0*,*x*) ≠ {} ∧ (∀ (*y*,*q''*) ∈ *f* (*q0*,*x*) . (*q''* ∈ *nS*))) *iL* = *Some x'*
by *auto*
show *?thesis*
unfolding *select-inputs-helper1*[*OF* ∗] **by** *auto*
qed

lemma *select-inputs-from-submachine* :
assumes *single-input S*
and *acyclic S*
and *is-submachine S M*
and ⋀ *q x* . *q* ∈ *reachable-states S* ⟹ *h S* (*q*,*x*) ≠ {} ⟹ *h S* (*q*,*x*) = *h M* (*q*,*x*)
and ⋀ *q* . *q* ∈ *reachable-states S* ⟹ *deadlock-state S q* ⟹ *q* ∈ *nS0* ∪ *set* (*map fst m*)
and *states M* = *insert* (*initial S*) (*set nL* ∪ *nS0* ∪ *set* (*map fst m*))
and (*initial S*) ∉ (*set nL* ∪ *nS0* ∪ *set* (*map fst m*))
shows *fst* (*last* (*select-inputs* (*h M*) (*initial S*) (*inputs-as-list M*) *nL* (*nS0* ∪ *set* (*map fst m*)) *m*)) = (*initial S*)
and *length* (*select-inputs* (*h M*) (*initial S*) (*inputs-as-list M*) *nL* (*nS0* ∪ *set* (*map fst m*)) *m*) > *0*
proof −
have *fst* (*last* (*select-inputs* (*h M*) (*initial S*) (*inputs-as-list M*) *nL* (*nS0* ∪ *set* (*map fst m*)) *m*)) = (*initial S*) ∧ *length* (*select-inputs* (*h M*) (*initial S*) (*inputs-as-list M*) *nL* (*nS0* ∪ *set* (*map fst m*)) *m*) > *0*
using *assms*(*5*,*6*,*7*) **proof** (*induction length nL arbitrary*: *nL m*)

case *0*
then have $nL = []$ **by** *auto*

have $\neg$ (*deadlock-state S* (*initial S*))
using *assms*(*5*,*6*,*3*,*7*) *reachable-states-initial* **by** *blast*
then obtain x **where** $x \in$ *set* (*inputs-as-list M*) **and** *h S* ((*initial S*),x) $\neq \{\}$
using *assms*(*3*) **unfolding** *deadlock-state.simps h.simps inputs-as-list-set*
by *fastforce*

then have *h M* ((*initial S*),x) $\neq \{\}$
using *assms*(*4*)[*OF reachable-states-initial*] **by** *fastforce*

have (*initial S*) $\in$ *reachable-states M*
using *assms*(*3*) *reachable-states-initial* **by** *auto*

then have (*initial S*) $\in$ *states M*
using *reachable-state-is-state* **by** *force*

have $\bigwedge$ y q'' . (y,q'') $\in$ *h M* ((*initial S*),x) $\Longrightarrow$ q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*))
proof −
fix y q'' **assume** (y,q'') $\in$ *h M* ((*initial S*),x)
then have q'' $\in$ *reachable-states M* **using** *fsm-transition-target* **unfolding** *h.simps*
using ‹*FSM.initial S* $\in$ *reachable-states M*› *reachable-states-next* **by** *fastforce*

then have q'' $\in$ *insert* (*initial S*) (*nS0* $\cup$ *set* (*map fst m*)) **using** *0.prems*(*2*) ‹$nL = []$›
using *reachable-state-is-state* **by** *force*
moreover have $q'' \neq$ (*initial S*)
using *acyclic-no-self-loop*[*OF* ‹*acyclic S*› *reachable-states-initial*]
using ‹(y,q'') $\in$ *h M* ((*initial S*),x)› *assms*(*4*)[*OF reachable-states-initial* ‹*h S* ((*initial S*),x) $\neq \{\}$›] **unfolding** *h-simps*
by *blast*
ultimately show q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*)) **by** *blast*
qed
then have $x \in$ *set* (*inputs-as-list M*) $\wedge$ *h M* ((*initial S*), x) $\neq \{\}$ $\wedge$ ($\forall$ (y, q'')$\in$*h M* ((*initial S*), x). q'' $\in$ *nS0* $\cup$ *set* (*map fst m*))
using ‹$x \in$ *set* (*inputs-as-list M*) › ‹*h M* ((*initial S*), x) $\neq \{\}$› **by** *blast*
then have *find* (λ x . (*h M*) ((*initial S*),x) $\neq \{\}$ $\wedge$ ($\forall$ (y,q'') $\in$ (*h M*) ((*initial S*),x) . (q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*))))) (*inputs-as-list M*) $\neq$ *None*
unfolding *find-None-iff* **by** *blast*
then show *?case*
unfolding ‹$nL = []$› *select-inputs.simps* **by** *auto*
next
case (*Suc k*)
then obtain n nL'' **where** $nL = n \# nL''$
by (*meson Suc-length-conv*)

have $\exists$ *q x* . *q* $\in$ *reachable-states S* $-$ (*nS0* $\cup$ *set* (*map fst m*)) $\wedge$ *h M* (*q*,*x*) $\neq$ {} $\wedge$ ($\forall$ (*y*,*q''*) $\in$ *h M* (*q*,*x*) . *q''* $\in$ (*nS0* $\cup$ *set* (*map fst m*)))
proof $-$
define *ndlps* **where** *ndlps-def*: *ndlps* = {*p* . *path S* (*initial S*) *p* $\wedge$ *target* (*initial S*) *p* $\notin$ (*nS0* $\cup$ *set* (*map fst m*))}

have *path S* (*initial S*) [] $\wedge$ *target* (*initial S*) [] $\notin$ (*nS0* $\cup$ *set* (*map fst m*))
using *Suc.prems*(*3*) **by** *auto*
then have [] $\in$ *ndlps*
unfolding *ndlps-def* **by** *blast*
then have *ndlps* $\neq$ {} **by** *auto*
moreover have *finite ndlps*
using *acyclic-finite-paths-from-reachable-state*[*OF* ‹*acyclic S*›, *of* []] **unfolding** *ndlps-def* **by** *fastforce*
ultimately have $\exists$ *p* $\in$ *ndlps* . $\forall$ *p'* $\in$ *ndlps* . *length p'* $\leq$ *length p*
by (*meson max-length-elem not-le-imp-less*)
then obtain *p* **where** *path S* (*initial S*) *p*
and *target* (*initial S*) *p* $\notin$ (*nS0* $\cup$ *set* (*map fst m*))
and $\bigwedge$ *p'* . *path S* (*initial S*) *p'* $\Longrightarrow$ *target* (*initial S*) *p'* $\notin$ (*nS0* $\cup$ *set* (*map fst m*)) $\Longrightarrow$ *length p'* $\leq$ *length p*
unfolding *ndlps-def* **by** *blast*

let *?q* = *target* (*initial S*) *p*
have $\neg$ *deadlock-state S ?q*
using *Suc.prems*(*1*) *reachable-states-intro*[*OF* ‹*path S* (*initial S*) *p*›] **using** ‹*?q* $\notin$ (*nS0* $\cup$ *set* (*map fst m*))› **by** *blast*
then obtain *x* **where** *h S* (*?q*,*x*) $\neq$ {}
unfolding *deadlock-state.simps h.simps* **by** *fastforce*
then have *h M* (*?q*,*x*) $\neq$ {}
using *assms*(*4*)[*of ?q -*] *reachable-states-intro*[*OF* ‹*path S* (*initial S*) *p*›]
by *blast*

moreover have $\bigwedge$ *y q''* . (*y*,*q''*) $\in$ *h M* (*?q*,*x*) $\Longrightarrow$ *q''* $\in$ (*nS0* $\cup$ *set* (*map fst m*))
proof (*rule ccontr*)
fix *y q''* **assume** (*y*,*q''*) $\in$ *h M* (*?q*,*x*) **and** *q''* $\notin$ *nS0* $\cup$ *set* (*map fst m*)
then have (*?q*,*x*,*y*,*q''*) $\in$ *transitions S*
using *assms*(*4*)[*OF reachable-states-intro*[*OF* ‹*path S* (*initial S*) *p*›] ‹*h S* (*?q*,*x*) $\neq$ {}›] **unfolding** *h-simps*
by *blast*
then have *path S* (*initial S*) (*p*@[(*?q*,*x*,*y*,*q''*)])
using ‹*path S* (*initial S*) *p*› **by** (*simp add*: *path-append-transition*)
moreover have *target* (*initial S*) (*p*@[(*?q*,*x*,*y*,*q''*)]) $\notin$ (*nS0* $\cup$ *set* (*map fst m*))
using ‹*q''* $\notin$ *nS0* $\cup$ *set* (*map fst m*)› **by** *auto*
ultimately show *False*

using ‹$\bigwedge$ p' . *path S* (*initial S*) p' $\Longrightarrow$ *target* (*initial S*) p' $\notin$ (*nS0* $\cup$ *set* (*map fst m*)) $\Longrightarrow$ *length* p' $\leq$ *length* p›[*of* (p@[(*?q*,*x*,*y*,q'')])] **by** *simp*
qed

moreover have *?q* $\in$ *reachable-states S* $-$ (*nS0* $\cup$ *set* (*map fst m*))
using ‹*?q* $\notin$ (*nS0* $\cup$ *set* (*map fst m*))› ‹*path S* (*initial S*) *p*› **by** *blast*

ultimately show *?thesis* **by** *blast*
qed

then obtain *q x* **where** *q* $\in$ *reachable-states S* **and** *q* $\notin$ (*nS0* $\cup$ *set* (*map fst m*)) **and** *h M* (*q*,*x*) $\neq$ {} **and** ($\forall$ (y,q'') $\in$ *h M* (*q*,*x*) . q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*)))
by *blast*
then have *x* $\in$ *set* (*inputs-as-list M*)
unfolding *h.simps* **using** *fsm-transition-input inputs-as-list-set* **by** *fastforce*

show *?case* **proof** (*cases q* = *initial S*)
case *True*
have *find* (λx. *h M* (*FSM.initial S*, *x*) $\neq$ {} $\wedge$ ($\forall$(y, q'')$\in$*h M* (*FSM.initial S*, *x*). q'' $\in$ *nS0* $\cup$ *set* (*map fst m*))) (*inputs-as-list M*) $\neq$ *None*
using ‹*h M* (*q*,*x*) $\neq$ {}› ‹($\forall$ (y,q'') $\in$ *h M* (*q*,*x*) . q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*)))› ‹*x* $\in$ *set* (*inputs-as-list M*)›
unfolding *True find-None-iff* **by** *blast*
then show *?thesis* **unfolding** ‹*nL* = *n* # *nL''*› **by** *auto*
next
case *False*
then have *q* $\in$ *set nL*
using *submachine-reachable-subset*[*OF* ‹*is-submachine S M*›]
unfolding *is-submachine.simps* ‹*states M* = *insert* (*initial S*) (*set nL* $\cup$ *nS0* $\cup$ *set* (*map fst m*))›
using ‹*q* $\in$ *reachable-states S*› ‹*q* $\notin$ (*nS0* $\cup$ *set* (*map fst m*))›
by (*metis* (*no-types*, *lifting*) *Suc.prems*(*2*) *UnE insertE reachable-state-is-state subsetD sup-assoc*)

show *?thesis* **proof** (*cases find* (λx. *h M* (*FSM.initial S*, *x*) $\neq$ {} $\wedge$ ($\forall$(y, q'')$\in$*h M* (*FSM.initial S*, *x*). q'' $\in$ *nS0* $\cup$ *set* (*map fst m*))) (*inputs-as-list M*))
case *None*

have *find-remove-2* (λ q' x . (*h M*) (q',*x*) $\neq$ {} $\wedge$ ($\forall$ (y,q'') $\in$ (*h M*) (q',*x*) . (q'' $\in$ *nS0* $\cup$ *set* (*map fst m*)))) (*nL*) (*inputs-as-list M*) $\neq$ *None*
using ‹*q* $\in$ *set nL*› ‹*h M* (*q*,*x*) $\neq$ {}› ‹($\forall$ (y,q'') $\in$ *h M* (*q*,*x*) . q'' $\in$ (*nS0* $\cup$ *set* (*map fst m*)))› ‹*x* $\in$ *set* (*inputs-as-list M*)›
unfolding *find-remove-2-None-iff* ‹*nL* = *n* # *nL''*›

by *blast*
then obtain q' x' nL' **where** $*$: *find-remove-2* (λ q' x . (h M) (q',x) $\neq$ {} $\wedge$ ($\forall$ (y,q'') $\in$ (h M) (q',x) . (q'' $\in$ *nS0* $\cup$ *set* (*map fst m*)))) (n#nL'') (*inputs-as-list* M) = *Some* (q',x',nL')
unfolding ‹nL = n # nL''› **by** *auto*
have k = *length* nL'
using *find-remove-2-length*[*OF* $*$] ‹*Suc* k = *length* nL› ‹nL = n # nL''› **by** *simp*

have $**$: *select-inputs* (h M) (*initial* S) (*inputs-as-list* M) nL (*nS0* $\cup$ *set* (*map fst m*)) m
= *select-inputs* (h M) (*initial* S) (*inputs-as-list* M) nL' (*nS0* $\cup$ *set* (*map fst* (m@[(q',x')]))) (m@[(q',x')])
unfolding ‹nL = n # nL''› *select-inputs.simps None* $*$ **by** *auto*

have *p1*: ($\bigwedge q$. q $\in$ *reachable-states* S $\Longrightarrow$ *deadlock-state* S q $\Longrightarrow$ q $\in$ *nS0* $\cup$ *set* (*map fst* (m@[(q',x')])))
using *Suc.prems*(*1*) **by** *fastforce*

have *set* nL = *insert* q' (*set* nL') **using** *find-remove-2-set*(*2*,*6*)[*OF* $*$] **unfolding** ‹nL = n # nL''› **by** *auto*
then have (*set* nL $\cup$ *set* (*map fst m*)) = (*set* nL' $\cup$ *set* (*map fst* (m @ [(q', x')]))) **by** *auto*
then have *p2*: *states* M = *insert* (*initial* S) (*set* nL' $\cup$ *nS0* $\cup$ *set* (*map fst* (m @ [(q', x')])))
using *Suc.prems*(*2*) **by** *auto*

have *p3*: *initial* S $\notin$ *set* nL' $\cup$ *nS0* $\cup$ *set* (*map fst* (m @ [(q', x')]))
using *Suc.prems*(*3*) *False* ‹*set* nL = *insert* q' (*set* nL')› **by** *auto*

show *?thesis* **unfolding** $**$
using *Suc.hyps*(*1*)[*OF* ‹k = *length* nL'› *p1 p2 p3*] **by** *blast*
next
case (*Some a*)
then show *?thesis* **unfolding** ‹nL = n # nL''› **by** *auto*
qed
qed
qed
then show *fst* (*last* (*select-inputs* (h M) (*initial* S) (*inputs-as-list* M) nL (*nS0* $\cup$ *set* (*map fst m*)) m)) = (*initial* S)
and *length* (*select-inputs* (h M) (*initial* S) (*inputs-as-list* M) nL (*nS0* $\cup$ *set* (*map fst m*)) m) > *0*
by *blast+*
qed

end

34 State Separators

This theory defined state separators. A state separator S of some pair of states *q1*, *q2* of some FSM M is an acyclic single-input FSM based on the product machine P of M with initial state *q1* and M with initial state *q2* such that every maximal length sequence in the language of S is either in the language of *q1* or the language of *q2*, but not both. That is, C represents a strategy of distinguishing *q1* and *q2* in every complete submachine of P. In testing, separators are used to distinguish states reached in the SUT to establish a lower bound on the number of distinct states in the SUT.

theory *State-Separator*
imports *../Product-FSM Backwards-Reachability-Analysis*
begin

34.1 Canonical Separators

34.1.1 Construction

fun *canonical-separator* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a \Rightarrow 'a \Rightarrow (('a \times 'a) + 'a,'b,'c)$ *fsm*
where
canonical-separator M q1 q2 = (*canonical-separator′ M* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))) *q1 q2*)

lemma *canonical-separator-simps* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*canonical-separator M q1 q2*) = *Inl* (*q1*,*q2*)
states (*canonical-separator M q1 q2*)
= (*image Inl* (*states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))) ∪ {*Inr q1*, *Inr q2*}
inputs (*canonical-separator M q1 q2*) = *inputs M*
outputs (*canonical-separator M q1 q2*) = *outputs M*
transitions (*canonical-separator M q1 q2*)
= *shifted-transitions* (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states* ((*product* (*from-FSM M q1*) (*from-FSM M q2*)))) (*inputs* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
proof −
have ∗: *initial* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))) = (*q1*,*q2*)
unfolding *restrict-to-reachable-states-simps product-simps* **using** *assms* **by** *auto*
have ∗∗∗: *inputs* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *inputs M*
unfolding *restrict-to-reachable-states-simps product-simps* **using** *assms* **by** *auto*
have ∗∗∗∗: *outputs* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *outputs M*
unfolding *restrict-to-reachable-states-simps product-simps* **using** *assms* **by** *auto*

show *initial* (*canonical-separator M q1 q2*) = *Inl* (*q1*,*q2*)

states (*canonical-separator M q1 q2*) = (*image Inl* (*states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))) ∪ {*Inr q1*, *Inr q2*}
inputs (*canonical-separator M q1 q2*) = *inputs M*
outputs (*canonical-separator M q1 q2*) = *outputs M*
transitions (*canonical-separator M q1 q2*) = *shifted-transitions* (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*)))) ∪ *distinguishing-transitions* (*h-out M*) *q1 q2* (*states* ((*product* (*from-FSM M q1*) (*from-FSM M q2*)))) (*inputs* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
unfolding *canonical-separator.simps canonical-separator′-simps*[*OF* ∗, *of M*] ∗∗∗ ∗∗∗∗ **by** *blast*+
qed

lemma *distinguishing-transitions-alt-def* :
distinguishing-transitions (*h-out M*) *q1 q2 PS* (*inputs M*) =
{(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *PS* ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
∪ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *PS* ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
(**is** *?dts* = *?dl* ∪ *?dr*)
proof −
have ⋀ *t* . *t* ∈ *?dts* ⟹ *t* ∈ *?dl* ∨ *t* ∈ *?dr*
unfolding *distinguishing-transitions-def h-out.simps* **by** *fastforce*
moreover have ⋀ *t* . *t* ∈ *?dl* ∨ *t* ∈ *?dr* ⟹ *t* ∈ *?dts*
proof −
fix *t* **assume** *t* ∈ *?dl* ∨ *t* ∈ *?dr*
then obtain *q1′ q2′* **where** *t-source t* = *Inl* (*q1′*,*q2′*) **and** (*q1′*,*q2′*) ∈ *PS*
by *auto*

consider (*a*) *t* ∈ *?dl* |
(*b*) *t* ∈ *?dr*
using ‹*t* ∈ *?dl* ∨ *t* ∈ *?dr*› **by** *blast*
then show *t* ∈ *?dts* **proof** *cases*
case *a*
then have *t-target t* = *Inr q1* **and** (∃ *q′* . (*q1′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)
and ¬(∃ *q′* . (*q2′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)
using ‹*t-source t* = *Inl* (*q1′*,*q2′*)› **by** *force*+
then have *t-output t* ∈ *h-out M* (*q1′*,*t-input t*) − *h-out M* (*q2′*,*t-input t*)
unfolding *h-out.simps* **by** *blast*
then have *t* ∈ (λ*y*. (*Inl* (*q1′*, *q2′*), *t-input t*, *y*, *Inr q1*)) ‘ (*h-out M* (*q1′*, *t-input t*) − *h-out M* (*q2′*, *t-input t*))
using ‹*t-source t* = *Inl* (*q1′*,*q2′*)› ‹*t-target t* = *Inr q1*›
by (*metis* (*mono-tags*, *lifting*) *imageI surjective-pairing*)
moreover have ((*q1′*,*q2′*),*t-input t*) ∈ *PS* × *inputs M*
using *fsm-transition-input* ‹(∃ *q′* . (*q1′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)›
‹(*q1′*,*q2′*) ∈ *PS*›
by *auto*

ultimately show *?thesis*
unfolding *distinguishing-transitions-def* **by** *fastforce*
next
case *b*
then have *t-target t* = *Inr q2* **and** ¬(∃ *q′* . (*q1′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)
and (∃ *q′* . (*q2′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)
using ‹*t-source t* = *Inl* (*q1′*,*q2′*)› **by** *force+*
then have *t-output t* ∈ *h-out M* (*q2′*,*t-input t*) − *h-out M* (*q1′*,*t-input t*)
unfolding *h-out.simps* **by** *blast*
then have *t* ∈ (λ*y*. (*Inl* (*q1′*, *q2′*), *t-input t*, *y*, *Inr q2*)) ‘ (*h-out M* (*q2′*, *t-input t*) − *h-out M* (*q1′*, *t-input t*))
using ‹*t-source t* = *Inl* (*q1′*,*q2′*)› ‹*t-target t* = *Inr q2*›
by (*metis* (*mono-tags*, *lifting*) *imageI surjective-pairing*)
moreover have ((*q1′*,*q2′*),*t-input t*) ∈ *PS* × *inputs M*
using *fsm-transition-input* ‹(∃ *q′* . (*q2′*,*t-input t*,*t-output t*,*q′*) ∈ *transitions M*)› ‹(*q1′*,*q2′*) ∈ *PS*›
by *auto*
ultimately show *?thesis*
unfolding *distinguishing-transitions-def* **by** *fastforce*
qed
qed
ultimately show *?thesis* **by** *blast*
qed

lemma *distinguishing-transitions-alt-alt-def* :
distinguishing-transitions (*h-out M*) *q1 q2 PS* (*inputs M*) =
{ *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *PS* ∧ *t-target t* = *Inr q1* ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}
∪ { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *PS* ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}

proof −
have {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *PS* ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
= { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *PS* ∧ *t-target t* = *Inr q1* ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}
by *force*
moreover have {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *PS* ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
= { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *PS* ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧

t-output t′ = t-output t) ∧ (∃ *t′* ∈ *transitions M* . *t-source t′ = q2′* ∧ *t-input t′ = t-input t* ∧ *t-output t′ = t-output t*)}
by *force*
ultimately show *?thesis*
unfolding *distinguishing-transitions-alt-def* **by** *force*
qed

lemma *shifted-transitions-alt-def* :
shifted-transitions ts = {(*Inl* (*q1′*,*q2′*), *x*, *y*, (*Inl* (*q1″*,*q2″*))) | *q1′ q2′ x y q1″ q2″* . ((*q1′*,*q2′*), *x*, *y*, (*q1″*,*q2″*)) ∈ *ts*}
unfolding *shifted-transitions-def* **by** *force*

lemma *canonical-separator-transitions-helper* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
(*shifted-transitions* (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))))
∪ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
∪ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
unfolding *canonical-separator-simps*[*OF assms*]
restrict-to-reachable-states-simps
product-simps from-FSM-simps[*OF assms*(*1*)] *from-FSM-simps*[*OF assms*(*2*)]
sup.idem
distinguishing-transitions-alt-def
by *blast*

definition *distinguishing-transitions-left* :: (*′a*, *′b*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a* + *′a*) × *′b* × *′c* × (*′a* × *′a* + *′a*)) *set* **where**
distinguishing-transitions-left M q1 q2 ≡ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ (∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ ¬(∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}
definition *distinguishing-transitions-right* :: (*′a*, *′b*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a* + *′a*) × *′b* × *′c* × (*′a* × *′a* + *′a*)) *set* **where**
distinguishing-transitions-right M q1 q2 ≡ {(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q2*) | *q1′ q2′ x y* . (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ ¬(∃ *q′* . (*q1′*,*x*,*y*,*q′*) ∈ *transitions M*) ∧ (∃ *q′* . (*q2′*,*x*,*y*,*q′*) ∈ *transitions M*)}

definition *distinguishing-transitions-left-alt* :: (*′a*, *′b*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a* + *′a*) × *′b* × *′c* × (*′a* × *′a* + *′a*)) *set* **where**
distinguishing-transitions-left-alt M q1 q2 ≡ { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧

t-target t = *Inr q1* ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}
definition *distinguishing-transitions-right-alt* :: (*′a*, *′b*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a* + *′a*) × *′b* × *′c* × (*′a* × *′a* + *′a*)) *set* **where**
distinguishing-transitions-right-alt M q1 q2 ≡ { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}

definition *shifted-transitions-for* :: (*′a*, *′b*, *′c*) *fsm* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a* + *′a*) × *′b* × *′c* × (*′a* × *′a* + *′a*)) *set* **where**
shifted-transitions-for M q1 q2 ≡ {(*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*)) | *t* . *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}

lemma *shifted-transitions-for-alt-def* :
shifted-transitions-for M q1 q2 = {(*Inl* (*q1′*,*q2′*), *x*, *y*, (*Inl* (*q1′′*,*q2′′*))) | *q1′ q2′ x y q1′′ q2′′* . ((*q1′*,*q2′*), *x*, *y*, (*q1′′*,*q2′′*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
unfolding *shifted-transitions-for-def* **by** *auto*

lemma *distinguishing-transitions-left-alt-alt-def* :
distinguishing-transitions-left M q1 q2 = *distinguishing-transitions-left-alt M q1 q2*
proof −
have ⋀ *t* . *t* ∈ *distinguishing-transitions-left M q1 q2* ⟹ *t* ∈ *distinguishing-transitions-left-alt M q1 q2*
proof −
fix *t* **assume** *t* ∈ *distinguishing-transitions-left M q1 q2*
then obtain *q1′ q2′ x y* **where** *t* = (*Inl* (*q1′*, *q2′*), *x*, *y*, *Inr q1*)
(*q1′*, *q2′*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
(∃ *q′*. (*q1′*, *x*, *y*, *q′*) ∈ *FSM.transitions M*)
(∄ *q′*. (*q2′*, *x*, *y*, *q′*) ∈ *FSM.transitions M*)
unfolding *distinguishing-transitions-left-def* **by** *blast*

have *t-source t* = *Inl* (*q1′*, *q2′*)
using ‹*t* = (*Inl* (*q1′*, *q2′*), *x*, *y*, *Inr q1*)› **by** *auto*
moreover note ‹(*q1′*, *q2′*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))›
moreover have *t-target t* = *Inr q1*
using ‹*t* = (*Inl* (*q1′*, *q2′*), *x*, *y*, *Inr q1*)› **by** *auto*
moreover have (∃ *t′*∈*FSM.transitions M*. *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
using ‹(∃ *q′*. (*q1′*, *x*, *y*, *q′*) ∈ *FSM.transitions M*)› **unfolding** ‹*t* = (*Inl* (*q1′*,

q2'), *x*, *y*, *Inr q1*)› **by** *force*
moreover have ¬(∃ *t'*∈*FSM.transitions M*. *t-source t'* = *q2'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
using ‹(∄ *q'*. (*q2'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*)› **unfolding** ‹*t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q1*)› **by** *force*
ultimately show *t* ∈ *distinguishing-transitions-left-alt M q1 q2*
unfolding *distinguishing-transitions-left-alt-def* **by** *simp*
qed
moreover have ⋀ *t* . *t* ∈ *distinguishing-transitions-left-alt M q1 q2* ⟹ *t* ∈ *distinguishing-transitions-left M q1 q2*
unfolding *distinguishing-transitions-left-alt-def distinguishing-transitions-left-def*

by *fastforce*
ultimately show *?thesis* **by** *blast*
qed

lemma *distinguishing-transitions-right-alt-alt-def* :
distinguishing-transitions-right M q1 q2 = *distinguishing-transitions-right-alt M q1 q2*
proof −
have ⋀ *t* . *t* ∈ *distinguishing-transitions-right M q1 q2* ⟹ *t* ∈ *distinguishing-transitions-right-alt M q1 q2*
proof −
fix *t* **assume** *t* ∈ *distinguishing-transitions-right M q1 q2*
then obtain *q1' q2' x y* **where** *t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q2*)
(*q1'*, *q2'*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
(∄ *q'*. (*q1'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*)
(∃ *q'*. (*q2'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*)
unfolding *distinguishing-transitions-right-def* **by** *blast*

have *t-source t* = *Inl* (*q1'*, *q2'*)
using ‹*t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q2*)› **by** *auto*
moreover note ‹(*q1'*, *q2'*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))›
moreover have *t-target t* = *Inr q2*
using ‹*t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q2*)› **by** *auto*
moreover have ¬(∃ *t'*∈*FSM.transitions M*. *t-source t'* = *q1'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
using ‹(∄ *q'*. (*q1'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*)› **unfolding** ‹*t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q2*)› **by** *force*
moreover have (∃ *t'*∈*FSM.transitions M*. *t-source t'* = *q2'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
using ‹(∃ *q'*. (*q2'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*)› **unfolding** ‹*t* = (*Inl* (*q1'*, *q2'*), *x*, *y*, *Inr q2*)› **by** *force*
ultimately show *t* ∈ *distinguishing-transitions-right-alt M q1 q2*
unfolding *distinguishing-transitions-right-def distinguishing-transitions-right-alt-def* **by** *simp*

qed
moreover have $\bigwedge$ *t* . *t* ∈ *distinguishing-transitions-right-alt M q1 q2* ⟹ *t* ∈ *distinguishing-transitions-right M q1 q2*
unfolding *distinguishing-transitions-right-def distinguishing-transitions-right-alt-def*
by *fastforce*
ultimately show *?thesis*
by *blast*
qed

lemma *canonical-separator-transitions-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
{(*Inl* (*q1'*,*q2'*), *x*, *y*, (*Inl* (*q1''*,*q2''*))) | *q1' q2' x y q1'' q2''* . ((*q1'*,*q2'*), *x*, *y*, (*q1''*,*q2''*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
∪ (*distinguishing-transitions-left M q1 q2*)
∪ (*distinguishing-transitions-right M q1 q2*)
unfolding *canonical-separator-transitions-helper*[*OF assms*]
shifted-transitions-alt-def
distinguishing-transitions-left-def
distinguishing-transitions-right-def **by** *simp*

lemma *canonical-separator-transitions-alt-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *transitions* (*canonical-separator M q1 q2*) =
(*shifted-transitions-for M q1 q2*)
∪ (*distinguishing-transitions-left-alt M q1 q2*)
∪ (*distinguishing-transitions-right-alt M q1 q2*)
proof −
have ∗: (*shift-Inl* ‘
{*t* ∈ *FSM.transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)).
t-source t ∈ *reachable-states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))})
= {(*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*)) |*t*.
t ∈ *FSM.transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) ∧
t-source t ∈ *reachable-states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))}
by *blast*

show *?thesis*
unfolding *canonical-separator-simps*[*OF assms*]
shifted-transitions-def
restrict-to-reachable-states-simps
product-simps from-FSM-simps[*OF assms*(*1*)] *from-FSM-simps*[*OF assms*(*2*)]
sup.idem

distinguishing-transitions-alt-alt-def
shifted-transitions-for-def
*

distinguishing-transitions-left-alt-def
distinguishing-transitions-right-alt-def
by *blast*
qed

34.1.2 State Separators as Submachines of Canonical Separators

definition *is-state-separator-from-canonical-separator* :: $(('a \times 'a) + 'a, 'b, 'c)$ *fsm* $\Rightarrow 'a \Rightarrow 'a \Rightarrow (('a \times 'a) + 'a, 'b, 'c)$ *fsm* $\Rightarrow$ *bool* **where**
is-state-separator-from-canonical-separator CSep q1 q2 S = (
is-submachine S CSep
$\wedge$ *single-input S*
$\wedge$ *acyclic S*
$\wedge$ *deadlock-state S* (*Inr q1*)
$\wedge$ *deadlock-state S* (*Inr q2*)
$\wedge$ ((*Inr q1*) $\in$ *reachable-states S*)
$\wedge$ ((*Inr q2*) $\in$ *reachable-states S*)
$\wedge$ ($\forall$ *q* $\in$ *reachable-states S* . (*q* $\neq$ *Inr q1* $\wedge$ *q* $\neq$ *Inr q2*) $\longrightarrow$ (*isl q* $\wedge$ $\neg$ *deadlock-state S q*))
$\wedge$ ($\forall$ *q* $\in$ *reachable-states S* . $\forall$ *x* $\in$ (*inputs CSep*) . ($\exists$ *t* $\in$ *transitions S* . *t-source t* = *q* $\wedge$ *t-input t* = *x*) $\longrightarrow$ ($\forall$ *t'* $\in$ *transitions CSep* . *t-source t'* = *q* $\wedge$ *t-input t'* = *x* $\longrightarrow$ *t'* $\in$ *transitions S*))
)

34.1.3 Canonical Separator Properties

lemma *is-state-separator-from-canonical-separator-simps* :
assumes *is-state-separator-from-canonical-separator CSep q1 q2 S*
shows *is-submachine S CSep*
and *single-input S*
and *acyclic S*
and *deadlock-state S* (*Inr q1*)
and *deadlock-state S* (*Inr q2*)
and ((*Inr q1*) $\in$ *reachable-states S*)
and ((*Inr q2*) $\in$ *reachable-states S*)
and $\bigwedge$ *q* . *q* $\in$ *reachable-states S* $\Longrightarrow$ *q* $\neq$ *Inr q1* $\Longrightarrow$ *q* $\neq$ *Inr q2* $\Longrightarrow$ (*isl q* $\wedge$ $\neg$ *deadlock-state S q*)
and $\bigwedge$ *q x t* . *q* $\in$ *reachable-states S* $\Longrightarrow$ *x* $\in$ (*inputs CSep*) $\Longrightarrow$ ($\exists$ *t* $\in$ *transitions S* . *t-source t* = *q* $\wedge$ *t-input t* = *x*) $\Longrightarrow$ *t* $\in$ *transitions CSep* $\Longrightarrow$ *t-source t* = *q* $\Longrightarrow$ *t-input t* = *x* $\Longrightarrow$ *t* $\in$ *transitions S*
using *assms* **unfolding** *is-state-separator-from-canonical-separator-def* **by** *blast+*

lemma *is-state-separator-from-canonical-separator-initial* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*

and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *initial A* = *Inl* (*q1*,*q2*)
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)]
using *canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)] **by** *auto*

lemma *path-shift-Inl* :
assumes (*image shift-Inl* (*transitions M*)) ⊆ (*transitions C*)
and ⋀ *t* . *t* ∈ (*transitions C*) ⟹ *isl* (*t-target t*) ⟹ ∃ *t′* ∈ *transitions M* . *t* = (*Inl* (*t-source t′*), *t-input t′*, *t-output t′*, *Inl* (*t-target t′*))
and *initial C* = *Inl* (*initial M*)
and (*inputs C*) = (*inputs M*)
and (*outputs C*) = (*outputs M*)
shows *path M* (*initial M*) *p* = *path C* (*initial C*) (*map shift-Inl p*)
proof (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc t p*)

have *path M* (*initial M*) (*p*@[*t*]) ⟹ *path C* (*initial C*) (*map shift-Inl* (*p*@[*t*]))
proof −
assume *path M* (*initial M*) (*p*@[*t*])
then have *path M* (*initial M*) *p* **by** *auto*
then have *path C* (*initial C*) (*map shift-Inl p*) **using** *snoc.IH*
by *auto*

have *t-source t* = *target* (*initial M*) *p*
using ‹*path M* (*initial M*) (*p*@[*t*])› **by** *auto*
then have *t-source* (*shift-Inl t*) = *target* (*Inl* (*initial M*)) (*map shift-Inl p*)
by (*cases p rule*: *rev-cases*; *auto*)
then have *t-source* (*shift-Inl t*) = *target* (*initial C*) (*map shift-Inl p*)
using *assms*(*3*) **by** *auto*
moreover have *target* (*initial C*) (*map shift-Inl p*) ∈ *states C*
using *path-target-is-state*[*OF* ‹*path C* (*initial C*) (*map shift-Inl p*)›] **by** *assumption*
ultimately have *t-source* (*shift-Inl t*) ∈ *states C*
by *auto*
moreover have *t* ∈ *transitions M*
using ‹*path M* (*initial M*) (*p*@[*t*])› **by** *auto*
ultimately have (*shift-Inl t*) ∈ *transitions C*
using *assms* **by** *auto*

show *path C* (*initial C*) (*map shift-Inl* (*p*@[*t*]))
using *path-append* [*OF* ‹*path C* (*initial C*) (*map shift-Inl p*)›, *of* [*shift-Inl t*]]
using ‹(*shift-Inl t*) ∈ *transitions C*› ‹*t-source* (*shift-Inl t*) = *target* (*initial C*) (*map shift-Inl p*)›
using *single-transition-path* **by** *force*

```
  qed

  moreover have path C (initial C) (map shift-Inl (p@[t])) ⟹ path M (initial
M) (p@[t])
  proof −
    assume path C (initial C) (map shift-Inl (p@[t]))
    then have path C (initial C) (map shift-Inl p) by auto
    then have path M (initial M) p using snoc.IH
      by blast

    have t-source (shift-Inl t) = target (initial C) (map shift-Inl p)
      using ‹path C (initial C) (map shift-Inl (p@[t]))› by auto
    then have t-source (shift-Inl t) = target (Inl (initial M)) (map shift-Inl p)
      using assms(3) by (cases p rule: rev-cases; auto)
    then have t-source t = target (initial M) p
      by (cases p rule: rev-cases; auto)
    moreover have target (initial M) p ∈ states M
      using path-target-is-state[OF ‹path M (initial M) p›] by assumption
    ultimately have t-source t ∈ states M
      by auto
    moreover have shift-Inl t ∈ transitions C
      using ‹path C (initial C) (map shift-Inl (p@[t]))› by auto
    moreover have isl (t-target (shift-Inl t))
      by auto
    ultimately have t ∈ transitions M using assms by fastforce

    show path M (initial M) (p@[t])
      using path-append [OF ‹path M (initial M) p›, of [t]]
            single-transition-path[OF ‹t ∈ transitions M›]
            ‹t-source t = target (initial M) p› by auto
  qed

  ultimately show ?case
    by linarith
qed


lemma canonical-separator-product-transitions-subset :
  assumes q1 ∈ states M and q2 ∈ states M
   shows image shift-Inl (transitions (product (from-FSM M q1) (from-FSM M
q2))) ⊆ (transitions (canonical-separator M q1 q2))
 unfolding canonical-separator-simps[OF assms] shifted-transitions-def restrict-to-reachable-states-simps

  by blast


lemma canonical-separator-transition-targets :
  assumes t ∈ (transitions (canonical-separator M q1 q2))
  and q1 ∈ states M
```

and *q2* ∈ *states M*
shows *isl* (*t-target t*) ⟹ *t* ∈ {(*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*)) | *t* . *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
and *t-target t* = *Inr q1* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)
and *t-target t* = *Inr q2* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ (*distinguishing-transitions-right-alt M q1 q2*)
and *isl* (*t-target t*) ∨ *t-target t* = *Inr q1* ∨ *t-target t* = *Inr q2*
unfolding *shifted-transitions-for-def*
distinguishing-transitions-left-alt-def
distinguishing-transitions-right-alt-def
proof −
let *?shftd* = {(*Inl* (*t-source t*),*t-input t*, *t-output t*, *Inl* (*t-target t*)) | *t* . *t* ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))}
let *?dl* = { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ *t-target t* = *Inr q1* ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}
let *?dr* = { *t* . ∃ *q1′ q2′* . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) ∧ *t-target t* = *Inr q2* ∧ ¬(∃ *t′* ∈ *transitions M* . *t-source t′* = *q1′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*) ∧ (∃ *t′* ∈ *transitions M* . *t-source t′* = *q2′* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)}

have *t* ∈ *?shftd* ∪ *?dl* ∪ *?dr*
using *assms*(*1*)
unfolding *canonical-separator-transitions-alt-def*[*OF assms*(*2*,*3*)]
shifted-transitions-for-def
distinguishing-transitions-left-alt-def
distinguishing-transitions-right-alt-def
by *force*

moreover have *p1*: ⋀ *t′* . *t′* ∈ *?shftd* ⟹ *isl* (*t-target t′*)
and *p2*: ⋀ *t′* . *t′* ∈ *?dl* ⟹ *t-target t′* = *Inr q1*
and *p3*: ⋀ *t′* . *t′* ∈ *?dr* ⟹ *t-target t′* = *Inr q2*
by *auto*
ultimately show *isl* (*t-target t*) ∨ *t-target t* = *Inr q1* ∨ *t-target t* = *Inr q2*
by *fast*

show *isl* (*t-target t*) ⟹ *t* ∈ *?shftd*
proof −
assume *isl* (*t-target t*)
then have *t-target t* ≠ *Inr q1* **and** *t-target t* ≠ *Inr q2* **by** *auto*
then have *t* ∉ *?dl* **and** *t* ∉ *?dr* **by** *force*+
then show *?thesis* **using** ‹*t* ∈ *?shftd* ∪ *?dl* ∪ *?dr*› **by** *fastforce*
qed

show *t-target t* = *Inr q1* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ *?dl*

proof −
assume *t-target t* = *Inr q1* **and** *q1* ≠ *q2*
then have ¬ *isl* (*t-target t*) **and** *t-target t* ≠ *Inr q2* **by** *auto*
then have *t* ∉ *?shftd* **and** *t* ∉ *?dr* **by** *force+*
then show *?thesis* **using** ‹*t* ∈ *?shftd* ∪ *?dl* ∪ *?dr*› **by** *fastforce*
qed

show *t-target t* = *Inr q2* ⟹ *q1* ≠ *q2* ⟹ *t* ∈ *?dr*
proof −
assume *t-target t* = *Inr q2* **and** *q1* ≠ *q2*
then have ¬ *isl* (*t-target t*) **and** *t-target t* ≠ *Inr q1* **by** *auto*
then have *t* ∉ *?shftd* **and** *t* ∉ *?dl* **by** *force+*
then show *?thesis* **using** ‹*t* ∈ *?shftd* ∪ *?dl* ∪ *?dr*› **by** *fastforce*
qed
qed

lemma *canonical-separator-path-shift* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *p*
= *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) (*map shift-Inl p*)
proof −
let *?C* = (*canonical-separator M q1 q2*)
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))
let *?PR* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

have (*inputs ?C*) = (*inputs ?P*)
and (*outputs ?C*) = (*outputs ?P*)
unfolding *canonical-separator-simps*(*3*,*4*)[*OF assms*] **using** *assms* **by** *auto*

have *p1*: *shift-Inl* ‘
FSM.transitions
((*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)))
⊆ *FSM.transitions* (*canonical-separator M q1 q2*)
using *canonical-separator-product-transitions-subset*[*OF assms*]
unfolding *restrict-to-reachable-states-simps* **by** *assumption*

have *p2*: (⋀*t*. *t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*) ⟹
isl (*t-target t*) ⟹
∃ *t'*∈*FSM.transitions*
((*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))).
t = *shift-Inl t'*)
using *canonical-separator-transition-targets*(*1*)[*OF - assms*] **unfolding** *restrict-to-reachable-states-simps* **by** *fastforce*

have *path ?PR* (*initial ?PR*) *p* = *path ?C* (*initial ?C*) (*map shift-Inl p*)

using *path-shift-Inl*[*of ?PR ?C*, *OF p1 p2*]
unfolding *restrict-to-reachable-states-simps canonical-separator-simps*(*1*,*2*,*3*,*4*)[*OF assms*] **using** *assms* **by** *auto*
moreover have *path ?P* (*initial ?P*) *p* = *path ?PR* (*initial ?PR*) *p*
unfolding *restrict-to-reachable-states-simps*
restrict-to-reachable-states-path[*OF reachable-states-initial*]
by *simp*
ultimately show *?thesis*
by *simp*
qed

lemma *canonical-separator-t-source-isl* :
assumes *t* ∈ (*transitions* (*canonical-separator M q1 q2*))
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *isl* (*t-source t*)
using *assms*(*1*)
unfolding *canonical-separator-transitions-alt-def*[*OF assms*(*2*,*3*)]
shifted-transitions-for-def
distinguishing-transitions-left-alt-def
distinguishing-transitions-right-alt-def
by *force*

lemma *canonical-separator-path-from-shift* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p*
and *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows ∃ *p′* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *p′*
∧ *p* = (*map shift-Inl p′*)
using *assms*(*1*,*2*) **proof** (*induction p rule*: *rev-induct*)
case *Nil*
show *?case* **using** *canonical-separator-path-shift*[*OF assms*(*3*,*4*), *of* []] **by** *fast*
next
case (*snoc t p*)
then have *isl* (*t-target t*) **by** *auto*

let *?C* = (*canonical-separator M q1 q2*)
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

have *t* ∈ *transitions ?C* **and** *t-source t* = *target* (*initial ?C*) *p*
using *snoc.prems* **by** *auto*
then have *isl* (*t-source t*)
using *canonical-separator-t-source-isl*[*of t M q1 q2*, *OF - assms*(*3*,*4*)] **by** *blast*

then have *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
using ‹*t-source t* = *target* (*initial ?C*) *p*› **by** *auto*

have *path ?C* (*initial ?C*) *p* **using** *snoc.prems* **by** *auto*
then obtain *p′* **where** *path ?P* (*initial ?P*) *p′*
and *p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p′*
using *snoc.IH*[*OF* - ‹*isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)›] **by** *blast*
then have *target* (*initial ?C*) *p* = *Inl* (*target* (*initial ?P*) *p′*)
proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis*
unfolding *target.simps visited-states.simps* **using** ‹*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p′*› *canonical-separator-simps*(*1*)[*OF assms*(*3*,*4*)]
by (*simp add*: *assms*(*3*) *assms*(*4*))
next
case (*snoc ys y*)
then show *?thesis*
unfolding *target.simps visited-states.simps* **using** ‹*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p′*› **by** (*cases p′ rule*: *rev-cases*; *auto*)
qed

obtain *t′* **where** *t′* ∈ *transitions ?P*
and *t* = (*Inl* (*t-source t′*), *t-input t′*, *t-output t′*, *Inl* (*t-target t′*))
using *canonical-separator-transition-targets*(*1*)[*OF* ‹*t* ∈ *transitions ?C*› *assms*(*3*,*4*) ‹*isl* (*t-target t*)›]
by *blast*

have *path ?P* (*initial ?P*) (*p′*@[*t′*])
by (*metis* ‹*path* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) (*FSM.initial* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))) *p′*›
‹*t* = *shift-Inl t′*› ‹*t′* ∈ *FSM.transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))›
‹*t-source t* = *target* (*FSM.initial* (*canonical-separator M q1 q2*)) *p*›
‹*target* (*FSM.initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*target* (*FSM.initial* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))) *p′*)›
fst-conv path-append-transition sum.inject(*1*))
moreover have *p*@[*t*] = *map shift-Inl* (*p′*@[*t′*])
using ‹*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p′*›

‹*t* = (*Inl* (*t-source t′*), *t-input t′*, *t-output t′*, *Inl* (*t-target t′*))›
by *auto*
ultimately show *?case*
by *meson*
qed

lemma *shifted-transitions-targets* :

assumes $t \in$ (*shifted-transitions ts*)
shows *isl* (*t-target t*)
using *assms* **unfolding** *shifted-transitions-def* **by** *force*

lemma *distinguishing-transitions-left-sources-targets* :
assumes $t \in$ (*distinguishing-transitions-left-alt M q1 q2*)
and *q2* $\in$ *states M*
obtains *q1′ q2′ t′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
q1′ $\in$ *states M*
q2′ $\in$ *states M*
t′ $\in$ *transitions M*
t-source t′ = *q1′*
t-input t′ = *t-input t*
t-output t′ = *t-output t*
$\neg$ ($\exists$ *t″*$\in$ *transitions M*. *t-source t″* = *q2′* $\wedge$ *t-input t″* = *t-input t* $\wedge$ *t-output t″* = *t-output t*)
t-target t = *Inr q1*
using *assms*(*1*) *assms*(*2*) *fsm-transition-source path-target-is-state*
unfolding *distinguishing-transitions-left-alt-def*
by *fastforce*

lemma *distinguishing-transitions-right-sources-targets* :
assumes $t \in$ (*distinguishing-transitions-right-alt M q1 q2*)
and *q1* $\in$ *states M*
obtains *q1′ q2′ t′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
q1′ $\in$ *states M*
q2′ $\in$ *states M*
t′ $\in$ *transitions M*
t-source t′ = *q2′*
t-input t′ = *t-input t*
t-output t′ = *t-output t*
$\neg$ ($\exists$ *t″*$\in$ *transitions M*. *t-source t″* = *q1′* $\wedge$ *t-input t″* = *t-input t* $\wedge$ *t-output t″* = *t-output t*)
t-target t = *Inr q2*
using *assms*(*1*) *assms*(*2*) *fsm-transition-source path-target-is-state*
unfolding *distinguishing-transitions-right-alt-def*
by *fastforce*

lemma *product-from-transition-split* :
assumes $t \in$ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
shows ($\exists$ *t′*$\in$ *transitions M*. *t-source t′* = *fst* (*t-source t*) $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*)
and ($\exists$ *t′*$\in$ *transitions M*. *t-source t′* = *snd* (*t-source t*) $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*)

using *product-transition-split-ob*[*OF assms*(*1*)]
unfolding *product-transitions-alt-def from-FSM-simps*[*OF assms*(*2*)] *from-FSM-simps*[*OF assms*(*3*)] **by** *blast+*

lemma *shifted-transitions-underlying-transition* :
assumes *tS* ∈ *shifted-transitions-for M q1 q2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
obtains *t* **where** *tS* = (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))
and *t* ∈ (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
and (∃ *t'*∈(*transitions M*).
t-source t' = *fst* (*t-source t*) ∧
t-input t' = *t-input t* ∧ *t-output t'* = *t-output t*)
and (∃ *t'*∈(*transitions M*).
t-source t' = *snd* (*t-source t*) ∧
t-input t' = *t-input t* ∧ *t-output t'* = *t-output t*)
proof −
obtain *t* **where** *tS* = (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))
and ∗: *t* ∈ (*transitions* ((*product* (*from-FSM M q1*) (*from-FSM M q2*))))
using *assms* **unfolding** *shifted-transitions-for-def shifted-transitions-def restrict-to-reachable-states-simps* **by** *blast*
moreover have (∃ *t'*∈(*transitions M*).
t-source t' = *fst* (*t-source t*) ∧
t-input t' = *t-input t* ∧ *t-output t'* = *t-output t*)
using *product-from-transition-split*(*1*)[*OF* - *assms*(*2*,*3*)]
∗
unfolding *restrict-to-reachable-states-simps* **by** *blast*
moreover have (∃ *t'*∈(*transitions M*).
t-source t' = *snd* (*t-source t*) ∧
t-input t' = *t-input t* ∧ *t-output t'* = *t-output t*)
using *product-from-transition-split*(*2*)[*OF* - *assms*(*2*,*3*)]
∗
unfolding *restrict-to-reachable-states-simps* **by** *blast*
ultimately show *?thesis*
using *that* **by** *blast*
qed

lemma *shifted-transitions-observable-against-distinguishing-transitions-left* :
assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-left M q1 q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
using *assms*(*1*,*2*)
unfolding *product-transitions-def from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)]

shifted-transitions-for-def distinguishing-transitions-left-def
by *force*

lemma *shifted-transitions-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
using *assms*
unfolding *product-transitions-def from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)]
shifted-transitions-for-def distinguishing-transitions-right-def
by *force*

lemma *distinguishing-transitions-left-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
shows ¬ (*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*)
using *assms*
unfolding *distinguishing-transitions-left-def distinguishing-transitions-right-def*
by *force*

lemma *distinguishing-transitions-left-observable-against-distinguishing-transitions-left* :
assumes *t1* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-left M q1 q2*)
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
using *assms* **unfolding** *distinguishing-transitions-left-def* **by** *force*

lemma *distinguishing-transitions-right-observable-against-distinguishing-transitions-right* :
assumes *t1* ∈ (*distinguishing-transitions-right M q1 q2*)
and *t2* ∈ (*distinguishing-transitions-right M q1 q2*)
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
using *assms* **unfolding** *distinguishing-transitions-right-def* **by** *force*

lemma *shifted-transitions-observable-against-shifted-transitions* :

assumes *t1* ∈ (*shifted-transitions-for M q1 q2*)
and *t2* ∈ (*shifted-transitions-for M q1 q2*)
and *observable M*
and *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*
shows *t1* = *t2*
proof −
obtain *t1′* **where** *d1*: *t1* = (*Inl* (*t-source t1′*), *t-input t1′*, *t-output t1′*, *Inl* (*t-target t1′*))
and *h1*: *t1′* ∈ (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
using *assms*(*1*) **unfolding** *shifted-transitions-for-def* **by** *auto*

obtain *t2′* **where** *d2*: *t2* = (*Inl* (*t-source t2′*), *t-input t2′*, *t-output t2′*, *Inl* (*t-target t2′*))
and *h2*: *t2′* ∈ (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
using *assms*(*2*) **unfolding** *shifted-transitions-for-def* **by** *auto*

have *observable* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *from-FSM-observable*[*OF assms*(*3*)]
product-observable
by *metis*

then have *t1′* = *t2′*
using *d1 d2 h1 h2* ‹*t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*›
by (*metis fst-conv observable.elims*(*2*) *prod.expand snd-conv sum.inject*(*1*))
then show *?thesis* **using** *d1 d2* **by** *auto*
qed

lemma *canonical-separator-observable* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *observable* (*canonical-separator M q1 q2*) (**is** *observable ?CSep*)
proof −

have ⋀ *t1 t2* . *t1* ∈ (*transitions ?CSep*) ⟹
t2 ∈ (*transitions ?CSep*) ⟹
t-source t1 = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2* ⟹ *t-target t1* = *t-target t2*
proof −
fix *t1 t2* **assume** *t1* ∈ (*transitions ?CSep*)
and *t2* ∈ (*transitions ?CSep*)
and ∗: *t-source t1* = *t-source t2* ∧ *t-input t1* = *t-input t2* ∧ *t-output t1* = *t-output t2*

moreover have *transitions ?CSep = shifted-transitions-for M q1 q2* ∪
distinguishing-transitions-left M q1 q2 ∪
distinguishing-transitions-right M q1 q2
using *canonical-separator-transitions-alt-def*[*OF assms*(*2*,*3*)]
unfolding *distinguishing-transitions-left-alt-alt-def distinguishing-transitions-right-alt-alt-def*
by *assumption*

ultimately consider *t1* ∈ *shifted-transitions-for M q1 q2* ∧ *t2* ∈ *shifted-transitions-for M q1 q2*
| *t1* ∈ *shifted-transitions-for M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-left M q1 q2*
| *t1* ∈ *shifted-transitions-for M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-right M q1 q2*
| *t1* ∈ *distinguishing-transitions-left M q1 q2* ∧ *t2* ∈ *shifted-transitions-for M q1 q2*
| *t1* ∈ *distinguishing-transitions-left M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-left M q1 q2*
| *t1* ∈ *distinguishing-transitions-left M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-right M q1 q2*
| *t1* ∈ *distinguishing-transitions-right M q1 q2* ∧ *t2* ∈ *shifted-transitions-for M q1 q2*
| *t1* ∈ *distinguishing-transitions-right M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-left M q1 q2*
| *t1* ∈ *distinguishing-transitions-right M q1 q2* ∧ *t2* ∈ *distinguishing-transitions-right M q1 q2*
by *force*
then show *t-target t1* = *t-target t2* **proof** *cases*
case *1*
then show *?thesis* **using** *shifted-transitions-observable-against-shifted-transitions*[*of t1 M q1 q2 t2*, *OF - - assms*(*1*) ∗] **by** *fastforce*
next
case *2*
then show *?thesis* **using** *shifted-transitions-observable-against-distinguishing-transitions-left*[*OF - - assms*(*2*,*3*), *of t1 t2*] ∗ **by** *fastforce*
next
case *3*
then show *?thesis* **using** *shifted-transitions-observable-against-distinguishing-transitions-right*[*OF - - assms*(*2*,*3*), *of t1 t2*] ∗ **by** *fastforce*
next
case *4*
then show *?thesis* **using** *shifted-transitions-observable-against-distinguishing-transitions-left*[*OF - - assms*(*2*,*3*), *of t2 t1*] ∗ **by** *fastforce*
next
case *5*
then show *?thesis* **using** ∗ **unfolding** *distinguishing-transitions-left-def* **by** *fastforce*
next
case *6*
then show *?thesis* **using** ∗ **unfolding** *distinguishing-transitions-left-def dis-*

tinguishing-transitions-right-def **by** *fastforce*
next
case *7*
then show *?thesis* **using** *shifted-transitions-observable-against-distinguishing-transitions-right*[*OF - - assms*(*2*,*3*), *of t2 t1*] ∗ **by** *fastforce*
next
case *8*
then show *?thesis* **using** ∗ **unfolding** *distinguishing-transitions-left-def distinguishing-transitions-right-def* **by** *fastforce*
next
case *9*
then show *?thesis* **using** ∗ **unfolding** *distinguishing-transitions-right-def* **by** *fastforce*
qed
qed
then show *?thesis* **unfolding** *observable.simps* **by** *blast*
qed

lemma *canonical-separator-targets-ineq* :
assumes $t \in$ *transitions* (*canonical-separator M q1 q2*)
and *q1* $\in$ *states M* **and** *q2* $\in$ *states M* **and** *q1* $\neq$ *q2*
shows *isl* (*t-target t*) $\Longrightarrow$ $t \in$ (*shifted-transitions-for M q1 q2*)
and *t-target t* = *Inr q1* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-left M q1 q2*)
and *t-target t* = *Inr q2* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-right M q1 q2*)
proof −
show *isl* (*t-target t*) $\Longrightarrow$ $t \in$ (*shifted-transitions-for M q1 q2*)
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *canonical-separator-transition-targets*(*1*) *shifted-transitions-for-def*)
show *t-target t* = *Inr q1* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-left M q1 q2*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *canonical-separator-transition-targets*(*2*) *distinguishing-transitions-left-alt-alt-def*)
show *t-target t* = *Inr q2* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-right M q1 q2*)
by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *canonical-separator-transition-targets*(*3*) *distinguishing-transitions-right-alt-alt-def*)
qed

lemma *canonical-separator-targets-observable* :
assumes $t \in$ *transitions* (*canonical-separator M q1 q2*)
and *q1* $\in$ *states M* **and** *q2* $\in$ *states M* **and** *q1* $\neq$ *q2*
shows *isl* (*t-target t*) $\Longrightarrow$ $t \in$ (*shifted-transitions-for M q1 q2*)
and *t-target t* = *Inr q1* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-left M q1 q2*)
and *t-target t* = *Inr q2* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-right M q1 q2*)
proof −
show *isl* (*t-target t*) $\Longrightarrow$ $t \in$ (*shifted-transitions-for M q1 q2*)
by (*metis assms canonical-separator-targets-ineq*(*1*))
show *t-target t* = *Inr q1* $\Longrightarrow$ $t \in$ (*distinguishing-transitions-left M q1 q2*)
by (*metis assms canonical-separator-targets-ineq*(*2*))

show *t-target t* = *Inr q2* $\Longrightarrow$ *t* $\in$ (*distinguishing-transitions-right M q1 q2*)
by (*metis assms canonical-separator-targets-ineq*(*3*))
qed

lemma *canonical-separator-maximal-path-distinguishes-left* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S* (**is** *is-state-separator-from-canonical-separator ?C q1 q2 S*)
and *path S* (*initial S*) *p*
and *target* (*initial S*) *p* = *Inr q1*
and *observable M*
and *q1* $\in$ *states M* **and** *q2* $\in$ *states M* **and** *q1* $\neq$ *q2*
shows *p-io p* $\in$ *LS M q1* − *LS M q2*
proof (*cases p rule*: *rev-cases*)
case *Nil*
then have *initial S* = *Inr q1* **using** *assms*(*3*) **by** *auto*
then have *initial ?C* = *Inr q1*
using *assms*(*1*) *assms*(*5*) *assms*(*6*) *is-state-separator-from-canonical-separator-initial* **by** *fastforce*
then show *?thesis* **using** *canonical-separator-simps*(*1*) *Inr-Inl-False*
using *assms*(*5*) *assms*(*6*) **by** *fastforce*
next
case (*snoc p′ t*)
then have *path S* (*initial S*) (*p′*@[*t*])
using *assms*(*2*) **by** *auto*
then have *t* $\in$ *transitions S* **and** *t-source t* = *target* (*initial S*) *p′* **by** *auto*

have *path ?C* (*initial ?C*) (*p′*@[*t*])
using ‹*path S* (*initial S*) (*p′*@[*t*])› *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*] **by** (*meson submachine-path-initial*)
then have *path ?C* (*initial ?C*) (*p′*) **and** *t* $\in$ *transitions ?C*
by *auto*

have *isl* (*target* (*initial S*) *p′*)
proof (*rule ccontr*)
assume ¬ *isl* (*target* (*initial S*) *p′*)
moreover have *target* (*initial S*) *p′* $\in$ *states S*
using ‹*path S* (*initial S*) (*p′*@[*t*])› **by** *auto*
ultimately have *target* (*initial S*) *p′* = *Inr q1* $\vee$ *target* (*initial S*) *p′* = *Inr q2*
using ‹*t* $\in$ *FSM.transitions* (*canonical-separator M q1 q2*)› ‹*t-source t* = *target* (*FSM.initial S*) *p′*› *assms*(*5*) *assms*(*6*) *canonical-separator-t-source-isl* **by** *fastforce*
moreover have *deadlock-state S* (*Inr q1*) **and** *deadlock-state S* (*Inr q2*)
using *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*] **by** *presburger*+
ultimately show *False*
using ‹*t* $\in$ *transitions S*› ‹*t-source t* = *target* (*initial S*) *p′*› **unfolding** *deadlock-state.simps*

by *metis*
qed
then obtain *q1′ q2′* **where** *target* (*initial S*) *p′* = *Inl* (*q1′*,*q2′*) **using** *isl-def prod.collapse* **by** *metis*
then have *isl* (*target* (*initial ?C*) *p′*)
using *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*]
by (*metis* (*no-types*, *lifting*) *Nil-is-append-conv assms*(*2*) *isl-def list.distinct*(*1*) *list.sel*(*1*) *path.cases snoc submachine-path-initial*)

obtain *pC* **where** *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *pC*
and *p′* = *map shift-Inl pC*
by (*metis* (*mono-tags*, *lifting*) ‹*isl* (*target* (*FSM.initial* (*canonical-separator M q1 q2*)) *p′*)›
‹*path* (*canonical-separator M q1 q2*) (*FSM.initial* (*canonical-separator M q1 q2*)) *p′*›
assms(*5*) *assms*(*6*) *canonical-separator-path-from-shift*)
then have *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *pC*
by (*simp add*: *assms*(*5*) *assms*(*6*))
then have *path* (*from-FSM M q1*) *q1* (*left-path pC*) **and** *path* (*from-FSM M q2*) *q2* (*right-path pC*)
using *product-path*[*of from-FSM M q1 from-FSM M q2 q1 q2 pC*] **by** *presburger+*

have *path M q1* (*left-path pC*)
using *from-FSM-path*[*OF assms*(*5*) ‹*path* (*from-FSM M q1*) *q1* (*left-path pC*)›] **by** *assumption*
have *path M q2* (*right-path pC*)
using *from-FSM-path*[*OF assms*(*6*) ‹*path* (*from-FSM M q2*) *q2* (*right-path pC*)›] **by** *assumption*

have *t-target t* = *Inr q1*
using ‹*path S* (*initial S*) (*p′*@[*t*])› *snoc assms*(*3*) **by** *auto*
then have *t* ∈ (*distinguishing-transitions-left M q1 q2*)
using *canonical-separator-targets-ineq*(*2*)[*OF* ‹*t* ∈ *transitions ?C*› *assms*(*5*,*6*,*7*)] **by** *auto*
then have *t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)
using *distinguishing-transitions-left-alt-alt-def* **by** *force*

have *t-source t* = *Inl* (*q1′*,*q2′*)
using ‹*target* (*initial S*) *p′* = *Inl* (*q1′*,*q2′*)› ‹*t-source t* = *target* (*initial S*) *p′*› **by** *auto*

then obtain *t′* **where** *q1′* ∈ *states M*
and *q2′* ∈ *states M*
and *t′* ∈ *transitions M*
and *t-source t′* = *q1′*
and *t-input t′* = *t-input t*

and *t-output t′ = t-output t*
and ¬ (∃ *t′′*∈ *transitions M. t-source t′′ = q2′* ∧ *t-input t′′ = t-input t* ∧ *t-output t′′ = t-output t*)
using ‹*t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)› *assms*(*5*,*6*) *fsm-transition-source path-target-is-state*
unfolding *distinguishing-transitions-left-alt-def reachable-states-def* **by** *fastforce*

have *initial S = Inl* (*q1*,*q2*)
by (*meson assms*(*1*) *assms*(*5*) *assms*(*6*) *is-state-separator-from-canonical-separator-initial*)
have *length p′ = length pC*
using ‹*p′ = map shift-Inl pC*› **by** *auto*
then have *target* (*initial S*) *p′ = Inl* (*target* (*q1*,*q2*) *pC*)
using ‹*p′ = map shift-Inl pC*› ‹*initial S = Inl* (*q1*,*q2*)› **by** (*induction p′ pC rule*: *list-induct2*; *auto*)
then have *target* (*q1*,*q2*) *pC* = (*q1′*,*q2′*)
using ‹*target* (*initial S*) *p′ = Inl* (*q1′*,*q2′*)› **by** *auto*
then have *target q2* (*right-path pC*) = *q2′*
using *product-target-split*(*2*) **by** *fastforce*
then have ¬ (∃ *t′*∈ *transitions M. t-source t′ = target q2* (*right-path pC*) ∧ *t-input t′ = t-input t* ∧ *t-output t′ = t-output t*)
using ‹¬ (∃ *t′*∈ *transitions M. t-source t′ = q2′* ∧ *t-input t′ = t-input t* ∧ *t-output t′ = t-output t*)› **by** *blast*

have *target q1* (*left-path pC*) = *q1′*
using ‹*target* (*q1*,*q2*) *pC* = (*q1′*,*q2′*)› *product-target-split*(*1*) **by** *fastforce*
then have *path M q1* ((*left-path pC*)@[*t′*])
using ‹*path M q1* (*left-path pC*)› ‹*t′* ∈ *transitions M*› ‹*t-source t′ = q1′*›
by (*simp add*: *path-append-transition*)
then have *p-io* ((*left-path pC*)@[*t′*]) ∈ *LS M q1*
unfolding *LS.simps* **by** *force*
moreover have *p-io p′ = p-io* (*left-path pC*)
using ‹*p′ = map shift-Inl pC*› **by** *auto*
ultimately have *p-io* (*p′*@[*t*]) ∈ *LS M q1*
using ‹*t-input t′ = t-input t*› ‹*t-output t′ = t-output t*› **by** *auto*

have *p-io* (*right-path pC*) @ [(*t-input t*, *t-output t*)] ∉ *LS M q2*
using *observable-path-language-step*[*OF assms*(*4*) ‹*path M q2* (*right-path pC*)› ‹¬ (∃ *t′*∈ *transitions M. t-source t′ = target q2* (*right-path pC*) ∧ *t-input t′ = t-input t* ∧ *t-output t′ = t-output t*)›] **by** *assumption*
moreover have *p-io p′ = p-io* (*right-path pC*)
using ‹*p′ = map shift-Inl pC*› **by** *auto*
ultimately have *p-io* (*p′*@[*t*]) ∉ *LS M q2*
by *auto*

show *?thesis*
using ‹*p-io* (*p′*@[*t*]) ∈ *LS M q1*› ‹*p-io* (*p′*@[*t*]) ∉ *LS M q2*› *snoc* **by** *blast*
qed

lemma *canonical-separator-maximal-path-distinguishes-right* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
(**is** *is-state-separator-from-canonical-separator ?C q1 q2 S*)
and *path S* (*initial S*) *p*
and *target* (*initial S*) *p* = *Inr q2*
and *observable M*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *q1* ≠ *q2*
shows *p-io p* ∈ *LS M q2* − *LS M q1*
proof (*cases p rule*: *rev-cases*)
case *Nil*
then have *initial S* = *Inr q2* **using** *assms*(*3*) **by** *auto*
then have *initial ?C* = *Inr q2*
using *assms*(*1*) *assms*(*5*) *assms*(*6*) *is-state-separator-from-canonical-separator-initial* **by** *fastforce*
then show *?thesis* **using** *canonical-separator-simps*(*1*) *Inr-Inl-False*
using *assms*(*5*) *assms*(*6*) **by** *fastforce*
next
case (*snoc p′ t*)
then have *path S* (*initial S*) (*p′*@[*t*])
using *assms*(*2*) **by** *auto*
then have *t* ∈ *transitions S* **and** *t-source t* = *target* (*initial S*) *p′*
by *auto*

have *path ?C* (*initial ?C*) (*p′*@[*t*])
using ‹*path S* (*initial S*) (*p′*@[*t*])› *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*]
by (*meson submachine-path-initial*)
then have *path ?C* (*initial ?C*) (*p′*) **and** *t* ∈ *transitions ?C*
by *auto*

have *isl* (*target* (*initial S*) *p′*)
proof (*rule ccontr*)
assume ¬ *isl* (*target* (*initial S*) *p′*)
moreover have *target* (*initial S*) *p′* ∈ *states S*
using ‹*path S* (*initial S*) (*p′*@[*t*])› **by** *auto*
ultimately have *target* (*initial S*) *p′* = *Inr q1* ∨ *target* (*initial S*) *p′* = *Inr q2*

using *assms*(*1*) **unfolding** *is-state-separator-from-canonical-separator-def*
by (*metis* ‹*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*)› ‹*t-source t* = *target* (*FSM.initial S*) *p′*›
assms(*5*) *assms*(*6*) *canonical-separator-t-source-isl*)
moreover have *deadlock-state S* (*Inr q1*) **and** *deadlock-state S* (*Inr q2*)
using *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*] **by** *presburger*+
ultimately show *False*
using ‹*t* ∈ *transitions S*› ‹*t-source t* = *target* (*initial S*) *p′*› **unfolding** *deadlock-state.simps*

by *metis*
qed
then obtain *q1′ q2′* **where** *target* (*initial S*) *p′* = *Inl* (*q1′*,*q2′*)
using *isl-def prod.collapse* **by** *metis*
then have *isl* (*target* (*initial ?C*) *p′*)
using *assms*(*1*) *is-state-separator-from-canonical-separator-def*[*of ?C q1 q2 S*]
by (*metis* (*no-types*, *lifting*) *Nil-is-append-conv assms*(*2*) *isl-def list.distinct*(*1*) *list.sel*(*1*)
path.cases snoc submachine-path-initial)

obtain *pC* **where** *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) *pC*
and *p′* = *map shift-Inl pC*
using *canonical-separator-path-from-shift*[*OF* ‹*path ?C* (*initial ?C*) (*p′*)› ‹*isl* (*target* (*initial ?C*) *p′*)›]
using *assms*(*5*) *assms*(*6*) **by** *blast*
then have *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *pC*
by (*simp add*: *assms*(*5*) *assms*(*6*))

then have *path* (*from-FSM M q1*) *q1* (*left-path pC*) **and** *path* (*from-FSM M q2*) *q2* (*right-path pC*)
using *product-path*[*of from-FSM M q1 from-FSM M q2 q1 q2 pC*] **by** *presburger+*

have *path M q1* (*left-path pC*)
using *from-FSM-path*[*OF assms*(*5*) ‹*path* (*from-FSM M q1*) *q1* (*left-path pC*)›] **by** *assumption*
have *path M q2* (*right-path pC*)
using *from-FSM-path*[*OF assms*(*6*) ‹*path* (*from-FSM M q2*) *q2* (*right-path pC*)›] **by** *assumption*

have *t-target t* = *Inr q2*
using ‹*path S* (*initial S*) (*p′*@[*t*])› *snoc assms*(*3*) **by** *auto*
then have *t* ∈ (*distinguishing-transitions-right M q1 q2*)
using *canonical-separator-targets-ineq*(*3*)[*OF* ‹*t* ∈ *transitions ?C*› *assms*(*5*,*6*,*7*)] **by** *auto*
then have *t* ∈ (*distinguishing-transitions-right-alt M q1 q2*)
unfolding *distinguishing-transitions-right-alt-alt-def* **by** *assumption*

have *t-source t* = *Inl* (*q1′*,*q2′*)
using ‹*target* (*initial S*) *p′* = *Inl* (*q1′*,*q2′*)› ‹*t-source t* = *target* (*initial S*) *p′*› **by** *auto*

then obtain *t′* **where** *q1′* ∈ *states M*
and *q2′* ∈ *states M*
and *t′* ∈ *transitions M*
and *t-source t′* = *q2′*
and *t-input t′* = *t-input t*
and *t-output t′* = *t-output t*

and ¬ (∃ *t''*∈ *transitions M*. *t-source t''* = *q1'* ∧ *t-input t''* = *t-input t* ∧ *t-output t''* = *t-output t*)
using ‹*t* ∈ (*distinguishing-transitions-right-alt M q1 q2*)› *assms*(*5*,*6*) *fsm-transition-source path-target-is-state*
unfolding *distinguishing-transitions-right-alt-def reachable-states-def* **by** *fastforce*

have *initial S* = *Inl* (*q1*,*q2*)
by (*meson assms*(*1*) *assms*(*5*) *assms*(*6*) *is-state-separator-from-canonical-separator-initial*)
have *length p'* = *length pC*
using ‹*p'* = *map shift-Inl pC*› **by** *auto*
then have *target* (*initial S*) *p'* = *Inl* (*target* (*q1*,*q2*) *pC*)
using ‹*p'* = *map shift-Inl pC*› ‹*initial S* = *Inl* (*q1*,*q2*)› **by** (*induction p' pC rule*: *list-induct2*; *auto*)
then have *target* (*q1*,*q2*) *pC* = (*q1'*,*q2'*)
using ‹*target* (*initial S*) *p'* = *Inl* (*q1'*,*q2'*)› **by** *auto*
then have *target q1* (*left-path pC*) = *q1'*
using *product-target-split*(*1*) **by** *fastforce*
then have ¬ (∃ *t'*∈ *transitions M*. *t-source t'* = *target q1* (*left-path pC*) ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
using ‹¬ (∃ *t'*∈ *transitions M*. *t-source t'* = *q1'* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)› **by** *blast*

have *target q2* (*right-path pC*) = *q2'*
using ‹*target* (*q1*,*q2*) *pC* = (*q1'*,*q2'*)› *product-target-split*(*2*) **by** *fastforce*
then have *path M q2* ((*right-path pC*)@[*t'*])
using ‹*path M q2* (*right-path pC*)› ‹*t'* ∈ *transitions M*› ‹*t-source t'* = *q2'*›
by (*simp add*: *path-append-transition*)
then have *p-io* ((*right-path pC*)@[*t'*]) ∈ *LS M q2*
unfolding *LS.simps* **by** *force*
moreover have *p-io p'* = *p-io* (*right-path pC*)
using ‹*p'* = *map shift-Inl pC*› **by** *auto*
ultimately have *p-io* (*p'*@[*t*]) ∈ *LS M q2*
using ‹*t-input t'* = *t-input t*› ‹*t-output t'* = *t-output t*› **by** *auto*

have *p-io* (*left-path pC*) @ [(*t-input t*, *t-output t*)] ∉ *LS M q1*
using *observable-path-language-step*[*OF assms*(*4*) ‹*path M q1* (*left-path pC*)› ‹¬ (∃ *t'*∈ *transitions M*. *t-source t'* = *target q1* (*left-path pC*) ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)›] **by** *assumption*
moreover have *p-io p'* = *p-io* (*left-path pC*)
using ‹*p'* = *map shift-Inl pC*› **by** *auto*
ultimately have *p-io* (*p'*@[*t*]) ∉ *LS M q1*
by *auto*

show *?thesis*
using ‹*p-io* (*p'*@[*t*]) ∈ *LS M q2*› ‹*p-io* (*p'*@[*t*]) ∉ *LS M q1*› *snoc*
by *blast*

qed

lemma *state-separator-from-canonical-separator-observable* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *observable A*
using *submachine-observable*[*OF - canonical-separator-observable*[*OF assms*(*2*,*3*,*4*)]]
using *assms*(*1*) **unfolding** *is-state-separator-from-canonical-separator-def*
by *metis*

lemma *canonical-separator-initial* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *initial* (*canonical-separator M q1 q2*) = *Inl* (*q1*,*q2*)
unfolding *canonical-separator-simps*[*OF assms*] **by** *simp*

lemma *canonical-separator-states* :
assumes *Inl* (*s1*,*s2*) ∈ *states* (*canonical-separator M q1 q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows (*s1*,*s2*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *assms*(*1*) *reachable-state-is-state*
unfolding *canonical-separator-simps*[*OF assms*(*2*,*3*)] **by** *fastforce*

lemma *canonical-separator-transition* :
assumes *t* ∈ *transitions* (*canonical-separator M q1 q2*) (**is** *t* ∈ *transitions ?C*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *t-source t* = *Inl* (*s1*,*s2*)
and *observable M*
and *q1* ≠ *q2*
shows ⋀ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*) ⟹ (*s1*, *t-input t*, *t-output t*, *s1′*) ∈ *transitions M* ∧ (*s2*, *t-input t*, *t-output t*, *s2′*) ∈ *transitions M*
and *t-target t* = *Inr q1* ⟹ (∃ *t′*∈ *transitions M* . *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
∧ (¬(∃ *t′*∈ *transitions M* . *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*))
and *t-target t* = *Inr q2* ⟹ (∃ *t′*∈ *transitions M* . *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
∧ (¬(∃ *t′*∈ *transitions M* . *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*))
and (∃ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*)) ∨ *t-target t* = *Inr q1* ∨ *t-target t* = *Inr q2*

proof −
show $\bigwedge$ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*) $\Longrightarrow$ (*s1*, *t-input t*, *t-output t*, *s1′*) $\in$ *transitions M* $\wedge$ (*s2*, *t-input t*, *t-output t*, *s2′*) $\in$ *transitions M*
using *canonical-separator-transition-targets*(*1*)[*OF assms*(*1*,*2*,*3*)] *assms*(*4*)
unfolding *shifted-transitions-for-def*[*symmetric*]
unfolding *shifted-transitions-for-alt-def*
unfolding *product-transitions-def from-FSM-simps*[*OF assms*(*2*)] *from-FSM-simps*[*OF assms*(*3*)] **by** *fastforce*

show *t-target t* = *Inr q1* $\Longrightarrow$ ($\exists$ *t′*$\in$ *transitions M* . *t-source t′* = *s1* $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*)
$\wedge$ ($\neg$($\exists$ *t′*$\in$ *transitions M* . *t-source t′* = *s2* $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*))
using *canonical-separator-targets-observable*(*2*)[*OF assms*(*1*,*2*,*3*,*6*)] *assms*(*4*)
unfolding *distinguishing-transitions-left-def* **by** *fastforce*

show *t-target t* = *Inr q2* $\Longrightarrow$ ($\exists$ *t′*$\in$ *transitions M* . *t-source t′* = *s2* $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*)
$\wedge$ ($\neg$($\exists$ *t′*$\in$ *transitions M* . *t-source t′* = *s1* $\wedge$ *t-input t′* = *t-input t* $\wedge$ *t-output t′* = *t-output t*))
using *canonical-separator-targets-observable*(*3*)[*OF assms*(*1*,*2*,*3*,*6*)] *assms*(*4*)
unfolding *distinguishing-transitions-right-def* **by** *fastforce*

show ($\exists$ *s1′ s2′* . *t-target t* = *Inl* (*s1′*,*s2′*)) $\vee$ *t-target t* = *Inr q1* $\vee$ *t-target t* = *Inr q2*
using *canonical-separator-transition-targets*(*4*)[*OF assms*(*1*,*2*,*3*)]
by (*simp add*: *isl-def*)
qed

lemma *canonical-separator-transition-source* :
assumes *t* $\in$ *transitions* (*canonical-separator M q1 q2*) (**is** *t* $\in$ *transitions ?C*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
obtains *q1′ q2′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
(*q1′*,*q2′*) $\in$ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
proof −
consider *t* $\in$ *shifted-transitions-for M q1 q2* | *t* $\in$ *distinguishing-transitions-left-alt M q1 q2* |
t $\in$ *distinguishing-transitions-right-alt M q1 q2*
using *assms*(*1*)
unfolding *canonical-separator-transitions-alt-def*[*OF assms*(*2*,*3*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis* **unfolding** *shifted-transitions-for-def* **using** *that*
using *fsm-transition-source* **by** *fastforce*
next
case *2*

then show *?thesis* **unfolding** *distinguishing-transitions-left-alt-def* **using** *that* **by** *fastforce*
next
case *3*
then show *?thesis* **unfolding** *distinguishing-transitions-right-alt-def* **using** *that* **by** *fastforce*
qed
qed

lemma *canonical-separator-transition-ex* :
assumes $t \in$ *transitions* (*canonical-separator M q1 q2*) (**is** $t \in$ *transitions ?C*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and *t-source t* = *Inl* (*s1*,*s2*)
shows ($\exists$ *t1* $\in$ *transitions M* . *t-source t1* = *s1* $\wedge$ *t-input t1* = *t-input t* $\wedge$ *t-output t1* = *t-output t*) $\vee$
($\exists$ *t2* $\in$ *transitions M* . *t-source t2* = *s2* $\wedge$ *t-input t2* = *t-input t* $\wedge$ *t-output t2* = *t-output t*)
proof −
consider $t \in$ *shifted-transitions-for M q1 q2* | $t \in$ *distinguishing-transitions-left-alt M q1 q2* |
$t \in$ *distinguishing-transitions-right-alt M q1 q2*
using *assms*(*1*)
unfolding *canonical-separator-transitions-alt-def*[*OF assms*(*2*,*3*)] **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then show *?thesis* **unfolding** *shifted-transitions-for-def*
using *product-from-transition-split*[*OF - assms*(*2*,*3*)]
using *assms*(*4*) **by** *force*
next
case *2*
then show *?thesis* **unfolding** *distinguishing-transitions-left-alt-def*
using *assms*(*4*) **by** *auto*

next
case *3*
then show *?thesis* **unfolding** *distinguishing-transitions-right-alt-def*
using *assms*(*4*) **by** *auto*
qed
qed

lemma *canonical-separator-path-split-target-isl* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) (*p*@[*t*])
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
shows *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)

proof −
  let *?C* = (*canonical-separator M q1 q2*)
  have *t* ∈ *transitions ?C*
    using *assms* **by** *auto*
  moreover have ¬ *deadlock-state ?C* (*t-source t*)
    using *assms* **unfolding** *deadlock-state.simps* **by** *blast*
  ultimately show *?thesis*
    using *canonical-separator-t-source-isl assms*
    by *fastforce*
qed

lemma *canonical-separator-path-initial* :
  assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p* (**is** *path ?C* (*initial ?C*) *p*)
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *observable M*
  and *q1* ≠ *q2*
shows ⋀ *s1′ s2′* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*,*s2′*) ⟹ (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io p* ∧ *target q1 p1* = *s1′* ∧ *target q2 p2* = *s2′*)
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ⟹ (∃ *p1 p2 t* . *path M q1* (*p1*@[*t*]) ∧ *path M q2 p2* ∧ *p-io* (*p1*@[*t*]) = *p-io p* ∧ *p-io p2* = *butlast* (*p-io p*)) ∧ (¬(∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*))
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2* ⟹ (∃ *p1 p2 t* . *path M q1 p1* ∧ *path M q2* (*p2*@[*t*]) ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io* (*p2*@[*t*]) = *p-io p*) ∧ (¬(∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*))
and (∃ *s1′ s2′* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*,*s2′*)) ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*
proof −

  let *?P1* = ∀ *s1′ s2′* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*,*s2′*) ⟶ (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io p* ∧ *target q1 p1* = *s1′* ∧ *target q2 p2* = *s2′*)
  let *?P2* = *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ⟶ (∃ *p1 p2 t* . *path M q1* (*p1*@[*t*]) ∧ *path M q2 p2* ∧ *p-io* (*p1*@[*t*]) = *p-io p* ∧ *p-io p2* = *butlast* (*p-io p*)) ∧ (¬(∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*))
  let *?P3* = *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2* ⟶ (∃ *p1 p2 t* . *path M q1 p1* ∧ *path M q2* (*p2*@[*t*]) ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io* (*p2*@[*t*]) = *p-io p*) ∧ (¬(∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*))

  have *?P1* ∧ *?P2* ∧ *?P3*
  using *assms*(*1*) **proof** (*induction p rule*: *rev-induct*)
    case *Nil*
    then have *target* (*FSM.initial* (*canonical-separator M q1 q2*)) [] = *Inl* (*q1*, *q2*)
      unfolding *canonical-separator-simps*[*OF assms*(*2*,*3*)] **by** *auto*
    then show *?case* **using** *assms*(*2*,*3*,*4*) **by** *fastforce*

next
case (*snoc t p*)

have *path ?C* (*initial ?C*) *p* **and** *t* ∈ *transitions ?C* **and** *t-source t* = *target* (*initial ?C*) *p*
using *snoc.prems*(*1*) **by** *auto*

let *?P1′* = (∀ *s1′ s2′*. *target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inl* (*s1′*, *s2′*) ⟶ (∃ *p1 p2*. *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io* (*p* @ [*t*]) ∧ *target q1 p1* = *s1′* ∧ *target q2 p2* = *s2′*))
let *?P2′* = (*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q1* ⟶ (∃ *p1 p2 ta*. *path M q1* (*p1* @ [*ta*]) ∧ *path M q2 p2* ∧ *p-io* (*p1* @ [*ta*]) = *p-io* (*p* @ [*t*]) ∧ *p-io p2* = *butlast* (*p-io* (*p* @ [*t*]))) ∧ (∄ *p2*. *path M q2 p2* ∧ *p-io p2* = *p-io* (*p* @ [*t*])))
let *?P3′* = (*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q2* ⟶ (∃ *p1 p2 ta*. *path M q1 p1* ∧ *path M q2* (*p2* @ [*ta*]) ∧ *p-io p1* = *butlast* (*p-io* (*p* @ [*t*])) ∧ *p-io* (*p2* @ [*ta*]) = *p-io* (*p* @ [*t*])) ∧ (∄ *p1*. *path M q1 p1* ∧ *p-io p1* = *p-io* (*p* @ [*t*])))

let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

obtain *p′* **where** *path ?P* (*initial ?P*) *p′*
and ∗:*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p′*
using *canonical-separator-path-from-shift*[*OF* ‹*path ?C* (*initial ?C*) *p*› *canonical-separator-path-split-target-isl*[*OF snoc.prems assms*(*2*,*3*)] *assms*(*2*,*3*)]
by *blast*

let *?pL* = (*map* (λ*t*. (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) *p′*)
let *?pR* = (*map* (λ*t*. (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) *p′*)

have *path ?P* (*q1*,*q2*) *p′*
using ‹*path ?P* (*initial ?P*) *p′*› *assms*(*2*,*3*) **unfolding** *product-simps*(*1*) *from-FSM-simps*(*1*) **by** *simp*
then have *pL*: *path* (*from-FSM M q1*) *q1 ?pL*
and *pR*: *path* (*from-FSM M q2*) *q2 ?pR*
using *product-path*[*of from-FSM M q1 from-FSM M q2 q1 q2 p′*] **by** *simp*+

have *p-io ?pL* = *p-io p* **and** *p-io ?pR* = *p-io p*
using ∗ **by** *auto*

have *pf1*: *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *?pL*
using *pL assms*(*2*) **unfolding** *from-FSM-simps*(*1*) **by** *auto*
have *pf2*: *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *?pR*
using *pR assms*(*3*) **unfolding** *from-FSM-simps*(*1*) **by** *auto*
have *pio*: *p-io ?pL* = *p-io ?pR*
by *auto*

have *p-io* (*zip-path ?pL ?pR*) = *p-io ?pL*
by (*induction p′*; *auto*)

have *zip1*: *path ?P* (*initial ?P*) (*zip-path ?pL ?pR*)
and *target* (*initial ?P*) (*zip-path ?pL ?pR*) = (*target q1 ?pL*, *target q2 ?pR*)
using *product-path-from-paths*[*OF pf1 pf2 pio*] *assms*(*2*,*3*)
unfolding *from-FSM-simps*(*1*) **by** *simp*+

have *p-io* (*zip-path ?pL ?pR*) = *p-io p*
using ‹*p-io ?pL* = *p-io p*› ‹*p-io* (*zip-path ?pL ?pR*) = *p-io ?pL*› **by** *auto*
have *observable ?P*
using *product-observable*[*OF from-FSM-observable*[*OF assms*(*4*)] *from-FSM-observable*[*OF assms*(*4*)]] **by** *assumption*

have *p-io p′* = *p-io p*
using ∗ **by** *auto*

obtain *s1 s2* **where** *t-source t* = *Inl* (*s1*,*s2*)
using *canonical-separator-path-split-target-isl*[*OF snoc.prems*(*1*) *assms*(*2*,*3*)]
by (*metis* ‹*t-source t* = *target* (*initial* (*canonical-separator M q1 q2*)) *p*› *isl-def old.prod.exhaust*)

have *map t-target p* = *map* (*Inl o t-target*) *p′*
using ∗ **by** *auto*
have *target* (*initial ?C*) *p* = *Inl* (*target* (*q1*,*q2*) *p′*)
unfolding *target.simps visited-states.simps canonical-separator-simps*[*OF assms*(*2*,*3*)]
unfolding ‹*map t-target p* = *map* (*Inl o t-target*) *p′*›
by (*simp add*: *last-map*)
then have *target* (*q1*,*q2*) *p′*= (*s1*,*s2*)
using ‹*t-source t* = *target* (*initial ?C*) *p*› ‹*t-source t* = *Inl* (*s1*,*s2*)›
by *auto*

have *target q1 ?pL* = *s1* **and** *target q2 ?pR* = *s2*
using *product-target-split*[*OF* ‹*target* (*q1*,*q2*) *p′*= (*s1*,*s2*)›] **by** *auto*

consider (*a*) (∃ *s1′ s2′*. *t-target t* = *Inl* (*s1′*, *s2′*)) |
(*b*) *t-target t* = *Inr q1* |
(*c*) *t-target t* = *Inr q2*
using *canonical-separator-transition*(*4*)[*OF* ‹*t* ∈ *transitions ?C*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*t-source t* = *Inl* (*s1*,*s2*)› ‹*observable M*› ‹*q1* ≠ *q2*›]
by *blast*
then show *?P1′* ∧ *?P2′* ∧ *?P3′* **proof** *cases*
case *a*
then obtain *s1′ s2′* **where** *t-target t* = *Inl* (*s1′*,*s2′*)
by *blast*

let *?t1* = (*s1*, *t-input t*, *t-output t*, *s1′*)

let *?t2* = (*s2*, *t-input t*, *t-output t*, *s2'*)

have *?t1* ∈ *transitions M*
and *?t2* ∈ *transitions M*
using *canonical-separator-transition*(*1*)[*OF* ‹*t* ∈ *transitions ?C*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*t-source t* = *Inl* (*s1*,*s2*)› ‹*observable M*› ‹*q1* ≠ *q2*› ‹*t-target t* = *Inl* (*s1'*,*s2'*)›]
by *auto*

have *target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inl* (*s1'*, *s2'*)
using ‹*t-target t* = *Inl* (*s1'*,*s2'*)› **by** *auto*

have *path M q1* (*?pL*@[*?t1*])
using *path-append-transition*[*OF from-FSM-path*[*OF* ‹*q1* ∈ *states M*› *pL*] ‹*?t1* ∈ *transitions M*›] ‹*target q1 ?pL* = *s1*› **by** *auto*
moreover have *path M q2* (*?pR*@[*?t2*])
using *path-append-transition*[*OF from-FSM-path*[*OF* ‹*q2* ∈ *states M*› *pR*] ‹*?t2* ∈ *transitions M*›] ‹*target q2 ?pR* = *s2*› **by** *auto*
moreover have *p-io* (*?pL*@[*?t1*]) = *p-io* (*?pR*@[*?t2*])
by *auto*
moreover have *p-io* (*?pL*@[*?t1*]) = *p-io* (*p*@[*t*])
using ‹*p-io ?pL* = *p-io p*› **by** *auto*
moreover have *target q1* (*?pL*@[*?t1*]) = *s1'* **and** *target q2* (*?pR*@[*?t2*]) = *s2'*
by *auto*
ultimately have *path M q1* (*?pL*@[*?t1*]) ∧ *path M q2* (*?pR*@[*?t2*]) ∧ *p-io* (*?pL*@[*?t1*]) = *p-io* (*?pR*@[*?t2*]) ∧ *p-io* (*?pL*@[*?t1*]) = *p-io* (*p*@[*t*]) ∧ *target q1* (*?pL*@[*?t1*]) = *s1'* ∧ *target q2* (*?pR*@[*?t2*]) = *s2'*
by *presburger*
then have (∃ *p1 p2*. *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io* (*p* @ [*t*]) ∧ *target q1 p1* = *s1'* ∧ *target q2 p2* = *s2'*)
by *meson*
then have *?P1'*
using ‹*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inl* (*s1'*, *s2'*)› **by** *auto*
then show *?thesis* **using** ‹*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inl* (*s1'*, *s2'*)›
by *auto*
next
case *b*
then have *target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q1*
by *auto*

have (∃ *t'*∈(*transitions M*). *t-source t'* = *s1* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
and ¬ (∃ *t'*∈(*transitions M*). *t-source t'* = *s2* ∧ *t-input t'* = *t-input t* ∧ *t-output t'* = *t-output t*)
using *canonical-separator-transition*(*2*)[*OF* ‹*t* ∈ *transitions ?C*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*t-source t* = *Inl* (*s1*,*s2*)› ‹*observable M*› ‹*q1* ≠ *q2*› *b*] **by**

blast+

then obtain t' **where** $t' \in$ *transitions M* **and** *t-source* $t' =$ *s1* **and** *t-input* $t' =$ *t-input t* **and** *t-output* $t' =$ *t-output t*
by *blast*

have *path M q1* (*?pL*@[t'])
using *path-append-transition*[*OF from-FSM-path*[*OF* ‹*q1* ∈ *states M*› *pL*] ‹$t' \in$ *transitions M*›] ‹*target q1 ?pL* = *s1*› ‹*t-source* t' = *s1*› **by** *auto*
moreover have *p-io* (*?pL*@[t']) = *p-io* (*p*@[*t*])
using ‹*p-io ?pL* = *p-io p*› ‹*t-input* t' = *t-input t*› ‹*t-output* t' = *t-output t*› **by** *auto*
moreover have *p-io ?pR* = *butlast* (*p-io* (*p* @ [*t*]))
using ‹*p-io ?pR* = *p-io p*› **by** *auto*
ultimately have *path M q1* (*?pL*@[t']) ∧ *path M q2 ?pR* ∧ *p-io* (*?pL*@[t']) = *p-io* (*p* @ [*t*]) ∧ *p-io ?pR* = *butlast* (*p-io* (*p* @ [*t*]))
using *from-FSM-path*[*OF* ‹*q2* ∈ *states M*› *pR*] **by** *linarith*
then have (∃ *p1 p2 ta. path M q1* (*p1* @ [*ta*]) ∧ *path M q2 p2* ∧ *p-io* (*p1* @ [*ta*]) = *p-io* (*p* @ [*t*]) ∧ *p-io p2* = *butlast* (*p-io* (*p* @ [*t*])))
by *meson*

moreover have (∄ *p2. path M q2 p2* ∧ *p-io p2* = *p-io* (*p* @ [*t*]))
proof
assume ∃ *p2. path M q2 p2* ∧ *p-io p2* = *p-io* (*p* @ [*t*])
then obtain p'' **where** *path M q2* p'' ∧ *p-io* p'' = *p-io* (*p* @ [*t*])
by *blast*
then have $p'' \neq []$ **by** *auto*
then obtain *p2 t2* **where** p'' = *p2*@[*t2*]
using *rev-exhaust* **by** *blast*
then have *path M q2* (*p2*@[*t2*]) **and** *p-io* (*p2*@[*t2*]) = *p-io* (*p* @ [*t*])
using ‹*path M q2* p'' ∧ *p-io* p'' = *p-io* (*p* @ [*t*])› **by** *auto*
then have *path M q2 p2* **by** *auto*

then have *pf2′*: *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*
using *from-FSM-path-initial*[*OF* ‹*q2* ∈ *states M*›, *of p2*] **by** *simp*
have *pio′*: *p-io ?pL* = *p-io p2*
using ‹*p-io* (*?pL*@[t']) = *p-io* (*p*@[*t*])› ‹*p-io* (*p2*@[*t2*]) = *p-io* (*p* @ [*t*])› **by** *auto*

have *zip2*: *path ?P* (*initial ?P*) (*zip-path ?pL p2*)
and *target* (*initial ?P*) (*zip-path ?pL p2*) = (*target q1 ?pL*, *target q2 p2*)
using *product-path-from-paths*[*OF pf1 pf2′ pio′*] *assms*(*2*,*3*)
unfolding *from-FSM-simps*(*1*) **by** *simp+*

have *length* p' = *length p2*
using ‹*p-io* (*p2*@[*t2*]) = *p-io* (*p* @ [*t*])›
by (*metis* (*no-types*, *lifting*) *length-map pio′*)
then have *p-io* (*zip-path ?pL p2*) = *p-io* p'

by (*induction p′ p2 rule*: *list-induct2*; *auto*)
then have *p-io* (*zip-path ?pL p2*) = *p-io p*
using ∗ **by** *auto*
then have *p-io* (*zip-path ?pL ?pR*) = *p-io* (*zip-path ?pL p2*)
using ‹*p-io* (*zip-path ?pL ?pR*) = *p-io p*› **by** *simp*

have *p-io ?pR* = *p-io p2*
using ‹*p-io ?pL* = *p-io p2*› *pio* **by** *auto*

have *l1*: *length ?pL* = *length ?pR* **by** *auto*
have *l2*: *length ?pR* = *length ?pL* **by** *auto*
have *l3*: *length ?pL* = *length p2* **using** ‹*length p′* = *length p2*› **by** *auto*

have *p2* = *?pR*
using *zip-path-eq-right*[*OF l1 l2 l3* ‹*p-io ?pR* = *p-io p2*› *observable-path-unique*[*OF* ‹*observable ?P*› *zip1 zip2* ‹*p-io* (*zip-path ?pL ?pR*) = *p-io* (*zip-path ?pL p2*)›]] **by** *simp*
then have *target q2 p2* = *s2*
using ‹*target q2 ?pR* = *s2*› **by** *auto*
then have *t2* ∈ *transitions M* **and** *t-source t2* = *s2*
using ‹*path M q2* (*p2*@[*t2*])› **by** *auto*
moreover have *t-input t2* = *t-input t* ∧ *t-output t2* = *t-output t*
using ‹*p-io* (*p2*@[*t2*]) = *p-io* (*p* @ [*t*])› **by** *auto*
ultimately show *False*
using ‹¬ (∃ *t′*∈(*transitions M*). *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)› **by** *blast*
qed

ultimately have *?P2′*
by *blast*
moreover have *?P3′*
using ‹*q1* ≠ *q2*› ‹*t-target t* = *Inr q1*› **by** *auto*
moreover have *?P1′*
using ‹*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q1*› **by** *auto*
ultimately show *?thesis*
by *blast*
next
case *c*
then have *target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q2*
by *auto*

have (∃ *t′*∈(*transitions M*). *t-source t′* = *s2* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
and ¬ (∃ *t′*∈(*transitions M*). *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)
using *canonical-separator-transition*(*3*)[*OF* ‹*t* ∈ *transitions ?C*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*t-source t* = *Inl* (*s1*,*s2*)› ‹*observable M*› ‹*q1* ≠ *q2*› *c*] **by**

blast+

then obtain t' **where** $t' \in$ *transitions M* **and** *t-source* $t' = s2$ **and** *t-input* $t' =$ *t-input t* **and** *t-output* $t' =$ *t-output t*
by *blast*

have *path M q2* (*?pR*@[t'])
using *path-append-transition*[*OF from-FSM-path*[*OF* ‹*q2* ∈ *states M*› *pR*] ‹$t' \in$ *transitions M*›] ‹*target q2 ?pR* = *s2*› ‹*t-source* t' = *s2*› **by** *auto*
moreover have *p-io* (*?pR*@[t']) = *p-io* (*p*@[*t*])
using ‹*p-io ?pR* = *p-io p*› ‹*t-input* t' = *t-input t*› ‹*t-output* t' = *t-output t*› **by** *auto*
moreover have *p-io ?pL* = *butlast* (*p-io* (*p* @ [*t*]))
using ‹*p-io ?pL* = *p-io p*› **by** *auto*
ultimately have *path M q2* (*?pR*@[t']) ∧ *path M q1 ?pL* ∧ *p-io* (*?pR*@[t']) = *p-io* (*p* @ [*t*]) ∧ *p-io ?pL* = *butlast* (*p-io* (*p* @ [*t*]))
using *from-FSM-path*[*OF* ‹*q1* ∈ *states M*› *pL*] **by** *linarith*
then have (∃ *p1 p2 ta. path M q1 p1* ∧ *path M q2* (*p2* @ [*ta*]) ∧ *p-io p1* = *butlast* (*p-io* (*p* @ [*t*])) ∧ *p-io* (*p2* @ [*ta*]) = *p-io* (*p* @ [*t*]))
by *meson*

moreover have (∄ *p1. path M q1 p1* ∧ *p-io p1* = *p-io* (*p* @ [*t*]))
proof
assume ∃ *p1. path M q1 p1* ∧ *p-io p1* = *p-io* (*p* @ [*t*])
then obtain p'' **where** *path M q1* p'' ∧ *p-io* p'' = *p-io* (*p* @ [*t*])
by *blast*
then have $p'' \neq []$ **by** *auto*
then obtain *p1 t1* **where** p'' = *p1*@[*t1*]
using *rev-exhaust* **by** *blast*
then have *path M q1* (*p1*@[*t1*]) **and** *p-io* (*p1*@[*t1*]) = *p-io* (*p* @ [*t*])
using ‹*path M q1* p'' ∧ *p-io* p'' = *p-io* (*p* @ [*t*])› **by** *auto*
then have *path M q1 p1*
by *auto*
then have *pf1′*: *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*
using *from-FSM-path-initial*[*OF* ‹*q1* ∈ *states M*›, *of p1*] **by** *simp*
have *pio′*: *p-io p1* = *p-io ?pR*
using ‹*p-io* (*?pR*@[t']) = *p-io* (*p*@[*t*])› ‹*p-io* (*p1*@[*t1*]) = *p-io* (*p* @ [*t*])› **by** *auto*

have *zip2*: *path ?P* (*initial ?P*) (*zip-path p1 ?pR*)
using *product-path-from-paths*[*OF pf1′ pf2 pio′*]
unfolding *from-FSM-simps*(*1*) **by** *simp*

have *length* p' = *length p1*
using ‹*p-io* (*p1*@[*t1*]) = *p-io* (*p* @ [*t*])›
by (*metis* (*no-types*, *lifting*) *length-map pio′*)
then have *p-io* (*zip-path p1 ?pR*) = *p-io* p'
using ‹*p-io p1* = *p-io ?pR*› **by** (*induction* p' *p1 rule*: *list-induct2*; *auto*)
then have *p-io* (*zip-path p1 ?pR*) = *p-io p*

using $*$ **by** *auto*
then have *p-io* (*zip-path ?pL ?pR*) = *p-io* (*zip-path p1 ?pR*)
using ‹*p-io* (*zip-path ?pL ?pR*) = *p-io p*› **by** *simp*

have *l1*: *length ?pL* = *length ?pR* **by** *auto*
have *l2*: *length ?pR* = *length p1* **using** ‹*length p′* = *length p1*› **by** *auto*
have *l3*: *length p1* = *length ?pR* **using** *l2* **by** *auto*

have *?pL* = *p1*
using *zip-path-eq-left*[*OF l1 l2 l3 observable-path-unique*[*OF* ‹*observable ?P*› *zip1 zip2* ‹*p-io* (*zip-path ?pL ?pR*) = *p-io* (*zip-path p1 ?pR*)›]] **by** *simp*
then have *target q1 p1* = *s1*
using ‹*target q1 ?pL* = *s1*› **by** *auto*
then have *t1* ∈ *transitions M* **and** *t-source t1* = *s1*
using ‹*path M q1* (*p1*@[*t1*])› **by** *auto*
moreover have *t-input t1* = *t-input t* ∧ *t-output t1* = *t-output t*
using ‹*p-io* (*p1*@[*t1*]) = *p-io* (*p* @ [*t*])› **by** *auto*
ultimately show *False*
using ‹¬ (∃ *t′*∈(*transitions M*). *t-source t′* = *s1* ∧ *t-input t′* = *t-input t* ∧ *t-output t′* = *t-output t*)› **by** *blast*
qed

ultimately have *?P3′*
by *blast*
moreover have *?P2′*
using ‹*q1* ≠ *q2*› ‹*t-target t* = *Inr q2*› **by** *auto*
moreover have *?P1′*
using ‹*target* (*initial* (*canonical-separator M q1 q2*)) (*p* @ [*t*]) = *Inr q2*› **by** *auto*
ultimately show *?thesis*
by *blast*
qed
qed

then show ⋀ *s1′ s2′* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*,*s2′*) ⟹ (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *p-io p2* ∧ *p-io p1* = *p-io p* ∧ *target q1 p1* = *s1′* ∧ *target q2 p2* = *s2′*)
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ⟹ (∃ *p1 p2 t* . *path M q1* (*p1*@[*t*]) ∧ *path M q2 p2* ∧ *p-io* (*p1*@[*t*]) = *p-io p* ∧ *p-io p2* = *butlast* (*p-io p*)) ∧ (¬(∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*))
and *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2* ⟹ (∃ *p1 p2 t* . *path M q1 p1* ∧ *path M q2* (*p2*@[*t*]) ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io* (*p2*@[*t*]) = *p-io p*) ∧ (¬(∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*))
by *blast*+

show (∃ *s1′ s2′* . *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*,*s2′*)) ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* ∨ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*

proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis* **unfolding** *canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)] **by** *auto*
next
case (*snoc p′ t*)
then have $t \in$ *transitions ?C* **and** *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *t-target t*
using *assms*(*1*) **by** *auto*
then have $t \in$ (*transitions ?C*)
by *auto*
obtain *s1 s2* **where** *t-source t* = *Inl* (*s1*,*s2*)
using *canonical-separator-t-source-isl*[*OF* ‹$t \in$ (*transitions ?C*)› *assms*(*2*,*3*)]
by (*metis sum.collapse*(*1*) *surjective-pairing*)
show *?thesis*
using *canonical-separator-transition*(*4*)[*OF* ‹$t \in$ *transitions ?C*› *assms*(*2*,*3*) ‹*t-source t* = *Inl* (*s1*,*s2*)› *assms*(*4*) ‹*q1* $\neq$ *q2*›]
‹*target* (*initial* (*canonical-separator M q1 q2*)) *p* = *t-target t*›
by *simp*
qed
qed

lemma *canonical-separator-path-initial-ex* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p* (**is** *path ?C* (*initial ?C*) *p*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
shows ($\exists$ *p1* . *path M q1 p1* $\wedge$ *p-io p1* = *p-io p*) $\vee$ ($\exists$ *p2* . *path M q2 p2* $\wedge$ *p-io p2* = *p-io p*)
and ($\exists$ *p1 p2* . *path M q1 p1* $\wedge$ *path M q2 p2* $\wedge$ *p-io p1* = *butlast* (*p-io p*) $\wedge$ *p-io p2* = *butlast* (*p-io p*))
proof −
have (($\exists$ *p1* . *path M q1 p1* $\wedge$ *p-io p1* = *p-io p*) $\vee$ ($\exists$ *p2* . *path M q2 p2* $\wedge$ *p-io p2* = *p-io p*))
$\wedge$ ($\exists$ *p1 p2* . *path M q1 p1* $\wedge$ *path M q2 p2* $\wedge$ *p-io p1* = *butlast* (*p-io p*) $\wedge$ *p-io p2* = *butlast* (*p-io p*))
using *assms* **proof** (*induction p rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc t p*)
then have *path ?C* (*initial ?C*) *p* **and** $t \in$ *transitions ?C* **and** *t-source t* = *target* (*initial ?C*) *p*
by *auto*

let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

obtain p' **where** *path ?P* (*initial ?P*) p'
and $*$:p = *map* (λt. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) p'
using *canonical-separator-path-from-shift*[*OF* ‹*path ?C* (*initial ?C*) *p*› *canonical-separator-path-split-target-isl*[*OF snoc.prems*(*1*) *assms*(*2*,*3*)] *assms*(*2*,*3*)]
by *blast*

let *?pL* = (*map* (λt. (*fst* (*t-source t*), *t-input t*, *t-output t*, *fst* (*t-target t*))) p')
let *?pR* = (*map* (λt. (*snd* (*t-source t*), *t-input t*, *t-output t*, *snd* (*t-target t*))) p')

have *path ?P* (*q1*,*q2*) p'
using ‹*path ?P* (*initial ?P*) p'› *assms*(*2*,*3*) **by** *simp*

then have *pL*: *path* (*from-FSM M q1*) *q1 ?pL*
and *pR*: *path* (*from-FSM M q2*) *q2 ?pR*
using *product-path*[*of from-FSM M q1 from-FSM M q2 q1 q2 p'*] **by** *auto*

have *p-io ?pL* = *butlast* (*p-io* (*p*@[*t*])) **and** *p-io ?pR* = *butlast* (*p-io* (*p*@[*t*]))
using $*$ **by** *auto*
then have *path M q1 ?pL* $\wedge$ *path M q2 ?pR* $\wedge$ *p-io ?pL* = *butlast* (*p-io* (*p*@[*t*])) $\wedge$ *p-io ?pR* = *butlast* (*p-io* (*p*@[*t*]))
using *from-FSM-path*[*OF* ‹*q1* $\in$ *states M*› *pL*] *from-FSM-path*[*OF* ‹*q2* $\in$ *states M*› *pR*] **by** *auto*
then have ($\exists$ *p1 p2*. *path M q1 p1* $\wedge$ *path M q2 p2* $\wedge$ *p-io p1* = *butlast* (*p-io* (*p* @ [*t*])) $\wedge$ *p-io p2* = *butlast* (*p-io* (*p* @ [*t*])))
by *blast*

obtain *s1 s2* **where** *t-source t* = *Inl* (*s1*,*s2*)
using *canonical-separator-path-split-target-isl*[*OF snoc.prems*(*1*) *assms*(*2*,*3*)]
by (*metis* ‹*t-source t* = *target* (*initial* (*canonical-separator M q1 q2*)) *p*› *isl-def old.prod.exhaust*)

have *map t-target p* = *map* (*Inl o t-target*) p'
using $*$ **by** *auto*
then have *target* (*initial ?C*) *p* = *Inl* (*target* (*q1*,*q2*) p')
unfolding *target.simps visited-states.simps canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)]
by (*simp add*: *last-map*)
then have *target* (*q1*,*q2*) p'= (*s1*,*s2*)
using ‹*t-source t* = *target* (*initial ?C*) *p*› ‹*t-source t* = *Inl* (*s1*,*s2*)›
by *auto*

have *target q1 ?pL* = *s1* **and** *target q2 ?pR* = *s2*
using *product-target-split*[*OF* ‹*target* (*q1*,*q2*) p'= (*s1*,*s2*)›] **by** *auto*

consider (*a*) ($\exists$ *t1*$\in$(*transitions M*). *t-source t1* = *s1* $\wedge$ *t-input t1* = *t-input t* $\wedge$ *t-output t1* = *t-output t*) |
(*b*) ($\exists$ *t2*$\in$(*transitions M*). *t-source t2* = *s2* $\wedge$ *t-input t2* = *t-input t* $\wedge$

t-output t2 = *t-output t*)
using *canonical-separator-transition-ex*[*OF* ‹*t* ∈ *transitions ?C*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*t-source t* = *Inl* (*s1*,*s2*)›] **by** *blast*
then show *?case* **proof** *cases*
case *a*
then obtain *t1* **where** *t1* ∈ *transitions M* **and** *t-source t1* = *s1* **and** *t-input t1* = *t-input t* **and** *t-output t1* = *t-output t*
by *blast*

have *t-source t1* = *target q1 ?pL*
using ‹*target q1 ?pL* = *s1*› ‹*t-source t1* = *s1*› **by** *auto*
then have *path M q1* (*?pL*@[*t1*])
using *pL* ‹*t1* ∈ *transitions M*›
by (*meson from-FSM-path path-append-transition snoc.prems*(*2*))
moreover have *p-io* (*?pL*@[*t1*]) = *p-io* (*p*@[*t*])
using ∗ ‹*t-input t1* = *t-input t*› ‹*t-output t1* = *t-output t*› **by** *auto*
ultimately show *?thesis*
using ‹(∃ *p1 p2*. *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *butlast* (*p-io* (*p* @ [*t*])) ∧ *p-io p2* = *butlast* (*p-io* (*p* @ [*t*])))›
by *meson*
next
case *b*
then obtain *t2* **where** *t2* ∈ *transitions M* **and** *t-source t2* = *s2* **and** *t-input t2* = *t-input t* **and** *t-output t2* = *t-output t*
by *blast*

have *t-source t2* = *target q2 ?pR*
using ‹*target q2 ?pR* = *s2*› ‹*t-source t2* = *s2*› **by** *auto*
then have *path M q2* (*?pR*@[*t2*])
using *pR* ‹*t2* ∈ *transitions M*›
by (*meson from-FSM-path path-append-transition snoc.prems*(*3*))
moreover have *p-io* (*?pR*@[*t2*]) = *p-io* (*p*@[*t*])
using ∗ ‹*t-input t2* = *t-input t*› ‹*t-output t2* = *t-output t*› **by** *auto*
ultimately show *?thesis*
using ‹(∃ *p1 p2*. *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *butlast* (*p-io* (*p* @ [*t*])) ∧ *p-io p2* = *butlast* (*p-io* (*p* @ [*t*])))›
by *meson*
qed
qed
then show (∃ *p1* . *path M q1 p1* ∧ *p-io p1* = *p-io p*) ∨ (∃ *p2* . *path M q2 p2* ∧ *p-io p2* = *p-io p*)
and (∃ *p1 p2* . *path M q1 p1* ∧ *path M q2 p2* ∧ *p-io p1* = *butlast* (*p-io p*) ∧ *p-io p2* = *butlast* (*p-io p*))
by *blast*+
qed

lemma *canonical-separator-language* :
assumes *q1* ∈ *states M*

and $q2 \in states\ M$
shows L (*canonical-separator M q1 q2*) $\subseteq L$ (*from-FSM M q1*) $\cup L$ (*from-FSM M q2*) (**is** L *?C* $\subseteq L$ *?M1* $\cup L$ *?M2*)
proof
fix *io* **assume** $io \in L$ (*canonical-separator M q1 q2*)
then obtain *p* **where** $*$: *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p* **and** $**$: *p-io p* = *io*
by *auto*

show $io \in L$ (*from-FSM M q1*) $\cup L$ (*from-FSM M q2*)
using *canonical-separator-path-initial-ex*[*OF* $*$ *assms*] **unfolding** $**$
using *from-FSM-path-initial*[*OF assms*(*1*)] *from-FSM-path-initial*[*OF assms*(*2*)]

unfolding *LS.simps* **by** *blast*
qed

lemma *canonical-separator-language-prefix* :
assumes *io*@[*xy*] $\in L$ (*canonical-separator M q1 q2*)
and $q1 \in states\ M$
and $q2 \in states\ M$
and *observable M*
and $q1 \neq q2$
shows $io \in LS\ M\ q1$
and $io \in LS\ M\ q2$
proof −
let *?C* = (*canonical-separator M q1 q2*)
obtain *p* **where** *path ?C* (*initial ?C*) *p* **and** *p-io p* = *io*@[*xy*]
using *assms*(*1*) **by** *auto*

consider (*a*) ($\exists$ *s1′ s2′*. *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1′*, *s2′*)) |
(*b*) *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1* |
(*c*) *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*
using *canonical-separator-path-initial*(*4*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*2*,*3*,*4*,*5*)] **by** *blast*
then have $io \in LS\ M\ q1 \wedge io \in LS\ M\ q2$
proof *cases*
case *a*
then obtain *s1 s2* **where** $*$: *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inl* (*s1*, *s2*)
by *blast*
show *?thesis* **using** *canonical-separator-path-initial*(*1*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*2*,*3*,*4*,*5*) $*$] *language-prefix*
by (*metis* (*mono-tags*, *lifting*) *LS.simps* ‹*p-io p* = *io* @ [*xy*]› *mem-Collect-eq*
)
next
case *b*
show *?thesis* **using** *canonical-separator-path-initial*(*2*)[*OF* ‹*path ?C* (*initial*

?C) *p*› *assms*(*2*,*3*,*4*,*5*) *b*]
using ‹*p-io p* = *io* @ [*xy*]› **by** *fastforce*
next
case *c*
show *?thesis* **using** *canonical-separator-path-initial*(*3*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*2*,*3*,*4*,*5*) *c*]
using ‹*p-io p* = *io* @ [*xy*]› **by** *fastforce*
qed
then show *io* ∈ *LS M q1* **and** *io* ∈ *LS M q2*
by *auto*
qed

lemma *canonical-separator-distinguishing-transitions-left-containment* :
assumes *t* ∈ (*distinguishing-transitions-left M q1 q2*)
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *t* ∈ *transitions* (*canonical-separator M q1 q2*)
using *assms*(*1*) **unfolding** *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)]
by *blast*

lemma *canonical-separator-distinguishing-transitions-right-containment* :
assumes *t* ∈ (*distinguishing-transitions-right M q1 q2*)
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *t* ∈ *transitions* (*canonical-separator M q1 q2*) (**is** *t* ∈ *transitions ?C*)
using *assms*(*1*) **unfolding** *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)]
by *blast*

lemma *distinguishing-transitions-left-alt-intro* :
assumes (*s1*,*s2*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
and (∃ *t* ∈ *transitions M*. *t-source t* = *s1* ∧ *t-input t* = *x* ∧ *t-output t* = *y*)
and ¬(∃ *t* ∈ *transitions M*. *t-source t* = *s2* ∧ *t-input t* = *x* ∧ *t-output t* = *y*)
shows (*Inl* (*s1*,*s2*), *x*, *y*, *Inr q1*) ∈ *distinguishing-transitions-left-alt M q1 q2*
using *assms* **unfolding** *distinguishing-transitions-left-alt-def*
by *auto*

lemma *distinguishing-transitions-left-right-intro* :
assumes (*s1*,*s2*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
and ¬(∃ *t* ∈ *transitions M*. *t-source t* = *s1* ∧ *t-input t* = *x* ∧ *t-output t* = *y*)
and (∃ *t* ∈ *transitions M*. *t-source t* = *s2* ∧ *t-input t* = *x* ∧ *t-output t* = *y*)
shows (*Inl* (*s1*,*s2*), *x*, *y*, *Inr q2*) ∈ *distinguishing-transitions-right-alt M q1 q2*
using *assms* **unfolding** *distinguishing-transitions-right-alt-def*
by *auto*

lemma *canonical-separator-io-from-prefix-left* :
  assumes *io* @ [*io1*] ∈ *LS M q1*
  and *io* ∈ *LS M q2*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
  and *observable M*
  and *q1* ≠ *q2*
shows *io* @ [*io1*] ∈ *L* (*canonical-separator M q1 q2*)
proof −
  let *?C* = *canonical-separator M q1 q2*

  obtain *p1* **where** *path M q1 p1* **and** *p-io p1* = *io* @ [*io1*]
    using ‹*io* @ [*io1*] ∈ *LS M q1*› **by** *auto*
  then have *p1* ≠ []
    by *auto*
  then obtain *pL tL* **where** *p1* = *pL* @ [*tL*]
    using *rev-exhaust* **by** *blast*
  then have *path M q1* (*pL*@[*tL*]) **and** *path M q1 pL* **and** *p-io pL* = *io* **and** *tL* ∈ *transitions M*
      and *t-input tL* = *fst io1* **and** *t-output tL* = *snd io1* **and** *p-io* (*pL*@[*tL*]) = *io* @ [*io1*]
    using ‹*path M q1 p1*› ‹*p-io p1* = *io* @ [*io1*]› **by** *auto*
  then have *pLf*: *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *pL*
      and *pLf'*: *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) (*pL*@[*tL*])
    using *from-FSM-path-initial*[*OF* ‹*q1* ∈ *states M*›] **by** *auto*

  obtain *pR* **where** *path M q2 pR* **and** *p-io pR* = *io*
    using ‹*io* ∈ *LS M q2*› **by** *auto*
  then have *pRf*: *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *pR*
    using *from-FSM-path-initial*[*OF* ‹*q2* ∈ *states M*›] **by** *auto*

  have *p-io pL* = *p-io pR*
    using ‹*p-io pL* = *io*› ‹*p-io pR* = *io*› **by** *auto*

  let *?pLR* = *zip-path pL pR*
  let *?pCLR* = *map shift-Inl ?pLR*
  let *?P* = *product* (*from-FSM M q1*) (*from-FSM M q2*)

  have *path ?P* (*initial ?P*) *?pLR*
  and *target* (*initial ?P*) *?pLR* = (*target q1 pL*, *target q2 pR*)
    using *product-path-from-paths*[*OF pLf pRf* ‹*p-io pL* = *p-io pR*›]
    unfolding *from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)] **by** *linarith+*

  have *path ?C* (*initial ?C*) *?pCLR*
      using *canonical-separator-path-shift*[*OF assms*(*3*,*4*)] ‹*path ?P* (*initial ?P*) *?pLR*›
    by *simp*

have *isl* (*target* (*initial ?C*) *?pCLR*)
unfolding *canonical-separator-simps*(*1*)[*OF assms*(*3*,*4*)] **by** (*cases ?pLR rule*: *rev-cases*; *auto*)
then obtain *s1 s2* **where** *target* (*initial ?C*) *?pCLR* = *Inl* (*s1*,*s2*)
by (*metis* (*no-types*, *lifting*) ‹*path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) (*map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) (*map* (λ*t*. ((*t-source* (*fst t*), *t-source* (*snd t*)), *t-input* (*fst t*), *t-output* (*fst t*), *t-target* (*fst t*), *t-target* (*snd t*))) (*zip pL pR*)))›
assms(*3*) *assms*(*4*) *assms*(*5*) *assms*(*6*) *canonical-separator-path-initial*(*4*) *sum.discI*(*2*))
then have *Inl* (*s1*,*s2*) ∈ *states ?C*
using *path-target-is-state*[*OF* ‹*path ?C* (*initial ?C*) *?pCLR*›] **by** *simp*
then have (*s1*,*s2*) ∈ *states ?P*
using *canonical-separator-states*[*OF* - *assms*(*3*,*4*)] **by** *force*

have *target* (*initial ?P*) *?pLR* = (*s1*,*s2*)
using ‹*target* (*initial ?C*) *?pCLR* = *Inl* (*s1*,*s2*)› *assms*(*3*,*4*)
unfolding *canonical-separator-simps*(*1*)[*OF assms*(*3*,*4*)] *product-simps*(*1*) *from-FSM-simps target.simps visited-states.simps*
by (*cases ?pLR rule*: *rev-cases*; *auto*)
then have *target q1 pL* = *s1* **and** *target q2 pR* = *s2*
using ‹*target* (*initial ?P*) *?pLR* = (*target q1 pL*, *target q2 pR*)› **by** *auto*
then have *t-source tL* = *s1*
using ‹*path M q1* (*pL*@[*tL*])› **by** *auto*

show *?thesis* **proof** (*cases* ∃ *tR* ∈ (*transitions M*) . *t-source tR* = *target q2 pR* ∧ *t-input tR* = *t-input tL* ∧ *t-output tR* = *t-output tL*)
case *True*
then obtain *tR* **where** *tR* ∈ (*transitions M*) **and** *t-source tR* = *target q2 pR* **and** *t-input tR* = *t-input tL* **and** *t-output tR* = *t-output tL*
by *blast*

have *t-source tR* ∈ *states M*
unfolding ‹*t-source tR* = *target q2 pR*› ‹*target q2 pR* = *s2*›
using ‹(*s1*,*s2*) ∈ *states ?P*› *product-simps*(*2*) *from-FSM-simps*(*2*) *assms*(*3*,*4*) **by** *simp*

then have *tR* ∈ *transitions M*
using ‹*tR* ∈ (*transitions M*)› ‹*t-input tR* = *t-input tL*› ‹*t-output tR* = *t-output tL*› ‹*tL* ∈ *transitions M*› **by** *auto*

then have *path M q2* (*pR*@[*tR*])
using ‹*path M q2 pR*› ‹*t-source tR* = *target q2 pR*› *path-append-transition* **by** *metis*
then have *pRf'*: *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) (*pR*@[*tR*])
using *from-FSM-path-initial*[*OF* ‹*q2* ∈ *states M*›] **by** *auto*

let *?PP* = (*zip-path* (*pL*@[*tL*]) (*pR*@[*tR*]))
let *?PC* = *map shift-Inl ?PP*

have *length pL* = *length pR*
using ‹*p-io pL* = *p-io pR*› *map-eq-imp-length-eq* **by** *blast*
moreover have *p-io* (*pL*@[*tL*]) = *p-io* (*pR*@[*tR*])
using ‹*p-io pR* = *io*› ‹*t-input tL* = *fst io1*› ‹*t-output tL* = *snd io1*› ‹*t-input tR* = *t-input tL*› ‹*t-output tR* = *t-output tL*› ‹*p-io* (*pL*@[*tL*]) = *io*@[*io1*]› **by** *auto*
ultimately have *p-io ?PP* = *p-io* (*pL*@[*tL*])
by (*induction pL pR rule*: *list-induct2*; *auto*)

have *p-io ?PC* = *p-io ?PP*
by *auto*

have *path ?P* (*initial ?P*) *?PP*
using *product-path-from-paths*(*1*)[*OF pLf′ pRf′* ‹*p-io* (*pL*@[*tL*]) = *p-io* (*pR*@[*tR*])›] **by** *assumption*
then have *path ?C* (*initial ?C*) *?PC*
using *canonical-separator-path-shift*[*OF assms*(*3*,*4*)] **by** *simp*
moreover have *p-io ?PC* = *io*@[*io1*]
using ‹*p-io* (*pL*@[*tL*]) = *io*@[*io1*]› ‹*p-io ?PP* = *p-io* (*pL*@[*tL*])› ‹*p-io ?PC* = *p-io ?PP*› **by** *simp*
ultimately have ∃ *p* . *path ?C* (*initial ?C*) *p* ∧ *p-io p* = *io*@[*io1*]
by *blast*
then show *?thesis* **unfolding** *LS.simps* **by** *force*
next
case *False*

let *?t* = (*Inl* (*s1*,*s2*), *t-input tL*, *t-output tL*, *Inr q1*)

have (*s1*,*s2*) ∈ *reachable-states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
by (*metis* (*no-types*, *lifting*) ‹*path* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) (*FSM.initial* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))) (*zip-path pL pR*)› ‹*target* (*FSM.initial* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))) (*zip-path pL pR*) = (*s1*, *s2*)› *reachable-states-intro*)
moreover have (∃ *tR*∈*FSM.transitions M*.
t-source tR = *target q1 pL* ∧ *t-input tR* = *t-input tL* ∧ *t-output tR* = *t-output tL*)
using ‹*tL* ∈ *transitions M*› ‹*path M q1* (*pL*@[*tL*])›
by *auto*
ultimately have *?t* ∈ (*distinguishing-transitions-left-alt M q1 q2*)
using *distinguishing-transitions-left-alt-intro*[*OF - - False*] ‹*q1* ≠ *q2*›
unfolding ‹*target q1 pL* = *s1*› ‹*target q2 pR* = *s2*›
using ‹(*s1*, *s2*) ∈ *FSM.states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))› **by** *blast*
then have *?t* ∈ *transitions ?C*
using *canonical-separator-distinguishing-transitions-left-containment*[*OF - assms*(*3*,*4*)] **unfolding** *distinguishing-transitions-left-alt-alt-def* **by** *blast*
then have *path ?C* (*initial ?C*) (*?pCLR*@[*?t*])

using ‹*path ?C* (*initial ?C*) *?pCLR*› ‹*target* (*initial ?C*) *?pCLR* = *Inl* (*s1*,*s2*)›

by (*simp add*: *path-append-transition*)

have *length pL* = *length pR*
using ‹*p-io pL* = *p-io pR*›
using *map-eq-imp-length-eq* **by** *blast*
then have *p-io ?pCLR* = *p-io pL*
by (*induction pL pR rule*: *list-induct2*; *auto*)
then have *p-io* (*?pCLR*@[*?t*]) = *io* @ [*io1*]
using ‹*p-io pL* = *io*› ‹*t-input tL* = *fst io1*› ‹*t-output tL* = *snd io1*›
by *auto*
then have ∃ *p* . *path ?C* (*initial ?C*) *p* ∧ *p-io p* = *io*@[*io1*]
using ‹*path ?C* (*initial ?C*) (*?pCLR*@[*?t*])› **by** *meson*
then show *?thesis*
unfolding *LS.simps* **by** *force*
qed
qed

lemma *canonical-separator-path-targets-language* :
assumes *path* (*canonical-separator M q1 q2*) (*initial* (*canonical-separator M q1 q2*)) *p*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
and (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) = *Inr q1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
and (*target* (*initial* (*canonical-separator M q1 q2*)) *p*) = *Inr q2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
and *p-io p* ∈ *LS M q1* ∩ *LS M q2* ⟹ *isl* (*target* (*initial* (*canonical-separator M q1 q2*)) *p*)
and *p-io p* ∈ *LS M q1* − *LS M q2* ⟹ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q1*
and *p-io p* ∈ *LS M q2* − *LS M q1* ⟹ *target* (*initial* (*canonical-separator M q1 q2*)) *p* = *Inr q2*
proof −
let *?C* = *canonical-separator M q1 q2*
let *?tgt* = *target* (*initial ?C*) *p*

show *isl ?tgt* ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
proof −
assume *isl ?tgt*

then obtain *s1 s2* **where** *?tgt* = *Inl* (*s1*,*s2*)
by (*metis isl-def old.prod.exhaust*)
then obtain *p1 p2* **where** *path M q1 p1* **and** *path M q2 p2* **and** *p-io p1* = *p-io p* **and** *p-io p2* = *p-io p*
using *canonical-separator-path-initial*(*1*)[*OF assms*(*1*) ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*observable M*› ‹*q1* ≠ *q2*› ‹*?tgt* = *Inl* (*s1*,*s2*)›] **by** *force*
then show *p-io p* ∈ *LS M q1* ∩ *LS M q2*
unfolding *LS.simps* **by** *force*
qed
moreover show *?tgt* = *Inr q1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
proof −
assume *?tgt* = *Inr q1*
obtain *p1 p2 t* **where** *path M q1* (*p1* @ [*t*]) **and** *path M q2 p2* **and** *p-io* (*p1* @ [*t*]) = *p-io p*
and *p-io p2* = *butlast* (*p-io p*) **and** (∄*p2*. *path M q2 p2* ∧ *p-io p2* = *p-io p*)
using *canonical-separator-path-initial*(*2*)[*OF assms*(*1*) ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*›
‹*observable M*› ‹*q1* ≠ *q2*› ‹*?tgt* = *Inr q1*›]
by *meson*

have *path M q1 p1*
using ‹*path M q1* (*p1*@[*t*])› **by** *auto*
have *p-io p1* = *butlast* (*p-io p*)
using ‹*p-io* (*p1* @ [*t*]) = *p-io p*›
by (*metis* (*no-types*, *lifting*) *butlast-snoc map-butlast*)

have *p-io p* ∈ *LS M q1*
using ‹*path M q1* (*p1*@[*t*])› ‹*p-io* (*p1* @ [*t*]) = *p-io p*› **unfolding** *LS.simps* **by** *force*
moreover have *p-io p* ∉ *LS M q2*
using ‹(∄*p2*. *path M q2 p2* ∧ *p-io p2* = *p-io p*)› **unfolding** *LS.simps* **by** *force*
moreover have *butlast* (*p-io p*) ∈ *LS M q1*
using ‹*path M q1 p1*› ‹*p-io p1* = *butlast* (*p-io p*)› **unfolding** *LS.simps* **by** *force*
moreover have *butlast* (*p-io p*) ∈ *LS M q2*
using ‹*path M q2 p2*› ‹*p-io p2* = *butlast* (*p-io p*)› **unfolding** *LS.simps* **by** *force*
ultimately show *p-io p* ∈ *LS M q1* − *LS M q2* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
by (*simp add*: *map-butlast*)
qed
moreover show *?tgt* = *Inr q2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1* ∧ *p-io* (*butlast p*) ∈ *LS M q1* ∩ *LS M q2*
proof −
assume *?tgt* = *Inr q2*
obtain *p1 p2 t* **where** *path M q2* (*p2* @ [*t*]) **and** *path M q1 p1* **and** *p-io* (*p2*

@ [t]) = p-io p
and p-io p1 = butlast (p-io p) **and** (∄ p2. path M q1 p2 ∧ p-io p2 = p-io p)
using canonical-separator-path-initial(3)[OF assms(1) ‹q1 ∈ states M› ‹q2 ∈ states M›
‹observable M› ‹q1 ≠ q2› ‹?tgt = Inr q2›]
by meson

have path M q2 p2
using ‹path M q2 (p2@[t])› **by** auto
have p-io p2 = butlast (p-io p)
using ‹p-io (p2 @ [t]) = p-io p›
by (metis (no-types, lifting) butlast-snoc map-butlast)

have p-io p ∈ LS M q2
using ‹path M q2 (p2@[t])› ‹p-io (p2 @ [t]) = p-io p› **unfolding** LS.simps **by** force
moreover have p-io p ∉ LS M q1
using ‹(∄ p2. path M q1 p2 ∧ p-io p2 = p-io p)› **unfolding** LS.simps **by** force
moreover have butlast (p-io p) ∈ LS M q1
using ‹path M q1 p1› ‹p-io p1 = butlast (p-io p)› **unfolding** LS.simps **by** force
moreover have butlast (p-io p) ∈ LS M q2
using ‹path M q2 p2› ‹p-io p2 = butlast (p-io p)› **unfolding** LS.simps **by** force
ultimately show p-io p ∈ LS M q2 − LS M q1 ∧ p-io (butlast p) ∈ LS M q1 ∩ LS M q2
by (simp add: map-butlast)
qed
moreover have isl ?tgt ∨ ?tgt = Inr q1 ∨ ?tgt = Inr q2
using canonical-separator-path-initial(4)[OF assms(1) ‹q1 ∈ states M› ‹q2 ∈ states M› ‹observable M› ‹q1 ≠ q2›] **by** force
ultimately show p-io p ∈ LS M q1 ∩ LS M q2 ⟹ isl (target (initial (canonical-separator M q1 q2)) p)
and p-io p ∈ LS M q1 − LS M q2 ⟹ target (initial (canonical-separator M q1 q2)) p = Inr q1
and p-io p ∈ LS M q2 − LS M q1 ⟹ target (initial (canonical-separator M q1 q2)) p = Inr q2
by blast+
qed

lemma canonical-separator-language-target :
assumes io ∈ L (canonical-separator M q1 q2)
and observable M
and q1 ∈ states M
and q2 ∈ states M
and q1 ≠ q2

shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q2*}
proof −
  let *?C* = *canonical-separator M q1 q2*
  obtain *p* **where** *path ?C* (*initial ?C*) *p* **and** *p-io p* = *io*
    using *assms*(*1*) **by** *force*

  show *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q1*}
  proof −
    assume *io* ∈ *LS M q1* − *LS M q2*
    then have *p-io p* ∈ *LS M q1* − *LS M q2*
      using ‹*p-io p* = *io*› **by** *auto*
    have *Inr q1* ∈ *io-targets ?C io* (*initial ?C*)
      using *canonical-separator-path-targets-language*(*5*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*2*,*3*,*4*,*5*) ‹*p-io p* ∈ *LS M q1* − *LS M q2*›]
      using ‹*path ?C* (*initial ?C*) *p*› **unfolding** *io-targets.simps*
      by (*metis* (*mono-tags*, *lifting*) ‹*p-io p* = *io*› *mem-Collect-eq*)
    then show *?thesis*
      by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *canonical-separator-observable observable-io-targets singletonD*)
  qed

  show *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets* (*canonical-separator M q1 q2*) *io* (*initial* (*canonical-separator M q1 q2*)) = {*Inr q2*}
  proof −
    assume *io* ∈ *LS M q2* − *LS M q1*
    then have *p-io p* ∈ *LS M q2* − *LS M q1*
      using ‹*p-io p* = *io*› **by** *auto*
    have *Inr q2* ∈ *io-targets ?C io* (*initial ?C*)
      using *canonical-separator-path-targets-language*(*6*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*2*,*3*,*4*,*5*) ‹*p-io p* ∈ *LS M q2* − *LS M q1*›]
      using ‹*path ?C* (*initial ?C*) *p*› **unfolding** *io-targets.simps*
      by (*metis* (*mono-tags*, *lifting*) ‹*p-io p* = *io*› *mem-Collect-eq*)
    then show *?thesis*
      by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *canonical-separator-observable observable-io-targets singletonD*)
  qed
qed

lemma *canonical-separator-language-intersection* :
  assumes *io* ∈ *LS M q1*
  and *io* ∈ *LS M q2*
  and *q1* ∈ *states M*
  and *q2* ∈ *states M*
shows *io* ∈ *L* (*canonical-separator M q1 q2*) (**is** *io* ∈ *L ?C*)

proof −
  let *?P* = *product* (*from-FSM M q1*) (*from-FSM M q2*)

  have *io* ∈ *L ?P*
    using ‹*io* ∈ *LS M q1*› ‹*io* ∈ *LS M q2*› *product-language*[*of from-FSM M q1 from-FSM M q2*]
    unfolding *from-FSM-language*[*OF* ‹*q1* ∈ *states M*›] *from-FSM-language*[*OF* ‹*q2* ∈ *states M*›]
    by *blast*
  then obtain *p* **where** *path ?P* (*initial ?P*) *p* **and** *p-io p* = *io*
    by *auto*
  then have ∗: *path ?C* (*initial ?C*) (*map shift-Inl p*)
    using *canonical-separator-path-shift*[*OF assms*(*3*,*4*)] **by** *auto*
  have ∗∗: *p-io* (*map shift-Inl p*) = *io*
    using ‹*p-io p* = *io*› **by** (*induction p*; *auto*)
  show *io* ∈ *L ?C*
    using *language-state-containment*[*OF* ∗ ∗∗] **by** *assumption*
qed

lemma *canonical-separator-deadlock* :
  assumes *q1* ∈ *states M*
      and *q2* ∈ *states M*
    shows *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q1*)
      and *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q2*)
  unfolding *deadlock-state.simps*
  by (*metis assms*(*1*) *assms*(*2*) *canonical-separator-t-source-isl sum.disc*(*2*))+

lemma *canonical-separator-isl-deadlock* :
  assumes *Inl* (*q1′*,*q2′*) ∈ *states* (*canonical-separator M q1 q2*)
      and *x* ∈ *inputs M*
      and *completely-specified M*
      and ¬(∃ *t* ∈ *transitions* (*canonical-separator M q1 q2*) . *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x* ∧ *isl* (*t-target t*))
      and *q1* ∈ *states M*
      and *q2* ∈ *states M*
obtains *y1 y2* **where** (*Inl* (*q1′*,*q2′*),*x*,*y1*,*Inr q1*) ∈ *transitions* (*canonical-separator M q1 q2*)
                (*Inl* (*q1′*,*q2′*),*x*,*y2*,*Inr q2*) ∈ *transitions* (*canonical-separator M q1 q2*)
proof −
  let *?C* = (*canonical-separator M q1 q2*)
  let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

  have (*q1′*,*q2′*) ∈ *states ?P*
    using *assms*(*1*) **unfolding** *canonical-separator-simps*[*OF assms*(*5*,*6*)] **by** *fast-force*
  then have (*q1′*,*q2′*) ∈ *states ?P*

using *reachable-state-is-state* **by** *force*
then have *q1′* ∈ *states M* **and** *q2′* ∈ *states M*
using *assms*(*5*,*6*) **by** *auto*
then obtain *y1 y2* **where** *y1* ∈ *h-out M* (*q1′*,*x*) **and** *y2* ∈ *h-out M* (*q2′*,*x*)
by (*metis* (*no-types*, *lifting*) *assms*(*2*,*3*) *h-out.simps completely-specified-alt-def mem-Collect-eq*)
moreover have *h-out M* (*q1′*,*x*) ∩ *h-out M* (*q2′*,*x*) = {}
proof (*rule ccontr*)
assume *h-out M* (*q1′*, *x*) ∩ *h-out M* (*q2′*, *x*) ≠ {}
then obtain *y* **where** *y* ∈ *h-out M* (*q1′*, *x*) ∩ *h-out M* (*q2′*, *x*) **by** *blast*
then obtain *q1″ q2″* **where** ((*q1′*,*q2′*),*x*,*y*,(*q1″*,*q2″*)) ∈ *transitions ?P*
unfolding *product-transitions-def h-out.simps* **using** *assms*(*5*,*6*) **by** *auto*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ *transitions ?C*
using ‹(*q1′*,*q2′*) ∈ *states ?P*› **unfolding** *canonical-separator-transitions-def*[*OF assms*(*5*,*6*)] *h-out.simps* **by** *blast*
then show *False*
using *assms*(*4*) **by** *auto*
qed
ultimately have *y1* ∈ *h-out M* (*q1′*,*x*) − *h-out M* (*q2′*,*x*)
and *y2* ∈ *h-out M* (*q2′*,*x*) − *h-out M* (*q1′*,*x*)
by *blast+*

let *?t1* = (*Inl* (*q1′*,*q2′*),*x*,*y1*,*Inr q1*)
let *?t2* = (*Inl* (*q1′*,*q2′*),*x*,*y2*,*Inr q2*)

have *?t1* ∈ *distinguishing-transitions-left M q1 q2*
using ‹(*q1′*,*q2′*) ∈ *states ?P*› ‹*y1* ∈ *h-out M* (*q1′*,*x*) − *h-out M* (*q2′*,*x*)›
unfolding *distinguishing-transitions-left-def* **by** *auto*
then have *?t1* ∈ *transitions* (*canonical-separator M q1 q2*)
unfolding *canonical-separator-transitions-def*[*OF assms*(*5*,*6*)] **by** *blast*

have *?t2* ∈ *distinguishing-transitions-right M q1 q2*
using ‹(*q1′*,*q2′*) ∈ *states ?P*› ‹*y2* ∈ *h-out M* (*q2′*,*x*) − *h-out M* (*q1′*,*x*)›
unfolding *distinguishing-transitions-right-def* **by** *auto*
then have *?t2* ∈ *transitions* (*canonical-separator M q1 q2*)
unfolding *canonical-separator-transitions-def*[*OF assms*(*5*,*6*)] **by** *blast*

show *?thesis*
using *that* ‹*?t1* ∈ *transitions* (*canonical-separator M q1 q2*)› ‹*?t2* ∈ *transitions* (*canonical-separator M q1 q2*)› **by** *blast*
qed

lemma *canonical-separator-deadlocks* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q1*)
and *deadlock-state* (*canonical-separator M q1 q2*) (*Inr q2*)
using *canonical-separator-t-source-isl*[*OF - assms*]
unfolding *deadlock-state.simps* **by** *force+*

lemma *state-separator-from-canonical-separator-language-target* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *io* ∈ *L A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*Inr q1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*Inr q2*}
and *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}
proof −
have *observable A*
using *state-separator-from-canonical-separator-observable*[*OF assms*(*1*,*3*,*4*,*5*)] **by** *assumption*

let *?C* = *canonical-separator M q1 q2*

obtain *p* **where** *path A* (*initial A*) *p* **and** *p-io p* = *io*
using *assms*(*2*) **by** *force*
then have *path ?C* (*initial ?C*) *p*
using *submachine-path-initial*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)]] **by** *auto*
then have *io* ∈ *L ?C*
using ‹*p-io p* = *io*› **by** *auto*

show *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*Inr q1*}
proof −
assume *io* ∈ *LS M q1* − *LS M q2*

have *target* (*initial A*) *p* = *Inr q1*
using *submachine-path-initial*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)] ‹*path A* (*initial A*) *p*›]
canonical-separator-language-target(*1*)[*OF* ‹*io* ∈ *L ?C*› *assms*(*3*,*4*,*5*,*6*) ‹*io* ∈ *LS M q1* − *LS M q2*›]
‹*p-io p* = *io*›
unfolding *io-targets.simps is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*4*,*5*)]
canonical-separator-simps product-simps from-FSM-simps[*OF assms*(*4*)] *from-FSM-simps*[*OF assms*(*5*)]
using *assms*(*4*) *assms*(*5*) *canonical-separator-initial* **by** *fastforce*
then have *Inr q1* ∈ *io-targets A io* (*initial A*)
using ‹*path A* (*initial A*) *p*› ‹*p-io p* = *io*› **unfolding** *io-targets.simps*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then show *io-targets A io* (*initial A*) = {*Inr q1*}
using *observable-io-targets*[*OF* ‹*observable A*› ‹*io* ∈ *L A*›]

by (*metis singletonD*)
qed

show *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*Inr q2*}
proof −
assume *io* ∈ *LS M q2* − *LS M q1*

have *target* (*initial A*) *p* = *Inr q2*
using *submachine-path-initial*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)] ‹*path A* (*initial A*) *p*›]
canonical-separator-language-target(*2*)[*OF* ‹*io* ∈ *L ?C*› *assms*(*3*,*4*,*5*,*6*) ‹*io* ∈ *LS M q2* − *LS M q1*›]
‹*p-io p* = *io*›
unfolding *io-targets.simps is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*4*,*5*)]
canonical-separator-simps product-simps from-FSM-simps[*OF assms*(*4*)] *from-FSM-simps*[*OF assms*(*5*)]
using *assms*(*4*) *assms*(*5*) *canonical-separator-initial* **by** *fastforce*
then have *Inr q2* ∈ *io-targets A io* (*initial A*)
using ‹*path A* (*initial A*) *p*› ‹*p-io p* = *io*› **unfolding** *io-targets.simps*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then show *io-targets A io* (*initial A*) = {*Inr q2*}
using *observable-io-targets*[*OF* ‹*observable A*› ‹*io* ∈ *L A*›]
by (*metis singletonD*)
qed

show *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}
proof −
let *?P* = *product* (*from-FSM M q1*) (*from-FSM M q2*)

assume *io* ∈ *LS M q1* ∩ *LS M q2*

have⋀ *q* . *q* ∈ *io-targets A io* (*initial A*) ⟹ *q* ∉ {*Inr q1*, *Inr q2*}
proof −
fix *q* **assume** *q* ∈ *io-targets A io* (*initial A*)
then obtain *p* **where** *q* = *target* (*initial A*) *p* **and** *path A* (*initial A*) *p* **and** *p-io p* = *io*
by *auto*
then have *path ?C* (*initial ?C*) *p*
using *submachine-path-initial*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)]] **by** *auto*
then have *isl* (*target* (*initial ?C*) *p*)
using *canonical-separator-path-targets-language*(*4*)[*OF* - ‹*observable M*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*q1* ≠ *q2*›]
using ‹*p-io p* = *io*› ‹*io* ∈ *LS M q1* ∩ *LS M q2*› **by** *auto*
then show *q* ∉ {*Inr q1*, *Inr q2*}
using ‹*q* = *target* (*initial A*) *p*›
unfolding *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*4*,*5*)]

unfolding *canonical-separator-simps product-simps from-FSM-simps* **by** *auto*
qed

then show *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}
by *blast*
qed
qed

lemma *state-separator-language-intersections-nonempty* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
shows ∃ *io* . *io* ∈ (*L A* ∩ *LS M q1*) − *LS M q2* **and** ∃ *io* . *io* ∈ (*L A* ∩ *LS M q2*) − *LS M q1*
proof −
have *Inr q1* ∈ *reachable-states A*
using *is-state-separator-from-canonical-separator-simps*(*6*)[*OF assms*(*1*)] **by** *assumption*
then obtain *p* **where** *path A* (*initial A*) *p* **and** *target* (*initial A*) *p* = *Inr q1*
unfolding *reachable-states-def* **by** *auto*
then have *p-io p* ∈ *LS M q1* − *LS M q2*
using *canonical-separator-maximal-path-distinguishes-left*[*OF assms*(*1*) - - *assms*(*2*,*3*,*4*,*5*)] **by** *blast*
moreover have *p-io p* ∈ *L A*
using ‹*path A* (*initial A*) *p*› **by** *auto*
ultimately show ∃ *io* . *io* ∈ (*L A* ∩ *LS M q1*) − *LS M q2* **by** *blast*

have *Inr q2* ∈ *reachable-states A*
using *is-state-separator-from-canonical-separator-simps*(*7*)[*OF assms*(*1*)] **by** *assumption*
then obtain *p′* **where** *path A* (*initial A*) *p′* **and** *target* (*initial A*) *p′* = *Inr q2*
unfolding *reachable-states-def* **by** *auto*
then have *p-io p′* ∈ *LS M q2* − *LS M q1*
using *canonical-separator-maximal-path-distinguishes-right*[*OF assms*(*1*) - - *assms*(*2*,*3*,*4*,*5*)] **by** *blast*
moreover have *p-io p′* ∈ *L A*
using ‹*path A* (*initial A*) *p′*› **by** *auto*
ultimately show ∃ *io* . *io* ∈ (*L A* ∩ *LS M q2*) − *LS M q1* **by** *blast*
qed

lemma *state-separator-language-inclusion* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*

and $q1 \in states\ M$
and $q2 \in states\ M$
shows $L\ A \subseteq LS\ M\ q1 \cup LS\ M\ q2$
using *canonical-separator-language*[*OF assms*(*2*,*3*)]
using *submachine-language*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)]]
unfolding *from-FSM-language*[*OF assms*(*2*)] *from-FSM-language*[*OF assms*(*3*)]
by *blast*

lemma *state-separator-from-canonical-separator-targets-left-inclusion* :
assumes *observable T*
and *observable M*
and $t1 \in states\ T$
and $q1 \in states\ M$
and $q2 \in states\ M$
and *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and $(inputs\ T) = (inputs\ M)$
and *path A* (*initial A*) *p*
and $p\text{-}io\ p \in LS\ M\ q1$
and $q1 \neq q2$
shows *target* (*initial A*) $p \neq Inr\ q2$
and *target* (*initial A*) $p = Inr\ q1 \vee isl$ (*target* (*initial A*) *p*)
proof −
let $?C = canonical\text{-}separator\ M\ q1\ q2$
have *c-path*: $\bigwedge p\ .\ path\ A\ (initial\ A)\ p \Longrightarrow path\ ?C\ (initial\ ?C)\ p$
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] *submachine-path-initial* **by** *metis*
have *path ?C* (*initial ?C*) *p*
using *assms*(*8*) *c-path* **by** *auto*

show *target* (*initial A*) $p \neq Inr\ q2$
proof
assume *target* (*initial A*) $p = Inr\ q2$
then have *target* (*initial ?C*) $p = Inr\ q2$
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **by** *auto*

have $(\nexists p1.\ path\ M\ q1\ p1 \wedge p\text{-}io\ p1 = p\text{-}io\ p)$
using *canonical-separator-path-initial*(*3*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*4*,*5*,*2*,*10*) ‹*target* (*initial ?C*) $p = Inr\ q2$›] **by** *blast*
then have $p\text{-}io\ p \notin LS\ M\ q1$
unfolding *LS.simps* **by** *force*
then show *False*
using ‹$p\text{-}io\ p \in LS\ M\ q1$› **by** *blast*
qed
then have *target* (*initial ?C*) $p \neq Inr\ q2$
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **un-**

folding *is-submachine.simps* **by** *simp*
then have *target* (*initial ?C*) *p* = *Inr q1* ∨ *isl* (*target* (*initial ?C*) *p*)
proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis* **unfolding** *canonical-separator-simps*[*OF assms*(*4*,*5*)] **by** *simp*
next
case (*snoc ys y*)
then show *?thesis*
by (*metis* ‹*path* (*canonical-separator M q1 q2*) (*FSM.initial* (*canonical-separator M q1 q2*)) *p*› ‹*target* (*FSM.initial* (*canonical-separator M q1 q2*)) *p* ≠ *Inr q2*› *assms*(*10*) *assms*(*2*) *assms*(*4*) *assms*(*5*) *canonical-separator-path-initial*(*4*) *isl-def*)

qed
then show *target* (*initial A*) *p* = *Inr q1* ∨ *isl* (*target* (*initial A*) *p*)
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **unfolding** *is-submachine.simps* **by** *simp*
qed

lemma *state-separator-from-canonical-separator-targets-right-inclusion* :
assumes *observable T*
and *observable M*
and *t1* ∈ *states T*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and (*inputs T*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS M q2*
and *q1* ≠ *q2*
shows *target* (*initial A*) *p* ≠ *Inr q1*
and *target* (*initial A*) *p* = *Inr q2* ∨ *isl* (*target* (*initial A*) *p*)
proof −
let *?C* = *canonical-separator M q1 q2*
have *c-path*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *path ?C* (*initial ?C*) *p*
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] *submachine-path-initial* **by** *metis*
have *path ?C* (*initial ?C*) *p*
using *assms*(*8*) *c-path* **by** *auto*

show *target* (*initial A*) *p* ≠ *Inr q1*
proof
assume *target* (*initial A*) *p* = *Inr q1*
then have *target* (*initial ?C*) *p* = *Inr q1*
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **by** *auto*

have ($\nexists$ *p1*. *path M q2 p1* $\wedge$ *p-io p1* = *p-io p*)
using *canonical-separator-path-initial*(*2*)[*OF* ‹*path ?C* (*initial ?C*) *p*› *assms*(*4*,*5*,*2*,*10*) ‹*target* (*initial ?C*) *p* = *Inr q1*›] **by** *blast*
then have *p-io p* $\notin$ *LS M q2*
unfolding *LS.simps* **by** *force*
then show *False*
using ‹*p-io p* $\in$ *LS M q2*› **by** *blast*
qed

then have *target* (*initial ?C*) *p* $\neq$ *Inr q1*
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **unfolding** *is-submachine.simps* **by** *simp*
then have *target* (*initial ?C*) *p* = *Inr q2* $\vee$ *isl* (*target* (*initial ?C*) *p*)
proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis* **unfolding** *canonical-separator-simps*[*OF assms*(*4*,*5*)] **by** *simp*
next
case (*snoc ys y*)
then show *?thesis*
by (*metis* ‹*path* (*canonical-separator M q1 q2*) (*FSM.initial* (*canonical-separator M q1 q2*)) *p*› ‹*target* (*FSM.initial* (*canonical-separator M q1 q2*)) *p* $\neq$ *Inr q1*› *assms*(*10*) *assms*(*2*) *assms*(*4*) *assms*(*5*) *canonical-separator-path-initial*(*4*) *isl-def*)
qed
then show *target* (*initial A*) *p* = *Inr q2* $\vee$ *isl* (*target* (*initial A*) *p*)
using *is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*6*)] **unfolding** *is-submachine.simps* **by** *simp*
qed

34.2 Calculating State Separators

34.2.1 Sufficient Condition to Induce a State Separator

definition *state-separator-from-input-choices* :: ('*a*,'*b*,'*c*) *fsm* $\Rightarrow$ (('*a* $\times$ '*a*) + '*a*,'*b*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ '*a* $\Rightarrow$ ((('*a* $\times$ '*a*) + '*a*) $\times$ '*b*) *list* $\Rightarrow$ (('*a* $\times$ '*a*) + '*a*, '*b*, '*c*) *fsm* **where**
state-separator-from-input-choices M CSep q1 q2 cs =
(*let css* = *set cs*;
cssQ = (*set* (*map fst cs*)) $\cup$ {*Inr q1*, *Inr q2*};
S0 = *filter-states CSep* (λ *q* . *q* $\in$ *cssQ*);
S1 = *filter-transitions S0* (λ *t* . (*t-source t*, *t-input t*) $\in$ *css*)
in S1)

lemma *state-separator-from-input-choices-simps* :
assumes *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and $\bigwedge$ *qq x* . (*qq*,*x*) $\in$ *set cs* $\Longrightarrow$ *qq* $\in$ *states* (*canonical-separator M q1 q2*) $\wedge$ *x* $\in$ *inputs M*

and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
shows
initial (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *Inl* (*q1*,*q2*)
states (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}
inputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *inputs M*
outputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *outputs M*
transitions (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ (*transitions* (*canonical-separator M q1 q2*)) . ∃ *q1′ q2′ x* . (*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x* ∧ *t-target t* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}}
proof −
let *?SS* = (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
let *?S0* = *filter-states* (*canonical-separator M q1 q2*) (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*})

have (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))
unfolding *canonical-separator-simps*[*OF assms*(*1*,*2*)]
using *assms*(*4*) **by** *simp*

have *states ?S0* = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}
proof −
have ⋀ *qq* . *qq* ∈ *states ?S0* ⟹ *qq* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}
unfolding *filter-states-simps*[*of* (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}),
OF ‹(λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))›]
by *fastforce*
moreover have ⋀ *qq* . *qq* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*} ⟹ *qq* ∈ *states ?S0*
proof −
fix *qq* **assume** *qq* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}
then consider (*a*) *qq* ∈ *set* (*map fst cs*) | (*b*) *qq* ∈ {*Inr q1*, *Inr q2*}
by *blast*
then show *qq* ∈ *states ?S0* **proof** *cases*
case *a*
then obtain *q1′ q2′* **where** *qq* = *Inl* (*q1′*,*q2′*)
using *assms*(*5*) **by** (*metis old.prod.exhaust*)
then show *?thesis*
using *a assms*(*3*)[*of qq*]
unfolding *filter-states-simps*[*of* (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}), *OF* ‹(λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial*

(*canonical-separator M q1 q2*))›]
canonical-separator-simps[*OF assms*(*1*,*2*)] **by** *force*
next
case *b*
then show *?thesis* **using** *assms*(*3*)
unfolding *filter-states-simps*[*of* (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}), *OF* ‹(λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))›]
canonical-separator-simps[*OF assms*(*1*,*2*)] **by** *force*
qed
qed

ultimately show *?thesis* **by** *blast*
qed

show *initial* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *Inl* (*q1*,*q2*)
states (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}
inputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *inputs M*
outputs (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) = *outputs M*
unfolding *canonical-separator-simps*[*OF assms*(*1*,*2*)]
filter-transitions-simps
state-separator-from-input-choices-def
Let-def
filter-states-simps(*1*,*3*,*4*,*5*)[*of* (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}), *OF* ‹(λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))›]
‹*states ?S0* = (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}›
by *simp*+

have *alt-def-shared*: {*t* ∈ {*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*). *t-source t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*} ∧ *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}}. (*t-source t*, *t-input t*) ∈ *set cs*}
= {*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*). ∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧ *t-source t* = *Inl* (*q1′*, *q2′*) ∧ *t-input t* = *x* ∧ *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}}
(**is** *?ts1* = *?ts2*)
proof −
have ⋀ *t* . *t* ∈ *?ts1* ⟹ *t* ∈ *?ts2*
proof −
fix *t* **assume** *t* ∈ *?ts1*
then have *t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*) **and** *t-source t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*} **and** *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*} **and** (*t-source t*, *t-input t*) ∈ *set cs*
by *blast*+

have *t-source t* ∈ *set* (*map fst cs*)
using ‹*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*)› ‹*t-source t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}›
using *canonical-separator-deadlocks*[*OF assms*(*1*,*2*)]
by *fastforce*
then obtain *q1′ q2′* **where** *t-source t* = *Inl* (*q1′*,*q2′*)
using *assms*(*5*) **by** (*metis old.prod.exhaust*)
then have ∃ *q1′ q2′ x*. (*Inl* (*q1′*, *q2′*), *x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1′*, *q2′*) ∧ *t-input t* = *x*
using ‹(*t-source t*, *t-input t*) ∈ *set cs*› **by** *auto*

then show *t* ∈ *?ts2*
using ‹*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*)› ‹*t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}›
by *simp*
qed
moreover have ⋀ *t* . *t* ∈ *?ts2* ⟹ *t* ∈ *?ts1*
by *force*
ultimately show *?thesis* **by** *blast*
qed

show *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ (*transitions* (*canonical-separator M q1 q2*)) . ∃ *q1′ q2′ x* . (*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x* ∧ *t-target t* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}}
unfolding *canonical-separator-simps*(*1*,*2*,*3*,*4*)[*OF assms*(*1*,*2*)]
unfolding *state-separator-from-input-choices-def Let-def*
unfolding *filter-transitions-simps*
unfolding *filter-states-simps*[*of* (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}), *OF* ‹(λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))›]
unfolding *alt-def-shared* **by** *blast*
qed

lemma *state-separator-from-input-choices-submachine* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
shows *is-submachine* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) (*canonical-separator M q1 q2*)
proof −
have (λ *q* . *q* ∈ (*set* (*map fst cs*)) ∪ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))

unfolding *canonical-separator-simps*[*OF assms*(*1*,*2*)]
using *assms*(*4*) **by** *simp*

show *?thesis*
unfolding *state-separator-from-input-choices-def Let-def*
using *submachine-transitive*[*OF filter-states-submachine*[*of* (λ *q* . *q* $\in$ (*set* (*map fst cs*)) $\cup$ {*Inr q1*, *Inr q2*}), *OF* ‹(λ *q* . *q* $\in$ (*set* (*map fst cs*)) $\cup$ {*Inr q1*, *Inr q2*}) (*initial* (*canonical-separator M q1 q2*))›]
filter-transitions-submachine[*of filter-states* (*canonical-separator M q1 q2*) (λ*q*. *q* $\in$ *set* (*map fst cs*) $\cup$ {*Inr q1*, *Inr q2*}) (λ*t*. (*t-source t*, *t-input t*) $\in$ *set cs*)]]
by *assumption*
qed

lemma *state-separator-from-input-choices-single-input* :
assumes *distinct* (*map fst cs*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and $\bigwedge$ *qq x* . (*qq*,*x*) $\in$ *set cs* $\Longrightarrow$ *qq* $\in$ *states* (*canonical-separator M q1 q2*) $\wedge$ *x* $\in$ *inputs M*
and *Inl* (*q1*,*q2*) $\in$ *set* (*map fst cs*)
and $\bigwedge$ *qq* . *qq* $\in$ *set* (*map fst cs*) $\Longrightarrow$ $\exists$ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
shows *single-input* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
proof −
have $\bigwedge$ *t1 t2* . *t1* $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) $\Longrightarrow$
t2 $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) $\Longrightarrow$
t-source t1 = *t-source t2* $\Longrightarrow$ *t-input t1* = *t-input t2*
proof −
fix *t1 t2*
assume *t1* $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
and *t2* $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
and *t-source t1* = *t-source t2*

obtain *q1′ q2′* **where** (*Inl* (*q1′*,*q2′*),*t-input t1*) $\in$ *set cs*
and *t-source t1* = *Inl* (*q1′*,*q2′*)
using ‹*t1* $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)›
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *fastforce*

obtain *q1″ q2″* **where** (*Inl* (*q1″*,*q2″*),*t-input t2*) $\in$ *set cs*
and *t-source t2* = *Inl* (*q1″*,*q2″*)
using ‹*t2* $\in$ *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator*

M q1 q2) q1 q2 cs)›
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *fastforce*

have (*Inl* (*q1′*,*q2′*),*t-input t2*) ∈ *set cs*
using ‹(*Inl* (*q1″*,*q2″*),*t-input t2*) ∈ *set cs*› ‹*t-source t1* = *Inl* (*q1′*,*q2′*)› ‹*t-source t2* = *Inl* (*q1″*,*q2″*)› ‹*t-source t1* = *t-source t2*›
by *simp*
then show *t-input t1* = *t-input t2*
using ‹(*Inl* (*q1′*,*q2′*),*t-input t1*) ∈ *set cs*› ‹*distinct* (*map fst cs*)›
by (*meson eq-key-imp-eq-value*)
qed
then show *?thesis*
by *fastforce*
qed

lemma *state-separator-from-input-choices-transition-list* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and *t* ∈ *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
shows (*t-source t*, *t-input t*) ∈ *set cs*
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*1*,*2*,*3*,*4*,*5*)] *assms*(*6*) **by** *auto*

lemma *state-separator-from-input-choices-transition-target* :
assumes *t* ∈ *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
shows *t* ∈ *transitions* (*canonical-separator M q1 q2*) ∨ *t-target t* ∈ {*Inr q1*, *Inr q2*}
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*2*−*6*)] *assms*(*1*) **by** *fastforce*

lemma *state-separator-from-input-choices-acyclic-paths′* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*

and $q2 \in states\ M$
and $\bigwedge qq\ x\ .\ (qq,x) \in set\ cs \Longrightarrow qq \in states\ (canonical\text{-}separator\ M\ q1\ q2)$
$\wedge\ x \in inputs\ M$
and $Inl\ (q1,q2) \in set\ (map\ fst\ cs)$
and $\bigwedge qq\ .\ qq \in set\ (map\ fst\ cs) \Longrightarrow \exists\ q1'\ q2'\ .\ qq = Inl\ (q1',q2')$
and $\bigwedge i\ t\ .\ i < length\ cs$
$\Longrightarrow t \in transitions\ (canonical\text{-}separator\ M\ q1\ q2)$
$\Longrightarrow t\text{-}source\ t = (fst\ (cs\ !\ i))$
$\Longrightarrow t\text{-}input\ \ t = snd\ (cs\ !\ i)$
$\Longrightarrow t\text{-}target\ t \in ((set\ (map\ fst\ (take\ i\ cs))) \cup \{Inr\ q1,\ Inr\ q2\})$
and $path\ (state\text{-}separator\text{-}from\text{-}input\text{-}choices\ M\ (canonical\text{-}separator\ M\ q1\ q2)\ q1\ q2\ cs)\ q'\ p$
and $target\ q'\ p = q'$
and $p \neq []$
shows *False*
proof −

let $?S = (state\text{-}separator\text{-}from\text{-}input\text{-}choices\ M\ (canonical\text{-}separator\ M\ q1\ q2)\ q1\ q2\ cs)$

from ‹$p \neq []$› **obtain** $p'\ t'$ **where** $p = t'\#p'$
using *list.exhaust* **by** *blast*
then have $path\ ?S\ q'\ (p@[t'])$
using *assms*(*8*,*9*) **by** *fastforce*

define $f :: (('a \times 'a + 'a) \times 'b \times 'c \times ('a \times 'a + 'a)) \Rightarrow nat$
where *f-def*: $f = (\lambda\ t\ .\ the\ (find\text{-}index\ (\lambda\ qx\ .\ (fst\ qx) = t\text{-}source\ t \wedge snd\ qx = t\text{-}input\ t)\ cs))$

have *f-prop*: $\bigwedge t\ .\ t \in set\ (p@[t']) \Longrightarrow (f\ t < length\ cs)$
$\wedge\ (\lambda(q,\ x).\ (q,\ x))\ (cs\ !\ (f\ t)) = (t\text{-}source\ t, t\text{-}input\ t)$
$\wedge\ (\forall\ j < f\ t\ .\ (fst\ (cs\ !\ j)) \neq t\text{-}source\ t)$
proof −
fix t **assume** $t \in set\ (p@[t'])$
then have $t \in set\ p$ **using** ‹$p = t'\#p'$› **by** *auto*
then have $t \in transitions\ ?S$
using *assms*(*8*)
by (*meson path-transitions subsetD*)
then have $(t\text{-}source\ t,\ t\text{-}input\ t) \in set\ cs$
using *state-separator-from-input-choices-transition-list*[*OF assms*(*2*,*3*,*4*,*5*,*6*)]
by *blast*
then have $\exists\ qx \in set\ cs\ .\ (\lambda\ qx\ .\ (fst\ qx) = t\text{-}source\ t \wedge snd\ qx = t\text{-}input\ t)\ qx$
by *force*
then have $find\text{-}index\ (\lambda\ qx\ .\ (fst\ qx) = t\text{-}source\ t \wedge snd\ qx = t\text{-}input\ t)\ cs \neq None$
by (*simp add*: *find-index-exhaustive*)
then obtain i **where** $*$: $find\text{-}index\ (\lambda\ qx\ .\ (fst\ qx) = t\text{-}source\ t \wedge snd\ qx = t\text{-}input\ t)\ cs = Some\ i$
by *auto*

have $**$: $\bigwedge j\ .\ j < i \Longrightarrow (fst\ (cs\ !\ j)) = t\text{-}source\ t \Longrightarrow cs\ !\ i = cs\ !\ j$
using *assms*(*1*)
using *nth-eq-iff-index-eq find-index-index*[*OF* $*$]
by (*metis* (*mono-tags*, *lifting*) *Suc-lessD length-map less-trans-Suc nth-map*)

have $f\ t < length\ cs$
unfolding *f-def* **using** *find-index-index*(*1*)[*OF* $*$] **unfolding** $*$ **by** *simp*
moreover have $(\lambda(q,\ x).\ (q,\ x))\ (cs\ !\ (f\ t)) = (t\text{-}source\ t,\ t\text{-}input\ t)$
unfolding *f-def* **using** *find-index-index*(*2*)[*OF* $*$]
by (*metis* $*$ *case-prod-Pair-iden option.sel prod.collapse*)
moreover have $\forall\ j < f\ t\ .\ (fst\ (cs\ !\ j)) \neq t\text{-}source\ t$
unfolding *f-def* **using** *find-index-index*(*3*)[*OF* $*$] **unfolding** $*$
using *assms*(*1*) $**$
by (*metis* (*no-types*, *lifting*) $*$ ‹$\exists\ qx \in set\ cs.\ fst\ qx = t\text{-}source\ t \wedge snd\ qx = t\text{-}input\ t$› *eq-key-imp-eq-value find-index-index*(*1*) *nth-mem option.sel prod.collapse*)

ultimately show $(f\ t < length\ cs)$
$\wedge\ (\lambda(q,\ x).\ (q,\ x))\ (cs\ !\ (f\ t)) = (t\text{-}source\ t, t\text{-}input\ t)$
$\wedge\ (\forall\ j < f\ t\ .\ (fst\ (cs\ !\ j)) \neq t\text{-}source\ t)$ **by** *simp*
qed

have $*$: $\bigwedge i\ .\ Suc\ i < length\ (p@[t']) \Longrightarrow f\ ((p@[t])\ !\ i) > f\ ((p@[t])\ !\ (Suc\ i))$
proof −
fix i **assume** $Suc\ i < length\ (p@[t'])$
then have $(p@[t])\ !\ i \in set\ (p@[t])$ **and** $(p@[t])\ !\ (Suc\ i) \in set\ (p@[t])$
using *Suc-lessD nth-mem* **by** *blast*+
then have $(p@[t])\ !\ i \in transitions\ ?S$ **and** $(p@[t])\ !\ Suc\ i \in transitions\ ?S$
using ‹*path ?S q′* $(p@[t])$›
by (*meson path-transitions subsetD*)+
then have $(p\ @\ [t])\ !\ i \in FSM.transitions\ (canonical\text{-}separator\ M\ q1\ q2) \vee t\text{-}target\ ((p\ @\ [t])\ !\ i) \in \{Inr\ q1,\ Inr\ q2\}$
using *state-separator-from-input-choices-transition-target*[*OF* - *assms*(*2*−*6*)] **by** *blast*

have $f\ ((p@[t])\ !\ i) < length\ cs$
and $(\lambda(q,\ x).\ (q,\ x))\ (cs\ !\ (f\ ((p@[t])\ !\ i))) = (t\text{-}source\ ((p@[t])\ !\ i),\ t\text{-}input\ ((p@[t])\ !\ i))$
and $(\forall j{<}f\ ((p@[t])\ !\ i).\ (fst\ (cs\ !\ j)) \neq t\text{-}source\ ((p@[t])\ !\ i))$
using *f-prop*[*OF* ‹$(p@[t])\ !\ i \in set\ (p@[t])$›] **by** *auto*

have $f\ ((p@[t])\ !\ Suc\ i) < length\ cs$
and $(\lambda(q,\ x).\ (q,\ x))\ (cs\ !\ (f\ ((p@[t])\ !\ Suc\ i))) = (t\text{-}source\ ((p@[t])\ !\ Suc\ i),\ t\text{-}input\ ((p@[t])\ !\ Suc\ i))$
and $(\forall j{<}f\ ((p@[t])\ !\ Suc\ i).\ (fst\ (cs\ !\ j)) \neq t\text{-}source\ ((p@[t])\ !\ Suc\ i))$
using *f-prop*[*OF* ‹$(p@[t])\ !\ Suc\ i \in set\ (p@[t])$›] **by** *auto*

have $t\text{-}source\ ((p\ @\ [t])\ !\ i) = (fst\ (cs\ !\ f\ ((p\ @\ [t])\ !\ i)))$ **and** $t\text{-}input\ ((p\ @\ [t])\ !\ i) = snd\ (cs\ !\ f\ ((p\ @\ [t])\ !\ i))$

using *f-prop*[*OF* ‹(*p*@[*t'*]) ! *i* ∈ *set* (*p*@[*t'*])›]
by (*simp add*: *prod.case-eq-if*)+

have *t-target* ((*p*@[*t'*]) ! *i*) = *t-source* ((*p*@[*t'*]) ! *Suc i*)
using ‹*Suc i* < *length* (*p*@[*t'*])› ‹*path ?S q'* (*p*@[*t'*])›
by (*simp add*: *path-source-target-index*)
then have *t-target* ((*p*@[*t'*]) ! *i*) ∉ {*Inr q1*, *Inr q2*}
using *state-separator-from-input-choices-transition-list*[*OF assms*(*2*,*3*,*4*,*5*,*6*) ‹(*p*@[*t'*]) ! *Suc i* ∈ *transitions ?S*›] *assms*(*6*) **by** *force*
then have *t-target* ((*p* @ [*t'*]) ! *i*) ∈ *set* (*map fst* (*take* (*f* ((*p* @ [*t'*]) ! *i*)) *cs*))
using *assms*(*7*)[*OF* ‹*f* ((*p*@[*t'*]) ! *i*) < *length cs*› - ‹*t-source* ((*p* @ [*t'*]) ! *i*) = (*fst* (*cs* ! *f* ((*p* @ [*t'*]) ! *i*)))› ‹*t-input* ((*p* @ [*t'*]) ! *i*) = *snd* (*cs* ! *f* ((*p* @ [*t'*]) ! *i*))›]
using ‹(*p* @ [*t'*]) ! *i* ∈ *FSM.transitions* (*canonical-separator M q1 q2*) ∨ *t-target* ((*p* @ [*t'*]) ! *i*) ∈ {*Inr q1*, *Inr q2*}› **by** *blast*
then have (∃ *qx'*∈*set* (*take* (*f* ((*p*@[*t'*]) ! *i*)) *cs*). (*fst qx'*) = *t-target* ((*p*@[*t'*]) ! *i*))
by *force*
then obtain *j* **where** (*fst* (*cs* ! *j*)) = *t-source* ((*p*@[*t'*]) ! *Suc i*) **and** *j* < *f* ((*p*@[*t'*]) ! *i*)
unfolding ‹*t-target* ((*p*@[*t'*]) ! *i*) = *t-source* ((*p*@[*t'*]) ! *Suc i*)›
by (*metis* (*no-types*, *lifting*) ‹*f* ((*p*@[*t'*]) ! *i*) < *length cs*› *in-set-conv-nth leD length-take min-def-raw nth-take*)
then show *f* ((*p*@[*t'*]) ! *i*) > *f* ((*p*@[*t'*]) ! (*Suc i*))
using ‹(∀ *j*<*f* ((*p*@[*t'*]) ! *Suc i*). (*fst* (*cs* ! *j*)) ≠ *t-source* ((*p*@[*t'*]) ! *Suc i*))›
using *leI le-less-trans* **by** *blast*
qed

have ⋀ *i j* . *j* < *i* ⟹ *i* < *length* (*p*@[*t'*]) ⟹ *f* ((*p*@[*t'*]) ! *j*) > *f* ((*p*@[*t'*]) ! *i*)
using *list-index-fun-gt*[*of p*@[*t'*] *f*] ∗ **by** *blast*
then have *f t'* < *f t'*
unfolding ‹*p* = *t'*#*p'*› **by** *fastforce*
then show *False*
by *auto*
qed

lemma *state-separator-from-input-choices-acyclic-paths* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})

and *path* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) *q′ p*
shows *distinct* (*visited-states q′ p*)
proof (*rule ccontr*)
assume ¬ *distinct* (*visited-states q′ p*)

obtain *i j* **where** *p1*:*take j* (*drop i p*) ≠ []
and *p2*:*target* (*target q′* (*take i p*)) (*take j* (*drop i p*)) = (*target q′* (*take i p*))
and *p3*:*path* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) (*target q′* (*take i p*)) (*take j* (*drop i p*))
using *cycle-from-cyclic-path*[*OF assms*(*8*) ‹¬ *distinct* (*visited-states q′ p*)›] **by** *blast*

show *False*
using *state-separator-from-input-choices-acyclic-paths′*[*OF assms*(*1*−*7*) *p3 p2 p1*] **by** *blast*
qed

lemma *state-separator-from-input-choices-acyclic* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
shows *acyclic* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
unfolding *acyclic.simps* **using** *state-separator-from-input-choices-acyclic-paths*[*OF assms*] **by** *blast*

lemma *state-separator-from-input-choices-target* :
assumes ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
and *t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*)
and ∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧ *t-source t* = *Inl* (*q1′*, *q2′*) ∧ *t-input t* = *x*
shows *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}

proof −
from *assms*(*3*) **obtain** *q1′ q2′ x* **where** (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* **and** *t-source t* = *Inl* (*q1′*, *q2′*) **and** *t-input t* = *x*
by *auto*
then obtain *i* **where** *i* < *length cs* **and** *t-source t* = (*fst* (*cs* ! *i*)) **and** *t-input t* = *snd* (*cs* ! *i*)
by (*metis fst-conv in-set-conv-nth snd-conv*)
then have *t-target t* ∈ *set* (*map fst* (*take i cs*)) ∪ {*Inr q1*, *Inr q2*} **using** *assms*(*1*)[*OF* - *assms*(*2*)] **by** *blast*
then consider *t-target t* ∈ *set* (*map fst* (*take i cs*)) | *t-target t* ∈ {*Inr q1*, *Inr q2*} **by** *blast*
then show *?thesis* **proof** *cases*
case *1*
then have *t-target t* ∈ *set* (*map fst cs*)
by (*metis in-set-takeD take-map*)
then show *?thesis* **by** *blast*
next
case *2*
then show *?thesis* **by** *auto*
qed
qed

lemma *state-separator-from-input-choices-transitions-alt-def* :
assumes *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})
shows *transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ (*transitions* (*canonical-separator M q1 q2*)) . ∃ *q1′ q2′ x* . (*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs* ∧ *t-source t* = *Inl* (*q1′*,*q2′*) ∧ *t-input t* = *x*}
proof −
have *FSM.transitions* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) =
{*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*).
∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧
t-source t = *Inl* (*q1′*, *q2′*) ∧
t-input t = *x* ∧ *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}}
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*1*,*2*,*3*,*4*,*5*)] **by** *blast*

moreover have $\bigwedge$ *t* . *t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*) ⟹ ∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧ *t-source t* = *Inl* (*q1′*, *q2′*) ∧ *t-input t* = *x* ⟹ *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}
using *state-separator-from-input-choices-target*[*OF assms*(*6*)] **by** *blast*

ultimately show *?thesis*
by *fast*
qed

lemma *state-separator-from-input-choices-deadlock* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and $\bigwedge$ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and $\bigwedge$ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and $\bigwedge$ *i t* . *i* < *length cs*
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* (*cs* ! *i*))
⟹ *t-input t* = *snd* (*cs* ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i cs*))) ∪ {*Inr q1*, *Inr q2*})

shows $\bigwedge$ *qq* . *qq* ∈ *states* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) ⟹ *deadlock-state* (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*) *qq* ⟹ *qq* ∈ {*Inr q1*, *Inr q2*} ∨ (∃ *q1′ q2′ x* . *qq* = *Inl* (*q1′*,*q2′*) ∧ *x* ∈ *inputs M* ∧ (*h-out M* (*q1′*,*x*) = {} ∧ *h-out M* (*q2′*,*x*) = {}))
proof −
let *?C* = (*canonical-separator M q1 q2*)
let *?S* = (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)

fix *qq* **assume** *qq* ∈ *states ?S* **and** *deadlock-state ?S qq*

then consider (*a*) *qq* ∈ (*set* (*map fst cs*)) | (*b*) *qq* ∈ {*Inr q1*, *Inr q2*}
using *state-separator-from-input-choices-simps*(*2*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *blast*
then show *qq* ∈ {*Inr q1*, *Inr q2*} ∨ (∃ *q1′ q2′ x* . *qq* = *Inl* (*q1′*,*q2′*) ∧ *x* ∈ *inputs M* ∧ (*h-out M* (*q1′*,*x*) = {} ∧ *h-out M* (*q2′*,*x*) = {}))
proof *cases*
case *a*
then obtain *q1′ q2′ x* **where** (*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs* **and** *qq* = *Inl* (*q1′*,*q2′*) **using** *assms*(*6*) **by** *fastforce*
then have *Inl* (*q1′*,*q2′*) ∈ *states* (*canonical-separator M q1 q2*) **and** *x* ∈ *inputs M* **using** *assms*(*4*) **by** *blast+*
then have (*q1′*, *q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *canonical-separator-simps*(*2*)[*OF assms*(*2*,*3*)] **by** *fastforce*

have *h-out M* (*q1′*,*x*) = {} ∧ *h-out M* (*q2′*,*x*) = {}
proof (*rule ccontr*)
assume ¬ (*h-out M* (*q1′*, *x*) = {} ∧ *h-out M* (*q2′*, *x*) = {})
then consider (*a1*) ∃ *y* ∈ (*h-out M* (*q1′*, *x*) ∩ *h-out M* (*q2′*, *x*)) . *True* |
(*a2*) ∃ *y* ∈ (*h-out M* (*q1′*, *x*) − *h-out M* (*q2′*, *x*)) . *True* |
(*a3*) ∃ *y* ∈ (*h-out M* (*q2′*, *x*) − *h-out M* (*q1′*, *x*)) . *True*
by *blast*
then show *False* **proof** *cases*
case *a1*
then obtain *y q1″ q2″* **where** (*y*,*q1″*) ∈ *h M* (*q1′*,*x*) **and** (*y*,*q2″*) ∈ *h M* (*q2′*,*x*) **by** *auto*
then have ((*q1′*,*q2′*),*x*,*y*,(*q1″*,*q2″*)) ∈ *transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
unfolding *product-transitions-def h.simps* **using** *assms*(*2*,*3*) **by** *auto*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ *transitions ?C*
using ‹(*q1′*, *q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
canonical-separator-transitions-def[*OF assms*(*2*,*3*)] **by** *fast*

then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ {*t* ∈ *FSM.transitions* (*canonical-separator M q1 q2*).
∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧
t-source t = *Inl* (*q1′*, *q2′*) ∧
t-input t = *x* ∧ *t-target t* ∈ *set* (*map fst cs*) ∪ {*Inr q1*, *Inr q2*}}
using *state-separator-from-input-choices-target*[*OF assms*(*7*) ‹(*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ *transitions ?C*›]
using ‹(*Inl* (*q1′*, *q2′*), *x*) ∈ *set cs*› **by** *force*

then have (*Inl* (*q1′*,*q2′*), *x*, *y*, *Inl* (*q1″*,*q2″*)) ∈ *transitions ?S*
using *state-separator-from-input-choices-simps*(*5*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *fastforce*
then show *False*
using ‹*deadlock-state ?S qq*› **unfolding** ‹*qq* = *Inl* (*q1′*,*q2′*)› **by** *auto*
next
case *a2*
then obtain *y* **where** *y* ∈ (*h-out M* (*q1′*, *x*) − *h-out M* (*q2′*, *x*)) **unfolding** *h-out.simps* **by** *blast*
then have (∃ *q′*. (*q1′*, *x*, *y*, *q′*) ∈ *FSM.transitions M*) ∧ (∄ *q′*. (*q2′*, *x*, *y*, *q′*) ∈ *FSM.transitions M*) **unfolding** *h-out.simps* **by** *blast*
then have (*Inl* (*q1′*,*q2′*), *x*, *y*, *Inr q1*) ∈ *distinguishing-transitions-left M q1 q2*
unfolding *distinguishing-transitions-left-def h.simps*
using ‹(*q1′*, *q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))› **by** *blast*
then have (*Inl* (*q1′*,*q2′*), *x*, *y*, *Inr q1*) ∈ *transitions ?C*
unfolding *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)] **by** *blast*
moreover have ∃ *q1″ q2″ x′* . (*Inl* (*q1″*, *q2″*), *x′*)∈*set cs* ∧ *t-source* (*Inl* (*q1′*,*q2′*), *x*, *y*, *Inr q1*) = *Inl* (*q1″*, *q2″*) ∧ *t-input* (*Inl* (*q1′*,*q2′*), *x*, *y*, *Inr q1*) =

x'
using ‹(*Inl* (*q1'*, *q2'*), *x*) ∈ *set cs*› **by** *auto*
ultimately have (*Inl* (*q1'*,*q2'*), *x*, *y*, *Inr q1*) ∈ *transitions ?S*
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)]
by *blast*
then show *False*
using ‹*deadlock-state ?S qq*› **unfolding** ‹*qq* = *Inl* (*q1'*,*q2'*)› **by** *auto*
next
case *a3*
then obtain *y* **where** *y* ∈ (*h-out M* (*q2'*, *x*) − *h-out M* (*q1'*, *x*)) **unfolding**
h-out.simps **by** *blast*
then have ¬(∃ *q'*. (*q1'*, *x*, *y*, *q'*) ∈ *FSM.transitions M*) ∧ (∃ *q'*. (*q2'*, *x*, *y*,
q') ∈ *FSM.transitions M*) **unfolding** *h-out.simps* **by** *blast*
then have (*Inl* (*q1'*,*q2'*), *x*, *y*, *Inr q2*) ∈ *distinguishing-transitions-right M*
q1 q2
unfolding *distinguishing-transitions-right-def h.simps*
using ‹(*q1'*, *q2'*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
by *blast*
then have (*Inl* (*q1'*,*q2'*), *x*, *y*, *Inr q2*) ∈ *transitions ?C*
unfolding *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)] **by** *blast*
moreover have ∃ *q1'' q2'' x'* . (*Inl* (*q1''*, *q2''*), *x'*)∈*set cs* ∧ *t-source* (*Inl*
(*q1'*,*q2'*), *x*, *y*, *Inr q2*) = *Inl* (*q1''*, *q2''*) ∧ *t-input* (*Inl* (*q1'*,*q2'*), *x*, *y*, *Inr q2*) =
x'
using ‹(*Inl* (*q1'*, *q2'*), *x*) ∈ *set cs*› **by** *auto*
ultimately have (*Inl* (*q1'*,*q2'*), *x*, *y*, *Inr q2*) ∈ *transitions ?S*
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)]
by *blast*
then show *False*
using ‹*deadlock-state ?S qq*› **unfolding** ‹*qq* = *Inl* (*q1'*,*q2'*)› **by** *auto*
qed
qed
then show *?thesis*
using ‹*qq* = *Inl* (*q1'*, *q2'*)› ‹*x* ∈ *FSM.inputs M*› **by** *blast*
next
case *b*
then show *?thesis* **by** *simp*
qed
qed

lemma *state-separator-from-input-choices-retains-io* :
assumes *distinct* (*map fst cs*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and ⋀ *qq x* . (*qq*,*x*) ∈ *set cs* ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*)
∧ *x* ∈ *inputs M*
and *Inl* (*q1*,*q2*) ∈ *set* (*map fst cs*)
and ⋀ *qq* . *qq* ∈ *set* (*map fst cs*) ⟹ ∃ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
and ⋀ *i t* . *i* < *length cs*

$\Longrightarrow$ *t* $\in$ *transitions* (*canonical-separator M q1 q2*)
$\Longrightarrow$ *t-source t* = (*fst* (*cs* ! *i*))
$\Longrightarrow$ *t-input t* = *snd* (*cs* ! *i*)
$\Longrightarrow$ *t-target t* $\in$ ((*set* (*map fst* (*take i cs*))) $\cup$ {*Inr q1*, *Inr q2*})
shows *retains-outputs-for-states-and-inputs* (*canonical-separator M q1 q2*) (*state-separator-from-input-choic*
M (*canonical-separator M q1 q2*) *q1 q2 cs*)
unfolding *retains-outputs-for-states-and-inputs-def*
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)]
by *fastforce*

lemma *state-separator-from-input-choices-is-state-separator* :
assumes *distinct* (*map fst cs*)
and *q1* $\in$ *states M*
and *q2* $\in$ *states M*
and $\bigwedge$ *qq x* . (*qq*,*x*) $\in$ *set cs* $\Longrightarrow$ *qq* $\in$ *states* (*canonical-separator M q1 q2*)
$\wedge$ *x* $\in$ *inputs M*
and *Inl* (*q1*,*q2*) $\in$ *set* (*map fst cs*)
and $\bigwedge$ *qq* . *qq* $\in$ *set* (*map fst cs*) $\Longrightarrow$ $\exists$ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and $\bigwedge$ *i t* . *i* < *length cs*
$\Longrightarrow$ *t* $\in$ *transitions* (*canonical-separator M q1 q2*)
$\Longrightarrow$ *t-source t* = (*fst* (*cs* ! *i*))
$\Longrightarrow$ *t-input t* = *snd* (*cs* ! *i*)
$\Longrightarrow$ *t-target t* $\in$ ((*set* (*map fst* (*take i cs*))) $\cup$ {*Inr q1*, *Inr q2*})
and *completely-specified M*
shows *is-state-separator-from-canonical-separator*
(*canonical-separator M q1 q2*)
q1
q2
(*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1*
q2 cs)
proof −
let *?C* = (*canonical-separator M q1 q2*)
let *?S* = (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1*
q2 cs)

have *submachine-prop*: *is-submachine ?S ?C*
using *state-separator-from-input-choices-submachine*[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by**
blast

have *single-input-prop*: *single-input ?S*
using *state-separator-from-input-choices-single-input*[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*)]
by *blast*

have *acyclic-prop* : *acyclic ?S*
using *state-separator-from-input-choices-acyclic*[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)] **by**
blast

have *i3*: $\bigwedge$ *qq* . *qq* $\in$ *states ?S*

$\Longrightarrow$ *deadlock-state ?S qq*
$\Longrightarrow$ *qq* $\in$ {*Inr q1*, *Inr q2*}
$\vee$ ($\exists$ *q1′ q2′ x* . *qq* = *Inl* (*q1′*,*q2′*)
$\wedge$ *x* $\in$ *inputs M*
$\wedge$ *h-out M* (*q1′*,*x*) = {}
$\wedge$ *h-out M* (*q2′*,*x*) = {})
using *state-separator-from-input-choices-deadlock*[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
by *blast*

have *i4*: *retains-outputs-for-states-and-inputs* (*canonical-separator M q1 q2*) (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
using *state-separator-from-input-choices-retains-io*[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
by *blast*

have *deadlock-prop-1*: *deadlock-state ?S* (*Inr q1*)
using *submachine-deadlock*[*OF* ‹*is-submachine ?S ?C*› *canonical-separator-deadlock*(*1*)[*OF assms*(*2*,*3*)]] **by** *assumption*

have *deadlock-prop-2*: *deadlock-state ?S* (*Inr q2*)
using *submachine-deadlock*[*OF* ‹*is-submachine ?S ?C*› *canonical-separator-deadlock*(*2*)[*OF assms*(*2*,*3*)]] **by** *assumption*

have *non-deadlock-prop′*: $\bigwedge$ *qq* . *qq* $\in$ *states ?S* $\Longrightarrow$ *qq* $\neq$ *Inr q1* $\Longrightarrow$ *qq* $\neq$ *Inr q2* $\Longrightarrow$ (*isl qq* $\wedge$ $\neg$ *deadlock-state ?S qq*)
proof −
fix *qq* **assume** *qq* $\in$ *states ?S* **and** *qq* $\neq$ *Inr q1* **and** *qq* $\neq$ *Inr q2*
then have *qq* $\in$ *set* (*map fst cs*)
using *state-separator-from-input-choices-simps*(*2*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *blast*
then obtain *q1′ q2′ x* **where** *qq* = *Inl* (*q1′*,*q2′*) **and** (*Inl* (*q1′*,*q2′*),*x*) $\in$ *set cs*
using *assms*(*6*) **by** *fastforce*
then have (*Inl* (*q1′*,*q2′*)) $\in$ *states* (*canonical-separator M q1 q2*) **and** *x* $\in$ *inputs M*
using *assms*(*4*) **by** *blast*+
then have (*q1′*,*q2′*) $\in$ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *canonical-separator-simps*(*2*)[*OF assms*(*2*,*3*)] **by** *fastforce*
then have (*q1′*,*q2′*) $\in$ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *reachable-state-is-state* **by** *fastforce*
then have *q1′* $\in$ *states M* **and** *q2′* $\in$ *states M*
using *assms*(*2*,*3*) **by** *auto*

obtain *y q1″* **where** (*y*,*q1″*) $\in$ *h M* (*q1′*,*x*)
using ‹*completely-specified M*› ‹*q1′* $\in$ *states M*› ‹*x* $\in$ *inputs M*›
unfolding *completely-specified.simps h.simps* **by** *fastforce*

consider (*a*) *y* ∈ *h-out M* (*q2′*,*x*) | (*b*) *y* ∉ *h-out M* (*q2′*,*x*) **by** *blast*
then have ¬ *deadlock-state ?S* (*Inl* (*q1′*,*q2′*))
proof *cases*
case *a*
then obtain *q2″* **where** (*y*,*q2″*) ∈ *h M* (*q2′*,*x*) **by** *auto*
then have ((*q1′*,*q2′*),*x*,*y*,(*q1″*,*q2″*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *assms*(*2*,*3*) ‹(*y*,*q1″*) ∈ *h M* (*q1′*,*x*)›
unfolding *h.simps product-transitions-def* **by** *fastforce*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ *transitions ?C*
using *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)]
using ‹(*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))› **by** *fast*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inl* (*q1″*,*q2″*)) ∈ *transitions ?S*
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)]

‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*› **by** *fastforce*
then show *?thesis*
unfolding *deadlock-state.simps* **by** *fastforce*
next
case *b*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) ∈ *distinguishing-transitions-left M q1 q2*
using ‹(*y*,*q1″*) ∈ *h M* (*q1′*,*x*)› ‹(*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
unfolding *h-simps h-out.simps distinguishing-transitions-left-def*
by *blast*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) ∈ *transitions ?C*
unfolding *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)] **by** *blast*
then have (*Inl* (*q1′*,*q2′*),*x*,*y*,*Inr q1*) ∈ *transitions ?S*
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)]

‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*› **by** *fastforce*
then show *?thesis*
unfolding *deadlock-state.simps* **by** *fastforce*
qed
then show (*isl qq* ∧ ¬ *deadlock-state ?S qq*)
unfolding ‹*qq* = *Inl* (*q1′*,*q2′*)› **by** *simp*
qed
then have *non-deadlock-prop*: (∀ *q* ∈ *reachable-states ?S* . (*q* ≠ *Inr q1* ∧ *q* ≠ *Inr q2*) ⟶ (*isl q* ∧ ¬ *deadlock-state ?S q*))
using *reachable-state-is-state* **by** *force*

define *ndlps* **where** *ndlps-def*: *ndlps* = {*p* . *path ?S* (*initial ?S*) *p* ∧ *isl* (*target* (*initial ?S*) *p*)}

obtain *qdl* **where** *qdl* ∈ *reachable-states ?S* **and** *deadlock-state ?S qdl*

using *acyclic-deadlock-reachable*[*OF* ‹*acyclic ?S*›] **by** *blast*

have *qdl* = *Inr q1* ∨ *qdl* = *Inr q2*
using *non-deadlock-prop'*[*OF reachable-state-is-state*[*OF* ‹*qdl* ∈ *reachable-states ?S*›]] ‹*deadlock-state ?S qdl*› **by** *fastforce*
then have *Inr q1* ∈ *reachable-states ?S* ∨ *Inr q2* ∈ *reachable-states ?S*
using ‹*qdl* ∈ *reachable-states ?S*› **by** *blast*

have *isl* (*target* (*initial ?S*) []) **using** *state-separator-from-input-choices-simps*(*1*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *auto*
then have [] ∈ *ndlps*
unfolding *ndlps-def* **by** *auto*
then have *ndlps* ≠ {}
by *blast*
moreover have *finite ndlps*
using *acyclic-finite-paths-from-reachable-state*[*OF* ‹*acyclic ?S*›, *of* []] **unfolding** *ndlps-def* **by** *fastforce*
ultimately have ∃ *p* ∈ *ndlps* . ∀ *p'* ∈ *ndlps* . *length p'* ≤ *length p*
by (*meson max-length-elem not-le-imp-less*)
then obtain *mndlp* **where** *path ?S* (*initial ?S*) *mndlp*
and *isl* (*target* (*initial ?S*) *mndlp*)
and ⋀ *p* . *path ?S* (*initial ?S*) *p* ⟹ *isl* (*target* (*initial ?S*) *p*) ⟹ *length p* ≤ *length mndlp*
unfolding *ndlps-def* **by** *blast*
then have (*target* (*initial ?S*) *mndlp*) ∈ *reachable-states ?S*
unfolding *reachable-states-def* **by** *auto*
then have (*target* (*initial ?S*) *mndlp*) ∈ *states ?S*
using *reachable-state-is-state* **by** *auto*
then have (*target* (*initial ?S*) *mndlp*) ∈ (*set* (*map fst cs*))
using ‹*isl* (*target* (*initial ?S*) *mndlp*)› *state-separator-from-input-choices-simps*(*2*)[*OF assms*(*2*,*3*,*4*,*5*,*6*)] **by** *force*
then obtain *q1' q2' x* **where** (*Inl* (*q1'*,*q2'*),*x*) ∈ *set cs*
and *target* (*initial ?S*) *mndlp* = *Inl* (*q1'*,*q2'*)
using *assms*(*6*) **by** *fastforce*
then obtain *i* **where** *i* < *length cs* **and** (*cs* ! *i*) = (*Inl* (*q1'*,*q2'*),*x*)
by (*metis in-set-conv-nth*)

have *Inl* (*q1'*, *q2'*) ∈ *FSM.states* (*canonical-separator M q1 q2*) **and** *x* ∈ *FSM.inputs M*
using *assms*(*4*)[*OF* ‹(*Inl* (*q1'*,*q2'*),*x*) ∈ *set cs*›] **by** *blast+*
then have (*q1'*,*q2'*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
using *canonical-separator-simps*(*2*)[*OF assms*(*2*,*3*)] **by** *blast*

have *q1'* ∈ *states M* **and** *q2'* ∈ *states M*
using *canonical-separator-states*[*OF* ‹*Inl* (*q1'*, *q2'*) ∈ *FSM.states* (*canonical-separator M q1 q2*)› *assms*(*2*,*3*)]
unfolding *product-simps* **using** *assms*(*2*,*3*) **by** *simp+*

have ¬(∃ *t′*∈*FSM.transitions* (*canonical-separator M q1 q2*). *t-source t′* = *target* (*initial ?S*) *mndlp* ∧ *t-input t′* = *x* ∧ *isl* (*t-target t′*))
proof
assume ∃ *t′*∈*FSM.transitions* (*canonical-separator M q1 q2*). *t-source t′* = *target* (*initial ?S*) *mndlp* ∧ *t-input t′* = *x* ∧ *isl* (*t-target t′*)
then obtain *t′* **where** *t′*∈*FSM.transitions* (*canonical-separator M q1 q2*)
and *t-source t′* = *target* (*initial ?S*) *mndlp*
and *t-input t′* = *x*
and *isl* (*t-target t′*)
by *blast*
then have ∃ *q1′ q2′ x* . (*Inl* (*q1′*, *q2′*), *x*)∈*set cs* ∧ *t-source t′* = *Inl* (*q1′*, *q2′*) ∧ *t-input t′* = *x*
using ‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*› **unfolding** ‹*target* (*initial ?S*) *mndlp* = *Inl* (*q1′*,*q2′*)› **by** *fast*
then have *t′* ∈ *transitions ?S*
using ‹*t′*∈*FSM.transitions* (*canonical-separator M q1 q2*)› ‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*›
using *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)] **by** *blast*

then have *path ?S* (*initial ?S*) (*mndlp* @ [*t′*])
using ‹*path ?S* (*initial ?S*) *mndlp*› ‹*t-source t′* = *target* (*initial ?S*) *mndlp*› **by** (*metis path-append-transition*)
moreover have *isl* (*target* (*initial ?S*) (*mndlp* @[*t′*]))
using ‹*isl* (*t-target t′*)› **by** *auto*
ultimately show *False*
using ‹⋀ *p* . *path ?S* (*initial ?S*) *p* ⟹ *isl* (*target* (*initial ?S*) *p*) ⟹ *length p* ≤ *length mndlp*›[*of mndlp*@[*t′*]] **by** *auto*
qed

then obtain *y1 y2* **where** (*Inl* (*q1′*,*q2′*),*x*,*y1*,*Inr q1*) ∈ *transitions* (*canonical-separator M q1 q2*)
and (*Inl* (*q1′*,*q2′*),*x*,*y2*,*Inr q2*) ∈ *transitions* (*canonical-separator M q1 q2*)
using *canonical-separator-isl-deadlock*[*OF* ‹*Inl* (*q1′*, *q2′*) ∈ *FSM.states* (*canonical-separator M q1 q2*)› ‹*x* ∈ *FSM.inputs M*› ‹*completely-specified M*› - *assms*(*2*,*3*)]
unfolding ‹*target* (*initial ?S*) *mndlp* = *Inl* (*q1′*,*q2′*)› **by** *blast*

have (*Inl* (*q1′*,*q2′*), *x*, *y1*, *Inr q1*) ∈ *transitions ?S*
using ‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*› *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)] ‹(*Inl* (*q1′*,*q2′*),*x*,*y1*,*Inr q1*) ∈ *transitions* (*canonical-separator M q1 q2*)› **by** *force*

have (*Inl* (*q1′*,*q2′*), *x*, *y2*, *Inr q2*) ∈ *transitions ?S*
using ‹(*Inl* (*q1′*,*q2′*),*x*) ∈ *set cs*› *state-separator-from-input-choices-transitions-alt-def*[*OF assms*(*2*,*3*,*4*,*5*,*6*,*7*)] ‹(*Inl* (*q1′*,*q2′*),*x*,*y2*,*Inr q2*) ∈ *transitions* (*canonical-separator M q1 q2*)› **by** *force*

have *path ?S* (*initial ?S*) (*mndlp*@[(*Inl* (*q1'*,*q2'*), *x*, *y1*, *Inr q1*)])
using ‹*target* (*initial ?S*) *mndlp* = *Inl* (*q1'*,*q2'*)›
using *path-append-transition*[*OF* ‹*path ?S* (*initial ?S*) *mndlp*› ‹(*Inl* (*q1'*,*q2'*), *x*, *y1*, *Inr q1*) ∈ *transitions ?S*›] **by** *force*
moreover have *target* (*initial ?S*) (*mndlp*@[(*Inl* (*q1'*,*q2'*), *x*, *y1*, *Inr q1*)]) = *Inr q1*
by *auto*
ultimately have *reachable-prop-1*: *Inr q1* ∈ *reachable-states ?S*
using *reachable-states-intro* **by** *metis*

have *path ?S* (*initial ?S*) (*mndlp*@[(*Inl* (*q1'*,*q2'*), *x*, *y2*, *Inr q2*)])
using ‹*target* (*initial ?S*) *mndlp* = *Inl* (*q1'*,*q2'*)›
using *path-append-transition*[*OF* ‹*path ?S* (*initial ?S*) *mndlp*› ‹(*Inl* (*q1'*,*q2'*), *x*, *y2*, *Inr q2*) ∈ *transitions ?S*›] **by** *force*
moreover have *target* (*initial ?S*) (*mndlp*@[(*Inl* (*q1'*,*q2'*), *x*, *y2*, *Inr q2*)]) = *Inr q2*
by *auto*
ultimately have *reachable-prop-2*: *Inr q2* ∈ *reachable-states ?S*
using *reachable-states-intro* **by** *metis*

have *retainment-prop* : ⋀ *q x t'* . *q* ∈ *reachable-states ?S*
⟹ *x* ∈ *FSM.inputs ?C*
⟹ (∃ *t*∈*FSM.transitions ?S*. *t-source t* = *q* ∧ *t-input t* = *x*)
⟹ *t'* ∈ *FSM.transitions ?C*
⟹ *t-source t'* = *q*
⟹ *t-input t'* = *x*
⟹ *t'* ∈ *FSM.transitions ?S*
proof −
fix *q x t'* **assume** *q* ∈ *reachable-states ?S*
and *x* ∈ *FSM.inputs ?C*
and (∃ *t*∈*FSM.transitions ?S*. *t-source t* = *q* ∧ *t-input t* = *x*)
and *t'* ∈ *FSM.transitions ?C*
and *t-source t'* = *q*
and *t-input t'* = *x*

obtain *t* **where** *t* ∈ *FSM.transitions ?S* **and** *t-source t* = *q* **and** *t-input t* = *x*
using ‹(∃ *t*∈*FSM.transitions ?S*. *t-source t* = *q* ∧ *t-input t* = *x*)› **by** *blast*
then have *t-source t* = *t-source t'* ∧ *t-input t* = *t-input t'*
using ‹*t-source t'* = *q*› ‹*t-input t'* = *x*› **by** *auto*

show *t'* ∈ *FSM.transitions ?S*
using *i4* **unfolding** *retains-outputs-for-states-and-inputs-def*
using ‹*t* ∈ *FSM.transitions ?S*› ‹*t'* ∈ *FSM.transitions ?C*› ‹*t-source t* = *t-source t'* ∧ *t-input t* = *t-input t'*›
by *blast*
qed

show *?thesis* **unfolding** *is-state-separator-from-canonical-separator-def*
using *submachine-prop*
single-input-prop
acyclic-prop
deadlock-prop-1
deadlock-prop-2
reachable-prop-1
reachable-prop-2
non-deadlock-prop
retainment-prop **by** *blast*
qed

34.2.2 Calculating a State Separator by Backwards Reachability Analysis

A state separator for states *q1* and *q2* can be calculated using backwards reachability analysis starting from the two deadlock states of their canonical separator until *Inl* (*q1*.*q2*) is reached or it is not possible to reach (*q1*,*q2*).

definition *s-states* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ ((('*a* × '*a*) + '*a*) × '*b*) *list* **where**
s-states M q1 q2 = (*let C* = *canonical-separator M q1 q2*
in select-inputs (*h C*) (*initial C*) (*inputs-as-list C*) (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list C*)))) {*Inr q1*, *Inr q2*} [])

definition *state-separator-from-s-states* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ (('*a* × '*a*) + '*a*, '*b*, '*c*) *fsm option*
where
state-separator-from-s-states M q1 q2 =
(*let cs* = *s-states M q1 q2*
in (*case length cs of*
0 ⇒ *None* |
- ⇒ *if fst* (*last cs*) = *Inl* (*q1*,*q2*)
then Some (*state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 cs*)
else None))

lemma *state-separator-from-s-states-code*[*code*] :
state-separator-from-s-states M q1 q2 =
(*let C* = *canonical-separator M q1 q2*;
cs = *select-inputs* (*h C*) (*initial C*) (*inputs-as-list C*) (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list C*)))) {*Inr q1*, *Inr q2*} []
in (*case length cs of*
0 ⇒ *None* |
- ⇒ *if fst* (*last cs*) = *Inl* (*q1*,*q2*)
then Some (*state-separator-from-input-choices M C q1 q2 cs*)

else None))
unfolding *s-states-def state-separator-from-s-states-def Let-def* **by** *simp*

lemma *s-states-properties* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *distinct* (*map fst* (*s-states M q1 q2*))
and ⋀ *qq x* . (*qq*,*x*) ∈ *set* (*s-states M q1 q2*) ⟹ *qq* ∈ *states* (*canonical-separator M q1 q2*) ∧ *x* ∈ *inputs M*
and ⋀ *qq* . *qq* ∈ *set* (*map fst* (*s-states M q1 q2*)) ⟹ ∃ *q1′ q2′* . *qq* = *Inl* (*q1′*,*q2′*)
and ⋀ *i t* . *i* < *length* (*s-states M q1 q2*)
⟹ *t* ∈ *transitions* (*canonical-separator M q1 q2*)
⟹ *t-source t* = (*fst* ((*s-states M q1 q2*) ! *i*))
⟹ *t-input t* = *snd* ((*s-states M q1 q2*) ! *i*)
⟹ *t-target t* ∈ ((*set* (*map fst* (*take i* (*s-states M q1 q2*)))) ∪ {*Inr q1*, *Inr q2*})
proof −
let *?C* = *canonical-separator M q1 q2*
let *?nS* = {*Inr q1*, *Inr q2*}
let *?nL* = (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list ?C*))))
let *?iL* = (*inputs-as-list ?C*)
let *?q0* = (*initial ?C*)
let *?f* = (*h ?C*)
let *?k* = (*size* (*canonical-separator M q1 q2*))

let *?cs* = (*s-states M q1 q2*)

have *pp1*: *distinct* (*map fst* []) **by** *auto*
have *pp2*: *set* (*map fst* []) ⊆ *?nS* **by** *auto*
have *pp3*: *?nS* = *?nS* ∪ *set* (*map fst* []) **by** *auto*
have *pp4*: *?q0* ∉ *?nS* **unfolding** *canonical-separator-simps*[*OF assms*] **by** *auto*
have *pp5* :*distinct ?nL* **using** *states-as-list-distinct* **by** *simp*
have *pp6*: *?q0* ∉ *set ?nL* **unfolding** *canonical-separator-simps*[*OF assms*] **by** *auto*
have *pp7*: *set ?nL* ∩ *?nS* = {} **by** *auto*

have ⋀ *i* . *length* [] ≤ *i* **by** *auto*

have *ip1*: ⋀ *i* . *i* < *length ?cs* ⟹ *fst* (*?cs* ! *i*) ∈ (*insert ?q0* (*set ?nL*))
and *ip2*: ⋀ *i* . *i* < *length ?cs* ⟹ *fst* (*?cs* ! *i*) ∉ *?nS0*
and *ip3*: ⋀ *i* . *i* < *length ?cs* ⟹ *snd* (*?cs* ! *i*) ∈ *set ?iL*
and *ip4*: ⋀ *i* . *i* < *length ?cs* ⟹ (∀ *qx′* ∈ *set* (*take i ?cs*) . *fst* (*?cs* ! *i*) ≠ *fst qx′*)
using *select-inputs-index-properties*[*OF* - ‹⋀ *i* . *length* [] ≤ *i*› *pp1 pp3 pp4 pp5*

pp6 pp7]
unfolding *s-states-def Let-def* **by** *blast+*
have *ip5*: $\bigwedge$ *i* . *i* < *length ?cs* $\Longrightarrow$ ($\exists$ *t* $\in$ *transitions ?C* . *t-source t* = *fst* (*?cs* ! *i*) $\wedge$ *t-input t* = *snd* (*?cs* ! *i*))
using *select-inputs-index-properties*(*5*)[*OF* - ‹$\bigwedge$ *i* . *length* [] $\leq$ *i*› *pp1 pp3 pp4 pp5 pp6 pp7*]
unfolding *s-states-def Let-def* **by** *blast*
have *ip6*: $\bigwedge$ *i t* . *i* < *length ?cs* $\Longrightarrow$ *t* $\in$ *transitions ?C* $\Longrightarrow$ *t-source t* = *fst* (*?cs* ! *i*) $\Longrightarrow$ *t-input t* = *snd* (*?cs* ! *i*) $\Longrightarrow$ (*t-target t* $\in$ *?nS0* $\vee$ ($\exists$ *qx'* $\in$ *set* (*take i ?cs*) . *fst qx'* = (*t-target t*)))
using *select-inputs-index-properties*(*6*)[*OF* - ‹$\bigwedge$ *i* . *length* [] $\leq$ *i*› *pp1 pp3 pp4 pp5 pp6 pp7*]
unfolding *s-states-def Let-def* **by** *blast*

show *distinct* (*map fst ?cs*)
using *select-inputs-distinct*[*OF pp1 pp2 pp4 pp5 pp6 pp7*]
unfolding *s-states-def Let-def* **by** *blast*

show $\bigwedge$ *qq x* . (*qq*,*x*) $\in$ *set ?cs* $\Longrightarrow$ *qq* $\in$ *states* (*canonical-separator M q1 q2*) $\wedge$ *x* $\in$ *inputs M*
proof −
fix *qq x* **assume** (*qq*,*x*) $\in$ *set ?cs*
then obtain *i* **where** *i* < *length ?cs* **and** *?cs* ! *i* = (*qq*,*x*)
by (*meson in-set-conv-nth*)
show *qq* $\in$ *states* (*canonical-separator M q1 q2*) $\wedge$ *x* $\in$ *inputs M*
using *ip1*[*OF* ‹*i* < *length ?cs*›] *ip3*[*OF* ‹*i* < *length ?cs*›]
states-as-list-set[*of ?C*] *inputs-as-list-set*[*of ?C*]
unfolding ‹*?cs* ! *i* = (*qq*,*x*)› *fst-conv snd-conv canonical-separator-simps*(*3*)[*OF assms*]
by *auto*
qed

show $\bigwedge$ *qq* . *qq* $\in$ *set* (*map fst ?cs*) $\Longrightarrow$ $\exists$ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
proof −
fix *qq* **assume** *qq* $\in$ *set* (*map fst ?cs*)
then obtain *i* **where** *i* < *length ?cs* **and** *fst* (*?cs* ! *i*) = *qq*
by (*metis* (*no-types*, *lifting*) *in-set-conv-nth length-map nth-map*)
show $\exists$ *q1' q2'* . *qq* = *Inl* (*q1'*,*q2'*)
using *ip1*[*OF* ‹*i* < *length ?cs*›] *states-as-list-set*[*of ?C*]
unfolding ‹*fst* (*?cs* ! *i*) = *qq*› *canonical-separator-simps*[*OF assms*]
by *auto*
qed

show $\bigwedge$ *i t* . *i* < *length ?cs*
$\Longrightarrow$ *t* $\in$ *transitions* (*canonical-separator M q1 q2*)
$\Longrightarrow$ *t-source t* = (*fst* (*?cs* ! *i*))
$\Longrightarrow$ *t-input t* = *snd* (*?cs* ! *i*)
$\Longrightarrow$ *t-target t* $\in$ ((*set* (*map fst* (*take i ?cs*))) $\cup$ {*Inr q1*, *Inr q2*})

proof −
fix *i t* **assume** *i* < *length ?cs*
and *t* ∈ *transitions ?C*
and *t-source t* = (*fst* (*?cs* ! *i*))
and *t-input t* = *snd* (*?cs* ! *i*)

show *t-target t* ∈ ((*set* (*map fst* (*take i ?cs*))) ∪ {*Inr q1*, *Inr q2*})
using *ip6*[*OF* ‹*i* < *length ?cs*› ‹*t* ∈ *transitions ?C*› ‹*t-source t* = (*fst* (*?cs* ! *i*))› ‹*t-input t* = *snd* (*?cs* ! *i*)›]
by (*metis Un-iff in-set-conv-nth length-map nth-map*)
qed
qed

lemma *state-separator-from-s-states-soundness* :
assumes *state-separator-from-s-states M q1 q2* = *Some A*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *completely-specified M*
shows *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
proof −
let *?cs* = *s-states M q1 q2*

have *length* (*s-states M q1 q2*) ≠ *0* ∧ *fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*)
and *A* = *state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 ?cs*
using *assms*(*1*) **unfolding** *state-separator-from-s-states-def Let-def*
by (*cases length* (*s-states M q1 q2*); *cases fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*); *auto*)+
then have *Inl* (*q1*,*q2*) ∈ *set* (*map fst ?cs*)
by (*metis last-in-set length-0-conv map-set*)

show *?thesis*
using *state-separator-from-input-choices-is-state-separator*[
OF - assms(*2*,*3*) - ‹*Inl* (*q1*,*q2*) ∈ *set* (*map fst ?cs*)›,
OF s-states-properties[*OF assms*(*2*,*3*)] *assms*(*4*)]
unfolding ‹*A* = *state-separator-from-input-choices M* (*canonical-separator M q1 q2*) *q1 q2 ?cs*›[*symmetric*] **by** *blast*
qed

lemma *state-separator-from-s-states-exhaustiveness* :
assumes ∃ *S* . *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
and *q1* ∈ *states M* **and** *q2* ∈ *states M* **and** *completely-specified M* **and** *observable M*
shows *state-separator-from-s-states M q1 q2* ≠ *None*
proof −
let *?CSep* = (*canonical-separator M q1 q2*)

obtain *S* **where** *S-def*: *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
using *assms*(*1*) **by** *blast*

then have *is-submachine S ?CSep*
and *single-input S*
and *acyclic S*
and $*$:$\bigwedge$ *q* . *q* $\in$ *reachable-states S* $\Longrightarrow$ *q* $\neq$ *Inr q1* $\Longrightarrow$ *q* $\neq$ *Inr q2* $\Longrightarrow$ (*isl q* $\wedge$ $\neg$ *deadlock-state S q*)
and $**$:$\bigwedge$ *q x t* . *q* $\in$ *reachable-states S* $\Longrightarrow$ *x* $\in$ (*inputs ?CSep*) $\Longrightarrow$ ($\exists$ *t* $\in$ *transitions S* . *t-source t* = *q* $\wedge$ *t-input t* = *x*) $\Longrightarrow$ *t* $\in$ *transitions ?CSep* $\Longrightarrow$ *t-source t* = *q* $\Longrightarrow$ *t-input t* = *x* $\Longrightarrow$ *t* $\in$ *transitions S*
using *assms* **unfolding** *is-state-separator-from-canonical-separator-def* **by** *blast+*

have *p1*: ($\bigwedge$*q x*. *q* $\in$ *reachable-states S* $\Longrightarrow$ *h S* (*q*, *x*) $\neq$ {} $\Longrightarrow$ *h S* (*q*, *x*) = *h ?CSep* (*q*, *x*))
proof −
fix *q x* **assume** *q* $\in$ *reachable-states S* **and** *h S* (*q*, *x*) $\neq$ {}

then have *x* $\in$ *inputs ?CSep*
using ‹*is-submachine S ?CSep*› *fsm-transition-input* **by** *force*
have ($\exists$ *t* $\in$ *transitions S* . *t-source t* = *q* $\wedge$ *t-input t* = *x*)
using ‹*h S* (*q*, *x*) $\neq$ {}› **by** *fastforce*

have $\bigwedge$ *y q''* . (*y*,*q''*) $\in$ *h S* (*q*,*x*) $\Longrightarrow$ (*y*,*q''*) $\in$ *h ?CSep* (*q*,*x*)
using ‹*is-submachine S ?CSep*› **by** *force*
moreover have $\bigwedge$ *y q''* . (*y*,*q''*) $\in$ *h ?CSep* (*q*,*x*) $\Longrightarrow$ (*y*,*q''*) $\in$ *h S* (*q*,*x*)
using $**$[*OF* ‹*q* $\in$ *reachable-states S*› ‹*x* $\in$ *inputs ?CSep*› ‹($\exists$ *t* $\in$ *transitions S* . *t-source t* = *q* $\wedge$ *t-input t* = *x*)›]
unfolding *h.simps* **by** *force*
ultimately show *h S* (*q*, *x*) = *h ?CSep* (*q*, *x*)
by *force*
qed

have *p2*: $\bigwedge$*q'*. *q'* $\in$ *reachable-states S* $\Longrightarrow$ *deadlock-state S q'* $\Longrightarrow$ *q'* $\in$ {*Inr q1*, *Inr q2*} $\cup$ *set* (*map fst* [])
using $*$ **by** *fast*

have *initial S* = *Inl* (*q1*,*q2*)
using *is-state-separator-from-canonical-separator-initial*[*OF S-def assms*(*2*,*3*)] **by** *assumption*

have $***$: (*set* (*remove1* (*Inl* (*q1*, *q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list ?CSep*)))) $\cup$ {*Inr q1*, *Inr q2*} $\cup$ *set* (*map fst* []) = (*states ?CSep* − {*Inl* (*q1*,*q2*)})
using *states-as-list-set*[*of ?CSep*] *states-as-list-distinct*[*of ?CSep*]
unfolding
‹*initial S* = *Inl* (*q1*,*q2*)›

canonical-separator-simps(*2*)[*OF assms*(*2*,*3*)]
by *auto*

have *Inl* (*q1*,*q2*) ∈ *reachable-states ?CSep*
using *reachable-states-initial*[*of S*] **unfolding** ‹*initial S* = *Inl* (*q1*,*q2*)›
using *submachine-reachable-subset*[*OF* ‹*is-submachine S ?CSep*›] **by** *blast*
then have *p3*: *states ?CSep* = *insert* (*FSM.initial S*) (*set* (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list ?CSep*)))) ∪ {*Inr q1*, *Inr q2*} ∪ *set* (*map fst* []))
unfolding ∗∗∗ ‹*initial S* = *Inl* (*q1*,*q2*)›
using *reachable-state-is-state* **by** *fastforce*

have *p4*: *initial S* ∉ (*set* (*remove1* (*Inl* (*q1*,*q2*)) (*remove1* (*Inr q1*) (*remove1* (*Inr q2*) (*states-as-list ?CSep*)))) ∪ {*Inr q1*, *Inr q2*} ∪ *set* (*map fst* []))
using ‹*FSM.initial S* = *Inl* (*q1*, *q2*)› **by** *auto*

have *fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*) **and** *length* (*s-states M q1 q2*) > *0*
using *select-inputs-from-submachine*[*OF* ‹*single-input S*› ‹*acyclic S*› ‹*is-submachine S ?CSep*› *p1 p2 p3 p4*]
unfolding *s-states-def submachine-simps*[*OF* ‹*is-submachine S ?CSep*›] *Let-def canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)]
by *auto*

obtain *k* **where***length* (*s-states M q1 q2*) = *Suc k*
using ‹*length* (*s-states M q1 q2*) > *0*› *gr0-conv-Suc* **by** *blast*
have (*fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*)) = *True*
using ‹*fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*)› **by** *simp*

show *?thesis*
unfolding *state-separator-from-s-states-def Let-def* ‹*length* (*s-states M q1 q2*) = *Suc k*› ‹*fst* (*last* (*s-states M q1 q2*)) = *Inl* (*q1*,*q2*)›
by *auto*
qed

34.3 Generalizing State Separators

State separators can be defined without reverence to the canonical separator:

definition *is-separator* :: ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ '*a* ⇒ ('*d*,'*b*,'*c*) *fsm* ⇒ '*d* ⇒ '*d* ⇒ *bool* **where**
is-separator M q1 q2 A t1 t2 =
(*single-input A*
∧ *acyclic A*
∧ *observable A*
∧ *deadlock-state A t1*
∧ *deadlock-state A t2*
∧ *t1* ∈ *reachable-states A*
∧ *t2* ∈ *reachable-states A*
∧ (∀ *t* ∈ *reachable-states A* . (*t* ≠ *t1* ∧ *t* ≠ *t2*) ⟶ ¬ *deadlock-state A t*)

$\wedge$ ($\forall$ *io* $\in$ *L A* . ($\forall$ *x yq yt* . (*io*@[(*x*,*yq*)] $\in$ *LS M q1* $\wedge$ *io*@[(*x*,*yt*)] $\in$ *L A*) $\longrightarrow$ (*io*@[(*x*,*yq*)] $\in$ *L A*))
$\wedge$ ($\forall$ *x yq2 yt* . (*io*@[(*x*,*yq2*)] $\in$ *LS M q2* $\wedge$ *io*@[(*x*,*yt*)] $\in$ *L A*) $\longrightarrow$ (*io*@[(*x*,*yq2*)] $\in$ *L A*)))
$\wedge$ ($\forall$ *p* . (*path A* (*initial A*) *p* $\wedge$ *target* (*initial A*) *p* = *t1*) $\longrightarrow$ *p-io p* $\in$ *LS M q1* − *LS M q2*)
$\wedge$ ($\forall$ *p* . (*path A* (*initial A*) *p* $\wedge$ *target* (*initial A*) *p* = *t2*) $\longrightarrow$ *p-io p* $\in$ *LS M q2* − *LS M q1*)
$\wedge$ ($\forall$ *p* . (*path A* (*initial A*) *p* $\wedge$ *target* (*initial A*) *p* $\neq$ *t1* $\wedge$ *target* (*initial A*) *p* $\neq$ *t2*) $\longrightarrow$ *p-io p* $\in$ *LS M q1* $\cap$ *LS M q2*)
$\wedge$ *q1* $\neq$ *q2*
$\wedge$ *t1* $\neq$ *t2*
$\wedge$ (*inputs A*) $\subseteq$ (*inputs M*))

lemma *is-separator-simps* :
assumes *is-separator M q1 q2 A t1 t2*
shows *single-input A*
and *acyclic A*
and *observable A*
and *deadlock-state A t1*
and *deadlock-state A t2*
and *t1* $\in$ *reachable-states A*
and *t2* $\in$ *reachable-states A*
and $\bigwedge$ *t* . *t* $\in$ *reachable-states A* $\Longrightarrow$ *t* $\neq$ *t1* $\Longrightarrow$ *t* $\neq$ *t2* $\Longrightarrow$ $\neg$ *deadlock-state A t*
and $\bigwedge$ *io x yq yt* . *io*@[(*x*,*yq*)] $\in$ *LS M q1* $\Longrightarrow$ *io*@[(*x*,*yt*)] $\in$ *L A* $\Longrightarrow$ (*io*@[(*x*,*yq*)] $\in$ *L A*)
and $\bigwedge$ *io x yq yt* . *io*@[(*x*,*yq*)] $\in$ *LS M q2* $\Longrightarrow$ *io*@[(*x*,*yt*)] $\in$ *L A* $\Longrightarrow$ (*io*@[(*x*,*yq*)] $\in$ *L A*)
and $\bigwedge$ *p* . *path A* (*initial A*) *p* $\Longrightarrow$ *target* (*initial A*) *p* = *t1* $\Longrightarrow$ *p-io p* $\in$ *LS M q1* − *LS M q2*
and $\bigwedge$ *p* . *path A* (*initial A*) *p* $\Longrightarrow$ *target* (*initial A*) *p* = *t2* $\Longrightarrow$ *p-io p* $\in$ *LS M q2* − *LS M q1*
and $\bigwedge$ *p* . *path A* (*initial A*) *p* $\Longrightarrow$ *target* (*initial A*) *p* $\neq$ *t1* $\Longrightarrow$ *target* (*initial A*) *p* $\neq$ *t2* $\Longrightarrow$ *p-io p* $\in$ *LS M q1* $\cap$ *LS M q2*
and *q1* $\neq$ *q2*
and *t1* $\neq$ *t2*
and (*inputs A*) $\subseteq$ (*inputs M*)
proof −
have *p01*: *single-input A*
and *p02*: *acyclic A*
and *p03*: *observable A*
and *p04*: *deadlock-state A t1*
and *p05*: *deadlock-state A t2*
and *p06*: *t1* $\in$ *reachable-states A*
and *p07*: *t2* $\in$ *reachable-states A*
and *p08*: ($\forall$ *t* $\in$ *reachable-states A* . (*t* $\neq$ *t1* $\wedge$ *t* $\neq$ *t2*) $\longrightarrow$ $\neg$ *deadlock-state A t*)
and *p09*: ($\forall$ *io* $\in$ *L A* . ($\forall$ *x yq yt* . (*io*@[(*x*,*yq*)] $\in$ *LS M q1* $\wedge$ *io*@[(*x*,*yt*)] $\in$ *L*

$A) \longrightarrow (io@[(x,yq)] \in L\ A))$
$\wedge\ (\forall\ x\ yq2\ yt\ .\ (io@[(x,yq2)] \in LS\ M\ q2 \wedge io@[(x,yt)] \in L\ A) \longrightarrow (io@[(x,yq2)] \in L\ A)))$
and *p10*: $(\forall\ p\ .\ (path\ A\ (initial\ A)\ p \wedge target\ (initial\ A)\ p = t1) \longrightarrow p\text{-}io\ p \in LS\ M\ q1 - LS\ M\ q2)$
and *p11*: $(\forall\ p\ .\ (path\ A\ (initial\ A)\ p \wedge target\ (initial\ A)\ p = t2) \longrightarrow p\text{-}io\ p \in LS\ M\ q2 - LS\ M\ q1)$
and *p12*: $(\forall\ p\ .\ (path\ A\ (initial\ A)\ p \wedge target\ (initial\ A)\ p \neq t1 \wedge target\ (initial\ A)\ p \neq t2) \longrightarrow p\text{-}io\ p \in LS\ M\ q1 \cap LS\ M\ q2)$
and *p13*: $q1 \neq q2$
and *p14*: $t1 \neq t2$
and *p15*: $(inputs\ A) \subseteq (inputs\ M)$
using *assms* **unfolding** *is-separator-def* **by** *presburger+*

show *single-input A* **using** *p01* **by** *assumption*
show *acyclic A* **using** *p02* **by** *assumption*
show *observable A* **using** *p03* **by** *assumption*
show *deadlock-state A t1* **using** *p04* **by** *assumption*
show *deadlock-state A t2* **using** *p05* **by** *assumption*
show $t1 \in reachable\text{-}states\ A$ **using** *p06* **by** *assumption*
show $t2 \in reachable\text{-}states\ A$ **using** *p07* **by** *assumption*
show $\bigwedge io\ x\ yq\ yt\ .\ io@[(x,yq)] \in LS\ M\ q1 \Longrightarrow io@[(x,yt)] \in L\ A \Longrightarrow (io@[(x,yq)] \in L\ A)$ **using** *p09 language-prefix*[*of - - A initial A*] **by** *blast*
show $\bigwedge io\ x\ yq\ yt\ .\ io@[(x,yq)] \in LS\ M\ q2 \Longrightarrow io@[(x,yt)] \in L\ A \Longrightarrow (io@[(x,yq)] \in L\ A)$ **using** *p09 language-prefix*[*of - - A initial A*] **by** *blast*
show $\bigwedge t\ .\ t \in reachable\text{-}states\ A \Longrightarrow t \neq t1 \Longrightarrow t \neq t2 \Longrightarrow \neg\ deadlock\text{-}state\ A\ t$ **using** *p08* **by** *blast*
show $\bigwedge p\ .\ path\ A\ (initial\ A)\ p \Longrightarrow target\ (initial\ A)\ p = t1 \Longrightarrow p\text{-}io\ p \in LS\ M\ q1 - LS\ M\ q2$ **using** *p10* **by** *blast*
show $\bigwedge p\ .\ path\ A\ (initial\ A)\ p \Longrightarrow target\ (initial\ A)\ p = t2 \Longrightarrow p\text{-}io\ p \in LS\ M\ q2 - LS\ M\ q1$ **using** *p11* **by** *blast*
show $\bigwedge p\ .\ path\ A\ (initial\ A)\ p \Longrightarrow target\ (initial\ A)\ p \neq t1 \Longrightarrow target\ (initial\ A)\ p \neq t2 \Longrightarrow p\text{-}io\ p \in LS\ M\ q1 \cap LS\ M\ q2$ **using** *p12* **by** *blast*
show $q1 \neq q2$ **using** *p13* **by** *assumption*
show $t1 \neq t2$ **using** *p14* **by** *assumption*
show $(inputs\ A) \subseteq (inputs\ M)$ **using** *p15* **by** *assumption*
qed

lemma *separator-initial* :
assumes *is-separator M q1 q2 A t1 t2*
shows $initial\ A \neq t1$
and $initial\ A \neq t2$
proof −
show $initial\ A \neq t1$
proof
assume $initial\ A = t1$
then have $deadlock\text{-}state\ A\ (initial\ A)$
using *is-separator-simps*(*4*)[*OF assms*] **by** *auto*

then have *reachable-states A* = {*initial A*}
using *states-initial-deadlock* **by** *blast*
then show *False*
using *is-separator-simps*(*7*,*15*)[*OF assms*] ‹*initial A* = *t1*› **by** *auto*
qed

show *initial A* ≠ *t2*
proof
assume *initial A* = *t2*
then have *deadlock-state A* (*initial A*)
using *is-separator-simps*(*5*)[*OF assms*] **by** *auto*
then have *reachable-states A* = {*initial A*}
using *states-initial-deadlock* **by** *blast*
then show *False*
using *is-separator-simps*(*6*,*15*)[*OF assms*] ‹*initial A* = *t2*› **by** *auto*
qed
qed

lemma *separator-path-targets* :
assumes *is-separator M q1 q2 A t1 t2*
and *path A* (*initial A*) *p*
shows *p-io p* ∈ *LS M q1* − *LS M q2* ⟹ *target* (*initial A*) *p* = *t1*
and *p-io p* ∈ *LS M q2* − *LS M q1* ⟹ *target* (*initial A*) *p* = *t2*
and *p-io p* ∈ *LS M q1* ∩ *LS M q2* ⟹ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*)
and *p-io p* ∈ *LS M q1* ∪ *LS M q2*
proof −
have *pt1*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*
and *pt2*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*
and *pt3*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* ≠ *t1* ⟹ *target* (*initial A*) *p* ≠ *t2* ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
and *t1* ≠ *t2*
and *observable A*
using *is-separator-simps*[*OF assms*(*1*)] **by** *blast*+

show *p-io p* ∈ *LS M q1* − *LS M q2* ⟹ *target* (*initial A*) *p* = *t1*
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*› **by** *blast*
show *p-io p* ∈ *LS M q2* − *LS M q1* ⟹ *target* (*initial A*) *p* = *t2*
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*› **by** *blast*
show *p-io p* ∈ *LS M q1* ∩ *LS M q2* ⟹ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*)
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*› **by** *blast*
show *p-io p* ∈ *LS M q1* ∪ *LS M q2*

using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*› **by** *blast*
qed

lemma *separator-language* :
assumes *is-separator M q1 q2 A t1 t2*
and *io* ∈ *L A*
shows *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*t1*}
and *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*t2*}
and *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*t1*,*t2*} = {}
and *io* ∈ *LS M q1* ∪ *LS M q2*
proof −

obtain *p* **where** *path A* (*initial A*) *p* **and** *p-io p* = *io*
using ‹*io* ∈ *L A*› **by** *auto*

have *pt1*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*
and *pt2*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *t2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*
and *pt3*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* ≠ *t1* ⟹ *target* (*initial A*) *p* ≠ *t2* ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
and *t1* ≠ *t2*
and *observable A*
using *is-separator-simps*[*OF assms*(*1*)] **by** *blast*+

show *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*t1*}
proof −
assume *io* ∈ *LS M q1* − *LS M q2*

then have *p-io p* ∈ *LS M q1* − *LS M q2*
using ‹*p-io p* = *io*› **by** *auto*
then have *target* (*initial A*) *p* = *t1*
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*›
by *blast*
then have *t1* ∈ *io-targets A io* (*initial A*)
using ‹*path A* (*initial A*) *p*› ‹*p-io p* = *io*› **unfolding** *io-targets.simps* **by** *force*
then show *io-targets A io* (*initial A*) = {*t1*}
using *observable-io-targets*[*OF* ‹*observable A*›]
by (*metis* ‹*io* ∈ *L A*› *singletonD*)
qed

show *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*t2*}
proof −
assume *io* ∈ *LS M q2* − *LS M q1*

then have *p-io p* ∈ *LS M q2* − *LS M q1*
using ‹*p-io p* = *io*› **by** *auto*
then have *target* (*initial A*) *p* = *t2*
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*›
by *blast*
then have *t2* ∈ *io-targets A io* (*initial A*)
using ‹*path A* (*initial A*) *p*› ‹*p-io p* = *io*› **unfolding** *io-targets.simps* **by** *force*
then show *io-targets A io* (*initial A*) = {*t2*}
using *observable-io-targets*[*OF* ‹*observable A*›]
by (*metis* ‹*io* ∈ *L A*› *singletonD*)
qed

show *io* ∈ *LS M q1* ∩ *LS M q2* ⟹ *io-targets A io* (*initial A*) ∩ {*t1*,*t2*} = {}
proof −
assume *io* ∈ *LS M q1* ∩ *LS M q2*

then have *p-io p* ∈ *LS M q1* ∩ *LS M q2*
using ‹*p-io p* = *io*› **by** *auto*
then have *target* (*initial A*) *p* ≠ *t1* **and** *target* (*initial A*) *p* ≠ *t2*
using *pt1*[*OF* ‹*path A* (*initial A*) *p*›] *pt2*[*OF* ‹*path A* (*initial A*) *p*›] *pt3*[*OF* ‹*path A* (*initial A*) *p*›] ‹*t1* ≠ *t2*›
by *blast*+
moreover have *target* (*initial A*) *p* ∈ *io-targets A io* (*initial A*)
using ‹*path A* (*initial A*) *p*› ‹*p-io p* = *io*› **unfolding** *io-targets.simps* **by** *force*
ultimately show *io-targets A io* (*initial A*) ∩ {*t1*,*t2*} = {}
using *observable-io-targets*[*OF* ‹*observable A*› ‹*io* ∈ *L A*›]
by (*metis* (*no-types*, *opaque-lifting*) *inf-bot-left insert-disjoint*(*2*) *insert-iff singletonD*)
qed

show *io* ∈ *LS M q1* ∪ *LS M q2*
using *separator-path-targets*(*4*)[*OF assms*(*1*) ‹*path A* (*initial A*) *p*›] ‹*p-io p* = *io*› **by** *auto*
qed

lemma *is-separator-sym* :
is-separator M q1 q2 A t1 t2 ⟹ *is-separator M q2 q1 A t2 t1*
unfolding *is-separator-def Int-commute*[*of LS M q2 LS M q1*] **by** *meson*

lemma *state-separator-from-canonical-separator-is-separator* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
and *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*

shows *is-separator M q1 q2 A* (*Inr q1*) (*Inr q2*)
proof −
let *?C* = *canonical-separator M q1 q2*
have *observable ?C*
using *canonical-separator-observable*[*OF assms*(*2*,*3*,*4*)] **by** *assumption*

have *is-submachine A ?C*
and *p1*: *single-input A*
and *p2*: *acyclic A*
and *p4*: *deadlock-state A* (*Inr q1*)
and *p5*: *deadlock-state A* (*Inr q2*)
and *p6*: ((*Inr q1*) ∈ *reachable-states A*)
and *p7*: ((*Inr q2*) ∈ *reachable-states A*)
and ⋀ *q* . *q* ∈ *reachable-states A* ⟹ *q* ≠ *Inr q1* ⟹ *q* ≠ *Inr q2* ⟹ (*isl q* ∧ ¬ *deadlock-state A q*)
and *compl*: ⋀ *q x t* . *q* ∈ *reachable-states A* ⟹ *x* ∈ (*inputs M*) ⟹ (∃ *t* ∈ *transitions A* . *t-source t* = *q* ∧ *t-input t* = *x*) ⟹ *t* ∈ *transitions ?C* ⟹ *t-source t* = *q* ⟹ *t-input t* = *x* ⟹ *t* ∈ *transitions A*
using *is-state-separator-from-canonical-separator-simps*[*OF assms*(*1*)]
unfolding *canonical-separator-simps*[*OF assms*(*3*,*4*)]
by *blast*+

have *p3*: *observable A*
using *state-separator-from-canonical-separator-observable*[*OF assms*(*1*−*4*)] **by** *assumption*

have *p8*: (∀ *t*∈*reachable-states A*. *t* ≠ *Inr q1* ∧ *t* ≠ *Inr q2* ⟶ ¬ *deadlock-state A t*)
using ‹⋀ *q* . *q* ∈ *reachable-states A* ⟹ *q* ≠ *Inr q1* ⟹ *q* ≠ *Inr q2* ⟹ (*isl q* ∧ ¬ *deadlock-state A q*)› **by** *simp*

have ⋀ *io* . *io* ∈ *L A* ⟹
(*io* ∈ *LS M q1* − *LS M q2* ⟶ *io-targets A io* (*initial A*) = {*Inr q1*}) ∧
(*io* ∈ *LS M q2* − *LS M q1* ⟶ *io-targets A io* (*initial A*) = {*Inr q2*}) ∧
(*io* ∈ *LS M q1* ∩ *LS M q2* ⟶ *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}) ∧
(∀ *x yq yt*. *io* @ [(*x*, *yq*)] ∈ *LS M q1* ∧ *io* @ [(*x*, *yt*)] ∈ *LS A* (*initial A*) ⟶ *io* @ [(*x*, *yq*)] ∈ *LS A* (*initial A*)) ∧
(∀ *x yq2 yt*. *io* @ [(*x*, *yq2*)] ∈ *LS M q2* ∧ *io* @ [(*x*, *yt*)] ∈ *LS A* (*initial A*) ⟶ *io* @ [(*x*, *yq2*)] ∈ *LS A* (*initial A*))
proof −
fix *io* **assume** *io* ∈ *L A*

have *io* ∈ *LS M q1* − *LS M q2* ⟹ *io-targets A io* (*initial A*) = {*Inr q1*}
using *state-separator-from-canonical-separator-language-target*(*1*)[*OF assms*(*1*) ‹*io* ∈ *L A*› *assms*(*2*,*3*,*4*,*5*)] **by** *assumption*
moreover have *io* ∈ *LS M q2* − *LS M q1* ⟹ *io-targets A io* (*initial A*) = {*Inr q2*}
using *state-separator-from-canonical-separator-language-target*(*2*)[*OF assms*(*1*)

‹io ∈ L A› assms(2,3,4,5)] **by** assumption
moreover have io ∈ LS M q1 ∩ LS M q2 ⟹ io-targets A io (initial A) ∩ {Inr q1, Inr q2} = {}
using state-separator-from-canonical-separator-language-target(3)[OF assms(1) ‹io ∈ L A› assms(2,3,4,5)] **by** assumption
moreover have ⋀ x yq yt. io @ [(x, yq)] ∈ LS M q1 ⟹ io @ [(x, yt)] ∈ L A ⟹ io @ [(x, yq)] ∈ L A
proof −
fix x yq yt **assume** io @ [(x, yq)] ∈ LS M q1 **and** io @ [(x, yt)] ∈ L A

obtain pA tA **where** path A (initial A) (pA@[tA]) **and** p-io (pA@[tA]) = io @ [(x, yt)]
using language-initial-path-append-transition[OF ‹io @ [(x, yt)] ∈ L A›] **by** blast
then have path A (initial A) pA **and** p-io pA = io
by auto
then have path ?C (initial ?C) pA
using submachine-path-initial[OF ‹is-submachine A ?C›] **by** auto

obtain p1 t1 **where** path M q1 (p1@[t1]) **and** p-io (p1@[t1]) = io @ [(x, yq)]
using language-path-append-transition[OF ‹io @ [(x, yq)] ∈ LS M q1›] **by** blast
then have path M q1 p1 **and** p-io p1 = io **and** t1 ∈ transitions M **and** t-input t1 = x **and** t-output t1 = yq **and** t-source t1 = target q1 p1
by auto

let ?q = target (initial A) pA
have ?q ∈ states A
using path-target-is-state ‹path A (initial A) (pA@[tA])› **by** auto
have ?q ∈ reachable-states A
using ‹path A (initial A) pA› reachable-states-intro **by** blast

have tA ∈ transitions A **and** t-input tA = x **and** t-output tA = yt **and** t-source tA = target (initial A) pA
using ‹path A (initial A) (pA@[tA])› ‹p-io (pA@[tA]) = io @ [(x, yt)]› **by** auto
then have x ∈ (inputs M)
using ‹is-submachine A ?C›
unfolding is-submachine.simps canonical-separator-simps[OF assms(3,4)] **by** auto

have ∃ t∈(transitions A). t-source t = target (initial A) pA ∧ t-input t = x
using ‹tA ∈ transitions A› ‹t-input tA = x› ‹t-source tA = target (initial A) pA› **by** blast

have io ∈ LS M q2
using submachine-language[OF ‹is-submachine A ?C›] ‹io @ [(x, yt)] ∈ L A›

using *canonical-separator-language-prefix*[*OF* - *assms*(*3*,*4*,*2*,*5*), *of io* (*x*,*yt*)] **by** *blast*
then obtain *p2* **where** *path M q2 p2* **and** *p-io p2* = *io*
by *auto*

show *io* @ [(*x*, *yq*)] ∈ *L A*
proof (*cases* ∃ *t2* ∈ *transitions M* . *t-source t2* = *target q2 p2* ∧ *t-input t2* = *x* ∧ *t-output t2* = *yq*)
case *True*
then obtain *t2* **where** *t2* ∈ *transitions M* **and** *t-source t2* = *target q2 p2* **and** *t-input t2* = *x* **and** *t-output t2* = *yq*
by *blast*
then have *path M q2* (*p2*@[*t2*]) **and** *p-io* (*p2*@[*t2*]) = *io*@[(*x*,*yq*)]
using *path-append-transition*[*OF* ‹*path M q2 p2*›] ‹*p-io p2* = *io*› **by** *auto*
then have *io* @ [(*x*, *yq*)] ∈ *LS M q2*
unfolding *LS.simps* **by** (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io*@[(*x*,*yq*)] ∈ *L ?C*
using *canonical-separator-language-intersection*[*OF* ‹*io* @ [(*x*, *yq*)] ∈ *LS M q1*› - *assms*(*3*,*4*)] **by** *blast*

obtain *pA′ tA′* **where** *path ?C* (*initial ?C*) (*pA′*@[*tA′*]) **and** *p-io* (*pA′*@[*tA′*]) = *io*@[(*x*,*yq*)]
using *language-initial-path-append-transition*[*OF* ‹*io* @ [(*x*, *yq*)] ∈ *L ?C*›] **by** *blast*
then have *path ?C* (*initial ?C*) *pA′* **and** *p-io pA′* = *io* **and** *tA′* ∈ *transitions ?C* **and** *t-source tA′* = *target* (*initial ?C*) *pA′* **and** *t-input tA′* = *x* **and** *t-output tA′* = *yq*
by *auto*

have *pA* = *pA′*
using *observable-path-unique*[*OF* ‹*observable ?C*› ‹*path ?C* (*initial ?C*) *pA′*› ‹*path ?C* (*initial ?C*) *pA*›]
using ‹*p-io pA′* = *io*› ‹*p-io pA* = *io*› **by** *auto*
then have *t-source tA′* = *target* (*initial A*) *pA*
using ‹*t-source tA′* = *target* (*initial ?C*) *pA′*›
using *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*3*,*4*)]
using *canonical-separator-initial*[*OF assms*(*3*,*4*)] **by** *fastforce*

have *tA′* ∈ *transitions A*
using *compl*[*OF* ‹*?q* ∈ *reachable-states A*› ‹*x* ∈ (*inputs M*)› ‹∃ *t*∈(*transitions A*). *t-source t* = *target* (*initial A*) *pA* ∧ *t-input t* = *x*› ‹*tA′* ∈ *transitions ?C*› ‹*t-source tA′* = *target* (*initial A*) *pA*› ‹*t-input tA′* = *x*›] **by** *assumption*
then have *path A* (*initial A*) (*pA*@[*tA′*])
using ‹*path A* (*initial A*) *pA*› ‹*t-source tA′* = *target* (*initial A*) *pA*› **using** *path-append-transition* **by** *metis*
moreover have *p-io* (*pA*@[*tA′*]) = *io*@[(*x*,*yq*)]

using ‹*t-input tA′ = x*› ‹*t-output tA′ = yq*› ‹*p-io pA = io*› **by** *auto*

ultimately show *?thesis*
using *language-state-containment*
by (*metis* (*mono-tags*, *lifting*))

next
case *False*

let *?P = product* (*from-FSM M q1*) (*from-FSM M q2*)
let *?qq* = (*target q1 p1*, *target q2 p2*)
let *?tA* = (*Inl* (*target q1 p1*, *target q2 p2*), *x*, *yq*, *Inr q1*)

have *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*
using *from-FSM-path-initial*[*OF* ‹*q1 ∈ states M*›] ‹*path M q1 p1*› **by** *auto*
have *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*
using *from-FSM-path-initial*[*OF* ‹*q2 ∈ states M*›] ‹*path M q2 p2*› **by** *auto*
have *p-io p1 = p-io p2*
using ‹*p-io p1 = io*› ‹*p-io p2 = io*› **by** *auto*

have *?qq ∈ states ?P*
using *reachable-states-intro*[*OF product-path-from-paths*(*1*)[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*› ‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*› ‹*p-io p1 = p-io p2*›]]
unfolding *product-path-from-paths*(*2*)[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*› ‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*› ‹*p-io p1 = p-io p2*›] *from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)]
using *reachable-state-is-state*
by *metis*
moreover have ∃ *q′*. (*target q1 p1*, *x*, *yq*, *q′*) ∈ *FSM.transitions M*
using ‹*t1 ∈ FSM.transitions M*› ‹*t-input t1 = x*› ‹*t-output t1 = yq*› ‹*t-source t1 = target q1 p1*›
by (*metis prod.collapse*)
moreover have ¬(∃ *q′*. (*target q2 p2*, *x*, *yq*, *q′*) ∈ *FSM.transitions M*)
using ‹*t1 ∈ FSM.transitions M*› ‹*t-input t1 = x*› ‹*t-output t1 = yq*› ‹*t-source t1 = target q1 p1*› *False*
by *fastforce*
ultimately have *?tA* ∈ (*distinguishing-transitions-left M q1 q2*)
unfolding *distinguishing-transitions-left-def*
by *blast*

then have (*Inl* (*target q1 p1*, *target q2 p2*), *x*, *yq*, *Inr q1*) ∈ *transitions ?C*
using *canonical-separator-distinguishing-transitions-left-containment*[*OF -assms*(*3*,*4*)] **by** *metis*

let *?pP = zip-path p1 p2*
let *?pC = map shift-Inl ?pP*
have *path ?P* (*initial ?P*) *?pP*

and *target* (*initial ?P*) *?pP* = (*target q1 p1*, *target q2 p2*)
using *product-path-from-paths*[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*›
‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*›
‹*p-io p1* = *p-io p2*›]
using *assms*(*3*,*4*) **by** *auto*

have *length p1* = *length p2*
using ‹*p-io p1* = *p-io p2*› *map-eq-imp-length-eq* **by** *blast*
then have *p-io ?pP* = *io*
using ‹*p-io p1* = *io*› **by** (*induction p1 p2 arbitrary*: *io rule*: *list-induct2*; *auto*)

have *path ?C* (*initial ?C*) *?pC*
using *canonical-separator-path-shift*[*OF assms*(*3*,*4*)] ‹*path ?P* (*initial ?P*) *?pP*› **by** *simp*

have *target* (*initial ?C*) *?pC* = *Inl* (*target q1 p1*, *target q2 p2*)
using *path-map-target*[*of Inl initial ?P Inl id id ?pP*]
using ‹*target* (*initial ?P*) *?pP* = (*target q1 p1*, *target q2 p2*)›
unfolding *canonical-separator-simps*[*OF assms*(*3*,*4*)] *product-simps from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)]
by *fastforce*

have *p-io ?pC* = *io*
using ‹*p-io ?pP* = *io*› **by** *auto*
have *p-io pA* = *p-io ?pC*
unfolding ‹*p-io ?pC* = *io*›
using ‹*p-io pA* = *io*› **by** *assumption*

then have *?pC* = *pA*
using *observable-path-unique*[*OF* ‹*observable ?C*› ‹*path ?C* (*initial ?C*) *pA*› ‹*path ?C* (*initial ?C*) *?pC*›] **by** *auto*
then have *t-source ?tA* = *target* (*initial A*) *pA*
using ‹*target* (*initial ?C*) *?pC* = *Inl* (*target q1 p1*, *target q2 p2*)›
unfolding *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*3*,*4*)]
canonical-separator-simps[*OF assms*(*3*,*4*)] **by** *force*

have *?tA* ∈ *transitions A*
using *compl*[*OF* ‹*?q* ∈ *reachable-states A*› ‹*x* ∈ (*inputs M*)› ‹∃ *t*∈(*transitions A*). *t-source t* = *target* (*initial A*) *pA* ∧ *t-input t* = *x*› ‹*?tA* ∈ *transitions ?C*› ‹*t-source ?tA* = *target* (*initial A*) *pA*›]
unfolding *snd-conv fst-conv* **by** *simp*

have ∗: *path A* (*initial A*) (*pA*@[*?tA*])
using *path-append-transition*[*OF* ‹*path A* (*initial A*) *pA*› ‹*?tA* ∈ *transitions A*› ‹*t-source ?tA* = *target* (*initial A*) *pA*›] **by** *assumption*

have $**$: *p-io* (*pA*@[*?tA*]) = *io*@[(*x*,*yq*)]
using ‹*p-io pA* = *io*› **by** *auto*

show *?thesis*
using *language-state-containment*[*OF* $*$ $**$] **by** *assumption*
qed
qed

moreover have $\bigwedge$ *x yq yt*. *io* @ [(*x*, *yq*)] ∈ *LS M q2* ⟹ *io* @ [(*x*, *yt*)] ∈ *L A* ⟹ *io* @ [(*x*, *yq*)] ∈ *L A*
proof −
fix *x yq yt* **assume** *io* @ [(*x*, *yq*)] ∈ *LS M q2* **and** *io* @ [(*x*, *yt*)] ∈ *L A*

obtain *pA tA* **where** *path A* (*initial A*) (*pA*@[*tA*]) **and** *p-io* (*pA*@[*tA*]) = *io* @ [(*x*, *yt*)]
using *language-initial-path-append-transition*[*OF* ‹*io* @ [(*x*, *yt*)] ∈ *L A*›] **by** *blast*
then have *path A* (*initial A*) *pA* **and** *p-io pA* = *io*
by *auto*
then have *path ?C* (*initial ?C*) *pA*
using *submachine-path-initial*[*OF* ‹*is-submachine A ?C*›] **by** *auto*

obtain *p2 t2* **where** *path M q2* (*p2*@[*t2*]) **and** *p-io* (*p2*@[*t2*]) = *io* @ [(*x*, *yq*)]
using *language-path-append-transition*[*OF* ‹*io* @ [(*x*, *yq*)] ∈ *LS M q2*›] **by** *blast*
then have *path M q2 p2* **and** *p-io p2* = *io* **and** *t2* ∈ *transitions M* **and** *t-input t2* = *x* **and** *t-output t2* = *yq* **and** *t-source t2* = *target q2 p2*
by *auto*

let *?q* = *target* (*initial A*) *pA*
have *?q* ∈ *states A*
using *path-target-is-state* ‹*path A* (*initial A*) (*pA*@[*tA*])› **by** *auto*

have *tA* ∈ *transitions A* **and** *t-input tA* = *x* **and** *t-output tA* = *yt* **and** *t-source tA* = *target* (*initial A*) *pA*
using ‹*path A* (*initial A*) (*pA*@[*tA*])› ‹*p-io* (*pA*@[*tA*]) = *io* @ [(*x*, *yt*)]› **by** *auto*
then have *x* ∈ (*inputs M*)
using ‹*is-submachine A ?C*›
unfolding *is-submachine.simps canonical-separator-simps*[*OF assms*(*3*,*4*)] **by** *auto*

have ∃ *t*∈(*transitions A*). *t-source t* = *target* (*initial A*) *pA* ∧ *t-input t* = *x*
using ‹*tA* ∈ *transitions A*› ‹*t-input tA* = *x*› ‹*t-source tA* = *target* (*initial A*) *pA*› **by** *blast*

have *io* ∈ *LS M q1*
using *submachine-language*[*OF* ‹*is-submachine A ?C*›] ‹*io* @ [(*x*, *yt*)] ∈ *L A*›
using *canonical-separator-language-prefix*[*OF - assms*(*3*,*4*,*2*,*5*), *of io* (*x*,*yt*)] **by** *blast*
then obtain *p1* **where** *path M q1 p1* **and** *p-io p1* = *io*
by *auto*

show *io* @ [(*x*, *yq*)] ∈ *L A*
proof (*cases* ∃ *t1* ∈ *transitions M* . *t-source t1* = *target q1 p1* ∧ *t-input t1* = *x* ∧ *t-output t1* = *yq*)
case *True*
then obtain *t1* **where** *t1* ∈ *transitions M* **and** *t-source t1* = *target q1 p1* **and** *t-input t1* = *x* **and** *t-output t1* = *yq*
by *blast*
then have *path M q1* (*p1*@[*t1*]) **and** *p-io* (*p1*@[*t1*]) = *io*@[(*x*,*yq*)]
using *path-append-transition*[*OF* ‹*path M q1 p1*›] ‹*p-io p1* = *io*› **by** *auto*
then have *io* @ [(*x*, *yq*)] ∈ *LS M q1*
unfolding *LS.simps* **by** (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io*@[(*x*,*yq*)] ∈ *L ?C*
using *canonical-separator-language-intersection*[*OF* - ‹*io* @ [(*x*, *yq*)] ∈ *LS M q2*› *assms*(*3*,*4*)] **by** *blast*

obtain *pA′ tA′* **where** *path ?C* (*initial ?C*) (*pA′*@[*tA′*]) **and** *p-io* (*pA′*@[*tA′*]) = *io*@[(*x*,*yq*)]
using *language-initial-path-append-transition*[*OF* ‹*io* @ [(*x*, *yq*)] ∈ *L ?C*›] **by** *blast*
then have *path ?C* (*initial ?C*) *pA′* **and** *p-io pA′* = *io* **and** *tA′* ∈ *transitions ?C* **and** *t-source tA′* = *target* (*initial ?C*) *pA′* **and** *t-input tA′* = *x* **and** *t-output tA′* = *yq*
by *auto*

have *pA* = *pA′*
using *observable-path-unique*[*OF* ‹*observable ?C*› ‹*path ?C* (*initial ?C*) *pA′*› ‹*path ?C* (*initial ?C*) *pA*›]
using ‹*p-io pA′* = *io*› ‹*p-io pA* = *io*› **by** *auto*
then have *t-source tA′* = *target* (*initial A*) *pA*
using ‹*t-source tA′* = *target* (*initial ?C*) *pA′*›
using *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*3*,*4*)]

unfolding *canonical-separator-simps*[*OF assms*(*3*,*4*)] **by** *auto*

have *?q* ∈ *reachable-states A*
using ‹*path A* (*initial A*) *pA*› *reachable-states-intro* **by** *blast*

have *tA′* ∈ *transitions A*
using *compl*[*OF* ‹*?q* ∈ *reachable-states A*› ‹*x* ∈ (*inputs M*)› ‹∃ *t*∈(*transitions*

A). *t-source t* = *target* (*initial A*) *pA* ∧ *t-input t* = *x*› ‹*tA′* ∈ *transitions ?C*› ‹*t-source tA′* = *target* (*initial A*) *pA*› ‹*t-input tA′* = *x*›] **by** *assumption*
then have *path A* (*initial A*) (*pA*@[*tA′*])
using ‹*path A* (*initial A*) *pA*› ‹*t-source tA′* = *target* (*initial A*) *pA*› **using** *path-append-transition* **by** *metis*
moreover have *p-io* (*pA*@[*tA′*]) = *io*@[(*x*,*yq*)]
using ‹*t-input tA′* = *x*› ‹*t-output tA′* = *yq*› ‹*p-io pA* = *io*› **by** *auto*

ultimately show *?thesis*
using *language-state-containment*
by (*metis* (*mono-tags*, *lifting*))

next
case *False*

let *?P* = *product* (*from-FSM M q1*) (*from-FSM M q2*)
let *?qq* = (*target q1 p1*, *target q2 p2*)
let *?tA* = (*Inl* (*target q1 p1*, *target q2 p2*), *x*, *yq*, *Inr q2*)

have *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*
using *from-FSM-path-initial*[*OF* ‹*q1* ∈ *states M*›] ‹*path M q1 p1*› **by** *auto*
have *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*
using *from-FSM-path-initial*[*OF* ‹*q2* ∈ *states M*›] ‹*path M q2 p2*› **by** *auto*
have *p-io p1* = *p-io p2*
using ‹*p-io p1* = *io*› ‹*p-io p2* = *io*› **by** *auto*

have *?qq* ∈ *states ?P*
using *reachable-states-intro*[*OF product-path-from-paths*(*1*)[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*› ‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*› ‹*p-io p1* = *p-io p2*›]]
unfolding *product-path-from-paths*(*2*)[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*› ‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*› ‹*p-io p1* = *p-io p2*›] *from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)]
using *reachable-state-is-state* **by** *metis*
moreover have ∃ *q′*. (*target q2 p2*, *x*, *yq*, *q′*) ∈ *FSM.transitions M*
using ‹*t2* ∈ *FSM.transitions M*› ‹*t-input t2* = *x*› ‹*t-output t2* = *yq*› ‹*t-source t2* = *target q2 p2*›
by (*metis prod.collapse*)
moreover have ¬(∃ *q′*. (*target q1 p1*, *x*, *yq*, *q′*) ∈ *FSM.transitions M*)
using ‹*t2* ∈ *FSM.transitions M*› ‹*t-input t2* = *x*› ‹*t-output t2* = *yq*› ‹*t-source t2* = *target q2 p2*› *False*
by *fastforce*
ultimately have *?tA* ∈ (*distinguishing-transitions-right M q1 q2*)
unfolding *distinguishing-transitions-right-def*
by *blast*

then have *?tA* ∈ *transitions ?C*
using *canonical-separator-distinguishing-transitions-right-containment*[*OF*

- *assms*(*3*,*4*)] **by** *metis*

let *?pP* = *zip-path p1 p2*
let *?pC* = *map shift-Inl ?pP*
have *path ?P* (*initial ?P*) *?pP*
and *target* (*initial ?P*) *?pP* = (*target q1 p1*, *target q2 p2*)
using *product-path-from-paths*[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1*›
‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2*›
‹*p-io p1* = *p-io p2*›]
using *assms*(*3*,*4*) **by** *auto*

have *length p1* = *length p2*
using ‹*p-io p1* = *p-io p2*› *map-eq-imp-length-eq* **by** *blast*
then have *p-io ?pP* = *io*
using ‹*p-io p1* = *io*› **by** (*induction p1 p2 arbitrary*: *io rule*: *list-induct2*; *auto*)

have *path ?C* (*initial ?C*) *?pC*
using *canonical-separator-path-shift*[*OF assms*(*3*,*4*)] ‹*path ?P* (*initial ?P*) *?pP*› **by** *simp*

have *target* (*initial ?C*) *?pC* = *Inl* (*target q1 p1*, *target q2 p2*)
using *path-map-target*[*of Inl initial ?P Inl id id ?pP*]
using ‹*target* (*initial ?P*) *?pP* = (*target q1 p1*, *target q2 p2*)›
unfolding *canonical-separator-simps*[*OF assms*(*3*,*4*)] *product-simps from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF assms*(*4*)] **by** *force*

have *p-io ?pC* = *io*
using ‹*p-io ?pP* = *io*› **by** *auto*
have *p-io pA* = *p-io ?pC*
unfolding ‹*p-io ?pC* = *io*›
using ‹*p-io pA* = *io*› **by** *assumption*

then have *?pC* = *pA*
using *observable-path-unique*[*OF* ‹*observable ?C*› ‹*path ?C* (*initial ?C*) *pA*› ‹*path ?C* (*initial ?C*) *?pC*›] **by** *auto*
then have *t-source ?tA* = *target* (*initial A*) *pA*
using ‹*target* (*initial ?C*) *?pC* = *Inl* (*target q1 p1*, *target q2 p2*)›
unfolding *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*3*,*4*)]
canonical-separator-simps[*OF assms*(*3*,*4*)] **by** *force*

have *?q* ∈ *reachable-states A*
using ‹*path A* (*initial A*) *pA*› *reachable-states-intro* **by** *blast*
have *?tA* ∈ *transitions A*
using *compl*[*OF* ‹*?q* ∈ *reachable-states A*› ‹*x* ∈ (*inputs M*)› ‹∃ *t*∈(*transitions A*). *t-source t* = *target* (*initial A*) *pA* ∧ *t-input t* = *x*› ‹*?tA* ∈ *transitions ?C*›

‹*t-source ?tA* = *target* (*initial A*) *pA*›]
unfolding *snd-conv fst-conv* **by** *simp*

have ∗: *path A* (*initial A*) (*pA*@[*?tA*])
using *path-append-transition*[*OF* ‹*path A* (*initial A*) *pA*› ‹*?tA* ∈ *transitions A*› ‹*t-source ?tA* = *target* (*initial A*) *pA*›] **by** *assumption*

have ∗∗: *p-io* (*pA*@[*?tA*]) = *io*@[(*x*,*yq*)]
using ‹*p-io pA* = *io*› **by** *auto*

show *?thesis*
using *language-state-containment*[*OF* ∗ ∗∗] **by** *assumption*
qed
qed

ultimately show (*io* ∈ *LS M q1* − *LS M q2* ⟶ *io-targets A io* (*initial A*) = {*Inr q1*}) ∧
(*io* ∈ *LS M q2* − *LS M q1* ⟶ *io-targets A io* (*initial A*) = {*Inr q2*}) ∧
(*io* ∈ *LS M q1* ∩ *LS M q2* ⟶ *io-targets A io* (*initial A*) ∩ {*Inr q1*, *Inr q2*} = {}) ∧
(∀ *x yq yt*. *io* @ [(*x*, *yq*)] ∈ *LS M q1* ∧ *io* @ [(*x*, *yt*)] ∈ *LS A* (*initial A*) ⟶ *io* @ [(*x*, *yq*)] ∈ *LS A* (*initial A*)) ∧
(∀ *x yq2 yt*. *io* @ [(*x*, *yq2*)] ∈ *LS M q2* ∧ *io* @ [(*x*, *yt*)] ∈ *LS A* (*initial A*) ⟶ *io* @ [(*x*, *yq2*)] ∈ *LS A* (*initial A*))
by *blast*
qed

moreover have ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *Inr q1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*
using *canonical-separator-maximal-path-distinguishes-left*[*OF assms*(*1*) - - ‹*observable M*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*q1* ≠ *q2*›] **by** *blast*
moreover have ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* = *Inr q2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*
using *canonical-separator-maximal-path-distinguishes-right*[*OF assms*(*1*) - - ‹*observable M*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*q1* ≠ *q2*›] **by** *blast*
moreover have ⋀ *p* . *path A* (*initial A*) *p* ⟹ *target* (*initial A*) *p* ≠ *Inr q1* ⟹ *target* (*initial A*) *p* ≠ *Inr q2* ⟹ *p-io p* ∈ *LS M q1* ∩ *LS M q2*
proof −
fix *p* **assume** *path A* (*initial A*) *p* **and** *target* (*initial A*) *p* ≠ *Inr q1* **and** *target* (*initial A*) *p* ≠ *Inr q2*

have *path ?C* (*initial ?C*) *p*
using *submachine-path-initial*[*OF is-state-separator-from-canonical-separator-simps*(*1*)[*OF assms*(*1*)] ‹*path A* (*initial A*) *p*›] **by** *assumption*

have *target* (*initial ?C*) *p* ≠ *Inr q1* **and** *target* (*initial ?C*) *p* ≠ *Inr q2*
using ‹*target* (*initial A*) *p* ≠ *Inr q1*› ‹*target* (*initial A*) *p* ≠ *Inr q2*›
unfolding *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*,*3*,*4*)]

canonical-separator-initial[*OF assms*(*3*,*4*)] **by** *blast+*

then have *isl* (*target* (*initial ?C*) *p*)
using *canonical-separator-path-initial*(*4*)[*OF* ‹*path ?C* (*initial ?C*) *p*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*observable M*› ‹*q1* ≠ *q2*›]
by *auto*

then show *p-io p* ∈ *LS M q1* ∩ *LS M q2*
using ‹*path ?C* (*initial ?C*) *p*› *canonical-separator-path-targets-language*(*1*)[*OF* - ‹*observable M*› ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› ‹*q1* ≠ *q2*›]
by *auto*
qed

moreover have (*inputs A*) ⊆ (*inputs M*)
using ‹*is-submachine A ?C*›
unfolding *is-submachine.simps canonical-separator-simps*[*OF assms*(*3*,*4*)] **by** *auto*

ultimately show *?thesis*
unfolding *is-separator-def*
using *p1 p2 p3 p4 p5 p6 p7 p8* ‹*q1* ≠ *q2*›
by (*meson sum.simps*(*2*))
qed

lemma *is-separator-separated-state-is-state* :
assumes *is-separator M q1 q2 A t1 t2*
shows *q1* ∈ *states M* **and** *q2* ∈ *states M*
proof −
have *initial A* ≠ *t1*
using *separator-initial*[*OF assms*(*1*)] **by** *blast*

have *t1* ∈ *reachable-states A*
and ⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *target* (*FSM.initial A*) *p* = *t1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*
and *t2* ∈ *reachable-states A*
and ⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *target* (*FSM.initial A*) *p* = *t2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*
using *is-separator-simps*[*OF assms*(*1*)]
by *blast+*

obtain *p1* **where** *path A* (*FSM.initial A*) *p1* **and** *target* (*FSM.initial A*) *p1* = *t1*
using ‹*t1* ∈ *reachable-states A*› **unfolding** *reachable-states-def* **by** *auto*
then have *p-io p1* ∈ *LS M q1* − *LS M q2*
using ‹⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *target* (*FSM.initial A*) *p* = *t1* ⟹ *p-io p* ∈ *LS M q1* − *LS M q2*›
by *blast*
then show *q1* ∈ *states M* **unfolding** *LS.simps*

using *path-begin-state* **by** *fastforce*

obtain *p2* **where** *path A* (*FSM.initial A*) *p2* **and** *target* (*FSM.initial A*) *p2* = *t2*
using ‹*t2* ∈ *reachable-states A*› **unfolding** *reachable-states-def* **by** *auto*
then have *p-io p2* ∈ *LS M q2* − *LS M q1*
using ‹⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *target* (*FSM.initial A*) *p* = *t2* ⟹ *p-io p* ∈ *LS M q2* − *LS M q1*›
by *blast*
then show *q2* ∈ *states M* **unfolding** *LS.simps*
using *path-begin-state* **by** *fastforce*
qed

end

35 Adaptive Test Cases

An ATC is a single input, acyclic, observable FSM, which is equivalent to a tree whose non-leaf states are labeled with inputs and whose edges are labeled with outputs.

theory *Adaptive-Test-Case*
imports *State-Separator*
begin

definition *is-ATC* :: (*'a*,*'b*,*'c*) *fsm* ⇒ *bool* **where**
is-ATC M = (*single-input M* ∧ *acyclic M* ∧ *observable M*)

lemma *is-ATC-from* :
assumes *t* ∈ *transitions A*
and *t-source t* ∈ *reachable-states A*
and *is-ATC A*
shows *is-ATC* (*from-FSM A* (*t-target t*))
using *from-FSM-single-input*[*of A*]
from-FSM-acyclic[*OF reachable-states-next*[*OF assms*(*2*,*1*)]]
from-FSM-observable[*of A*]
assms(*3*)
unfolding *is-ATC-def* **by** *fast*

35.1 Applying Adaptive Test Cases

fun *pass-ATC'* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'d*,*'b*,*'c*) *fsm* ⇒ *'d set* ⇒ *nat* ⇒ *bool* **where**
pass-ATC' M A fail-states 0 = (¬ (*initial A* ∈ *fail-states*)) |
pass-ATC' M A fail-states (*Suc k*) = ((¬ (*initial A* ∈ *fail-states*)) ∧
(∀ *x* ∈ *inputs A* . *h A* (*initial A*,*x*) ≠ {} ⟶ (∀ (*yM*,*qM*) ∈ *h M* (*initial M*,*x*) . ∃ (*yA*,*qA*) ∈ *h A* (*initial A*,*x*) . *yM* = *yA* ∧ *pass-ATC'* (*from-FSM M qM*) (*from-FSM A qA*) *fail-states k*)))

fun *pass-ATC* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('d,'b,'c)$ *fsm* $\Rightarrow$ $'d$ *set* $\Rightarrow$ *bool* **where**
  pass-ATC M A fail-states = *pass-ATC′ M A fail-states* (*size A*)

lemma *pass-ATC′-initial* :
  assumes *pass-ATC′ M A FS k*
  shows *initial A* $\notin$ *FS*
using *assms* **by** (*cases k*; *auto*)

lemma *pass-ATC′-io* :
  assumes *pass-ATC′ M A FS k*
  and *is-ATC A*
  and *observable M*
  and (*inputs A*) $\subseteq$ (*inputs M*)
  and *io*@[*ioA*] $\in$ *L A*
  and *io*@[*ioM*] $\in$ *L M*
  and *fst ioA* = *fst ioM*
  and *length* (*io*@[*ioA*]) $\leq$ *k*
shows *io*@[*ioM*] $\in$ *L A*
and *io-targets A* (*io*@[*ioM*]) (*initial A*) $\cap$ *FS* = {}
proof −
  have *io*@[*ioM*] $\in$ *L A* $\wedge$ *io-targets A* (*io*@[*ioM*]) (*initial A*) $\cap$ *FS* = {}
    using *assms* **proof** (*induction k arbitrary*: *io A M*)
    case *0*
    then show *?case* **by** *auto*
  next
    case (*Suc k*)
    then show *?case* **proof** (*cases io*)
      case *Nil*

      obtain *tA* **where** *tA* $\in$ *transitions A*
                and *t-source tA* = *initial A*
                and *t-input tA* = *fst ioA*
                and *t-output tA* = *snd ioA*
        using *Nil* ‹*io*@[*ioA*] $\in$ *L A*› **by** *auto*
      then have *fst ioA* $\in$ (*inputs A*)
        using *fsm-transition-input* **by** *fastforce*

      have (*t-output tA*,*t-target tA*) $\in$ *h A* (*initial A*,*t-input tA*)
        using ‹*tA* $\in$ *transitions A*› ‹*t-source tA* = *initial A*› **unfolding** *h-simps*
        by (*metis* (*no-types*, *lifting*) *case-prodI mem-Collect-eq prod.collapse*)
      then have *h A* (*initial A*,*fst ioA*) $\neq$ {}
        unfolding ‹*t-input tA* = *fst ioA*› **by** *blast*

      then have $*$: $\bigwedge$ *yM qM* . (*yM*,*qM*) $\in$ *h M* (*initial M*,*fst ioA*) $\Longrightarrow$ ($\exists$ (*yA*,*qA*)
$\in$ *h A* (*initial A*,*fst ioA*) . *yM* = *yA* $\wedge$ *pass-ATC′* (*from-FSM M qM*) (*from-FSM*

A qA) *FS k*)
using *Suc.prems*(*1*) *pass-ATC′-initial*[*OF Suc.prems*(*1*)] **unfolding** *pass-ATC′.simps*
using ‹*fst ioA* ∈ *FSM.inputs A*› **by** *auto*

obtain *tM* **where** *tM* ∈ *transitions M*
and *t-source tM* = *initial M*
and *t-input tM* = *fst ioA*
and *t-output tM* = *snd ioM*
using *Nil* ‹*io*@[*ioM*] ∈ *L M*› ‹*fst ioA* = *fst ioM*› **by** *auto*
have (*t-output tM*,*t-target tM*) ∈ *h M* (*initial M*,*fst ioA*)
using ‹*tM* ∈ *transitions M*› ‹*t-source tM* = *initial M*› ‹*t-input tM* = *fst ioA*› **unfolding** *h-simps*
by (*metis* (*mono-tags*, *lifting*) *case-prodI mem-Collect-eq prod.collapse*)

obtain *tA′* **where** *tA′* ∈ *transitions A*
and *t-source tA′* = *initial A*
and *t-input tA′* = *fst ioA*
and *t-output tA′* = *snd ioM*
and *pass-ATC′* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA′*)) *FS k*
using ∗[*OF* ‹(*t-output tM*,*t-target tM*) ∈ *h M* (*initial M*,*fst ioA*)›]
unfolding *h.simps* ‹*t-output tM* = *snd ioM*› **by** *fastforce*

then have *path A* (*initial A*) [*tA′*]
using *single-transition-path*[*OF* ‹*tA′* ∈ *transitions A*›] **by** *auto*
moreover have *p-io* [*tA′*] = [*ioM*]
using ‹*t-input tA′* = *fst ioA*› ‹*t-output tA′* = *snd ioM*› **unfolding** ‹*fst ioA* = *fst ioM*› **by** *auto*
ultimately have [*ioM*] ∈ *LS A* (*initial A*)
unfolding *LS.simps* **by** (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io* @ [*ioM*] ∈ *LS A* (*initial A*)
using *Nil* **by** *auto*

have *target* (*initial A*) [*tA′*] = *t-target tA′*
by *auto*
then have *t-target tA′* ∈ *io-targets A* [*ioM*] (*initial A*)
unfolding *io-targets.simps*
using ‹*path A* (*initial A*) [*tA′*]› ‹*p-io* [*tA′*] = [*ioM*]›
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io-targets A* (*io* @ [*ioM*]) (*initial A*) = {*t-target tA′*}
using *observable-io-targets*[*OF* - ‹*io* @ [*ioM*] ∈ *LS A* (*initial A*)›] ‹*is-ATC A*› *Nil*
unfolding *is-ATC-def*
by (*metis append-self-conv2 singletonD*)
moreover have *t-target tA′* ∉ *FS*
using *pass-ATC′-initial*[*OF* ‹*pass-ATC′* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA′*)) *FS k*›]
unfolding *from-FSM-simps*(*1*)[*OF fsm-transition-target*[*OF* ‹*tA′* ∈ *transi-*

tions A›]] **by** *assumption*
ultimately have *io-targets A* (*io* @ [*ioM*]) (*initial A*) ∩ *FS* = {}
by *auto*

then show *?thesis*
using ‹*io* @ [*ioM*] ∈ *LS A* (*initial A*)› **by** *auto*
next
case (*Cons io′ io″*)

have [*io′*] ∈ *L A*
using *Cons* ‹*io*@[*ioA*] ∈ *L A*›
by (*metis append.left-neutral append-Cons language-prefix*)
then obtain *tA* **where** *tA* ∈ *transitions A*
and *t-source tA* = *initial A*
and *t-input tA* = *fst io′*
and *t-output tA* = *snd io′*
by *auto*
then have *fst io′* ∈ (*inputs A*)
using *fsm-transition-input* **by** *metis*

have (*t-output tA*,*t-target tA*) ∈ *h A* (*initial A*,*t-input tA*)
using ‹*tA* ∈ *transitions A*› ‹*t-source tA* = *initial A*› **unfolding** *h-simps*
by (*metis* (*no-types*, *lifting*) *case-prodI mem-Collect-eq prod.collapse*)
then have *h A* (*initial A*,*fst io′*) ≠ {}
unfolding ‹*t-input tA* = *fst io′*› **by** *blast*

then have ∗: ⋀ *yM qM* . (*yM*,*qM*) ∈ *h M* (*initial M*,*fst io′*) ⟹ (∃ (*yA*,*qA*) ∈ *h A* (*initial A*,*fst io′*) . *yM* = *yA* ∧ *pass-ATC′* (*from-FSM M qM*) (*from-FSM A qA*) *FS k*)
using *Suc.prems*(*1*) *pass-ATC′-initial*[*OF Suc.prems*(*1*)] **unfolding** *pass-ATC′.simps*
using ‹*fst io′* ∈ *FSM.inputs A*› **by** *auto*

obtain *tM* **where** *tM* ∈ *transitions M*
and *t-source tM* = *initial M*
and *t-input tM* = *fst io′*
and *t-output tM* = *snd io′*
using *Cons* ‹*io*@[*ioM*] ∈ *L M*› ‹*fst ioA* = *fst ioM*› **by** *auto*
have (*t-output tM*,*t-target tM*) ∈ *h M* (*initial M*,*fst io′*)
using ‹*tM* ∈ *transitions M*› ‹*t-source tM* = *initial M*› ‹*t-input tM* = *fst io′*› **unfolding** *h-simps*
by (*metis* (*mono-tags*, *lifting*) *case-prodI mem-Collect-eq prod.collapse*)

obtain *tA′* **where** *tA′* ∈ *transitions A*
and *t-source tA′* = *initial A*
and *t-input tA′* = *fst io′*
and *t-output tA′* = *snd io′*
and *pass-ATC′* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA′*)) *FS k*

using ∗[*OF* ‹(*t-output tM*,*t-target tM*) ∈ *h M* (*initial M*,*fst io′*)›]
unfolding *h.simps* ‹*t-output tM* = *snd io′*› **by** *fastforce*

then have *tA* = *tA′*
using ‹*is-ATC A*›
unfolding *is-ATC-def observable.simps*
by (*metis* ‹*tA* ∈ *transitions A*› ‹*t-input tA* = *fst io′*› ‹*t-output tA* = *snd io′*› ‹*t-source tA* = *initial A*› *prod.collapse*)
then have *pass-ATC′* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA*)) *FS k*
using ‹*pass-ATC′* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA′*)) *FS k*› **by** *auto*

have (*inputs* (*from-FSM A* (*t-target tA*))) ⊆ (*inputs* (*from-FSM M* (*t-target tM*)))
using *Suc.prems*(*4*)
unfolding *from-FSM-simps*(*2*)[*OF fsm-transition-target*[*OF* ‹*tM* ∈ *transitions M*›]]
from-FSM-simps(*2*)[*OF fsm-transition-target*[*OF* ‹*tA* ∈ *transitions A*›]] **by** *assumption*

have *length* (*io′′* @ [*ioA*]) ≤ *k*
using *Cons* ‹*length* (*io* @ [*ioA*]) ≤ *Suc k*› **by** *auto*

have (*io′* # (*io′′*@[*ioA*])) ∈ *LS A* (*t-source tA*)
using ‹*t-source tA* = *initial A*› ‹*io*@[*ioA*] ∈ *L A*› *Cons* **by** *auto*
have *io′′* @ [*ioA*] ∈ *LS* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*)))
using *observable-language-next*[*OF* ‹(*io′* # (*io′′*@[*ioA*])) ∈ *LS A* (*t-source tA*)›]
‹*is-ATC A*› ‹*tA* ∈ *transitions A*› ‹*t-input tA* = *fst io′*› ‹*t-output tA* = *snd io′*›
using *is-ATC-def* **by** *blast*

have (*io′* # (*io′′*@[*ioM*])) ∈ *LS M* (*t-source tM*)
using ‹*t-source tM* = *initial M*› ‹*io*@[*ioM*] ∈ *L M*› *Cons* **by** *auto*
have *io′′* @ [*ioM*] ∈ *LS* (*from-FSM M* (*t-target tM*)) (*initial* (*from-FSM M* (*t-target tM*)))
using *observable-language-next*[*OF* ‹(*io′* # (*io′′*@[*ioM*])) ∈ *LS M* (*t-source tM*)›]
‹*observable M*› ‹*tM* ∈ *transitions M*› ‹*t-input tM* = *fst io′*› ‹*t-output tM* = *snd io′*›
by *blast*

have *observable* (*from-FSM M* (*t-target tM*))
using *from-FSM-observable*[*OF* ‹*observable M*›] **by** *blast*

have *is-ATC* (*FSM.from-FSM A* (*t-target tA*))
using *is-ATC-from*[*OF* ‹*tA* ∈ *transitions A*› - ‹*is-ATC A*›] *reachable-states-initial*

unfolding ‹*t-source tA* = *initial A*› **by** *blast*

have *io''* @ [*ioM*] ∈ *LS* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*)))
and *io-targets* (*from-FSM A* (*t-target tA*)) (*io''* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*))) ∩ *FS* = {}
using *Suc.IH*[*OF* ‹*pass-ATC'* (*from-FSM M* (*t-target tM*)) (*from-FSM A* (*t-target tA*)) *FS k*›
‹*is-ATC* (*FSM.from-FSM A* (*t-target tA*))›
‹*observable* (*from-FSM M* (*t-target tM*))›
‹(*inputs* (*from-FSM A* (*t-target tA*))) ⊆ (*inputs* (*from-FSM M* (*t-target tM*)))›
‹*io''* @ [*ioA*] ∈ *LS* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*)))›
‹*io''* @ [*ioM*] ∈ *LS* (*from-FSM M* (*t-target tM*)) (*initial* (*from-FSM M* (*t-target tM*)))›
‹*fst ioA* = *fst ioM*›
‹*length* (*io''* @ [*ioA*]) ≤ *k*›]
by *blast+*

then obtain *pA* **where** *path* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*))) *pA* **and** *p-io pA* = *io''* @ [*ioM*]
by *auto*

have *path A* (*initial A*) (*tA*#*pA*)
using ‹*path* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*))) *pA*› ‹*tA* ∈ *transitions A*›
by (*metis* ‹*t-source tA* = *initial A*› *cons from-FSM-path-initial fsm-transition-target*)
moreover have *p-io* (*tA*#*pA*) = *io'* # *io''* @ [*ioM*]
using ‹*t-input tA* = *fst io'*› ‹*t-output tA* = *snd io'*› ‹*p-io pA* = *io''* @ [*ioM*]› **by** *auto*
ultimately have *io'* # *io''* @ [*ioM*] ∈ *L A*
unfolding *LS.simps*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io* @ [*ioM*] ∈ *L A*
using *Cons* **by** *auto*

have *observable A*
using *Suc.prems*(*2*) *is-ATC-def* **by** *blast*

have *io-targets A* (*io* @ [*ioM*]) (*FSM.initial A*) ∩ *FS* = {}
proof −
have ⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *p-io p* = (*io'* # *io''*) @ [*ioM*] ⟹ *p* = *tA* # (*tl p*)
using ‹*observable A*› **unfolding** *observable.simps*
using ‹*tA* ∈ *transitions A*› ‹*t-source tA* = *initial A*› ‹*t-input tA* = *fst io'*› ‹*t-output tA* = *snd io'*› **by** *fastforce*

have ⋀ *q* . *q* ∈ *io-targets A* (*io* @ [*ioM*]) (*FSM.initial A*) ⟹ *q* ∈ *io-targets*

(*from-FSM A* (*t-target tA*)) (*io''* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*)))
proof −
fix *q* **assume** *q* ∈ *io-targets A* (*io* @ [*ioM*]) (*FSM.initial A*)
then obtain *p* **where** *q* = *target* (*FSM.initial A*) *p* **and** *path A* (*FSM.initial A*) *p* **and** *p-io p* = (*io'* # *io''*) @ [*ioM*]
unfolding *io-targets.simps Cons* **by** *blast*
then have *p* = *tA* # (*tl p*)
using ‹⋀ *p* . *path A* (*FSM.initial A*) *p* ⟹ *p-io p* = (*io'* # *io''*) @ [*ioM*] ⟹ *p* = *tA* # (*tl p*)› **by** *blast*

have *path A* (*FSM.initial A*) (*tA*#(*tl p*))
using ‹*path A* (*FSM.initial A*) *p*› ‹*p* = *tA* # (*tl p*)› **by** *simp*
then have *path* (*from-FSM A* (*t-target tA*)) (*initial* (*from-FSM A* (*t-target tA*))) (*tl p*)
by (*meson from-FSM-path-initial fsm-transition-target path-cons-elim*)
moreover have *p-io* (*tl p*) = (*io''*) @ [*ioM*]
using ‹*p-io p* = (*io'* # *io''*) @ [*ioM*]› ‹*p* = *tA* # (*tl p*)› **by** *auto*
moreover have *q* = *target* (*initial* (*from-FSM A* (*t-target tA*))) (*tl p*)
using ‹*q* = *target* (*FSM.initial A*) *p*› ‹*p* = *tA* # (*tl p*)›
unfolding *target.simps visited-states.simps from-FSM-simps*[*OF fsm-transition-target*[*OF* ‹*tA* ∈ *transitions A*›]]
by (*cases p*; *auto*)
ultimately show *q* ∈ *io-targets* (*from-FSM A* (*t-target tA*)) (*io''* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*)))
unfolding *io-targets.simps* **by** *blast*
qed
moreover have ⋀ *q* . *q* ∈ *io-targets* (*from-FSM A* (*t-target tA*)) (*io''* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*))) ⟹ *q* ∈ *io-targets A* (*io* @ [*ioM*]) (*FSM.initial A*)
proof −
fix *q* **assume** *q* ∈ *io-targets* (*from-FSM A* (*t-target tA*)) (*io''* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*)))
then obtain *p* **where** *q* = *target* (*FSM.initial* (*FSM.from-FSM A* (*t-target tA*))) *p* **and** *path* (*FSM.from-FSM A* (*t-target tA*)) (*FSM.initial* (*FSM.from-FSM A* (*t-target tA*))) *p* **and** *p-io p* = *io''* @ [*ioM*]
unfolding *io-targets.simps Cons* **by** *blast*

have *q* = *target* (*FSM.initial A*) (*tA*#*p*)
unfolding ‹*q* = *target* (*FSM.initial* (*FSM.from-FSM A* (*t-target tA*))) *p*› *from-FSM-simps*[*OF fsm-transition-target*[*OF* ‹*tA* ∈ *transitions A*›]] **by** *auto*
moreover have *path A* (*initial A*) (*tA*#*p*)
using ‹*path* (*FSM.from-FSM A* (*t-target tA*)) (*FSM.initial* (*FSM.from-FSM A* (*t-target tA*))) *p*›
unfolding *from-FSM-path-initial*[*OF fsm-transition-target*[*OF* ‹*tA* ∈ *transitions A*›], *symmetric*]
using ‹*tA* ∈ *transitions A*› ‹*t-source tA* = *initial A*› *cons*
by *fastforce*
moreover have *p-io* (*tA*#*p*) = *io* @ [*ioM*]
using ‹*p-io p* = *io''* @ [*ioM*]› ‹*t-input tA* = *fst io'*› ‹*t-output tA* = *snd*

io′› **unfolding** *Cons* **by** *simp*
ultimately show *q* ∈ *io-targets A* (*io* @ [*ioM*]) (*FSM.initial A*)
unfolding *io-targets.simps* **by** *fastforce*
qed
ultimately show *?thesis*
using ‹*io-targets* (*from-FSM A* (*t-target tA*)) (*io′′* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target tA*))) ∩ *FS* = {}› **by** *blast*
qed

then show *?thesis*
using ‹*io* @ [*ioM*] ∈ *L A*› **by** *simp*
qed
qed

then show *io*@[*ioM*] ∈ *L A*
and *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* = {}
by *simp*+
qed

lemma *pass-ATC-io* :
assumes *pass-ATC M A FS*
and *is-ATC A*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and *io*@[*ioA*] ∈ *L A*
and *io*@[*ioM*] ∈ *L M*
and *fst ioA* = *fst ioM*
shows *io*@[*ioM*] ∈ *L A*
and *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* = {}
proof −

have *acyclic A*
using ‹*is-ATC A*› *is-ATC-def* **by** *blast*

then have *length* (*io* @ [*ioA*]) ≤ (*size A*)
using ‹*io*@[*ioA*] ∈ *L A*› **unfolding** *LS.simps*
using *acyclic-path-length-limit* **unfolding** *acyclic.simps* **by** *fastforce*

show *io*@[*ioM*] ∈ *L A*
and *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* = {}
using *pass-ATC′-io*[*OF - assms*(*2*−*7*) ‹*length* (*io* @ [*ioA*]) ≤ (*size A*)›]
using *assms*(*1*) **by** *simp*+
qed

lemma *pass-ATC-io-explicit-io-tuple* :
assumes *pass-ATC M A FS*
and *is-ATC A*

and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and *io*@[(*x*,*y*)] ∈ *L A*
and *io*@[(*x*,*y*′)] ∈ *L M*
shows *io*@[(*x*,*y*′)] ∈ *L A*
and *io-targets A* (*io*@[(*x*,*y*′)]) (*initial A*) ∩ *FS* = {}
apply (*metis pass-ATC-io*(*1*) *assms fst-conv*)
by (*metis pass-ATC-io*(*2*) *assms fst-conv*)

lemma *pass-ATC-io-fail-fixed-io* :
assumes *is-ATC A*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and *io*@[*ioA*] ∈ *L A*
and *io*@[*ioM*] ∈ *L M*
and *fst ioA* = *fst ioM*
and *io*@[*ioM*] ∉ *L A* ∨ *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* ≠ {}
shows ¬*pass-ATC M A FS*
proof −
consider (*a*) *io*@[*ioM*] ∉ *L A* |
(*b*) *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩ *FS* ≠ {}
using *assms*(*7*) **by** *blast*
then show *?thesis* **proof** (*cases*)
case *a*
then show *?thesis* **using** *pass-ATC-io*(*1*)[*OF* - *assms*(*1*−*6*)] **by** *blast*
next
case *b*
then show *?thesis* **using** *pass-ATC-io*(*2*)[*OF* - *assms*(*1*−*6*)] **by** *blast*
qed
qed

lemma *pass-ATC′-io-fail* :
assumes ¬*pass-ATC′ M A FS k*
and *is-ATC A*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
shows *initial A* ∈ *FS* ∨ (∃ *io ioA ioM* . *io*@[*ioA*] ∈ *L A*
∧ *io*@[*ioM*] ∈ *L M*
∧ *fst ioA* = *fst ioM*
∧ (*io*@[*ioM*] ∉ *L A* ∨ *io-targets A* (*io*@[*ioM*]) (*initial A*) ∩
FS ≠ {}))
using *assms* **proof** (*induction k arbitrary*: *M A*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
then show *?case* **proof** (*cases initial A* ∈ *FS*)

case *True*
then show *?thesis* **by** *auto*
next
case *False*
then obtain x **where** $x \in$ *inputs A*
and *h A* (*FSM.initial A*, x) $\neq \{\}$
and $\neg$($\forall$ (yM, qM)$\in$*h M* (*initial M*, x). $\exists$ (yA, qA)$\in$*h A* (*FSM.initial A*, x). $yM = yA \wedge$ *pass-ATC′* (*FSM.from-FSM M qM*) (*FSM.from-FSM A qA*) *FS k*)
using *Suc.prems*(*1*) **unfolding** *pass-ATC′.simps*
by *fastforce*

obtain *tM* **where** *tM* $\in$ *transitions M*
and *t-source tM* = *initial M*
and *t-input tM* = x
and $\neg$($\exists$ (yA, qA)$\in$*h A* (*FSM.initial A*, x). *t-output tM* = $yA \wedge$ *pass-ATC′* (*FSM.from-FSM M* (*t-target tM*)) (*FSM.from-FSM A qA*) *FS k*)
using ‹$\neg$($\forall$ (yM, qM)$\in$*h M* (*initial M*, x). $\exists$ (yA, qA)$\in$*h A* (*FSM.initial A*, x). $yM = yA \wedge$ *pass-ATC′* (*FSM.from-FSM M qM*) (*FSM.from-FSM A qA*) *FS k*)›
unfolding *h.simps*
by *auto*

have $\neg$ ($\exists$ *tA* . *tA* $\in$ *transitions A* $\wedge$ *t-source tA* = *initial A* $\wedge$ *t-input tA* = x $\wedge$ *t-output tA* = *t-output tM* $\wedge$ *pass-ATC′* (*FSM.from-FSM M* (*t-target tM*)) (*FSM.from-FSM A* (*t-target tA*)) *FS k*)
using ‹$\neg$($\exists$ (yA, qA)$\in$*h A* (*FSM.initial A*, x). *t-output tM* = $yA \wedge$ *pass-ATC′* (*FSM.from-FSM M* (*t-target tM*)) (*FSM.from-FSM A qA*) *FS k*)›
unfolding *h.simps* **by** *force*
moreover have $\exists$ *tA* . *tA* $\in$ *transitions A* $\wedge$ *t-source tA* = *initial A* $\wedge$ *t-input tA* = x
using ‹*h A* (*FSM.initial A*, x) $\neq \{\}$› **unfolding** *h.simps* **by** *force*
ultimately consider
(*a*) $\bigwedge$ *tA* . *tA* $\in$ *transitions A* $\Longrightarrow$ *t-source tA* = *initial A* $\Longrightarrow$ *t-input tA* = x $\Longrightarrow$ *t-output tM* $\neq$ *t-output tA* |
(*b*) $\exists$ *tA* . *tA* $\in$ *transitions A* $\wedge$ *t-source tA* = *initial A* $\wedge$ *t-input tA* = x $\wedge$ *t-output tA* = *t-output tM* $\wedge$ $\neg$*pass-ATC′* (*FSM.from-FSM M* (*t-target tM*)) (*FSM.from-FSM A* (*t-target tA*)) *FS k*
by *force*
then show *?thesis* **proof** *cases*
case *a*
then have [(x,*t-output tM*)] $\notin$ *L A*
unfolding *LS.simps* **by** *fastforce*
moreover have $\exists$ y . [(x,y)] $\in$ *L A*
using ‹*h A* (*FSM.initial A*, x) $\neq \{\}$› **unfolding** *h.simps LS.simps*
proof −
obtain *pp* :: *′d* $\times$ *′b* $\times$ *′c* $\times$ *′d* **where**
f1: *pp* $\in$ *FSM.transitions A* $\wedge$ *t-source pp* = *FSM.initial A* $\wedge$ *t-input pp* = x

using ‹∃ *tA*. *tA* ∈ *FSM.transitions A* ∧ *t-source tA* = *FSM.initial A* ∧ *t-input tA* = *x*› **by** *blast*
then have *path A* (*FSM.initial A*) [*pp*]
by (*metis single-transition-path*)
then have (*t-input pp*, *t-output pp*) # *p-io* ([]::('*d* × '*b* × '*c* × -) *list*) ∈ {*p-io ps* |*ps*. *path A* (*FSM.initial A*) *ps*}
by *force*
then show ∃ *c*. [(*x*, *c*)] ∈ {*p-io ps* |*ps*. *path A* (*FSM.initial A*) *ps*}
using *f1* **by** *force*
qed
moreover have [(*x*,*t-output tM*)] ∈ *L M*
unfolding *LS.simps* **using** ‹*tM* ∈ *transitions M*› ‹*t-input tM* = *x*› ‹*t-source tM* = *initial M*›
proof −
have ∃ *ps*. *p-io* [*tM*] = *p-io ps* ∧ *path M* (*FSM.initial M*) *ps*
by (*metis* (*no-types*) ‹*tM* ∈ *FSM.transitions M*› ‹*t-source tM* = *FSM.initial M*› *single-transition-path*)
then show [(*x*, *t-output tM*)] ∈ {*p-io ps* |*ps*. *path M* (*FSM.initial M*) *ps*}
by (*simp add*: ‹*t-input tM* = *x*›)
qed
ultimately have (∃ *io ioA ioM*. *io* @ [*ioA*] ∈ *L A* ∧ *io* @ [*ioM*] ∈ *L M* ∧ *fst ioA* = *fst ioM* ∧ (*io* @ [*ioM*] ∉ *L A*))
by (*metis append-self-conv2 fst-conv*)
then show *?thesis* **by** *blast*
next
case *b*
then obtain *t'* **where** *t'* ∈ *transitions A*
and *t-source t'* = *initial A*
and *t-input t'* = *x*
and *t-output t'* = *t-output tM*
and ¬*pass-ATC'* (*FSM.from-FSM M* (*t-target tM*)) (*FSM.from-FSM A* (*t-target t'*)) *FS k*
by *blast*

have *is-ATC* (*FSM.from-FSM A* (*t-target t'*))
using *is-ATC-from*[*OF* ‹*t'* ∈ *transitions A*› - ‹*is-ATC A*›] *reachable-states-initial*
unfolding ‹*t-source t'* = *initial A*› **by** *blast*

have (*inputs* (*from-FSM A* (*t-target t'*))) ⊆ (*inputs* (*from-FSM M* (*t-target tM*)))
by (*simp add*: *Suc.prems*(*4*) ‹*t'* ∈ *FSM.transitions A*› ‹*tM* ∈ *FSM.transitions M*› *fsm-transition-target*)

let *?ioM* = (*x*,*t-output tM*)
let *?ioA* = (*x*,*t-output t'*)

consider (*b1*) *initial* (*from-FSM A* (*t-target t'*)) ∈ *FS* |
(*b2*) (∃ *io ioA ioM*.

io @ [ioA] ∈ LS (from-FSM A (t-target t′)) (initial (from-FSM A (t-target t′))) ∧
io @ [ioM] ∈ LS (from-FSM M (t-target tM)) (initial (from-FSM M (t-target tM))) ∧
fst ioA = fst ioM ∧
(io @ [ioM] ∉ LS (from-FSM A (t-target t′)) (initial (from-FSM A (t-target t′))) ∨
io-targets (from-FSM A (t-target t′)) (io @ [ioM]) (initial (from-FSM A (t-target t′))) ∩ FS ≠ {}))
using Suc.IH[OF ‹¬pass-ATC′ (FSM.from-FSM M (t-target tM)) (FSM.from-FSM A (t-target t′)) FS k›
‹is-ATC (FSM.from-FSM A (t-target t′))›
from-FSM-observable[OF ‹observable M›]
‹(inputs (from-FSM A (t-target t′))) ⊆ (inputs (from-FSM M (t-target tM)))›]
by *blast*
then show *?thesis* **proof** *cases*
case *b1*

have [*?ioA*] ∈ L A
unfolding LS.simps
proof −
have ∃ ps. [(x, t-output t′)] = p-io ps ∧ path A (t-source t′) ps
using ‹t′ ∈ FSM.transitions A› ‹t-input t′ = x› **by** *force*
then show [(x, t-output t′)] ∈ {p-io ps |ps. path A (FSM.initial A) ps}
by (*simp add*: ‹t-source t′ = FSM.initial A›)
qed

have [*?ioM*] ∈ L M
unfolding LS.simps
proof −
have path M (FSM.initial M) [tM]
by (*metis* ‹tM ∈ FSM.transitions M› ‹t-source tM = FSM.initial M› *single-transition-path*)
then have ∃ ps. [(x, t-output tM)] = p-io ps ∧ path M (FSM.initial M) ps
using ‹t-input tM = x› **by** *force*
then show [(x, t-output tM)] ∈ {p-io ps |ps. path M (FSM.initial M) ps}
by *simp*
qed

have p-io [t′] = [(x, t-output tM)]
using ‹t-input t′ = x› ‹t-output t′ = t-output tM›
by *auto*
moreover have target (initial A) [t′] = t-target t′
using ‹t-source t′ = initial A› **by** *auto*
ultimately have t-target t′ ∈ io-targets A [(x,t-output tM)] (initial A)
unfolding io-targets.simps
using single-transition-path[OF ‹t′ ∈ transitions A›]
by (*metis* (*mono-tags*, *lifting*) ‹t-source t′ = initial A› *mem-Collect-eq*)

then have *initial* (*from-FSM A* (*t-target t′*)) ∈ *io-targets A* [(*x*,*t-output tM*)] (*initial A*)
unfolding *io-targets.simps from-FSM-simps*[*OF fsm-transition-target*[*OF* ‹*t′* ∈ *transitions A*›]] **by** *simp*
then have *io-targets A* ([] @ [*?ioM*]) (*initial A*) ∩ *FS* ≠ {}
using *b1* **by** (*metis IntI append-Nil empty-iff*)

then have ∃ *io ioA ioM* . *io*@[*ioA*] ∈ *L A*
∧ *io*@[*ioM*] ∈ *L M*
∧ *fst ioA* = *fst ioM*
∧ *io-targets A* (*io* @ [*ioM*]) (*initial A*) ∩ *FS* ≠ {}
using ‹[*?ioA*] ∈ *L A*› ‹[*?ioM*] ∈ *L M*›
by (*metis* ‹*t-output t′* = *t-output tM*› *append.left-neutral*)

then show *?thesis* **by** *blast*

next
case *b2*

then obtain *io ioA ioM* **where**
io @ [*ioA*] ∈ *LS* (*from-FSM A* (*t-target t′*)) (*initial* (*from-FSM A* (*t-target t′*)))
and *io* @ [*ioM*] ∈ *LS* (*from-FSM M* (*t-target tM*)) (*initial* (*from-FSM M* (*t-target tM*)))
and *fst ioA* = *fst ioM*
and (*io* @ [*ioM*] ∉ *LS* (*from-FSM A* (*t-target t′*)) (*initial* (*from-FSM A* (*t-target t′*))) ∨ *io-targets* (*from-FSM A* (*t-target t′*)) (*io* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target t′*))) ∩ *FS* ≠ {})
by *blast*

have *observable A*
using *Suc.prems*(*2*) *is-ATC-def* **by** *blast*

have (*?ioM* # *io*) @ [*ioA*] ∈ *L A*
using *language-state-prepend-transition*[*OF* ‹*io* @ [*ioA*] ∈ *LS* (*from-FSM A* (*t-target t′*)) (*initial* (*from-FSM A* (*t-target t′*)))› ‹*t′* ∈ *transitions A*›]
using ‹*t-input t′* = *x*› ‹*t-source t′* = *initial A*› ‹*t-output t′* = *t-output tM*›
by *simp*

moreover have (*?ioM* # *io*) @ [*ioM*] ∈ *L M*
using *language-state-prepend-transition*[*OF* ‹*io* @ [*ioM*] ∈ *L* (*from-FSM M* (*t-target tM*))› ‹*tM* ∈ *transitions M*›]
using ‹*t-input tM* = *x*› ‹*t-source tM* = *initial M*›
by *simp*

moreover have ((*?ioM* # *io*) @ [*ioM*] ∉ *L A* ∨ *io-targets A* ((*?ioM* # *io*) @ [*ioM*]) (*initial A*) ∩ *FS* ≠ {})
proof −
consider (*f1*) *io* @ [*ioM*] ∉ *L* (*from-FSM A* (*t-target t′*)) |

(*f2*) *io-targets* (*from-FSM A* (*t-target t'*)) (*io* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target t'*))) ∩ *FS* ≠ {}
using ‹(*io* @ [*ioM*] ∉ *LS* (*from-FSM A* (*t-target t'*)) (*initial* (*from-FSM A* (*t-target t'*))) ∨ *io-targets* (*from-FSM A* (*t-target t'*)) (*io* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target t'*))) ∩ *FS* ≠ {})›
by *blast*
then show *?thesis* **proof** *cases*
case *f1*

have *p-io* [*t'*] = [(*x*, *t-output tM*)]
using ‹*t-input t'* = *x*› ‹*t-output t'* = *t-output tM*›
by *auto*
moreover have *target* (*initial A*) [*t'*] = *t-target t'*
using ‹*t-source t'* = *initial A*› **by** *auto*
ultimately have *t-target t'* ∈ *io-targets A* [*?ioM*] (*initial A*)
unfolding *io-targets.simps*
using *single-transition-path*[*OF* ‹*t'* ∈ *transitions A*›]
by (*metis* (*mono-tags*, *lifting*) ‹*t-source t'* = *initial A*› *mem-Collect-eq*)

show *?thesis*
using *observable-io-targets-language*[*of* [(*x*, *t-output tM*)] *io*@[*ioM*] *A initial A t-target t'*, *OF* - ‹*observable A*› ‹*t-target t'* ∈ *io-targets A* [*?ioM*] (*initial A*)›]
f1
by (*metis* ‹*observable A*› ‹*t'* ∈ *FSM.transitions A*› ‹*t-input t'* = *x*› ‹*t-output t'* = *t-output tM*› ‹*t-source t'* = *FSM.initial A*› *append-Cons fst-conv observable-language-next snd-conv*)

next
case *f2*

have *io-targets A* (*p-io* [*t'*] @ *io* @ [*ioM*]) (*t-source t'*) = *io-targets A* ([*?ioM*] @ *io* @ [*ioM*]) (*t-source t'*)
using ‹*t-input t'* = *x*› ‹*t-output t'* = *t-output tM*› **by** *auto*
moreover have *io-targets A* (*io* @ [*ioM*]) (*t-target t'*) = *io-targets* (*from-FSM A* (*t-target t'*)) (*io* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target t'*)))
unfolding *io-targets.simps*
using *from-FSM-path-initial*[*OF fsm-transition-target*[*OF* ‹*t'* ∈ *transitions A*›]] **by** *auto*
ultimately have *io-targets A* ([*?ioM*] @ *io* @ [*ioM*]) (*t-source t'*) = *io-targets* (*from-FSM A* (*t-target t'*)) (*io* @ [*ioM*]) (*initial* (*from-FSM A* (*t-target t'*)))
using *observable-io-targets-next*[*OF* ‹*observable A*› ‹*t'* ∈ *transitions A*›, *of io* @ [*ioM*]] **by** *auto*

then show *?thesis*

```
            using f2 ‹t-source t′ = initial A› by auto
          qed
        qed

        ultimately show ?thesis
          using ‹fst ioA = fst ioM› by blast
      qed
    qed
  qed
qed


lemma pass-ATC-io-fail :
  assumes ¬pass-ATC M A FS
  and     is-ATC A
  and     observable M
  and     (inputs A) ⊆ (inputs M)
shows initial A ∈ FS ∨ (∃ io ioA ioM . io@[ioA] ∈ L A
                        ∧ io@[ioM] ∈ L M
                        ∧ fst ioA = fst ioM
                         ∧ (io@[ioM] ∉ L A ∨ io-targets A (io@[ioM]) (initial A) ∩
FS ≠ {}))
  using pass-ATC′-io-fail[OF - assms(2−4)] using assms(1) by auto


lemma pass-ATC-fail :
  assumes is-ATC A
  and     observable M
  and     (inputs A) ⊆ (inputs M)
  and     io@[(x,y)] ∈ L A
  and     io@[(x,y′)] ∈ L M
  and     io@[(x,y′)] ∉ L A
shows ¬ pass-ATC M A FS
  using assms(6) pass-ATC-io-explicit-io-tuple(1)[OF - assms(1,2,3,4,5)]
  by blast


lemma pass-ATC-reduction :
  assumes L M2 ⊆ L M1
  and     is-ATC A
  and     observable M1
  and     observable M2
  and     (inputs A) ⊆ (inputs M1)
  and     (inputs M2) = (inputs M1)
  and     pass-ATC M1 A FS
shows pass-ATC M2 A FS
proof (rule ccontr)
  assume ¬ pass-ATC M2 A FS
  have (inputs A) ⊆ (inputs M2)
```

using *assms*(*5*,*6*) **by** *blast*

have *initial A* $\notin$ *FS*
using ‹*pass-ATC M1 A FS*› **by** (*cases size A*; *auto*)
then show *False*
using *pass-ATC-io-fail*[*OF* ‹¬ *pass-ATC M2 A FS*› *assms*(*2*,*4*) ‹(*inputs A*) ⊆ (*inputs M2*)›]
using *assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*5*) *assms*(*7*) *pass-ATC-io-fail-fixed-io*
by *blast*
qed

lemma *pass-ATC-fail-no-reduction* :
assumes *is-ATC A*
and *observable T*
and *observable M*
and (*inputs A*) ⊆ (*inputs M*)
and (*inputs T*) = (*inputs M*)
and *pass-ATC M A FS*
and ¬*pass-ATC T A FS*
shows ¬ (*L T* ⊆ *L M*)
using *pass-ATC-reduction*[*OF* - *assms*(*1*,*3*,*2*,*4*,*5*,*6*)] *assms*(*7*) **by** *blast*

35.2 State Separators as Adaptive Test Cases

fun *pass-separator-ATC* :: (*'a*,*'b*,*'c*) *fsm* ⇒ (*'d*,*'b*,*'c*) *fsm* ⇒ *'a* ⇒ *'d* ⇒ *bool* **where**
pass-separator-ATC M S q1 t2 = *pass-ATC* (*from-FSM M q1*) *S* {*t2*}

lemma *separator-is-ATC* :
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *q1* ∈ *states M*
shows *is-ATC A*
unfolding *is-ATC-def*
using *is-separator-simps*(*1*,*2*,*3*)[*OF assms*(*1*)] **by** *blast*

lemma *pass-separator-ATC-from-separator-left* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
shows *pass-separator-ATC M A q1 t2*
proof (*rule ccontr*)
assume ¬ *pass-separator-ATC M A q1 t2*

then have ¬ *pass-ATC* (*from-FSM M q1*) *A* {*t2*}
by *auto*

```
  have is-ATC A
    using separator-is-ATC[OF assms(4,1,2)] by assumption

  have initial A ∉ {t2}
    using separator-initial(2)[OF assms(4)] by blast
  then obtain io ioA ioM where
    io @ [ioA] ∈ L A
    io @ [ioM] ∈ LS M q1
    fst ioA = fst ioM
    io @ [ioM] ∉ L A ∨ io-targets A (io @ [ioM]) (initial A) ∩ {t2} ≠ {}

    using pass-ATC-io-fail[OF ‹¬ pass-ATC (from-FSM M q1) A {t2}› ‹is-ATC
A› from-FSM-observable[OF ‹observable M›] ]
    using is-separator-simps(16)[OF assms(4)]
    using from-FSM-language[OF ‹q1 ∈ states M›]
    unfolding from-FSM-simps[OF ‹q1 ∈ states M›] by blast
  then obtain x ya ym where
    io @ [(x,ya)] ∈ L A
    io @ [(x,ym)] ∈ LS M q1
    io @ [(x,ym)] ∉ L A ∨ io-targets A (io @ [(x,ym)]) (initial A) ∩ {t2} ≠ {}
    by (metis fst-eqD old.prod.exhaust)

  have io @ [(x,ym)] ∈ L A
     using is-separator-simps(9)[OF assms(4) ‹io @ [(x,ym)] ∈ LS M q1› ‹io @
[(x,ya)] ∈ L A›] by assumption

  have t1 ≠ t2 using is-separator-simps(15)[OF assms(4)] by assumption

  consider (a) io @ [(x, ym)] ∈ LS M q1 − LS M q2 |
           (b) io @ [(x, ym)] ∈ LS M q1 ∩ LS M q2
    using ‹io @ [(x,ym)] ∈ LS M q1› by blast
  then have io-targets A (io @ [(x,ym)]) (initial A) ∩ {t2} = {}

  proof (cases)
    case a
    show ?thesis using separator-language(1)[OF assms(4) ‹io @ [(x,ym)] ∈ L A›
a] ‹t1 ≠ t2› by auto
  next
    case b
    show ?thesis using separator-language(3)[OF assms(4) ‹io @ [(x,ym)] ∈ L A›
b] ‹t1 ≠ t2› by auto
  qed

  then show False
    using ‹io @ [(x,ym)] ∈ L A›
    using ‹io @ [(x,ym)] ∉ L A ∨ io-targets A (io @ [(x,ym)]) (initial A) ∩ {t2}
≠ {}› by blast
qed
```

lemma *pass-separator-ATC-from-separator-right* :
assumes *observable M*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
shows *pass-separator-ATC M A q2 t1*
using *assms*(*1*−*3*) *is-separator-sym*[*OF assms*(*4*)] *pass-separator-ATC-from-separator-left*
by *metis*

lemma *pass-separator-ATC-path-left* :
assumes *pass-separator-ATC S A s1 t2*
and *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *path A* (*initial A*) *pA*
and *path S s1 pS*
and *p-io pA* = *p-io pS*
shows *target* (*initial A*) *pA* ≠ *t2*
and ∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*
proof −

have *pass-ATC* (*from-FSM S s1*) *A* {*t2*}
using ‹*pass-separator-ATC S A s1 t2*› **by** *auto*

have *is-ATC A*
using *separator-is-ATC*[*OF assms*(*7*,*3*,*5*)] **by** *assumption*

have *observable* (*from-FSM S s1*)
using *from-FSM-observable*[*OF assms*(*2*)] **by** *assumption*

have (*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))
using *is-separator-simps*(*16*)[*OF assms*(*7*)] ‹(*inputs S*) = (*inputs M*)›
unfolding *from-FSM-simps*[*OF* ‹*s1* ∈ *states S*›] **by** *blast*

have *target* (*initial A*) *pA* ≠ *t2* ∧ (∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*)
proof (*cases pA rule*: *rev-cases*)
case *Nil*
then have *target* (*initial A*) *pA* ≠ *t2*
using *separator-initial*(*2*)[*OF assms*(*7*)] **by** *auto*
moreover have (∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*)
unfolding *Nil* **using** ‹*q1* ∈ *states M*› **by** *auto*

ultimately show *?thesis* **by** *auto*
next
case (*snoc ys y*)
then have *p-io pA* = (*p-io ys*)@[(*t-input y*,*t-output y*)]
by *auto*
then have ∗: (*p-io ys*)@[(*t-input y*,*t-output y*)] ∈ *L A*
using *language-state-containment*[*OF* ‹*path A* (*initial A*) *pA*›] **by** *blast*
then have *p-io pS* = (*p-io ys*)@[(*t-input y*,*t-output y*)]
using ‹*p-io pA* = (*p-io ys*)@[(*t-input y*,*t-output y*)]› ‹*p-io pA* = *p-io pS*› **by** *auto*
then have ∗∗: (*p-io ys*)@[(*t-input y*,*t-output y*)] ∈ *L* (*from-FSM S s1*)
using *language-state-containment*[*OF* ‹*path S s1 pS*›]
unfolding *from-FSM-language*[*OF* ‹*s1* ∈ *states S*›] **by** *blast*

have *io-targets A* ((*p-io ys*)@[(*t-input y*,*t-output y*)]) (*initial A*) ∩ {*t2*} = {}
using *pass-ATC-io*(*2*)[*OF* ‹*pass-ATC* (*from-FSM S s1*) *A* {*t2*}› ‹*is-ATC A*› ‹*observable* (*from-FSM S s1*)› ‹(*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))› ∗ ∗∗]
unfolding *fst-conv* **by** *auto*
then have *target* (*initial A*) *pA* ≠ *t2*
using ‹*p-io pA* = (*p-io ys*)@[(*t-input y*,*t-output y*)]› ‹*path A* (*initial A*) *pA*›
unfolding *io-targets.simps*
by *blast*

have *p-io ys* @ [(*t-input y*, *t-output y*)] ∈ *LS M q1*
using *separator-language*(*2*,*4*)[*OF assms*(*7*) ‹(*p-io ys*)@[(*t-input y*,*t-output y*)] ∈ *L A*›]
using ‹*io-targets A* ((*p-io ys*)@[(*t-input y*,*t-output y*)]) (*initial A*) ∩ {*t2*} = {}› **by** *blast*
then have ∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*
using ‹*p-io pA* = (*p-io ys*)@[(*t-input y*,*t-output y*)]› **by** *auto*

then show *?thesis* **using** ‹*target* (*initial A*) *pA* ≠ *t2*› **by** *auto*
qed

then show *target* (*initial A*) *pA* ≠ *t2* **and** ∃ *pM* . *path M q1 pM* ∧ *p-io pM* = *p-io pA*
by *blast*+
qed

lemma *pass-separator-ATC-path-right* :
assumes *pass-separator-ATC S A s2 t1*
and *observable S*
and *observable M*
and *s2* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)

and $q1 \neq q2$
and *path A (initial A) pA*
and *path S s2 pS*
and *p-io pA = p-io pS*
shows *target (initial A) pA ≠ t1*
and $\exists$ *pM . path M q2 pM ∧ p-io pM = p-io pA*
using *pass-separator-ATC-path-left*[*OF assms*(*1*−*4*,*6*,*5*) *is-separator-sym*[*OF assms*(*7*)] *assms*(*8*) - *assms*(*10*−*12*)] *assms*(*9*) **by** *blast*+

lemma *pass-separator-ATC-fail-no-reduction* :
assumes *observable S*
and *observable M*
and *s1 ∈ states S*
and *q1 ∈ states M*
and *q2 ∈ states M*
and *is-separator M q1 q2 A t1 t2*
and *(inputs S) = (inputs M)*
and *¬pass-separator-ATC S A s1 t2*
shows *¬ (LS S s1 ⊆ LS M q1)*
proof
assume *LS S s1 ⊆ LS M q1*

have *is-ATC A*
using *separator-is-ATC*[*OF assms*(*6*,*2*,*4*)] **by** *assumption*

have ∗: *(inputs A) ⊆ (inputs (from-FSM M q1))*
using *is-separator-simps*(*16*)[*OF assms*(*6*)]
unfolding *is-submachine.simps canonical-separator-simps from-FSM-simps*[*OF* ‹*q1 ∈ states M*›] **by** *auto*

have *pass-ATC (from-FSM M q1) A {t2}*
using *pass-separator-ATC-from-separator-left*[*OF assms*(*2*,*4*,*5*,*6*)] **by** *auto*

have *¬ pass-ATC (from-FSM S s1) A {t2}*
using ‹*¬pass-separator-ATC S A s1 t2*› **by** *auto*

moreover have *pass-ATC (from-FSM S s1) A {t2}*
using *pass-ATC-reduction*[*OF* - ‹*is-ATC A*› *from-FSM-observable*[*OF* ‹*observable M*›] *from-FSM-observable*[*OF* ‹*observable S*›] ∗]
using ‹*LS S s1 ⊆ LS M q1*› ‹*pass-ATC (from-FSM M q1) A {t2}*›
unfolding *from-FSM-language*[*OF assms*(*3*)] *from-FSM-language*[*OF assms*(*4*)]
using ‹*L (FSM.from-FSM S s1) = LS S s1*› *assms*(*3*) *assms*(*4*) *assms*(*7*) **by** *auto*
ultimately show *False* **by** *simp*
qed

lemma *pass-separator-ATC-pass-left* :

assumes *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS S s1*
and *q1* ≠ *q2*
and *pass-separator-ATC S A s1 t2*
shows *target* (*initial A*) *p* ≠ *t2*
and *target* (*initial A*) *p* = *t1* ∨ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*)
proof −

from ‹*p-io p* ∈ *LS S s1*› **obtain** *pS* **where** *path S s1 pS* **and** *p-io p* = *p-io pS*
by *auto*

then show *target* (*initial A*) *p* ≠ *t2*
using *pass-separator-ATC-path-left*[*OF assms*(*11*,*1*−*7*,*10*,*8*)] **by** *simp*

obtain *pM* **where** *path M q1 pM* **and** *p-io pM* = *p-io p*
using *pass-separator-ATC-path-left*[*OF assms*(*11*,*1*−*7*,*10*,*8*) ‹*path S s1 pS*› ‹*p-io p* = *p-io pS*›] **by** *blast*
then have *p-io p* ∈ *LS M q1*
unfolding *LS.simps* **by** *force*

then show *target* (*initial A*) *p* = *t1* ∨ (*target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*)
using *separator-path-targets*(*1*,*3*,*4*)[*OF assms*(*6*,*8*)] **by** *blast*
qed

lemma *pass-separator-ATC-pass-right* :
assumes *observable S*
and *observable M*
and *s2* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *path A* (*initial A*) *p*
and *p-io p* ∈ *LS S s2*
and *q1* ≠ *q2*
and *pass-separator-ATC S A s2 t1*
shows *target* (*initial A*) *p* ≠ *t1*
and *target* (*initial A*) *p* = *t2* ∨ (*target* (*initial A*) *p* ≠ *t2* ∧ *target* (*initial A*) *p* ≠ *t2*)

using *pass-separator-ATC-pass-left*[*OF assms*(*1*,*2*,*3*,*5*,*4*) *is-separator-sym*[*OF assms*(*6*)] *assms*(*7*−*9*) - *assms*(*11*)] *assms*(*10*) **by** *blast*+

lemma *pass-separator-ATC-completely-specified-left* :
assumes *observable S*
and *observable M*
and *s1* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *pass-separator-ATC S A s1 t2*
and *completely-specified S*
shows ∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) *p* = *t1*
and ¬ (∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) *p* = *t2*)
proof −
have *p1*: *pass-ATC* (*from-FSM S s1*) *A* {*t2*}
using *assms*(*9*) **by** *auto*

have *p2*: *is-ATC A*
using *separator-is-ATC*[*OF assms*(*6*,*2*,*4*)] **by** *assumption*

have *p3*: *observable* (*from-FSM S s1*)
using *from-FSM-observable*[*OF assms*(*1*)] **by** *assumption*

have *p4*: (*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))
using *is-separator-simps*(*16*)[*OF assms*(*6*)]
unfolding *from-FSM-simps*[*OF* ‹*s1* ∈ *states S*›] *is-submachine.simps canonical-separator-simps assms*(*7*) **by** *auto*

have *t1* ≠ *t2* **and** *observable A*
using *is-separator-simps*(*15*,*3*)[*OF assms*(*6*)] **by** *linarith*+

have *path-ext*: ⋀ *p* . *path A* (*initial A*) *p* ⟹ *p-io p* ∈ *LS S s1* ⟹ (*target* (*initial A*) *p* ≠ *t2*) ∧ (*target* (*initial A*) *p* = *t1* ∨ (∃ *t* . *path A* (*initial A*) (*p*@[*t*]) ∧ *p-io* (*p*@[*t*]) ∈ *LS S s1*))
proof −
fix *p* **assume** *path A* (*initial A*) *p* **and** *p-io p* ∈ *LS S s1*

consider (*a*) *target* (*initial A*) *p* = *t1* |
(*b*) *target* (*initial A*) *p* ≠ *t1* ∧ *target* (*initial A*) *p* ≠ *t2*
using *pass-separator-ATC-pass-left*(*2*)[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*) ‹*path A* (*initial A*) *p*› ‹*p-io p* ∈ *LS S s1*› *assms*(*8*,*9*)] **by** *blast*
then have *target* (*initial A*) *p* = *t1* ∨ (∃ *t* . *path A* (*initial A*) (*p*@[*t*]) ∧ *p-io* (*p*@[*t*]) ∈ *LS S s1*)

```
  proof cases
    case a
    then show ?thesis by blast
  next
    case b

    let ?t3 = target (initial A) p
    have ?t3 ≠ t1 and ?t3 ≠ t2
      using b by auto
    moreover have ?t3 ∈ reachable-states A
      using ‹path A (initial A) p› reachable-states-intro by blast
    ultimately have ¬ deadlock-state A ?t3
      using is-separator-simps(8)[OF assms(6)] by blast
    then obtain tt where tt ∈ transitions A and t-source tt = ?t3
      by auto

    then have path A (initial A) (p@[tt])
      using ‹path A (initial A) p› using path-append-transition by metis

    moreover have p-io (p@[tt]) = (p-io p)@[(t-input tt, t-output tt)]
      by auto

    ultimately have (p-io p)@[(t-input tt,t-output tt)] ∈ L A
      using language-state-containment[of A initial A p@[tt]] by metis

    let ?x = t-input tt
    have ?x ∈ (inputs S)
       using ‹tt ∈ transitions A› is-separator-simps(16)[OF assms(6)] assms(7)
by auto
    then obtain y where (p-io p)@[(?x,y)] ∈ LS S s1
       using completely-specified-language-extension[OF ‹completely-specified S›
‹s1 ∈ states S› ‹p-io p ∈ LS S s1› ] by auto

    then have p-io p @ [(?x, y)] ∈ LS A (initial A)
       using pass-ATC-io-explicit-io-tuple(1)[OF p1 p2 p3 p4 ‹(p-io p)@[(t-input
tt,t-output tt)] ∈ L A›]
       unfolding from-FSM-language[OF ‹s1 ∈ states S›] by auto

    then obtain tt' where path A (initial A) (p@[tt']) and t-input tt' = ?x and
t-output tt' = y
       using language-path-append-transition-observable[OF - ‹path A (initial A)
p› ‹observable A›] by blast

    then have p-io (p @ [tt']) ∈ LS S s1
      using ‹(p-io p)@[(?x,y)] ∈ LS S s1› by auto
    then show ?thesis
      using ‹path A (initial A) (p@[tt'])› by meson
  qed
```

moreover have *target* (*initial A*) $p \neq t2$
using *pass-separator-ATC-pass-left*(*1*)[*OF assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*) ‹*path A* (*initial A*) *p*› ‹*p-io p* ∈ *LS S s1*› *assms*(*8*,*9*)] **by** *assumption*

ultimately show (*target* (*initial A*) $p \neq t2$) ∧ (*target* (*initial A*) $p = t1$ ∨ (∃ *t* . *path A* (*initial A*) (*p*@[*t*]) ∧ *p-io* (*p*@[*t*]) ∈ *LS S s1*))
by *simp*
qed

have *acyclic A*
using ‹*is-ATC A*› *is-ATC-def* **by** *auto*
then have *finite* {*p* . *path A* (*initial A*) *p*}
using *acyclic-paths-finite*[*of A initial A*] **unfolding** *acyclic.simps*
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
then have *finite* {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*}
by *auto*

have [] ∈ {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*}
using ‹*s1* ∈ *states S*› **by** *auto*
then have {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*} ≠ {}
by *blast*

have *scheme*: ⋀ *S* . *finite S* ⟹ *S* ≠ {} ⟹ ∃ *x* ∈ *S* . ∀ *y* ∈ *S* . *length y* ≤ *length x*
by (*meson leI max-length-elem*)

obtain *p* **where** *p* ∈ {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*} **and** ⋀ *p′* . *p′* ∈ {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*} ⟹ *length p′* ≤ *length p*
using *scheme*[*OF* ‹*finite* {*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*}› ‹{*p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1*} ≠ {}›]
by *blast*
then have *path A* (*initial A*) *p* **and** *p-io p* ∈ *LS S s1* **and** ⋀ *p′* . *path A* (*initial A*) *p′* ⟹ *p-io p′* ∈ *LS S s1* ⟹ *length p′* ≤ *length p*
by *blast*+

have *target* (*initial A*) $p = t1$
using *path-ext*[*OF* ‹*path A* (*initial A*) *p*› ‹*p-io p* ∈ *LS S s1*›] ‹⋀ *p′* . *path A* (*initial A*) *p′* ⟹ *p-io p′* ∈ *LS S s1* ⟹ *length p′* ≤ *length p*›
by (*metis* (*no-types*, *lifting*) *Suc-n-not-le-n length-append-singleton*)

then show ∃ *p*. *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) $p = t1$
using ‹*path A* (*initial A*) *p*› ‹*p-io p* ∈ *LS S s1*› **by** *blast*

show ∄ *p*. *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) $p = t2$
using *path-ext* **by** *blast*

qed

lemma *pass-separator-ATC-completely-specified-right* :
assumes *observable S*
and *observable M*
and *s2* ∈ *states S*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *is-separator M q1 q2 A t1 t2*
and (*inputs S*) = (*inputs M*)
and *q1* ≠ *q2*
and *pass-separator-ATC S A s2 t1*
and *completely-specified S*
shows ∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s2* ∧ *target* (*initial A*) *p* = *t2*
and ¬ (∃ *p* . *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s2* ∧ *target* (*initial A*) *p* = *t1*)
using *pass-separator-ATC-completely-specified-left*[*OF assms*(*1*,*2*,*3*,*5*,*4*) *is-separator-sym*[*OF assms*(*6*)] *assms*(*7*) - *assms*(*9*,*10*)] *assms*(*8*) **by** *blast*+

lemma *pass-separator-ATC-reduction-distinction* :
assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *pass-separator-ATC S A s1 t2*
and *pass-separator-ATC S A s2 t1*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s1* ∈ *states S*
and *s2* ∈ *states S*
and *is-separator M q1 q2 A t1 t2*
and *completely-specified S*
shows *s1* ≠ *s2*
proof −

have ∃*p*. *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s1* ∧ *target* (*initial A*) *p* = *t1*
using *pass-separator-ATC-completely-specified-left*[*OF assms*(*2*,*1*,*9*,*6*,*7*,*11*,*3*,*8*,*4*,*12*)]
by *blast*

moreover have ¬ (∃*p*. *path A* (*initial A*) *p* ∧ *p-io p* ∈ *LS S s2* ∧ *target* (*initial A*) *p* = *t1*)
using *pass-separator-ATC-completely-specified-right*[*OF assms*(*2*,*1*,*10*,*6*,*7*,*11*,*3*,*8*,*5*,*12*)]
by *blast*

ultimately show *s1* ≠ *s2* **by** *blast*
qed

lemma *pass-separator-ATC-failure-left* :
assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *is-separator M q1 q2 A t1 t2*
and ¬ *pass-separator-ATC S A s1 t2*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s1* ∈ *states S*
shows *LS S s1* − *LS M q1* ≠ {}
proof −

have *p1*: *is-ATC A*
using *separator-is-ATC*[*OF assms*(*4*,*1*,*6*)] **by** *assumption*

have *p2*: *observable* (*from-FSM S s1*)
using *from-FSM-observable*[*OF assms*(*2*)] **by** *assumption*

have *p3*: *observable* (*from-FSM M q1*)
using *from-FSM-observable*[*OF assms*(*1*)] **by** *assumption*

have *p4*: (*inputs A*) ⊆ (*inputs* (*from-FSM M q1*))
using *is-separator-simps*(*16*)[*OF assms*(*4*)]
unfolding *from-FSM-simps*[*OF* ‹*q1* ∈ *states M*›] *is-submachine.simps canonical-separator-simps assms*(*3*) **by** *auto*

have *p5*: (*inputs* (*from-FSM S s1*)) = (*inputs* (*from-FSM M q1*))
using *assms*(*3*,*6*,*9*) **by** *simp*

have *p6*: *pass-ATC* (*from-FSM M q1*) *A* {*t2*}
using *pass-separator-ATC-from-separator-left*[*OF assms*(*1*,*6*,*7*,*4*)] **by** *auto*

have *p7*: ¬ *pass-ATC* (*from-FSM S s1*) *A* {*t2*}
using *assms*(*5*) **by** *auto*

show *?thesis*
using *pass-ATC-fail-no-reduction*[*OF p1 p2 p3 p4 p5 p6 p7*]
unfolding *from-FSM-language*[*OF* ‹*q1* ∈ *states M*›] *from-FSM-language*[*OF* ‹*s1* ∈ *states S*›] **by** *blast*
qed

lemma *pass-separator-ATC-failure-right* :
assumes *observable M*
and *observable S*
and (*inputs S*) = (*inputs M*)
and *is-separator M q1 q2 A t1 t2*
and ¬ *pass-separator-ATC S A s2 t1*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and *q1* ≠ *q2*
and *s2* ∈ *states S*
shows *LS S s2* − *LS M q2* ≠ {}
using *pass-separator-ATC-failure-left*[*OF assms*(*1*−*3*) *is-separator-sym*[*OF assms*(*4*)] *assms*(*5*,*7*,*6*) - *assms*(*9*)] *assms*(*8*) **by** *blast*

35.3 ATCs Represented as Sets of IO Sequences

fun *atc-to-io-set* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*d*,'*b*,'*c*) *fsm* ⇒ ('*b* × '*c*) *list set* **where**
atc-to-io-set M A = *L M* ∩ *L A*

lemma *atc-to-io-set-code* :
assumes *acyclic A*
shows *atc-to-io-set M A* = *acyclic-language-intersection M A*
using *acyclic-language-intersection-completeness*[*OF assms*] **unfolding** *atc-to-io-set.simps* **by** *blast*

lemma *pass-io-set-from-pass-separator* :
assumes *is-separator M q1 q2 A t1 t2*
and *pass-separator-ATC S A s1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
shows *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
proof (*rule ccontr*)
assume ¬ *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
then obtain *io x y y'* **where** *io*@[(*x*,*y*)] ∈ (*atc-to-io-set* (*from-FSM M q1*) *A*) **and** *io*@[(*x*,*y'*)] ∈ *L* (*from-FSM S s1*) **and** *io*@[(*x*,*y'*)] ∉ (*atc-to-io-set* (*from-FSM M q1*) *A*)
unfolding *pass-io-set-def* **by** *blast*

have *is-ATC A*
using *separator-is-ATC*[*OF assms*(*1*,*3*,*5*)] **by** *assumption*
then have *acyclic A*
unfolding *is-ATC-def* **by** *auto*
have *observable* (*from-FSM S s1*)

using *from-FSM-observable*[*OF* ‹*observable S*›] **by** *assumption*
have (*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))
by (*metis* (*no-types*) *assms*(*1*) *assms*(*6*) *assms*(*7*) *from-FSM-simps*(*2*) *is-separator-simps*(*16*))

obtain y'' **where** *io* @ [(*x*, y'')] ∈ *LS A* (*initial A*)
using ‹*io*@[(*x*,*y*)] ∈ (*atc-to-io-set* (*from-FSM M q1*) *A*)› **unfolding** *atc-to-io-set.simps*
by *blast*

have *pass-ATC* (*from-FSM S s1*) *A* {*t2*}
using ‹*pass-separator-ATC S A s1 t2*› **by** *auto*

then have *io* @ [(*x*, y')] ∈ *L A*
using *pass-ATC-fail*[*OF* ‹*is-ATC A*›
‹*observable* (*from-FSM S s1*)›
‹(*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))›
‹*io* @ [(*x*, y'')] ∈ *LS A* (*initial A*)›
‹*io*@[(*x*,y')] ∈ *L* (*from-FSM S s1*)›,
of {*t2*}]
by *auto*

have *io-targets A* (*io* @ [(*x*, y')]) (*initial A*) ∩ {*t2*} = {}
using *pass-ATC-io*(*2*)[*OF* ‹*pass-ATC* (*from-FSM S s1*) *A* {*t2*}› ‹*is-ATC A*› ‹*observable* (*from-FSM S s1*)› ‹(*inputs A*) ⊆ (*inputs* (*from-FSM S s1*))› ‹*io* @ [(*x*, y')] ∈ *L A*› ‹*io*@[(*x*,y')] ∈ *L* (*from-FSM S s1*)›]
unfolding *fst-conv* **by** *blast*

then have *io* @ [(*x*, y')] ∈ *LS M q1*
using *separator-language*(*1*,*3*,*4*)[*OF assms*(*1*) ‹*io* @ [(*x*, y')] ∈ *L A*›]
by (*metis UnE Un-Diff-cancel* ‹*io* @ [(*x*, y')] ∈ *LS A* (*initial A*)› *assms*(*1*) *disjoint-insert*(*2*) *is-separator-sym separator-language*(*1*) *singletonI*)
then show *False*
using ‹*io* @ [(*x*, y')] ∈ *L A*› ‹*io*@[(*x*,y')] ∉ (*atc-to-io-set* (*from-FSM M q1*) *A*)›

unfolding *atc-to-io-set.simps from-FSM-language*[*OF* ‹*q1* ∈ *states M*›]
by *blast*
qed

lemma *separator-language-last-left* :
assumes *is-separator M q1 q2 A t1 t2*
and *completely-specified M*
and *q1* ∈ *states M*
and *io* @ [(*x*, *y*)] ∈ *L A*
obtains y'' **where** *io*@[(*x*,y'')] ∈ *L A* ∩ *LS M q1*
proof −
obtain *p t* **where** *path A* (*initial A*) (*p*@[*t*]) **and** *p-io* (*p*@[*t*]) = *io*@[(*x*,*y*)]
using *language-initial-path-append-transition*[*OF* ‹*io* @ [(*x*, *y*)] ∈ *L A*›] **by** *blast*
then have ¬ *deadlock-state A* (*target* (*initial A*) *p*)
unfolding *deadlock-state.simps* **by** *fastforce*

have *path A* (*initial A*) *p*
using ‹*path A* (*initial A*) (*p*@[*t*])› **by** *auto*

have *p-io p* ∈ *LS M q1*
using *separator-path-targets*(*1*,*2*,*4*)[*OF assms*(*1*) ‹*path A* (*initial A*) *p*›]
using *is-separator-simps*(*4*,*5*)[*OF assms*(*1*)]
using ‹¬ *deadlock-state A* (*target* (*initial A*) *p*)› **by** *fastforce*
then have *io* ∈ *LS M q1*
using ‹*p-io* (*p*@[*t*]) = *io*@[(*x*,*y*)]› **by** *auto*

have *x* ∈ (*inputs A*)
using ‹*io* @ [(*x*, *y*)] ∈ *L A*› *language-io*(*1*)
by (*metis in-set-conv-decomp*)
then have *x* ∈ (*inputs M*)
using *is-separator-simps*(*16*)[*OF assms*(*1*)] **by** *blast*

then obtain *y''* **where** *io*@[(*x*,*y''*)] ∈ *LS M q1*
using *completely-specified-language-extension*[*OF* ‹*completely-specified M*› ‹*q1* ∈ *states M*› ‹*io* ∈ *LS M q1*›] **by** *blast*
then have *io*@[(*x*,*y''*)] ∈ *L A* ∩ *LS M q1*
using *is-separator-simps*(*9*)[*OF assms*(*1*) - ‹*io* @ [(*x*, *y*)] ∈ *L A*›] **by** *blast*
then show *?thesis*
using *that* **by** *blast*
qed

lemma *separator-language-last-right* :
assumes *is-separator M q1 q2 A t1 t2*
and *completely-specified M*
and *q2* ∈ *states M*
and *io* @ [(*x*, *y*)] ∈ *L A*
obtains *y''* **where** *io*@[(*x*,*y''*)] ∈ *L A* ∩ *LS M q2*
using *separator-language-last-left*[*OF is-separator-sym*[*OF assms*(*1*)] *assms*(*2*,*3*,*4*)]
by *blast*

lemma *pass-separator-from-pass-io-set* :
assumes *is-separator M q1 q2 A t1 t2*
and *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2*
proof (*rule ccontr*)
assume ¬ *pass-separator-ATC S A s1 t2*
then have ¬ *pass-ATC* (*from-FSM S s1*) *A* {*t2*} **by** *auto*

```
  have is-ATC A
    using separator-is-ATC[OF assms(1,3,5)]  by assumption
  then have acyclic A
    unfolding is-ATC-def by auto
  have observable (from-FSM S s1)
    using from-FSM-observable[OF ‹observable S›] by assumption
  have (inputs A) ⊆ (inputs (from-FSM S s1))
    using assms(1) assms(6) assms(7) is-separator-simps(16) by fastforce

  obtain io x y y′ where io @ [(x,y)] ∈ L A
                         io @ [(x,y′)] ∈ L (from-FSM S s1)
                         (io @ [(x,y′)] ∉ L A ∨ io-targets A (io @ [(x,y′)]) (initial A) ∩
{t2} ≠ {})
     using pass-ATC-io-fail[OF ‹¬ pass-ATC (from-FSM S s1) A {t2}› ‹is-ATC
A› ‹observable (from-FSM S s1)› ‹(inputs A) ⊆ (inputs (from-FSM S s1))›]
    using separator-initial(2)[OF assms(1)]
    using prod.exhaust fst-conv
    by (metis empty-iff insert-iff)

  show False
  proof (cases io-targets A (io @ [(x,y′)]) (initial A) ∩ {t2} ≠ {})
    case True
    then have io @ [(x,y′)] ∈ L A
      unfolding io-targets.simps LS.simps by force

    have io @ [(x,y′)] ∈ LS M q2 − LS M q1
    proof −
      have t2 ≠ t1
       by (metis (full-types) ‹is-separator M q1 q2 A t1 t2› is-separator-simps(15))
      then show ?thesis
        using True separator-language[OF assms(1) ‹io @ [(x,y′)] ∈ L A›]
        by blast
    qed
    then have io @ [(x,y′)] ∉ LS M q1 by blast

    obtain y″ where io @ [(x, y″)] ∈ LS M q1 ∩ L A
      using separator-language-last-left[OF assms(1,8,5) ‹io @ [(x,y)] ∈ L A›] by
blast
    then have io @ [(x, y′)] ∈ LS M q1 ∩ LS A (initial A)
      using ‹pass-io-set (from-FSM S s1) (atc-to-io-set (from-FSM M q1) A)›
      using ‹io @ [(x,y′)] ∈ L (from-FSM S s1)›
       unfolding pass-io-set-def atc-to-io-set.simps from-FSM-language[OF ‹q1 ∈
states M›] by blast

    then show False
      using ‹io @ [(x,y′)] ∉ LS M q1› by blast

  next
```

case *False*
then have *io* @ [(*x*,*y*′)] ∉ *L A*
using ‹(*io* @ [(*x*,*y*′)] ∉ *L A* ∨ *io-targets A* (*io* @ [(*x*,*y*′)]) (*initial A*) ∩ {*t2*} ≠ {})›
by *blast*

obtain *y*′′ **where** *io* @ [(*x*, *y*′′)] ∈ *LS M q1* ∩ *L A*
using *separator-language-last-left*[*OF assms*(*1*,*8*,*5*) ‹*io* @ [(*x*,*y*)] ∈ *L A*›] **by** *blast*
then have *io* @ [(*x*, *y*′)] ∈ *L A*
using ‹*pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)›
using ‹*io* @ [(*x*,*y*′)] ∈ *L* (*from-FSM S s1*)›
unfolding *pass-io-set-def atc-to-io-set.simps from-FSM-language*[*OF* ‹*q1* ∈ *states M*›] **by** *blast*

then show *False*
using ‹*io* @ [(*x*,*y*′)] ∉ *L A*› **by** *blast*
qed
qed

lemma *pass-separator-pass-io-set-iff*:
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2* ⟷ *pass-io-set* (*from-FSM S s1*) (*atc-to-io-set* (*from-FSM M q1*) *A*)
using *pass-separator-from-pass-io-set*[*OF assms*(*1*) - *assms*(*2*−*7*)]
pass-io-set-from-pass-separator[*OF assms*(*1*) - *assms*(*2*−*6*)] **by** *blast*

lemma *pass-separator-pass-io-set-maximal-iff*:
assumes *is-separator M q1 q2 A t1 t2*
and *observable M*
and *observable S*
and *q1* ∈ *states M*
and *s1* ∈ *states S*
and (*inputs S*) = (*inputs M*)
and *completely-specified M*
shows *pass-separator-ATC S A s1 t2* ⟷ *pass-io-set-maximal* (*from-FSM S s1*) (*remove-proper-prefixes* (*atc-to-io-set* (*from-FSM M q1*) *A*))
proof −

have *is-ATC A*
using *separator-is-ATC*[*OF assms*(*1*,*2*,*4*)] **by** *assumption*

then have *acyclic A*
unfolding *is-ATC-def* **by** *auto*
then have *finite* (*L A*)
unfolding *acyclic-alt-def* **by** *assumption*
then have ∗: *finite* (*atc-to-io-set* (*from-FSM M q1*) *A*)
unfolding *atc-to-io-set.simps* **by** *blast*

have ∗∗: $\bigwedge io'\ io''.\ io'$ @ $io'' \in$ *atc-to-io-set* (*from-FSM M q1*) *A* $\Longrightarrow io' \in$ *atc-to-io-set* (*from-FSM M q1*) *A*
unfolding *atc-to-io-set.simps*
using *language-prefix*[*of - - from-FSM M q1 initial* (*from-FSM M q1*)]
using *language-prefix*[*of - - A initial A*] **by** *blast*

show *?thesis*
unfolding *pass-separator-pass-io-set-iff*[*OF assms*] *remove-proper-prefixes-def*
using *pass-io-set-maximal-from-pass-io-set*[*of* (*atc-to-io-set* (*from-FSM M q1*) *A*) (*from-FSM S s1*), *OF* ∗] ∗∗ **by** *blast*
qed

end

36 State Preambles

This theory defines state preambles. A state preamble P of some state q of some FSM M is an acyclic single-input submachine of M that contains for each of its states and defined inputs in that state all transitions of M and has q as its only deadlock state. That is, P represents a strategy of reaching q in every complete submachine of M. In testing, preambles are used to reach states in the SUT that must conform to a single known state in the specification.

theory *State-Preamble*
imports *../Product-FSM Backwards-Reachability-Analysis*
begin

definition *is-preamble* :: ('*a*,'*b*,'*c*) *fsm* ⇒ ('*a*,'*b*,'*c*) *fsm* ⇒ '*a* ⇒ *bool* **where**
is-preamble S M q =
(*acyclic S*
∧ *single-input S*
∧ *is-submachine S M*
∧ *q* ∈ *reachable-states S*
∧ *deadlock-state S q*
∧ (∀ *q'* ∈ *reachable-states S* .
(*q* = *q'* ∨ ¬ *deadlock-state S q'*) ∧
(∀ *x* ∈ *inputs M* .

$(\exists\ t \in \textit{transitions } S\ .\ \textit{t-source } t = q' \wedge \textit{t-input } t = x)$
$\longrightarrow (\forall\ t' \in \textit{transitions } M\ .\ \textit{t-source } t' = q' \wedge \textit{t-input } t' = x \longrightarrow t' \in$
$\textit{transitions } S))))$

fun *definitely-reachable* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ *bool* **where**
definitely-reachable M q = ($\exists$ *S* . *is-preamble S M q*)

36.1 Basic Properties

lift-definition *initial-preamble* :: $('a,'b,'c)$ *fsm* $\Rightarrow$ $('a,'b,'c)$ *fsm* **is** *FSM-Impl.initial-singleton*

by *auto*

lemma *initial-preamble-simps*[*simp*] :
initial (*initial-preamble M*) = *initial M*
states (*initial-preamble M*) = {*initial M*}
inputs (*initial-preamble M*) = *inputs M*
outputs (*initial-preamble M*) = *outputs M*
transitions (*initial-preamble M*) = {}
by (*transfer*; *auto*)+

lemma *is-preamble-initial* :
is-preamble (*initial-preamble M*) *M* (*initial M*)
proof −
have *acyclic* (*initial-preamble M*)
by (*metis acyclic-code empty-iff initial-preamble-simps*(*5*))
moreover have *single-input* (*initial-preamble M*)
by *auto*
moreover have *is-submachine* (*initial-preamble M*) *M*
by (*simp add*: *fsm-initial*)
moreover have (*initial M*) $\in$ *reachable-states* (*initial-preamble M*)
unfolding *reachable-states-def* **using** *reachable-states-intro* **by** *auto*
moreover have *deadlock-state* (*initial-preamble M*) (*initial M*)
by *simp*
ultimately show *?thesis*
unfolding *is-preamble-def*
using *reachable-state-is-state* **by** *force*
qed

lemma *is-preamble-next* :
assumes *is-preamble S M q*
and $q \neq$ *initial M*
and $t \in$ *transitions S*
and *t-source t* = *initial M*
shows *is-preamble* (*from-FSM S* (*t-target t*)) (*from-FSM M* (*t-target t*)) *q*

(**is** *is-preamble ?S ?M q*)
proof −

have *acyclic S*
and *single-input S*
and *is-submachine S M*
and $q \in$ *reachable-states S*
and *deadlock-state S q*
and $*$: ($\forall$ $q' \in$ *reachable-states S* . ($q = q' \vee \neg$ *deadlock-state S* q')
$\wedge$ ($\forall$ $x \in$ *inputs M* . ($\exists$ $t \in$ *transitions S* . *t-source t* $= q' \wedge$ *t-input t* $=$ x)
$\longrightarrow$ ($\forall$ $t' \in$ *transitions M* . *t-source* $t' = q' \wedge$ *t-input* t' $= x$
$\longrightarrow$ $t' \in$ *transitions S*)))
using *assms*(*1*) **unfolding** *is-preamble-def* **by** *linarith+*

have *t-target t* $\in$ *states S*
using *assms*(*3*) *fsm-transition-target* **by** *metis*
have *t-target t* $\in$ *states M*
using ‹*is-submachine S M*› ‹*t-target t* $\in$ *FSM.states S*› **by** *auto*

have *is-acyclic*: *acyclic ?S*
using *from-FSM-path-initial*[*OF* ‹*t-target t* $\in$ *states S*›]
unfolding *acyclic.simps from-FSM-simps*[*OF* ‹*t-target t* $\in$ *states S*›]
using *acyclic-paths-from-reachable-states*[*OF* ‹*acyclic S*›, *of* [*t*] *t-target t*]
by (*metis* ‹*is-submachine S M*› *assms*(*3*) *assms*(*4*) *is-submachine.elims*(*2*)
prod.collapse single-transition-path target-single-transition)

have *is-single-input*: *single-input ?S*
using ‹*single-input S*›
unfolding *single-input.simps from-FSM-simps*[*OF* ‹*t-target t* $\in$ *states S*›] **by** *blast*

have *initial ?S* = *initial ?M*
by (*simp add*: ‹*t-target t* $\in$ *FSM.states M*› ‹*t-target t* $\in$ *FSM.states S*›)
moreover have *inputs ?S* = *inputs ?M*
using ‹*is-submachine S M*› **by** (*simp add*: ‹*t-target t* $\in$ *FSM.states M*› ‹*t-target t* $\in$ *FSM.states S*›)
moreover have *outputs ?S* = *outputs ?M*
using ‹*is-submachine S M*› **by** (*simp add*: ‹*t-target t* $\in$ *FSM.states M*› ‹*t-target t* $\in$ *FSM.states S*›)
moreover have *transitions ?S* $\subseteq$ *transitions ?M*
using ‹*is-submachine S M*›
by (*simp add*: ‹*t-target t* $\in$ *FSM.states M*› ‹*t-target t* $\in$ *FSM.states S*›)
ultimately have *is-sub* : *is-submachine ?S ?M*
using ‹*is-submachine S M*› ‹*t-target t* $\in$ *FSM.states M*› ‹*t-target t* $\in$ *FSM.states S*› **by** *auto*

have *contains-q* : $q \in$ *reachable-states ?S*
proof −
obtain *qd* **where** $qd \in$ *reachable-states ?S* **and** *deadlock-state ?S qd*
using *is-acyclic*
using *acyclic-deadlock-reachable* **by** *blast*

have $qd \in$ *reachable-states S*
by (*metis* (*no-types*, *lifting*) ‹*is-submachine S M*› ‹$qd \in$ *reachable-states* (*FSM.from-FSM S* (*t-target t*))›
assms(*3*) *assms*(*4*) *from-FSM-reachable-states in-mono is-submachine.elims*(*2*) *prod.collapse*
reachable-states-intro single-transition-path target-single-transition)
then have *deadlock-state S qd*
using ‹*deadlock-state ?S qd*› **unfolding** *deadlock-state.simps*
by (*simp add*: ‹*t-target t* $\in$ *FSM.states S*›)
then have $qd = q$
using $*$ ‹$qd \in$ *reachable-states S*›
by *fastforce*
then show *?thesis*
using ‹$qd \in$ *reachable-states ?S*› **by** *auto*
qed

have *has-deadlock-q* : *deadlock-state ?S q*
using $*$
by (*metis* ‹*deadlock-state S q*› ‹*t-target t* $\in$ *FSM.states S*› *deadlock-state.simps* *from-FSM-simps*(*4*))

have *has-states-prop-1*: $\bigwedge q' . q' \in$ *reachable-states ?S* $\Longrightarrow$ *deadlock-state ?S* q' $\Longrightarrow q = q'$
proof −
fix q' **assume** $q' \in$ *reachable-states ?S* **and** *deadlock-state ?S* q'
have $q' \in$ *reachable-states S*
by (*metis* (*no-types*, *lifting*) ‹*is-submachine S M*› ‹$q' \in$ *reachable-states* (*FSM.from-FSM S* (*t-target t*))›
assms(*3*) *assms*(*4*) *from-FSM-reachable-states in-mono is-submachine.elims*(*2*) *prod.collapse*
reachable-states-intro single-transition-path target-single-transition)
then have *deadlock-state S* q'
using ‹*deadlock-state ?S* q'› **unfolding** *deadlock-state.simps*
using ‹$q' \in$ *reachable-states ?S*› **by** (*simp add*: ‹*t-target t* $\in$ *FSM.states S*›)
then show $q = q'$
using $*$ ‹$q' \in$ *reachable-states S*› **by** *fastforce*
qed

moreover have *has-states-prop-2*: $\bigwedge x\ t\ t'\ q'$.
$q' \in$ *reachable-states ?S* $\Longrightarrow$
$t \in$ *transitions ?S* $\Longrightarrow$ *t-source* $t = q' \Longrightarrow$ *t-input* $t = x \Longrightarrow$

$t' \in$ *transitions ?M* $\Longrightarrow$ *t-source* $t' = q' \Longrightarrow$ *t-input* $t' = x \Longrightarrow t' \in$ *transitions ?S*
proof −
fix x tS tM q' **assume** $q' \in$ *reachable-states ?S* **and** $tS \in$ *transitions ?S* **and** *t-source* $tS = q'$
and *t-input* $tS = x$ **and** $tM \in$ *transitions ?M* **and** *t-source* $tM = q'$
and *t-input* $tM = x$

have $q' \in$ *reachable-states S*
by (*metis* (*no-types*, *lifting*) ‹*is-submachine S M*› ‹$q' \in$ *reachable-states* (*FSM.from-FSM S* (*t-target t*))›
assms(*3*) *assms*(*4*) *from-FSM-reachable-states in-mono is-submachine.elims*(*2*) *prod.collapse*
reachable-states-intro single-transition-path target-single-transition)

have $tS \in$ *transitions S*
using ‹$tS \in$ *transitions ?S*› **by** (*simp add*: ‹*t-target t* $\in$ *FSM.states S*›)
have $tM \in$ *transitions M*
using ‹$tM \in$ *transitions ?M*›
using ‹*t-target t* $\in$ *FSM.states M*› **by** (*simp add*: ‹*t-target t* $\in$ *FSM.states S*›)
have *t-source* $tS \in$ *states* (*from-FSM S* (*t-target t*))
using ‹$tS \in$ *transitions ?S*› **by** *auto*
have *t-source* $tM \in$ *states* (*from-FSM M* (*t-target t*))
using ‹$tM \in$ *transitions ?M*› **by** *auto*

have $q' \in$ *reachable-states ?M*
using ‹$q' \in$ *reachable-states ?S*› *submachine-path*[*OF* ‹*is-submachine ?S ?M*›]
unfolding *reachable-states-def*
proof −
assume $q' \in \{$*target* (*FSM.initial* (*FSM.from-FSM S* (*t-target t*))) *p* |*p*.
path (*FSM.from-FSM S* (*t-target t*)) (*FSM.initial* (*FSM.from-FSM S* (*t-target t*))) *p*$\}$
then show $q' \in \{$*target* (*FSM.initial* (*FSM.from-FSM M* (*t-target t*))) *ps* |*ps*.
path (*FSM.from-FSM M* (*t-target t*)) (*FSM.initial* (*FSM.from-FSM M* (*t-target t*))) *ps*$\}$
using ‹*FSM.initial* (*FSM.from-FSM S* (*t-target t*)) = *FSM.initial* (*FSM.from-FSM M* (*t-target t*))›
‹$\bigwedge q\ p$. *path* (*FSM.from-FSM S* (*t-target t*)) $q\ p \Longrightarrow$ *path* (*FSM.from-FSM M* (*t-target t*)) $q\ p$›
by *fastforce*
qed

show $tM \in$ *transitions ?S*

using $*$ ‹$q' \in$ *reachable-states* S›
‹$tM \in$ *FSM.transitions* M› ‹$tS \in$ *FSM.transitions* S› ‹*t-input* $tM = x$› ‹*t-input* $tS = x$›
‹*t-source* $tM = q'$› ‹*t-source* $tS = q'$› ‹*t-target* $t \in$ *FSM.states* S›
by *fastforce*
qed

show *?thesis*
unfolding *is-preamble-def*
using *is-acyclic*
is-single-input
is-sub
contains-q
has-deadlock-q
has-states-prop-1
using *has-states-prop-2* **by** *blast*
qed

lemma *observable-preamble-paths* :
assumes *is-preamble* P M q'
and *observable* M
and *path* M q p
and *p-io* $p \in$ *LS* P q
and $q \in$ *reachable-states* P
shows *path* P q p
using *assms*(*3*,*4*,*5*) **proof** (*induction* p *arbitrary*: q *rule*: *list.induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons* t p)

have *is-submachine* P M
and $*$: $\bigwedge q'\ x\ t\ t'$. $q' \in$*reachable-states* $P \Longrightarrow x \in$*FSM.inputs* $M \Longrightarrow$
$t \in$*FSM.transitions* $P \Longrightarrow$ *t-source* $t = q' \Longrightarrow$ *t-input* $t = x \Longrightarrow$
$t' \in$*FSM.transitions* $M \Longrightarrow$ *t-source* $t' = q' \Longrightarrow$ *t-input* $t' = x \Longrightarrow t' \in$ *FSM.transitions* P
using *assms*(*1*) **unfolding** *is-preamble-def* **by** *blast+*

have *observable* P
using *submachine-observable*[*OF* ‹*is-submachine* P M› ‹*observable* M›] **by** *blast*

obtain t' **where** $t' \in$*FSM.transitions* P **and** *t-source* $t' = q$ **and** *t-input* t' = *t-input* t
using ‹*p-io* $(t \# p) \in$ *LS* P q› **by** *auto*

have *t-source* $t = q$ **and** $t \in$ *transitions* M **and** *t-input* $t \in$ *inputs* M
using ‹*path* M q $(t \# p)$› **by** *auto*

have $t \in$ *transitions P*
using $*$[*OF* ‹$q \in$ *reachable-states P*› ‹*t-input t* $\in$ *inputs M*› ‹$t' \in$*FSM.transitions P*›
‹*t-source* $t' = q$› ‹*t-input* t' = *t-input t*› ‹$t \in$ *transitions M*› ‹*t-source t* = q›]
by *auto*

have *path M* (*t-target t*) *p*
using ‹*path M q* (t # *p*)› **by** *auto*
moreover have *p-io p* $\in$ *LS P* (*t-target t*)
proof −
have *f1*: *t-input t* = *fst* (*t-input t*, *t-output t*)
by (*metis fst-conv*)
have *f2*: *t-output t* = *snd* (*t-input t*, *t-output t*)
by *auto*
have *f3*: (*t-input t*, *t-output t*) # *p-io p* $\in$ *LS P* (*t-source t*)
using *Cons.prems*(*2*) ‹*t-source t* = q› **by** *fastforce*
have *L* (*FSM.from-FSM P* (*t-target t*)) = *LS P* (*t-target t*)
by (*meson* ‹$t \in$ *FSM.transitions P*› *from-FSM-language fsm-transition-target*)
then show *?thesis*
using *f3 f2 f1* ‹*observable P*› ‹$t \in$ *FSM.transitions P*› *observable-language-next*
by *blast*
qed
moreover have *t-target t* $\in$ *reachable-states P*
using ‹$t \in$ *transitions P*› ‹*t-source t* = q› ‹$q \in$ *reachable-states P*›
by (*meson reachable-states-next*)
ultimately have *path P* (*t-target t*) *p*
using *Cons.IH* **by** *blast*
then show *?case*
using ‹$t \in$ *transitions P*› ‹*t-source t* = q› **by** *auto*
qed

lemma *preamble-pass-path* :
assumes *is-preamble P M q*
and $\bigwedge$ *io x y y'* . *io*@[(*x*,*y*)] $\in$ *L P* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L M'* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L P*
and *completely-specified M'*
and *inputs M'* = *inputs M*
obtains *p* **where** *path P* (*initial P*) *p* **and** *target* (*initial P*) *p* = *q* **and** *p-io p* $\in$ *L M'*
proof −

let *?ps* = {*p* . *path P* (*initial P*) *p* $\wedge$ *p-io p* $\in$ *L M'*}
have *?ps* $\neq$ {}
proof −
have [] $\in$ *?ps* **by** *auto*

```
    then show ?thesis by blast
  qed
  moreover have finite ?ps
  proof −
    have acyclic P
      using assms(1) unfolding is-preamble-def by blast
    have finite {p. path P (FSM.initial P) p}
      using acyclic-finite-paths-from-reachable-state[OF ‹acyclic P›, of [] initial P]
by auto
    then show ?thesis
      by simp
  qed
  ultimately obtain p where p ∈ ?ps and ⋀ p′ . p′ ∈ ?ps ⟹ length p′ ≤ length
p
    by (meson leI max-length-elem)
  then have path P (initial P) p
       and  p-io p ∈ L M′
    by blast+

  show ?thesis
  proof (cases target (initial P) p = q)
    case True
    then show ?thesis using that[OF ‹path P (initial P) p› - ‹p-io p ∈ L M′›] by
blast
  next
    case False



    then have ¬ deadlock-state P (target (initial P) p)
      using reachable-states-intro[OF ‹path P (initial P) p›] assms(1) unfolding
is-preamble-def by fastforce
    then obtain t where t ∈ transitions P and t-source t = target (initial P) p
      by auto
    then have path P (initial P) (p@[t])
      using ‹path P (initial P) p› path-append-transition by simp
    have (p-io p) @ [(t-input t, t-output t)] ∈ L P
      using language-intro[OF ‹path P (initial P) (p@[t])›] by simp

    have t-input t ∈ inputs M′
      using assms(1,4) fsm-transition-input[OF ‹t ∈ transitions P›] unfolding
is-preamble-def is-submachine.simps by blast

    obtain p′ where path M′ (initial M′) p′ and p-io p′ = p-io p
      using ‹p-io p ∈ L M′› by auto
    obtain t′ where t′ ∈ transitions M′ and t-source t′ = target (initial M′) p′
and t-input t′ = t-input t
     using ‹completely-specified M′› ‹t-input t ∈ inputs M′› path-target-is-state[OF
‹path M′ (initial M′) p′›]
```

unfolding *completely-specified.simps* **by** *blast*
then have *path M′ (initial M′) (p′@[t′])*
using ‹*path M′ (initial M′) p′*› *path-append-transition* **by** *simp*
have *(p-io p) @ [(t-input t, t-output t′)] ∈ L M′*
using *language-intro*[*OF* ‹*path M′ (initial M′) (p′@[t′])*›]
unfolding ‹*p-io p′ = p-io p*›[*symmetric*] ‹*t-input t′ = t-input t*›[*symmetric*]
by *simp*

have *(p-io p) @ [(t-input t, t-output t′)] ∈ L P*
using *assms*(*2*)[*OF* ‹*(p-io p) @ [(t-input t, t-output t)] ∈ L P*› ‹*(p-io p) @ [(t-input t, t-output t′)] ∈ L M′*›]
by *assumption*
then obtain *pt′* **where** *path P (initial P) pt′* **and** *p-io pt′ = (p-io p) @ [(t-input t, t-output t′)]*
by *auto*
then have *pt′ ∈ ?ps*
using ‹*(p-io p) @ [(t-input t, t-output t′)] ∈ L M′*› **by** *auto*
then have *length pt′ ≤ length p*
using ‹*⋀ p′ . p′ ∈ ?ps ⟹ length p′ ≤ length p*› **by** *blast*
moreover have *length pt′ > length p*
using ‹*p-io pt′ = (p-io p) @ [(t-input t, t-output t′)]*›
unfolding *length-map*[*of (λ t . (t-input t, t-output t)), symmetric*] **by** *simp*
ultimately have *False*
by *simp*
then show *?thesis*
by *simp*
qed
qed

lemma *preamble-maximal-io-paths* :
assumes *is-preamble P M q*
and *observable M*
and *path P (initial P) p*
and *target (initial P) p = q*
shows *∄ io′ . io′ ≠ [] ∧ p-io p @ io′ ∈ L P*
proof −
have *deadlock-state P q*
and *is-submachine P M*
using *assms*(*1*) **unfolding** *is-preamble-def* **by** *blast+*

have *observable P*
using ‹*observable M*› ‹*is-submachine P M*›
using *submachine-observable* **by** *blast*

show *∄ io′ . io′ ≠ [] ∧ p-io p @ io′ ∈ L P*
proof
assume *∃ io′. io′ ≠ [] ∧ p-io p @ io′ ∈ L P*
then obtain *io′* **where** *io′ ≠ []* **and** *p-io p @ io′ ∈ L P*

```
      by blast

    obtain p1 p2 where path P (FSM.initial P) p1
                  and path P (target (FSM.initial P) p1) p2
                  and p-io p1 = p-io p
                  and p-io p2 = io'
      using language-state-split[OF ‹p-io p @ io' ∈ L P›] by blast

    have p1 = p
     using observable-path-unique[OF ‹observable P› ‹path P (FSM.initial P) p1›
‹path P (FSM.initial P) p› ‹p-io p1 = p-io p›]
      by assumption

    have io' ∈ LS P q
      using ‹path P (target (FSM.initial P) p1) p2› ‹p-io p2 = io'›
      unfolding ‹p1 = p› assms(4) by auto
    then show False
      using ‹io' ≠ []› ‹deadlock-state P q›
      unfolding deadlock-state-alt-def
      by blast
  qed
qed


lemma preamble-maximal-io-paths-rev :
  assumes is-preamble P M q
  and     observable M
  and     io ∈ L P
  and     ∄ io' . io' ≠ [] ∧ io @ io' ∈ L P
obtains p where path P (initial P) p
         and   p-io p = io
         and   target (initial P) p = q
proof −
  have acyclic P
  and  deadlock-state P q
  and  is-submachine P M
  and  ⋀ q' . q'∈reachable-states P ⟹ (q = q' ∨ ¬ deadlock-state P q')
    using assms(1) unfolding is-preamble-def by blast+

  have observable P
    using ‹observable M› ‹is-submachine P M›
    using submachine-observable by blast

  obtain p where path P (initial P) p and p-io p = io
    using ‹io ∈ L P› by auto

  moreover have target (initial P) p = q
  proof (rule ccontr)
    assume target (FSM.initial P) p ≠ q
```

then have $\neg$ *deadlock-state P* (*target* (*FSM.initial P*) *p*)
using ‹$\bigwedge$ *q'* . *q'*∈*reachable-states P* $\Longrightarrow$ (*q* = *q'* ∨ $\neg$ *deadlock-state P q'*)›[*OF reachable-states-intro*[*OF* ‹*path P* (*initial P*) *p*›]] **by** *simp*
then obtain *t* **where** *t* ∈ *transitions P* **and** *t-source t* = *target* (*initial P*) *p*
by *auto*
then have *path P* (*initial P*) (*p* @ [*t*])
using *path-append-transition*[*OF* ‹*path P* (*initial P*) *p*›] **by** *auto*
then have *p-io* (*p*@[*t*]) ∈ *L P*
unfolding *LS.simps* **by** (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
then have *io* @ [(*t-input t*, *t-output t*)] ∈ *L P*
using ‹*p-io p* = *io*› **by** *auto*
then show *False*
using *assms*(*4*) **by** *auto*
qed

ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *is-preamble-is-state* :
assumes *is-preamble P M q*
shows *q* ∈ *states M*
using *assms* **unfolding** *is-preamble-def*
by (*meson nil path-nil-elim reachable-state-is-state submachine-path*)

36.2 Calculating State Preambles via Backwards Reachability Analysis

fun *d-states* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm* ⇒ *'a* ⇒ (*'a* × *'b*) *list* **where**
d-states M q = (*if q* = *initial M*
then []
else select-inputs (*h M*) (*initial M*) (*inputs-as-list M*) (*removeAll q* (*removeAll* (*initial M*) (*states-as-list M*))) {*q*} [])

lemma *d-states-index-properties* :
assumes *i* < *length* (*d-states M q*)
shows *fst* (*d-states M q* ! *i*) ∈ (*states M* − {*q*})
fst (*d-states M q* ! *i*) ≠ *q*
snd (*d-states M q* ! *i*) ∈ *inputs M*
(∀ *qx'* ∈ *set* (*take i* (*d-states M q*)) . *fst* (*d-states M q* ! *i*) ≠ *fst qx'*)
(∃ *t* ∈ *transitions M* . *t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*))
(∀ *t* ∈ *transitions M* . (*t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*)) ⟶ (*t-target t* = *q* ∨ (∃ *qx'* ∈ *set* (*take i* (*d-states M q*)) . *fst qx'* = (*t-target t*))))
proof −
have *combined-goals* : *fst* (*d-states M q* ! *i*) ∈ (*states M* − {*q*})

$\wedge$ *fst* (*d-states M q* ! *i*) $\neq$ *q*
$\wedge$ *snd* (*d-states M q* ! *i*) $\in$ *inputs M*
$\wedge$ ($\forall$ *qx'* $\in$ *set* (*take i* (*d-states M q*)) . *fst* (*d-states M q* ! *i*) $\neq$ *fst qx'*)
$\wedge$ ($\exists$ *t* $\in$ *transitions M* . *t-source t* = *fst* (*d-states M q* ! *i*) $\wedge$ *t-input t* = *snd* (*d-states M q* ! *i*))
$\wedge$ ($\forall$ *t* $\in$ *transitions M* . (*t-source t* = *fst* (*d-states M q* ! *i*) $\wedge$ *t-input t* = *snd* (*d-states M q* ! *i*)) $\longrightarrow$ (*t-target t* = *q* $\vee$ ($\exists$ *qx'* $\in$ *set* (*take i* (*d-states M q*)) . *fst qx'* = (*t-target t*))))
proof (*cases q* = *initial M*)
case *True*
then have *d-states M q* = [] **by** *auto*
then have *False* **using** *assms* **by** *auto*
then show *?thesis* **by** *simp*
next
case *False*
then have $*$: *d-states M q* = *select-inputs* (*h M*) (*initial M*) (*inputs-as-list M*) (*removeAll q* (*removeAll* (*initial M*) (*states-as-list M*))) {*q*} [] **by** *auto*

have *initial M* $\in$ *states M* **by** *auto*
then have *insert* (*FSM.initial M*) (*set* (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*)))) = *states M* $-$ {*q*}
using *states-as-list-set False* **by** *auto*

have *i* $<$ *length* (*select-inputs* (*h M*) (*FSM.initial M*) (*inputs-as-list M*) (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*))) {*q*} [])
using *assms* $*$ **by** *simp*
moreover have *length* [] $\leq$ *i* **by** *auto*
moreover have *distinct* (*map fst* []) **by** *auto*
moreover have {*q*} = {*q*} $\cup$ *set* (*map fst* []) **by** *auto*
moreover have *initial M* $\notin$ {*q*} **using** *False* **by** *auto*
moreover have *distinct* (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*))) **using** *states-as-list-distinct*
by (*simp add*: *distinct-removeAll*)
moreover have *FSM.initial M* $\notin$ *set* (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*))) **by** *auto*
moreover have *set* (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*))) $\cap$ {*q*} = {} **by** *auto*

moreover show *?thesis*
using *select-inputs-index-properties*[*OF calculation*]
unfolding $*$[*symmetric*] *inputs-as-list-set* ‹*insert* (*FSM.initial M*) (*set* (*removeAll q* (*removeAll* (*FSM.initial M*) (*states-as-list M*)))) = *states M* $-$ {*q*}› **by** *blast*
qed

then show *fst* (*d-states M q* ! *i*) $\in$ (*states M* $-$ {*q*})
fst (*d-states M q* ! *i*) $\neq$ *q*

snd (*d-states M q* ! *i*) ∈ *inputs M*
(∀ *qx′* ∈ *set* (*take i* (*d-states M q*)) . *fst* (*d-states M q* ! *i*) ≠ *fst qx′*)
(∃ *t* ∈ *transitions M* . *t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*))
(∀ *t* ∈ *transitions M* . (*t-source t* = *fst* (*d-states M q* ! *i*) ∧ *t-input t* = *snd* (*d-states M q* ! *i*)) ⟶ (*t-target t* = *q* ∨ (∃ *qx′* ∈ *set* (*take i* (*d-states M q*)) . *fst qx′* = (*t-target t*))))
by *blast+*
qed

lemma *d-states-distinct* :
distinct (*map fst* (*d-states M q*))
proof −
have ∗: ⋀ *i q* . *i* < *length* (*map fst* (*d-states M q*)) ⟹ *q* ∈ *set* (*take i* (*map fst* (*d-states M q*))) ⟹ ((*map fst* (*d-states M q*)) ! *i*) ≠ *q*
using *d-states-index-properties*(*2*,*4*) **by** *fastforce*
then have (⋀*i*. *i* < *length* (*map fst* (*d-states M q*)) ⟹
map fst (*d-states M q*) ! *i* ∉ *set* (*take i* (*map fst* (*d-states M q*))))
proof −
fix *i* :: *nat*
assume *a1*: *i* < *length* (*map fst* (*d-states M q*))
then have ∀ *p*. *p* ∉ *set* (*take i* (*d-states M q*)) ∨ *fst* (*d-states M q* ! *i*) ≠ *fst p*
by (*metis* (*no-types*) *d-states-index-properties*(*4*) *length-map*)
then show *map fst* (*d-states M q*) ! *i* ∉ *set* (*take i* (*map fst* (*d-states M q*)))
using *a1* **by** (*metis* (*no-types*) *length-map list-map-source-elem nth-map take-map*)
qed
then show *?thesis*
using *list-distinct-prefix*[*of map fst* (*d-states M q*)] **by** *blast*
qed

lemma *d-states-states* :
set (*map fst* (*d-states M q*)) ⊆ *states M* − {*q*}
using *d-states-index-properties*(*1*)[*of - M q*] *list-property-from-index-property*[*of map fst* (*d-states M q*) λ*q′* . *q′* ∈ *states M* − {*q*}]
by (*simp add*: *subsetI*)

lemma *d-states-size* :
assumes *q* ∈ *states M*
shows *length* (*d-states M q*) ≤ *size M* − *1*
proof −
show *?thesis*
using *d-states-states*[*of M q*]
d-states-distinct[*of M q*]
fsm-states-finite[*of M*]
assms

by (*metis card-Diff-singleton-if card-mono distinct-card finite-Diff length-map size-def*)
qed

lemma *d-states-initial* :
assumes $qx \in$ *set* (*d-states M q*)
and *fst qx* = *initial M*
shows (*last* (*d-states M q*)) = *qx*
using *assms*(*1*) *select-inputs-initial*[*of qx h M initial M - - -* [], *OF - assms*(*2*)]
by (*cases q* = *initial M*; *auto*)

lemma *d-states-q-noncontainment* :
shows ¬(∃ *qqx* ∈ *set* (*d-states M q*) . *fst qqx* = *q*)
using *d-states-index-properties*(*2*)
by (*metis in-set-conv-nth*)

lemma *d-states-acyclic-paths′* :
fixes *M* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm*
assumes *path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q′ p*
and *target q′ p* = *q′*
and *p* ≠ []
shows *False*
proof −
from ‹*p* ≠ []› **obtain** *p′ t′* **where** *p* = *t′*#*p′*
using *list.exhaust* **by** *blast*
then have *path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q′* (*p*@[*t′*])
using *assms*(*1*,*2*) **by** *fastforce*

define *f* :: (*′a* × *′b* × *′c* × *′a*) ⇒ *nat*
where *f-def*: *f* = (λ *t* . *the* (*find-index* (λ *qx* . *fst qx* = *t-source t* ∧ *snd qx* = *t-input t*) (*d-states M q*)))

have *f-prop*: ⋀ *t* . *t* ∈ *set* (*p*@[*t′*]) ⟹ (*f t* < *length* (*d-states M q*))
∧ ((*d-states M q*) ! (*f t*) = (*t-source t*, *t-input t*))
∧ (∀ *j* < *f t* . *fst* (*d-states M q* ! *j*) ≠ *t-source t*)
proof −
fix *t* **assume** *t* ∈ *set* (*p*@[*t′*])
then have *t* ∈ *set p* **using** ‹*p* = *t′*#*p′*› **by** *auto*
then have *t* ∈ *transitions M* **and** (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)
using *assms*(*1*) *path-transitions* **by** *fastforce*+
then have ∃ *qx* ∈ *set* (*d-states M q*) . (λ *qx* . *fst qx* = *t-source t* ∧ *snd qx* = *t-input t*) *qx*
by (*meson fst-conv snd-conv*)
then have *find-index* (λ *qx* . *fst qx* = *t-source t* ∧ *snd qx* = *t-input t*) (*d-states*

M q) ≠ *None*
by (*simp add*: *find-index-exhaustive*)
then obtain *i* **where** ∗: *find-index* (λ *qx* . *fst qx* = *t-source t* ∧ *snd qx* = *t-input t*) (*d-states M q*) = *Some i*
by *auto*

have *f t* < *length* (*d-states M q*)
unfolding *f-def* **using** *find-index-index*(*1*)[*OF* ∗] **unfolding** ∗ **by** *simp*
moreover have ((*d-states M q*) ! (*f t*) = (*t-source t*, *t-input t*))
unfolding *f-def* **using** *find-index-index*(*2*)[*OF* ∗]
by (*metis* ∗ *option.sel prod.collapse*)
moreover have ∀ *j* < *f t* . *fst* (*d-states M q* ! *j*) ≠ *t-source t*
unfolding *f-def* **using** *find-index-index*(*3*)[*OF* ∗] **unfolding** ∗
using *d-states-distinct*[*of M q*]
by (*metis* (*mono-tags*, *lifting*) *calculation*(*1*) *calculation*(*2*) *distinct-conv-nth fst-conv length-map less-imp-le less-le-trans not-less nth-map option.sel snd-conv*)
ultimately show (*f t* < *length* (*d-states M q*))
∧ ((*d-states M q*) ! (*f t*) = (*t-source t*, *t-input t*))
∧ (∀ *j* < *f t* . *fst* (*d-states M q* ! *j*) ≠ *t-source t*) **by** *simp*
qed

have ∗: ⋀ *i* . *Suc i* < *length* (*p*@[*t'*]) ⟹ *f* ((*p*@[*t'*]) ! *i*) > *f* ((*p*@[*t'*]) ! (*Suc i*))
proof −
fix *i* **assume** *Suc i* < *length* (*p*@[*t'*])
then have (*p*@[*t'*]) ! *i* ∈ *set* (*p*@[*t'*]) **and** (*p*@[*t'*]) ! (*Suc i*) ∈ *set* (*p*@[*t'*])
using *Suc-lessD nth-mem* **by** *blast*+
then have (*p*@[*t'*]) ! *i* ∈ *transitions M* **and** (*p*@[*t'*]) ! *Suc i* ∈ *transitions M*
using *path-transitions*[*OF* ‹*path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q'* (*p*@[*t'*])›]
using *filter-transitions-simps*(*5*) **by** *blast*+

have *f* ((*p*@[*t'*]) ! *i*) < *length* (*d-states M q*)
and (*d-states M q*) ! (*f* ((*p*@[*t'*]) ! *i*)) = (*t-source* ((*p*@[*t'*]) ! *i*), *t-input* ((*p*@[*t'*]) ! *i*))
and (∀*j*<*f* ((*p*@[*t'*]) ! *i*). *fst* (*d-states M q* ! *j*) ≠ *t-source* ((*p*@[*t'*]) ! *i*))
using *f-prop*[*OF* ‹(*p*@[*t'*]) ! *i* ∈ *set* (*p*@[*t'*])›] **by** *auto*

have *f* ((*p*@[*t'*]) ! *Suc i*) < *length* (*d-states M q*)
and (*d-states M q*) ! (*f* ((*p*@[*t'*]) ! *Suc i*)) = (*t-source* ((*p*@[*t'*]) ! *Suc i*), *t-input* ((*p*@[*t'*]) ! *Suc i*))
and (∀*j*<*f* ((*p*@[*t'*]) ! *Suc i*). *fst* (*d-states M q* ! *j*) ≠ *t-source* ((*p*@[*t'*]) ! *Suc i*))
using *f-prop*[*OF* ‹(*p*@[*t'*]) ! *Suc i* ∈ *set* (*p*@[*t'*])›] **by** *auto*

have *t-target* ((*p*@[*t'*]) ! *i*) = *t-source* ((*p*@[*t'*]) ! *Suc i*)
using ‹*Suc i* < *length* (*p*@[*t'*])› ‹*path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q'* (*p*@[*t'*])›

by (*simp add*: *path-source-target-index*)
then have *t-target* ((*p*@[*t'*]) ! *i*) ≠ *q*
using *d-states-index-properties*(*2*)[*OF* ‹*f* ((*p*@[*t'*]) ! *Suc i*) < *length* (*d-states M q*)›]
unfolding ‹(*d-states M q*) ! (*f* ((*p*@[*t'*]) ! *Suc i*)) = (*t-source* ((*p*@[*t'*]) ! *Suc i*), *t-input* ((*p*@[*t'*]) ! *Suc i*))› **by** *auto*
then have (∃ *qx'*∈*set* (*take* (*f* ((*p*@[*t'*]) ! *i*)) (*d-states M q*)). *fst qx'* = *t-target* ((*p*@[*t'*]) ! *i*))
using *d-states-index-properties*(*6*)[*OF* ‹*f* ((*p*@[*t'*]) ! *i*) < *length* (*d-states M q*)›] **unfolding** ‹(*d-states M q*) ! (*f* ((*p*@[*t'*]) ! *i*)) = (*t-source* ((*p*@[*t'*]) ! *i*), *t-input* ((*p*@[*t'*]) ! *i*))› *fst-conv snd-conv*
using ‹(*p*@[*t'*]) ! *i* ∈ *transitions M*›
by *blast*
then have (∃ *qx'*∈*set* (*take* (*f* ((*p*@[*t'*]) ! *i*)) (*d-states M q*)). *fst qx'* = *t-source* ((*p*@[*t'*]) ! *Suc i*))
unfolding ‹*t-target* ((*p*@[*t'*]) ! *i*) = *t-source* ((*p*@[*t'*]) ! *Suc i*)› **by** *assumption*
then obtain *j* **where** *fst* (*d-states M q* ! *j*) = *t-source* ((*p*@[*t'*]) ! *Suc i*) **and** *j* < *f* ((*p*@[*t'*]) ! *i*)
by (*metis* (*no-types*, *lifting*) ‹*f* ((*p*@[*t'*]) ! *i*) < *length* (*d-states M q*)› *in-set-conv-nth leD length-take min-def-raw nth-take*)

then show *f* ((*p*@[*t'*]) ! *i*) > *f* ((*p*@[*t'*]) ! (*Suc i*))
using ‹(∀ *j*<*f* ((*p*@[*t'*]) ! *Suc i*). *fst* (*d-states M q* ! *j*) ≠ *t-source* ((*p*@[*t'*]) ! *Suc i*))›
using *leI le-less-trans* **by** *blast*
qed

have ⋀ *i j* . *j* < *i* ⟹ *i* < *length* (*p*@[*t'*]) ⟹ *f* ((*p*@[*t'*]) ! *j*) > *f* ((*p*@[*t'*]) ! *i*)
using *list-index-fun-gt*[*of p*@[*t'*] *f*] ∗ **by** *blast*
then have *f t'* < *f t'*
unfolding ‹*p* = *t'*#*p'*› **by** *fastforce*
then show *False*
by *auto*
qed

lemma *d-states-acyclic-paths* :
fixes *M* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm*
assumes *path* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *q' p*
(**is** *path ?FM q' p*)
shows *distinct* (*visited-states q' p*)
proof (*rule ccontr*)
assume ¬ *distinct* (*visited-states q' p*)

obtain *i j* **where** *p1*:*take j* (*drop i p*) ≠ []
and *p2*:*target* (*target q'* (*take i p*)) (*take j* (*drop i p*)) = (*target q'* (*take i p*))
and *p3*:*path ?FM* (*target q'* (*take i p*)) (*take j* (*drop i p*))

using *cycle-from-cyclic-path*[*OF assms* ‹¬ *distinct* (*visited-states* q' *p*)›] **by** *blast*

show *False*
using *d-states-acyclic-paths'*[*OF p3 p2 p1*] **by** *assumption*
qed

lemma *d-states-induces-state-preamble-helper-acyclic* :
shows *acyclic* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)))
unfolding *acyclic.simps*
using *d-states-acyclic-paths* **by** *force*

lemma *d-states-induces-state-preamble-helper-single-input* :
shows *single-input* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)))
(**is** *single-input ?FM*)
unfolding *single-input.simps filter-transitions-simps*
by (*metis* (*no-types*, *lifting*) *d-states-distinct eq-key-imp-eq-value mem-Collect-eq*)

lemma *d-states-induces-state-preamble* :
assumes ∃ *qx* ∈ *set* (*d-states M q*) . *fst qx* = *initial M*
shows *is-preamble* (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*))) *M q*
(**is** *is-preamble ?S M q*)
proof (*cases q* = *initial M*)
case *True*
then have *d-states M q* = [] **by** *auto*
then show *?thesis* **using** *assms*(*1*) **by** *auto*
next
case *False*

have *is-acyclic*: *acyclic ?S*
using *d-states-induces-state-preamble-helper-acyclic*[*of M q*] **by** *presburger*

have *is-single-input*: *single-input ?S*
using *d-states-induces-state-preamble-helper-single-input*[*of M q*] **by** *presburger*

have *is-sub* : *is-submachine ?S M*
unfolding *is-submachine.simps filter-transitions-simps* **by** *blast*

have *has-deadlock-q* : *deadlock-state ?S q*
using *d-states-q-noncontainment*[*of M q*] **unfolding** *deadlock-state.simps*
by *fastforce*

have ⋀ q' . q' ∈ *reachable-states ?S* ⟹ $q' \neq q$ ⟹ ¬ *deadlock-state ?S* q'
proof −
fix q' **assume** q' ∈ *reachable-states ?S* **and** $q' \neq q$

then obtain p **where** *path ?S* (*initial ?S*) p **and** *target* (*initial ?S*) $p = q'$
unfolding *reachable-states-def* **by** *auto*

have $\exists$ $qx \in$ *set* (*d-states M q*) . *fst qx* $= q'$
proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis*
using *assms*(*1*) ‹*target* (*initial ?S*) $p = q'$› **unfolding** *filter-transitions-simps*
by *simp*
next
case (*snoc* p' t)
then have $t \in$ *transitions ?S* **and** *t-target* $t = q'$
using ‹*path ?S* (*initial ?S*) p› ‹*target* (*initial ?S*) $p = q'$› **by** *auto*
then have (*t-source t*, *t-input t*) $\in$ *set* (*d-states M q*)
by *simp*
then obtain i **where** $i <$ *length* (*d-states M q*) **and** *d-states M q* ! $i =$ (*t-source t*, *t-input t*)
by (*meson in-set-conv-nth*)

have $t \in$ *transitions M*
using ‹$t \in$ *transitions ?S*›
using *is-sub* **by** *auto*
then show *?thesis*
using ‹*t-target* $t = q'$› ‹$q' \neq q$›
using *d-states-index-properties*(*6*)[*OF* ‹$i <$ *length* (*d-states M q*)›]
unfolding ‹*d-states M q* ! $i =$ (*t-source t*, *t-input t*)› *fst-conv snd-conv*
by (*metis in-set-takeD*)
qed

then obtain qx **where** $qx \in$ *set* (*d-states M q*) **and** *fst qx* $= q'$ **by** *blast*

then have ($\exists$ $t \in$ *transitions M* . *t-source t* $=$ *fst qx* $\wedge$ *t-input t* $=$ *snd qx*)
using *d-states-index-properties*(*5*)[*of - M q*]
by (*metis in-set-conv-nth*)
then have ($\exists$ $t \in$ *transitions ?S* . *t-source t* $=$ *fst qx* $\wedge$ *t-input t* $=$ *snd qx*)
using ‹$qx \in$ *set* (*d-states M q*)› **by** *fastforce*

then show $\neg$ *deadlock-state ?S* q'
unfolding *deadlock-state.simps* **using** ‹*fst qx* $= q'$› **by** *blast*
qed

then have *has-states-prop-1* : $\bigwedge$ q' . $q' \in$ *reachable-states ?S* $\Longrightarrow$ ($q = q' \vee \neg$ *deadlock-state ?S* q')
by *blast*

have *has-states-prop-2* : $\bigwedge$ q' x t t'. $q' \in$ *reachable-states ?S* $\Longrightarrow$ $x \in$ *inputs M* $\Longrightarrow$
$t \in$ *transitions ?S* $\Longrightarrow$ *t-source t* $= q' \Longrightarrow$ *t-input t* $= x \Longrightarrow$
$t' \in$ *transitions M* $\Longrightarrow$ *t-source* $t' = q' \Longrightarrow$ *t-input* $t' = x \Longrightarrow t' \in$

transitions ?S
  by *simp*

  have *contains-q* : $q \in$ *reachable-states ?S*
    using ‹$\bigwedge q'$. $[\![q' \in$ *reachable-states ?S*; $q' \neq q]\!] \Longrightarrow \neg$ *deadlock-state ?S* q'› *acyclic-deadlock-reachable is-acyclic*
  by *blast*

  show *?thesis*
  unfolding *is-preamble-def*
  using *is-acyclic*
    is-single-input
    is-sub
    contains-q
    has-deadlock-q
    has-states-prop-1 has-states-prop-2
  by *blast*
qed

fun *calculate-state-preamble-from-input-choices* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* $\Rightarrow$ '*a* $\Rightarrow$ ('*a*,'*b*,'*c*) *fsm option*
  where
  calculate-state-preamble-from-input-choices M q = (*if q* = *initial M*
  then Some (*initial-preamble M*)
  else
    (*let DS* = (*d-states M q*);
      DSS = *set DS*
    in (*case DS of*
      [] $\Rightarrow$ *None* |
      - $\Rightarrow$ *if fst* (*last DS*) = *initial M*
        then Some (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) $\in$ *DSS*))
          else None)))

lemma *calculate-state-preamble-from-input-choices-soundness* :
  assumes *calculate-state-preamble-from-input-choices M q* = *Some S*
  shows *is-preamble S M q*
proof (*cases q* = *initial M*)
  case *True*
  then have *S* = *initial-preamble M* **using** *assms* **by** *auto*
  then show *?thesis*
  using *is-preamble-initial*[*of M*] *True* **by** *presburger*
next
  case *False*

  then have *S* = (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) $\in$ *set* (*d-states M q*)))
    and *length* (*d-states M q*) $\neq$ *0*

and *fst* (*last* (*d-states M q*)) = *initial M*
using *assms* **by** (*cases* (*d-states M q*); *cases fst* (*last* (*d-states M q*)) = *initial M*; *simp*)+

then have ∃ *qx* ∈ *set* (*d-states M q*) . *fst qx* = *initial M*
by *auto*

then show *?thesis*
using *d-states-induces-state-preamble*
unfolding ‹*S* = (*filter-transitions M* (λ *t* . (*t-source t*, *t-input t*) ∈ *set* (*d-states M q*)))›
by *blast*
qed

lemma *calculate-state-preamble-from-input-choices-exhaustiveness* :
assumes ∃ *S* . *is-preamble S M q*
shows *calculate-state-preamble-from-input-choices M q* ≠ *None*
proof (*cases q* = *initial M*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*

obtain *S* **where** *is-preamble S M q*
using *assms* **by** *blast*

then have *acyclic S*
and *single-input S*
and *is-submachine S M*
and *q* ∈ *reachable-states S*
and ⋀ *q′* . *q′* ∈ *reachable-states S* ⟹ (*q* = *q′* ∨ ¬ *deadlock-state S q′*)
and ∗: ⋀ *q′ x* . *q′* ∈ *reachable-states S* ⟹ *x* ∈ *inputs M* ⟹ (∃ *t* ∈ *transitions S* . *t-source t* = *q′* ∧ *t-input t* = *x*) ⟹ (∀ *t′* ∈ *transitions M* . *t-source t′* = *q′* ∧ *t-input t′* = *x* ⟶ *t′* ∈ *transitions S*)
unfolding *is-preamble-def* **by** *blast*+

have *p1*: (⋀*q x*. *q* ∈ *reachable-states S* ⟹ *h S* (*q*, *x*) ≠ {} ⟹ *h S* (*q*, *x*) = *h M* (*q*, *x*))
proof −
fix *q x* **assume** *q* ∈ *reachable-states S* **and** *h S* (*q*, *x*) ≠ {}

then have *x* ∈ *inputs M*
using ‹*is-submachine S M*› *fsm-transition-input* **by** *force*
have (∃ *t* ∈ *transitions S* . *t-source t* = *q* ∧ *t-input t* = *x*)
using ‹*h S* (*q*, *x*) ≠ {}› **by** *fastforce*

have $\bigwedge$ *y q''* . (*y*,*q''*) ∈ *h S* (*q*,*x*) ⟹ (*y*,*q''*) ∈ *h M* (*q*,*x*)
using ‹*is-submachine S M*› **by** *force*
moreover have $\bigwedge$ *y q''* . (*y*,*q''*) ∈ *h M* (*q*,*x*) ⟹ (*y*,*q''*) ∈ *h S* (*q*,*x*)
using ∗[*OF* ‹*q* ∈ *reachable-states S*› ‹*x* ∈ *inputs M*› ‹(∃ *t* ∈ *transitions S* . *t-source t* = *q* ∧ *t-input t* = *x*)›]
unfolding *h.simps* **by** *force*
ultimately show *h S* (*q*, *x*) = *h M* (*q*, *x*)
by *force*
qed

have *p2*: $\bigwedge$*q'*. *q'* ∈ *reachable-states S* ⟹ *deadlock-state S q'* ⟹ *q'* ∈ {*q*} ∪ *set* (*map fst* []) **using** ‹$\bigwedge$ *q'* . *q'* ∈ *reachable-states S* ⟹ (*q* = *q'* ∨ ¬ *deadlock-state S q'*)› **by** *fast*

have *q* ∈ *states M*
using ‹*q* ∈ *reachable-states S*› *submachine-reachable-subset*[*OF* ‹*is-submachine S M*›]
by (*meson assms is-preamble-is-state*)
then have *p3*: *states M* = *insert* (*FSM.initial S*) (*set* (*removeAll q* (*removeAll* (*initial M*) (*states-as-list M*))) ∪ {*q*} ∪ *set* (*map fst* []))
using *states-as-list-set*[*of M*] *fsm-initial*[*of M*]
unfolding *submachine-simps*[*OF* ‹*is-submachine S M*›]
by *auto*

have *p4*: *initial S* ∉ *set* (*removeAll q* (*removeAll* (*initial M*) (*states-as-list M*))) ∪ {*q*} ∪ *set* (*map fst* [])
using *False*
unfolding *submachine-simps*[*OF* ‹*is-submachine S M*›] **by** *force*

have *fst* (*last* (*d-states M q*)) = *FSM.initial M* **and** *length* (*d-states M q*) > *0*
using *False select-inputs-from-submachine*[*OF* ‹*single-input S*› ‹*acyclic S*› ‹*is-submachine S M*› *p1 p2 p3 p4*]
unfolding *d-states.simps submachine-simps*[*OF* ‹*is-submachine S M*›]
by *auto*

have (*d-states M q*) ≠ []
using ‹*length* (*d-states M q*) > *0*› **by** *auto*
then obtain *dl dl'* **where** (*d-states M q*) = *dl* # *dl'*
using *list.exhaust* **by** *blast*
then have (*fst* (*last* (*dl*#*dl'*)) = *FSM.initial M*) = *True* **using** ‹*fst* (*last* (*d-states M q*)) = *FSM.initial M*› **by** *simp*
then show *?thesis*
using *False*
unfolding *calculate-state-preamble-from-input-choices.simps Let-def* ‹(*d-states M q*) = *dl* # *dl'*›
by *auto*
qed

36.3 Minimal Sequences to Failures extending Preambles

definition *sequence-to-failure-extending-preamble-path* ::
($'a$,$'b$,$'c$) *fsm* ⇒ ($'d$,$'b$,$'c$) *fsm* ⇒ ($'a$ × ($'a$,$'b$,$'c$) *fsm*) *set* ⇒ ($'a$×$'b$×$'c$×$'a$) *list* ⇒ ($'b$ × $'c$) *list* ⇒ *bool*
where
sequence-to-failure-extending-preamble-path M M′ PS p io = (∃ *q P* . *q* ∈ *states M*
∧ (*q*,*P*) ∈ *PS*
∧ *path P* (*initial P*) *p*
∧ *target* (*initial P*) *p* = *q*
∧ ((*p-io p*) @ *butlast io*) ∈ *L M*
∧ ((*p-io p*) @ *io*) ∉ *L M*
∧ ((*p-io p*) @ *io*) ∈ *L M′*)

lemma *sequence-to-failure-extending-preamble-ex* :
assumes (*initial M*, (*initial-preamble M*)) ∈ *PS* (**is** (*initial M*,*?P*) ∈ *PS*)
and ¬ *L M′* ⊆ *L M*
obtains *p io* **where** *sequence-to-failure-extending-preamble-path M M′ PS p io*
proof −
obtain *io* **where** *io* ∈ *L M′* − *L M*
using ‹¬ *L M′* ⊆ *L M*› **by** *auto*

obtain *j* **where** *take j io* ∈ *L M* **and** *take* (*Suc j*) *io* ∉ *L M*
proof −
have ∃ *j* . *take j io* ∈ *L M* ∧ *take* (*Suc j*) *io* ∉ *L M*
proof (*rule ccontr*)
assume ∄*j*. *take j io* ∈ *LS M* (*initial M*) ∧ *take* (*Suc j*) *io* ∉ *LS M* (*initial M*)
then have ∗: ⋀ *j* . *take j io* ∈ *LS M* (*initial M*) ⟹ *take* (*Suc j*) *io* ∈ *LS M* (*initial M*) **by** *blast*

have ⋀ *j* . *take j io* ∈ *LS M* (*initial M*)
proof −
fix *j*
show *take j io* ∈ *LS M* (*initial M*)
using ∗ **by** (*induction j*; *auto*)
qed
then have *take* (*length io*) *io* ∈ *L M* **by** *blast*
then show *False*
using ‹*io* ∈ *L M′* − *L M*› **by** *auto*
qed
then show *?thesis* **using** *that* **by** *blast*
qed

have ⋀ *i* . *take i io* ∈ *L M′*
proof −
fix *i* **show** *take i io* ∈ *L M′* **using** ‹*io* ∈ *L M′* − *L M*› *language-prefix*[*of take i io drop i io M′ initial M′*] **by** *auto*

qed

let *?io* = *take* (*Suc j*) *io*

have *initial M* ∈ *states M*
by *auto*
moreover note ‹(*initial M*, (*initial-preamble M*)) ∈ *PS*›
moreover have *path ?P* (*initial ?P*) [] **by** *force*
moreover have ((*p-io* []) @ *butlast ?io*) ∈ *L M*
using ‹*take j io* ∈ *L M*›
unfolding *List.list.map*(*1*) *append-Nil*
by (*metis Diff-iff One-nat-def* ‹*io* ∈ *LS M′* (*initial M′*) − *LS M* (*initial M*)› *butlast-take*
diff-Suc-Suc minus-nat.diff-0 not-less-eq-eq take-all)
moreover have ((*p-io* []) @ *?io*) ∉ *L M*
using ‹*take* (*Suc j*) *io* ∉ *L M*› **by** *auto*
moreover have ((*p-io* []) @ *?io*) ∈ *L M′*
using ‹⋀ *i* . *take i io* ∈ *L M′*› **by** *auto*
ultimately have *sequence-to-failure-extending-preamble-path M M′ PS* [] *?io*
unfolding *sequence-to-failure-extending-preamble-path-def* **by** *force*
then show *?thesis*
using *that* **by** *blast*
qed

definition *minimal-sequence-to-failure-extending-preamble-path* ::
(′*a*,′*b*,′*c*) *fsm* ⇒ (′*d*,′*b*,′*c*) *fsm* ⇒ (′*a* × (′*a*,′*b*,′*c*) *fsm*) *set* ⇒ (′*a*×′*b*×′*c*×′*a*) *list* ⇒ (′*b* × ′*c*) *list* ⇒ *bool*
where
minimal-sequence-to-failure-extending-preamble-path M M′ PS p io
= ((*sequence-to-failure-extending-preamble-path M M′ PS p io*)
∧ (∀ *p′ io′* . *sequence-to-failure-extending-preamble-path M M′ PS p′ io′*
⟶ *length io* ≤ *length io′*))

lemma *minimal-sequence-to-failure-extending-preamble-ex* :
assumes (*initial M*, (*initial-preamble M*)) ∈ *PS* (**is** (*initial M*,*?P*) ∈ *PS*)
and ¬ *L M′* ⊆ *L M*
obtains *p io* **where** *minimal-sequence-to-failure-extending-preamble-path M M′ PS p io*
proof −
let *?ios* = {*io* . ∃ *p* . *sequence-to-failure-extending-preamble-path M M′ PS p io*}
let *?io-min* = *arg-min length* (λ*io* . *io* ∈ *?ios*)

have *?ios* ≠ {}
using *sequence-to-failure-extending-preamble-ex*[*OF assms*] **by** *blast*
then have *?io-min* ∈ *?ios* **and** (∀ *io′* ∈ *?ios* . *length ?io-min* ≤ *length io′*)
by (*meson arg-min-nat-lemma some-in-eq*)+

obtain p **where** *sequence-to-failure-extending-preamble-path M M′ PS p ?io-min*
using ‹*?io-min* ∈ *?ios*›
by *auto*
moreover have (∀ *p′ io′* . *sequence-to-failure-extending-preamble-path M M′ PS p′ io′* ⟶ *length ?io-min* ≤ *length io′*)
using ‹(∀ *io′* ∈ *?ios* . *length ?io-min* ≤ *length io′*)› **by** *blast*
ultimately show *?thesis*
using *that*[*of p ?io-min*]
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast*
qed

lemma *minimal-sequence-to-failure-extending-preamble-no-repetitions-along-path* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *path M* (*target* (*initial M*) *pP*) *p*
and *p-io p* = *butlast io*
and *q′* ∈ *io-targets M′* (*p-io pP*) (*initial M′*)
and *path M′ q′ p′*
and *p-io p′* = *io*
and *i* < *j*
and *j* < *length* (*butlast io*)
and ⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*
shows *t-target* (*p* ! *i*) ≠ *t-target* (*p* ! *j*) ∨ *t-target* (*p′* ! *i*) ≠ *t-target* (*p′* ! *j*)
proof (*rule ccontr*)
assume ¬ (*t-target* (*p* ! *i*) ≠ *t-target* (*p* ! *j*) ∨ *t-target* (*p′* ! *i*) ≠ *t-target* (*p′* ! *j*))

then have *t-target* (*p* ! *i*) = *t-target* (*p* ! *j*)
and *t-target* (*p′* ! *i*) = *t-target* (*p′* ! *j*)
by *blast*+

have *sequence-to-failure-extending-preamble-path M M′ PS pP io*
and ⋀ *p′ io′* . *sequence-to-failure-extending-preamble-path M M′ PS p′ io′*
⟹ *length io* ≤ *length io′*
using ‹*minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast*+

obtain *q P* **where** (*q*,*P*) ∈ *PS*
and *path P* (*initial P*) *pP*
and *target* (*initial P*) *pP* = *q*
and ((*p-io pP*) @ *butlast io*) ∈ *L M*
and ((*p-io pP*) @ *io*) ∉ *L M*
and ((*p-io pP*) @ *io*) ∈ *L M′*

using ‹*sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *sequence-to-failure-extending-preamble-path-def*
by *blast*

have *is-preamble P M q*
using ‹(*q*,*P*) ∈ *PS*› ‹⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*› **by** *blast*
then have *q* ∈ *states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*›
path-target-is-state submachine-path)

have *initial P* = *initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *pP*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P* (*FSM.initial P*) *pP*› **by** *blast*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*› **unfolding** ‹*initial P* = *initial M*› **by** *assumption*

then have *path M q p*
using ‹*path M* (*target* (*initial M*) *pP*) *p*› **by** *auto*

have *io* ≠ []
using ‹((*p-io pP*) @ *butlast io*) ∈ *L M*› ‹((*p-io pP*) @ *io*) ∉ *L M*› **by** *auto*
then have *length p′* > *0*
using ‹*p-io p′* = *io*› **by** *auto*
then have *p′* = (*butlast p′*)@[*last p′*]
by *auto*
then have *path M′ q′* ((*butlast p′*)@[*last p′*])
using ‹*path M′ q′ p′*› **by** *simp*
then have *path M′ q′* (*butlast p′*) **and** (*last p′*) ∈ *transitions M′* **and** *t-source* (*last p′*) = *target q′* (*butlast p′*)
by *auto*

have *p-io* (*butlast p′*) = *butlast io*
using ‹*p′* = (*butlast p′*)@[*last p′*]› ‹*p-io p′* = *io*›
using *map-butlast* **by** *auto*

let *?p* = ((*take* (*Suc i*) *p*) @ (*drop* (*Suc j*) *p*))
let *?pCut* = (*drop* (*Suc i*) (*take* (*Suc j*) *p*))
let *?p′* = ((*take* (*Suc i*) (*butlast p′*)) @ (*drop* (*Suc j*) (*butlast p′*)))

have *j* < *length p*
using ‹*j* < *length* (*butlast io*)› ‹*p-io p* = *butlast io*›
by (*metis* (*no-types*, *lifting*) *length-map*)
have *j* < *length* (*butlast p′*)
using ‹*j* < *length* (*butlast io*)› ‹*p-io p′* = *io*› ‹*p′* = (*butlast p′*)@[*last p′*]›
by *auto*

then have *t-target* ((*butlast p′*) ! *i*) = *t-target* ((*butlast p′*) ! *j*)
using ‹*t-target* (*p′* ! *i*) = *t-target* (*p′* ! *j*)›
by (*simp add*: ‹*i* < *j*› *dual-order.strict-trans nth-butlast*)

have *path M q ?p*
and *target q ?p* = *target q p*
and *length ?p* < *length p*
and *path M* (*target q* (*take* (*Suc i*) *p*)) *?pCut*
and *target* (*target q* (*take* (*Suc i*) *p*)) *?pCut* = *target q* (*take* (*Suc i*) *p*)
using *path-loop-cut*[*OF* ‹*path M q p*› ‹*t-target* (*p* ! *i*) = *t-target* (*p* ! *j*)› ‹*i* < *j*› ‹*j* < *length p*›]
by *blast*+

have *path M′ q′ ?p′*
and *target q′ ?p′* = *target q′* (*butlast p′*)
and *length ?p′* < *length* (*butlast p′*)
using *path-loop-cut*[*OF* ‹*path M′ q′* (*butlast p′*)› ‹*t-target* ((*butlast p′*) ! *i*) = *t-target* ((*butlast p′*) ! *j*)› ‹*i* < *j*› ‹*j* < *length* (*butlast p′*)›]
by *blast*+

have *path M′ q′* (*?p′*@[*last p′*])
using ‹*t-source* (*last p′*) = *target q′* (*butlast p′*)›
using *path-append-transition*[*OF* ‹*path M′ q′ ?p′*› ‹(*last p′*) ∈ *transitions M′*›]
unfolding ‹*target q′ ?p′* = *target q′* (*butlast p′*)› **by** *simp*

have *p-io ?p′* = *p-io ?p*
using ‹*p-io p* = *butlast io*› ‹*p-io* (*butlast p′*) = *butlast io*›
by (*metis* (*no-types*, *lifting*) *drop-map map-append take-map*)

have *min-prop*: *length* (*p-io* (*?p′*@[*last p′*])) < *length io*
using ‹*length ?p′* < *length* (*butlast p′*)› ‹*p-io p′* = *io*›
unfolding *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*))]
by *auto*

have *q* ∈ *io-targets M* (*p-io pP*) (*initial M*)
using ‹*path M* (*initial M*) *pP*› ‹*target* (*initial M*) *pP* = *q*› **unfolding** *io-targets.simps*
by *blast*

have ((*p-io pP*) @ (*p-io ?p*)) ∈ *L M*
using *language-io-target-append*[*OF* ‹*q* ∈ *io-targets M* (*p-io pP*) (*initial M*)›, *of p-io ?p*]
‹*path M q ?p*›
unfolding *LS.simps* **by** *blast*
then have *p1*: ((*p-io pP*) @ *butlast* (*p-io* (*?p′* @ [*last p′*]))) ∈ *L M*
unfolding ‹*p-io ?p′* = *p-io ?p*›[*symmetric*]
by (*metis* (*no-types*, *lifting*) *butlast-snoc map-butlast*)

have *p2*: ((*p-io pP*) @ (*p-io* (*?p′* @ [*last p′*]))) ∉ *L M*
proof
assume ((*p-io pP*) @ (*p-io* (*?p′* @ [*last p′*]))) ∈ *L M*
then obtain *pCntr* **where** *path M* (*initial M*) *pCntr*
and *p-io pCntr* = (*p-io pP*) @ (*p-io* (*?p′* @ [*last p′*]))
by *auto*

let *?pCntr1* = (*take* (*length* (*p-io pP*)) *pCntr*)
let *?pCntr23* = (*drop* (*length* (*p-io pP*)) *pCntr*)

have *path M* (*initial M*) *?pCntr1*
and *p-io ?pCntr1* = *p-io pP*
and *path M* (*target* (*initial M*) *?pCntr1*) *?pCntr23*
and *p-io ?pCntr23* = *p-io* (*?p′* @ [*last p′*])
using *path-io-split*[*OF* ‹*path M* (*initial M*) *pCntr*› ‹*p-io pCntr* = (*p-io pP*) @ (*p-io* (*?p′* @ [*last p′*]))›]
by *blast+*

let *?pCntr2* = (*take* (*length* (*p-io* (*take* (*Suc i*) (*butlast p′*) @ *drop* (*Suc j*) (*butlast p′*)))) (*drop* (*length* (*p-io pP*)) *pCntr*))
let *?pCntr3* = (*drop* (*length* (*p-io* (*take* (*Suc i*) (*butlast p′*) @ *drop* (*Suc j*) (*butlast p′*)))) (*drop* (*length* (*p-io pP*)) *pCntr*))

have *p-io ?pCntr23* = *p-io ?p′* @ *p-io* [*last p′*]
using ‹*p-io ?pCntr23* = *p-io* (*?p′* @ [*last p′*])› **by** *auto*
have *path M* (*target* (*initial M*) *?pCntr1*) *?pCntr2*
and *p-io ?pCntr2* = *p-io ?p′*
and *path M* (*target* (*target* (*initial M*) *?pCntr1*) *?pCntr2*) *?pCntr3*
and *p-io ?pCntr3* = *p-io* [*last p′*]
using *path-io-split*[*OF* ‹*path M* (*target* (*initial M*) *?pCntr1*) *?pCntr23*› ‹*p-io ?pCntr23* = *p-io ?p′* @ *p-io* [*last p′*]›]
by *blast+*

have *?pCntr1* = *pP*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*initial M*) *?pCntr1*› ‹*path M* (*initial M*) *pP*› ‹*p-io ?pCntr1* = *p-io pP*›]
by *assumption*

then have (*target* (*initial M*) *?pCntr1*) = *q*
using ‹*target* (*initial M*) *pP* = *q*› **by** *auto*
then have *path M q ?pCntr2*
and *path M* (*target q ?pCntr2*) *?pCntr3*
using ‹*path M* (*target* (*initial M*) *?pCntr1*) *?pCntr2*›
‹*path M* (*target* (*target* (*initial M*) *?pCntr1*) *?pCntr2*) *?pCntr3*›
by *auto*

```
    have ?pCntr2 = ?p
      using observable-path-unique[OF ‹observable M› ‹path M q ?pCntr2› ‹path
M q ?p› ]
            ‹p-io ?pCntr2 = p-io ?p'›
      unfolding ‹p-io ?p' = p-io ?p›
      by blast
    then have (target q ?pCntr2) = (target q ?p)
      by auto
    then have (target q ?pCntr2) = (target q p)
      using ‹target q ?p = target q p› by auto

    have p-io ?pCntr3 = [last io]
      using ‹p-io ?pCntr3 = p-io [last p']›
        by (metis (mono-tags, lifting) ‹io ≠ []› assms(7) last-map list.simps(8)
list.simps(9))

    have path M (initial M) (pP @ p @ ?pCntr3)
     using ‹path M (initial M) pP› ‹target (initial M) pP = q› ‹path M q p› ‹path
M (target q ?pCntr2) ?pCntr3›
      unfolding ‹(target q ?pCntr2) = (target q p)›
      by auto
    moreover have p-io (pP @ p @ ?pCntr3) = ((p-io pP) @ io)
      using ‹p-io p = butlast io› ‹p-io ?pCntr3 = [last io]›
      by (simp add: ‹io ≠ []›)
    ultimately have ((p-io pP) @ io) ∈ L M
      by (metis (mono-tags, lifting) language-state-containment)
    then show False
      using ‹((p-io pP) @ io) ∉ L M›
      by simp
  qed

  have p3: ((p-io pP) @ (p-io (?p' @ [last p']))) ∈ L M'
   using language-io-target-append[OF ‹q' ∈ io-targets M' (p-io pP) (initial M')›,
of (p-io (?p' @ [last p']))]
    using ‹path M' q' (?p'@[last p'])›
    unfolding LS.simps
    by (metis (mono-tags, lifting) mem-Collect-eq)

  have sequence-to-failure-extending-preamble-path M M' PS pP (p-io (?p' @ [last
p']))
    unfolding sequence-to-failure-extending-preamble-path-def
    using ‹q ∈ states M›
          ‹(q,P) ∈ PS›
          ‹path P (FSM.initial P) pP›
          ‹target (FSM.initial P) pP = q›
          p1 p2 p3 by blast

  show False
    using ‹⋀ p' io' . sequence-to-failure-extending-preamble-path M M' PS p' io'
```

$\Longrightarrow$ *length io* $\leq$ *length io'*›[*OF* ‹*sequence-to-failure-extending-preamble-path M M' PS pP* (*p-io* (*?p'* @ [*last p'*]))›]
min-prop
by *simp*
qed

lemma *minimal-sequence-to-failure-extending-preamble-no-repetitions-with-other-preambles*
:
assumes *minimal-sequence-to-failure-extending-preamble-path M M' PS pP io*
and *observable M*
and *path M* (*target* (*initial M*) *pP*) *p*
and *p-io p* = *butlast io*
and *q'* $\in$ *io-targets M'* (*p-io pP*) (*initial M'*)
and *path M' q' p'*
and *p-io p'* = *io*
and $\bigwedge$ *q P*. (*q*, *P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*
and *i* < *length* (*butlast io*)
and (*t-target* (*p* ! *i*), *P'*) $\in$ *PS*
and *path P'* (*initial P'*) *pP'*
and *target* (*initial P'*) *pP'* = *t-target* (*p* ! *i*)
shows *t-target* (*p'* ! *i*) $\notin$ *io-targets M'* (*p-io pP'*) (*initial M'*)
proof
assume *t-target* (*p'* ! *i*) $\in$ *io-targets M'* (*p-io pP'*) (*FSM.initial M'*)

have *sequence-to-failure-extending-preamble-path M M' PS pP io*
and $\bigwedge$ *p' io'* . *sequence-to-failure-extending-preamble-path M M' PS p' io'* $\Longrightarrow$
length io $\leq$ *length io'*
using ‹*minimal-sequence-to-failure-extending-preamble-path M M' PS pP io*›
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast+*

obtain *q P* **where** (*q*,*P*) $\in$ *PS*
and *path P* (*initial P*) *pP*
and *target* (*initial P*) *pP* = *q*
and ((*p-io pP*) @ *butlast io*) $\in$ *L M*
and ((*p-io pP*) @ *io*) $\notin$ *L M*
and ((*p-io pP*) @ *io*) $\in$ *L M'*

using ‹*sequence-to-failure-extending-preamble-path M M' PS pP io*›
unfolding *sequence-to-failure-extending-preamble-path-def*
by *blast*

have *is-preamble P M q*
using ‹(*q*,*P*) $\in$ *PS*› ‹$\bigwedge$ *q P*. (*q*, *P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*› **by** *blast*
then have *q* $\in$ *states M*
unfolding *is-preamble-def*

by (*metis* ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› *path-target-is-state submachine-path*)

have *initial P* = *initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *pP*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P* (*FSM.initial P*) *pP*› **by** *blast*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*› **unfolding** ‹*initial P* = *initial M*› **by** *assumption*

then have *path M q p*
using ‹*path M* (*target* (*initial M*) *pP*) *p*› **by** *auto*

have *is-preamble P′ M* (*t-target* (*p* ! *i*))
using ‹(*t-target* (*p* ! *i*), *P′*) ∈ *PS*› ‹⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*› **by** *blast*
then have (*t-target* (*p* ! *i*)) ∈ *states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P′* (*initial P′*) *pP′*› ‹*target* (*initial P′*) *pP′* = *t-target* (*p* ! *i*)› *path-target-is-state submachine-path*)

have *initial P′* = *initial M*
using ‹*is-preamble P′ M* (*t-target* (*p* ! *i*))› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *pP′*
using ‹*is-preamble P′ M* (*t-target* (*p* ! *i*))› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P′* (*initial P′*) *pP′*› **by** *blast*
have *target* (*initial M*) *pP′* = *t-target* (*p* ! *i*)
using ‹*target* (*initial P′*) *pP′* = *t-target* (*p* ! *i*)› **unfolding** ‹*initial P′* = *initial M*› **by** *simp*

have *io* ≠ []
using ‹((*p-io pP*) @ *butlast io*) ∈ *L M*› ‹((*p-io pP*) @ *io*) ∉ *L M*› **by** *auto*
then have *length p′* > *0*
using ‹*p-io p′* = *io*› **by** *auto*
then have *p′* = (*butlast p′*)@[*last p′*]
by *auto*
then have *path M′ q′* ((*butlast p′*)@[*last p′*])
using ‹*path M′ q′ p′*› **by** *simp*
then have *path M′ q′* (*butlast p′*) **and** (*last p′*) ∈ *transitions M′* **and** *t-source* (*last p′*) = *target q′* (*butlast p′*)
by *auto*

have *p-io* (*butlast p′*) = *butlast io*
using ‹*p′* = (*butlast p′*)@[*last p′*]› ‹*p-io p′* = *io*›
using *map-butlast* **by** *auto*

```
  have butlast io ≠ []
    using assms(9) by fastforce

  let ?p = (drop (Suc i) p)
  let ?p' = (drop (Suc i) (butlast p'))

  have i < length p
     using ‹i < length (butlast io)› unfolding ‹p-io p = butlast io›[symmetric]
length-map[of (λ t . (t-input t, t-output t))]
    by assumption
  then have p ! i = last (take (Suc i) p)
    by (simp add: take-last-index)
  then have t-target (p ! i) = target q (take (Suc i) p)
    unfolding target.simps visited-states.simps
   by (metis (no-types, lifting) ‹i < length p› gr-implies-not0 last-ConsR length-0-conv
length-map nth-map old.nat.distinct(2) take-eq-Nil take-last-index take-map)

  have p = (take (Suc i) p @ ?p)
    by simp
  then have p-io p = (p-io (take (Suc i) p)) @ (p-io ?p)
    by (metis map-append)
  have (length (p-io (take (Suc i) p))) = Suc i
    using ‹i < length p›
    unfolding length-map[of (λ t . (t-input t, t-output t))]
    by auto

  have path M (t-target (p ! i)) ?p
   using path-io-split(3)[OF ‹path M q p› ‹p-io p = (p-io (take (Suc i) p)) @ (p-io
?p)›]
     unfolding ‹(length (p-io (take (Suc i) p))) = Suc i› ‹t-target (p ! i) = target
q (take (Suc i) p)›
    by assumption
  then have path M (initial M) (pP' @ ?p)
    using ‹path M (initial M) pP'› ‹target (initial M) pP' = t-target (p ! i)›
    by (simp add: path-append)

  let ?io = (p-io ?p) @ [last io]
  have is-shorter: length ?io < length io
  proof −
    have p-io ?p = drop (Suc i) (butlast io)
      by (metis assms(4) drop-map)
    moreover have length (drop (Suc i) (butlast io)) < length (butlast io)
      using assms(9) by auto
    ultimately have length (p-io ?p) < length (butlast io)
      by simp
    then show ?thesis
      by auto
  qed
```

have *p1*: $((p\text{-}io\ pP')\ @\ (p\text{-}io\ ?p)) \in L\ M$
using ‹*path M* (*initial M*) (*pP′* @ *?p*)›
by (*metis* (*mono-tags*, *lifting*) *language-state-containment map-append*)

have *p2*: $((p\text{-}io\ pP')\ @\ ?io) \notin L\ M$
proof
assume $((p\text{-}io\ pP')\ @\ ?io) \in L\ M$
then obtain *pCntr* **where** *path M* (*initial M*) *pCntr* **and** *p-io pCntr* = (*p-io pP′*) @ (*p-io ?p*) @ [*last io*]
by *auto*

let *?pCntr1* = (*take* (*length* (*p-io pP′*)) *pCntr*)
let *?pCntr23* = (*drop* (*length* (*p-io pP′*)) *pCntr*)

have *path M* (*initial M*) *?pCntr1*
and *p-io ?pCntr1* = *p-io pP′*
and *path M* (*target* (*initial M*) *?pCntr1*) *?pCntr23*
and *p-io ?pCntr23* = (*p-io ?p*) @ [*last io*]
using *path-io-split*[*OF* ‹*path M* (*initial M*) *pCntr*› ‹*p-io pCntr* = (*p-io pP′*) @ (*p-io ?p*) @ [*last io*]›]
by *blast+*

have *?pCntr1* = *pP′*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*initial M*) *?pCntr1*› ‹*path M* (*initial M*) *pP′*› ‹*p-io ?pCntr1* = *p-io pP′*›]
by *assumption*
then have (*target* (*initial M*) *?pCntr1*) = (*t-target* (*p* ! *i*))
using ‹*target* (*initial M*) *pP′* = (*t-target* (*p* ! *i*))› **by** *auto*
then have *path M* (*t-target* (*p* ! *i*)) *?pCntr23*
using ‹*path M* (*target* (*initial M*) *?pCntr1*) *?pCntr23*›
by *simp*

have *path M q* (*take* (*Suc i*) *p*)
using ‹*path M q p*›
by (*metis append-take-drop-id path-prefix*)

then have *path M q* ((*take* (*Suc i*) *p*) @ *?pCntr23*)
using ‹*path M* (*target* (*initial M*) *?pCntr1*) *?pCntr23*›
unfolding ‹(*target* (*initial M*) *?pCntr1*) = (*t-target* (*p* ! *i*))›
unfolding ‹*t-target* (*p* ! *i*) = *target q* (*take* (*Suc i*) *p*)›
by *auto*
then have *path M* (*initial M*) (*pP* @ ((*take* (*Suc i*) *p*) @ *?pCntr23*))
using ‹*path M* (*initial M*) *pP*› ‹*target* (*initial M*) *pP* = *q*›
by *auto*

moreover have *p-io* (*pP* @ ((*take* (*Suc i*) *p*) @ *?pCntr23*)) = *p-io pP* @ *io*
using ‹*io* ≠ []› ‹*p-io* (*drop* (*length* (*p-io pP′*)) *pCntr*) = *p-io* (*drop* (*Suc i*) *p*) @ [*last io*]› ‹*p-io p* = *p-io* (*take* (*Suc i*) *p*) @ *p-io* (*drop* (*Suc i*) *p*)› *append-butlast-last-id assms*(*4*)

by *fastforce*

ultimately have (*p-io pP* @ *io*) ∈ *L M*
by (*metis* (*mono-tags*, *lifting*) *language-state-containment*)

then show *False*
using ‹(*p-io pP* @ *io*) ∉ *L M*›
by *simp*
qed

have *p3*: ((*p-io pP′*) @ *?io*) ∈ *L M′*
proof −
have *i* < *length* (*butlast p′*)
using ‹*i* < *length* (*butlast io*)› **unfolding** ‹*p-io p′* = *io*›[*symmetric*]
using *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*))]
by *simp*
then have *butlast p′* ! *i* = *last* (*take* (*Suc i*) (*butlast p′*))
by (*simp add*: *nth-butlast take-last-index*)
moreover have (*take* (*Suc i*) (*butlast p′*)) ≠ []
by (*metis Zero-not-Suc* ‹*i* < *length* (*butlast p′*)› *list.size*(*3*) *not-less0 take-eq-Nil*)

ultimately have (*target q′* (*take* (*Suc i*) (*butlast p′*))) = *t-target* ((*butlast p′*) ! *i*)
unfolding *target.simps visited-states.simps*
by (*simp add*: *last-map*)
moreover have (*butlast p′*) ! *i* = *p′* ! *i*
using ‹*i* < *length* (*butlast p′*)›
by (*simp add*: *nth-butlast*)
ultimately have (*target q′* (*take* (*Suc i*) (*butlast p′*))) = *t-target* (*p′* ! *i*)
by *simp*

have *p′* = (*take* (*Suc i*) (*butlast p′*)) @ *?p′* @ [*last p′*]
by (*metis* ‹*p′* = *butlast p′* @ [*last p′*]› *append.assoc append-take-drop-id*)
then have *path M′* (*target q′* (*take* (*Suc i*) (*butlast p′*))) (*?p′* @ [*last p′*])
by (*metis assms*(*6*) *path-suffix*)
then have *path M′* (*t-target* (*p′* ! *i*)) (*?p′* @ [*last p′*])
unfolding ‹(*target q′* (*take* (*Suc i*) (*butlast p′*))) = *t-target* (*p′* ! *i*)› **by** *assumption*
moreover have *p-io* (*?p′* @ [*last p′*]) = *?io*
by (*metis* (*no-types*, *lifting*) ‹*io* ≠ []› ‹*p′* = *butlast p′* @ [*last p′*]› ‹*p-io* (*butlast p′*) = *butlast io*› *append-butlast-last-id assms*(*4*) *assms*(*7*) *drop-map map-append same-append-eq*)
ultimately have *?io* ∈ *LS M′* (*t-target* (*p′* ! *i*))
by (*metis* (*mono-tags*, *lifting*) *language-state-containment*)

show ((*p-io pP′*) @ *?io*) ∈ *L M′*
using *language-io-target-append*[*OF* ‹*t-target* (*p′* ! *i*) ∈ *io-targets M′* (*p-io pP′*) (*FSM.initial M′*)› ‹*?io* ∈ *LS M′* (*t-target* (*p′* ! *i*))›]
by *assumption*

qed

have $*$: $\bigwedge$ *xs x* . *butlast* (*xs* @ [*x*]) = *xs* **by** *auto*

have *sequence-to-failure-extending-preamble-path M M′ PS pP′ ?io*
unfolding *sequence-to-failure-extending-preamble-path-def*
using ‹*t-target* (*p* ! *i*) ∈ *states M*› *assms*(*10*−*12*) *p1 p2 p3*
unfolding $*$ **by** *blast*

then have *length io* ≤ *length ?io*
using ‹$\bigwedge$ *p′ io′* . *sequence-to-failure-extending-preamble-path M M′ PS p′ io′* ⟹ *length io* ≤ *length io′*›
by *blast*

then show *False*
using *is-shorter*
by *simp*
qed

end

37 Helper Algorithms

This theory contains several algorithms used to calculate components of a test suite.

theory *Helper-Algorithms*
imports *State-Separator State-Preamble*
begin

37.1 Calculating r-distinguishable State Pairs with Separators

definition *r-distinguishable-state-pairs-with-separators* ::
(′*a*::*linorder*,′*b*::*linorder*,′*c*) *fsm* ⇒ ((′*a* × ′*a*) × ((′*a* × ′*a*) + ′*a*,′*b*,′*c*) *fsm*) *set*
where
r-distinguishable-state-pairs-with-separators M =
{((*q1*,*q2*),*Sep*) | *q1 q2 Sep* . *q1* ∈ *states M*
∧ *q2* ∈ *states M*
∧ ((*q1* < *q2* ∧ *state-separator-from-s-states M q1 q2* = *Some Sep*)
∨ (*q2* < *q1* ∧ *state-separator-from-s-states M q2 q1* = *Some Sep*)) }

lemma *r-distinguishable-state-pairs-with-separators-alt-def* :
r-distinguishable-state-pairs-with-separators M =
⋃ (*image* (λ ((*q1*,*q2*),*A*) . {((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)})
(*Set.filter* (λ (*qq*,*A*) . *A* ≠ *None*)

(*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))))
(**is** *?P1* = *?P2*)
proof −
have ⋀ *x* . *x* ∈ *?P1* ⟹ *x* ∈ *?P2*
proof −
fix *x* **assume** *x* ∈ *?P1*
then obtain *q1 q2 A* **where** *x* = ((*q1*,*q2*),*A*)
by (*metis eq-snd-iff*)
then have ((*q1*,*q2*),*A*) ∈ *?P1* **using** ‹*x* ∈ *?P1*› **by** *auto*
then have *q1* ∈ *states M*
and *q2* ∈ *states M*
and ((*q1* < *q2* ∧ *state-separator-from-s-states M q1 q2* = *Some A*)
∨ (*q2* < *q1* ∧ *state-separator-from-s-states M q2 q1* = *Some A*))
unfolding *r-distinguishable-state-pairs-with-separators-def* **by** *blast+*

then consider (*a*) *q1* < *q2* ∧ *state-separator-from-s-states M q1 q2* = *Some A* |
(*b*) *q2* < *q1* ∧ *state-separator-from-s-states M q2 q1* = *Some A*
by *blast*
then show *x* ∈ *?P2*
using ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› **unfolding** ‹*x* = ((*q1*,*q2*),*A*)› **by** (*cases*; *force*)
qed
moreover have ⋀ *x* . *x* ∈ *?P2* ⟹ *x* ∈ *?P1*
proof −
fix *x* **assume** *x* ∈ *?P2*
then obtain *q1 q2 A* **where** *x* = ((*q1*,*q2*),*A*)
by (*metis eq-snd-iff*)
then have ((*q1*,*q2*),*A*) ∈ *?P2* **using** ‹*x* ∈ *?P2*› **by** *auto*
then obtain *q1′ q2′ A′* **where** ((*q1*,*q2*),*A*) ∈ {((*q1′*,*q2′*),*the A′*),((*q2′*,*q1′*),*the A′*)}
and *A′* ≠ *None*
and ((*q1′*,*q2′*), *A′*) ∈ (*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))
by *force*

then have *A′* = *Some A*
by (*metis* (*no-types*, *lifting*) *empty-iff insert-iff old.prod.inject option.collapse*)

moreover have *A′* = *state-separator-from-s-states M q1′ q2′*
and *q1′* < *q2′*
and *q1′* ∈ *states M*
and *q2′* ∈ *states M*

```
    using ‹((q1',q2'), A') ∈ (image (λ (q1,q2) . ((q1,q2),state-separator-from-s-states
M q1 q2))
                                (Set.filter (λ (q1,q2) . q1 < q2) (states M × states
M)))›
      by force+
    ultimately have state-separator-from-s-states M q1' q2' = Some A by simp

    consider ((q1',q2'),the A') = ((q1,q2),A) | ((q1',q2'),the A') = ((q2,q1),A)
      using ‹((q1,q2),A) ∈ {((q1',q2'),the A'),((q2',q1'),the A')}›
      by force
    then show x ∈ ?P1
    proof cases
      case 1
      then have ∗: q1' = q1 and ∗∗: q2' = q2 by auto

      show ?thesis
      using ‹q1' ∈ states M› ‹q2' ∈ states M› ‹q1' < q2'› ‹state-separator-from-s-states
M q1' q2' = Some A›
        unfolding r-distinguishable-state-pairs-with-separators-def
        unfolding ∗ ∗∗ ‹x = ((q1,q2),A)› by blast
    next
      case 2
      then have ∗: q1' = q2 and ∗∗: q2' = q1 by auto

      show ?thesis
      using ‹q1' ∈ states M› ‹q2' ∈ states M› ‹q1' < q2'› ‹state-separator-from-s-states
M q1' q2' = Some A›
        unfolding r-distinguishable-state-pairs-with-separators-def
        unfolding ∗ ∗∗ ‹x = ((q1,q2),A)› by blast
    qed
  qed
  ultimately show ?thesis by blast
qed


lemma r-distinguishable-state-pairs-with-separators-code[code] :
  r-distinguishable-state-pairs-with-separators M =
    set (concat (map
                (λ ((q1,q2),A) . [((q1,q2),the A),((q2,q1),the A)])
                (filter (λ (qq,A) . A ≠ None)
                      (map (λ (q1,q2) . ((q1,q2),state-separator-from-s-states M q1
q2))
                            (filter (λ (q1,q2) . q1 < q2)
                                (List.product(states-as-list M) (states-as-list M)))))))
  (is r-distinguishable-state-pairs-with-separators M = ?C2)
proof −
  let ?C1 = ⋃ (image (λ ((q1,q2),A) . {((q1,q2),the A),((q2,q1),the A)})
                    (Set.filter (λ (qq,A) . A ≠ None)
                          (image (λ (q1,q2) . ((q1,q2),state-separator-from-s-states
```

M q1 q2))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))))

have *r-distinguishable-state-pairs-with-separators M* = *?C1*
using *r-distinguishable-state-pairs-with-separators-alt-def* **by** *assumption*
also have ... = *?C2*
proof
show *?C1* ⊆ *?C2*
proof
fix *x* **assume** *x* ∈ *?C1*
then obtain *q1 q2 A* **where** *x* = ((*q1*,*q2*),*A*)
by (*metis eq-snd-iff*)
then have ((*q1*,*q2*),*A*) ∈ *?C1* **using** ‹*x* ∈ *?C1*› **by** *auto*
then obtain *q1′ q2′ A′* **where** ((*q1*,*q2*),*A*) ∈ {((*q1′*,*q2′*),*the A′*),((*q2′*,*q1′*),*the A′*)}
and *A′* ≠ *None*
and ((*q1′*,*q2′*), *A′*) ∈ (*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))
by *force*

then have *A′* = *Some A*
by (*metis* (*no-types*, *lifting*) *empty-iff insert-iff old.prod.inject option.collapse*)

moreover have *A′* = *state-separator-from-s-states M q1′ q2′*
and *q1′* < *q2′*
and *q1′* ∈ *states M*
and *q2′* ∈ *states M*
using ‹((*q1′*,*q2′*), *A′*) ∈ (*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))›
by *force+*
ultimately have *state-separator-from-s-states M q1′ q2′* = *Some A*
and (*q1′*,*q2′*) ∈ *set* (*filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*List.product*(*states-as-list M*) (*states-as-list M*)))
unfolding *states-as-list-set*[*symmetric*] **by** *auto*

then have ((*q1′*,*q2′*),*A′*) ∈ *set* (*filter* (λ (*qq*,*A*) . *A* ≠ *None*)
(*map* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*filter* (λ (*q1*,*q2*) . *q1* < *q2*)
(*List.product*(*states-as-list M*) (*states-as-list M*)))))
using ‹*A′* = *state-separator-from-s-states M q1′ q2′*› ‹*A′* = *Some A*› **by** *force*

have *scheme1*: $\bigwedge$ *f xs x* . *x* $\in$ *set xs* $\Longrightarrow$ *f x* $\in$ *set* (*map f xs*) **by** *auto*
have *scheme2*: $\bigwedge$ *x xs xss* . *xs* $\in$ *set xss* $\Longrightarrow$ *x* $\in$ *set xs* $\Longrightarrow$ *x* $\in$ *set* (*concat xss*) **by** *auto*
have $*$:[((*q1'*,*q2'*),*the A'*),((*q2'*,*q1'*),*the A'*)] $\in$
set (*map* (λ ((*q1*,*q2*),*A*) . [((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)])
(*filter* (λ (*qq*,*A*) . *A* $\neq$ *None*)
(*map* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*filter* (λ (*q1*,*q2*) . *q1* < *q2*)
(*List.product*(*states-as-list M*) (*states-as-list M*))))))
using *scheme1*[*OF* ‹((*q1'*,*q2'*),*A'*) $\in$ *set* (*filter* (λ (*qq*,*A*) . *A* $\neq$ *None*) (*map* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*)) (*filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*List.product*(*states-as-list M*) (*states-as-list M*)))))›, *of* λ ((*q1'*, *q2'*), *A'*) . [((*q1'*,*q2'*),*the A'*),((*q2'*,*q1'*),*the A'*)]]
by *force*
have $**$: ((*q1*,*q2*),*A*) $\in$ *set* [((*q1'*,*q2'*),*the A'*),((*q2'*,*q1'*),*the A'*)]
using ‹((*q1*,*q2*),*A*) $\in$ {((*q1'*,*q2'*),*the A'*),((*q2'*,*q1'*),*the A'*)}› **by** *auto*

show *x* $\in$ *?C2*
unfolding ‹*x* = ((*q1*,*q2*),*A*)› **using** *scheme2*[*OF* $*$ $**$] **by** *assumption*
qed

show *?C2* $\subseteq$ *?C1*
proof
fix *x* **assume** *x* $\in$ *?C2*
obtain *q1q2A* **where** *x* $\in$ *set* ((λ ((*q1'*, *q2'*), *A'*) . [((*q1'*,*q2'*),*the A'*),((*q2'*,*q1'*),*the A'*)]) *q1q2A*)
and *q1q2A* $\in$ *set* (*filter* (λ (*qq*,*A*) . *A* $\neq$ *None*)
(*map* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*filter* (λ (*q1*,*q2*) . *q1* < *q2*)
(*List.product*(*states-as-list M*) (*states-as-list M*)))))
using *concat-map-elem*[*OF* ‹*x* $\in$ *?C2*›] **by** *blast*

moreover obtain *q1 q2 A* **where** *q1q2A* = ((*q1*,*q2*),*A*)
by (*metis prod.collapse*)

ultimately have *x* $\in$ *set* [((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)]
and ((*q1*,*q2*),*A*) $\in$ *set* (*filter* (λ (*qq*,*A*) . *A* $\neq$ *None*)
(*map* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*filter* (λ (*q1*,*q2*) . *q1* < *q2*)
(*List.product*(*states-as-list M*) (*states-as-list M*)))))
by *force+*

then have A = *state-separator-from-s-states M q1 q2*
and $A \neq$ *None*
and (*q1*,*q2*) ∈ *set* (*filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*List.product*(*states-as-list M*) (*states-as-list M*)))
by *auto*

then have *q1* < *q2* **and** *q1* ∈ *states M* **and** *q2* ∈ *states M*
unfolding *states-as-list-set*[*symmetric*] **by** *auto*
then have (*q1*,*q2*) ∈ *Set.filter* (λ(*q1*, *q2*). *q1* < *q2*) (*FSM.states M* × *FSM.states M*)
by *auto*
then have ((*q1*,*q2*),*A*) ∈ (*Set.filter* (λ (*qq*,*A*) . *A* ≠ *None*)
(*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*))))
using ‹*A* ≠ *None*› **unfolding** ‹*A* = *state-separator-from-s-states M q1 q2*› **by** *auto*
then have {((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)} ∈
(*image* (λ ((*q1*,*q2*),*A*) . {((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)})
(*Set.filter* (λ (*qq*,*A*) . *A* ≠ *None*)
(*image* (λ (*q1*,*q2*) . ((*q1*,*q2*),*state-separator-from-s-states M q1 q2*))
(*Set.filter* (λ (*q1*,*q2*) . *q1* < *q2*) (*states M* × *states M*)))))
by (*metis* (*no-types*, *lifting*) ‹*q1q2A* = ((*q1*, *q2*), *A*)› *case-prod-conv image-iff*)
then show *x* ∈ *?C1*
using ‹*x* ∈ *set* [((*q1*,*q2*),*the A*),((*q2*,*q1*),*the A*)]›
by (*metis* (*no-types*, *lifting*) *UnionI list.simps*(*15*) *set-empty2*)
qed
qed

finally show *?thesis* **.**
qed

lemma *r-distinguishable-state-pairs-with-separators-same-pair-same-separator* :
assumes ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M*
and ((*q1*,*q2*),*A′*) ∈ *r-distinguishable-state-pairs-with-separators M*
shows $A = A'$
using *assms* **unfolding** *r-distinguishable-state-pairs-with-separators-def*
by *force*

lemma *r-distinguishable-state-pairs-with-separators-sym-pair-same-separator* :
assumes ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M*
and ((*q2*,*q1*),*A′*) ∈ *r-distinguishable-state-pairs-with-separators M*
shows $A = A'$
using *assms* **unfolding** *r-distinguishable-state-pairs-with-separators-def*

by *force*

lemma *r-distinguishable-state-pairs-with-separators-elem-is-separator*:
assumes $((q1,q2),A) \in$ *r-distinguishable-state-pairs-with-separators M*
and *observable M*
and *completely-specified M*
shows *is-separator M q1 q2 A* (*Inr q1*) (*Inr q2*)
proof −
have $*$:$q1 \in$ *states M*
and $**$:$q2 \in$ *states M*
and $***$:$q1 \neq q2$
and $****$: $q2 \neq q1$
and $*****$: *state-separator-from-s-states M q1 q2* = *Some A* ∨ *state-separator-from-s-states M q2 q1* = *Some A*
using *assms*(*1*) **unfolding** *r-distinguishable-state-pairs-with-separators-def* **by** *auto*

from $*****$ **have** *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 A*
∨ *is-state-separator-from-canonical-separator* (*canonical-separator M q2 q1*) *q2 q1 A*
using *state-separator-from-s-states-soundness*[*of M q1 q2 A*, *OF* - $*$ $**$ *assms*(*3*)]
using *state-separator-from-s-states-soundness*[*of M q2 q1 A*, *OF* - $**$ $*$ *assms*(*3*)]
by *auto*
then show *?thesis*
using *state-separator-from-canonical-separator-is-separator*[*of M q1 q2 A*, *OF* - ‹*observable M*› $*$ $**$ $***$]
using *state-separator-from-canonical-separator-is-separator*[*of M q2 q1 A*, *OF* - ‹*observable M*› $**$ $*$ $****$]
using *is-separator-sym*[*of M q2 q1 A Inr q2 Inr q1*] **by** *auto*
qed

37.2 Calculating Pairwise r-distinguishable Sets of States

definition *pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set set* **where**
pairwise-r-distinguishable-state-sets-from-separators M
= { *S* . *S* ⊆ *states M* ∧ (∀ *q1* ∈ *S* . ∀ *q2* ∈ *S* . *q1* ≠ *q2* ⟶ (*q1*,*q2*) ∈ *image fst* (*r-distinguishable-state-pairs-with-separators M*))}

definition *pairwise-r-distinguishable-state-sets-from-separators-list* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ '*a set list* **where**
pairwise-r-distinguishable-state-sets-from-separators-list M =
(*let RDS* = *image fst* (*r-distinguishable-state-pairs-with-separators M*)
in filter (λ *S* . ∀ *q1* ∈ *S* . ∀ *q2* ∈ *S* . *q1* ≠ *q2* ⟶ (*q1*,*q2*) ∈ *RDS*)
(*map set* (*pow-list* (*states-as-list M*))))

lemma *pairwise-r-distinguishable-state-sets-from-separators-code*[*code*] :
pairwise-r-distinguishable-state-sets-from-separators M = *set* (*pairwise-r-distinguishable-state-sets-from-separ*
M)
using *pow-list-set*[*of states-as-list M*]
unfolding *states-as-list-set*[*of M*]
pairwise-r-distinguishable-state-sets-from-separators-def
pairwise-r-distinguishable-state-sets-from-separators-list-def
by *auto*

lemma *pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes $q \in$ *states M*
shows $\exists$ $S \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) . $q \in S$
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def* **using** *assms*
by *blast*

definition *maximal-pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*)
fsm $\Rightarrow$ '*a set set* **where**
maximal-pairwise-r-distinguishable-state-sets-from-separators M
= { *S* . *S* $\in$ (*pairwise-r-distinguishable-state-sets-from-separators M*)
$\wedge$ ($\nexists$ *S'* . *S'* $\in$ (*pairwise-r-distinguishable-state-sets-from-separators M*)
$\wedge$ $S \subset S'$)}

definition *maximal-pairwise-r-distinguishable-state-sets-from-separators-list* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*)
fsm $\Rightarrow$ '*a set list* **where**
maximal-pairwise-r-distinguishable-state-sets-from-separators-list M =
remove-subsets (*pairwise-r-distinguishable-state-sets-from-separators-list M*)

lemma *maximal-pairwise-r-distinguishable-state-sets-from-separators-code*[*code*] :
maximal-pairwise-r-distinguishable-state-sets-from-separators M
= *set* (*maximal-pairwise-r-distinguishable-state-sets-from-separators-list M*)
unfolding *maximal-pairwise-r-distinguishable-state-sets-from-separators-list-def*
Let-def remove-subsets-set pairwise-r-distinguishable-state-sets-from-separators-code[*symmetric*]

maximal-pairwise-r-distinguishable-state-sets-from-separators-def
by *blast*

lemma *maximal-pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes $q \in$ *states M*
shows $\exists$ $S \in$ (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*).
$q \in S$
proof −

have $*$: $\{q\} \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*)
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def* **using** *assms* **by** *blast*
have $**$: *finite* (*pairwise-r-distinguishable-state-sets-from-separators M*)
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def* **by** (*simp add*: *fsm-states-finite*)

have (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*) =
$\{S \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*).
$\neg(\exists\ S' \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) . $S \subset S')\}$
unfolding *maximal-pairwise-r-distinguishable-state-sets-from-separators-def*
pairwise-r-distinguishable-state-sets-from-separators-def
by *metis*
then have (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*) =
$\{S \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) .
$(\forall\ S' \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) . $\neg\ S \subset S')\}$
by *blast*
moreover have $\exists\ S \in \{S \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) .
$(\forall\ S' \in$ (*pairwise-r-distinguishable-state-sets-from-separators M*) . $\neg\ S \subset S')\}$. $q \in S$
using *maximal-set-cover*[*OF* $** *$]
by *blast*
ultimately show *?thesis*
by *blast*
qed

37.3 Calculating d-reachable States with Preambles

definition *d-reachable-states-with-preambles* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* $\Rightarrow$ ('*a* $\times$ ('*a*,'*b*,'*c*) *fsm*) *set* **where**
d-reachable-states-with-preambles M =
image (λ *qp* . (*fst qp*, *the* (*snd qp*)))
(*Set.filter* (λ *qp* . *snd qp* $\neq$ *None*)
(*image* (λ *q* . (*q*, *calculate-state-preamble-from-input-choices M q*))
(*states M*)))

lemma *d-reachable-states-with-preambles-exhaustiveness* :
assumes $\exists$ *P* . *is-preamble P M q*
and *q* $\in$ *states M*
shows $\exists$ *P* . (*q*,*P*) $\in$ (*d-reachable-states-with-preambles M*)
using *calculate-state-preamble-from-input-choices-exhaustiveness*[*OF assms*(*1*)] *assms*(*2*)
unfolding *d-reachable-states-with-preambles-def* **by** *force*

lemma *d-reachable-states-with-preambles-soundness* :
assumes $(q,P) \in$ (*d-reachable-states-with-preambles M*)
and *observable M*
shows *is-preamble P M q*
and $q \in$ *states M*
using *assms*(*1*) *calculate-state-preamble-from-input-choices-soundness*[*of M q P*]
by (*auto simp add*: *d-reachable-states-with-preambles-def split*: *if-splits*)

37.4 Calculating Repetition Sets

Repetition sets are sets of tuples each containing a maximal set of pairwise r-distinguishable states and the subset of those states that have a preamble.

definition *maximal-repetition-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ ('*a set* × '*a set*) *set* **where**
maximal-repetition-sets-from-separators M
= {(*S*, *S* ∩ (*image fst* (*d-reachable-states-with-preambles M*))) | *S* .
S ∈ (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*)}

definition *maximal-repetition-sets-from-separators-list-naive* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* ⇒ ('*a set* × '*a set*) *list* **where**
maximal-repetition-sets-from-separators-list-naive M
= (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
in map (λ *S* . (*S*, *S* ∩ *DR*)) (*maximal-pairwise-r-distinguishable-state-sets-from-separators-list M*))

lemma *maximal-repetition-sets-from-separators-code*[*code*]:
maximal-repetition-sets-from-separators M = (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
in image (λ *S* . (*S*, *S* ∩ *DR*)) (*maximal-pairwise-r-distinguishable-state-sets-from-separators M*))
unfolding *maximal-repetition-sets-from-separators-def Let-def* **by** *force*

lemma *maximal-repetition-sets-from-separators-code-alt*:
maximal-repetition-sets-from-separators M = *set* (*maximal-repetition-sets-from-separators-list-naive M*)
unfolding *maximal-repetition-sets-from-separators-def*
maximal-repetition-sets-from-separators-list-naive-def
maximal-pairwise-r-distinguishable-state-sets-from-separators-code
by *force*

37.4.1 Calculating Sub-Optimal Repetition Sets

Finding maximal pairwise r-distinguishable subsets of the state set of some FSM is likely too expensive for FSMs containing a large number of r-distinguishable pairs of states. The following functions calculate only subset of all repetition sets while maintaining the property that every state is con-

tained in some repetition set.

fun *extend-until-conflict* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *list* **where**
extend-until-conflict non-confl-set candidates xs 0 = xs |
extend-until-conflict non-confl-set candidates xs (*Suc k*) = (*case dropWhile* (λ *x* . *find* (λ *y* . (*x*,*y*) $\notin$ *non-confl-set*) *xs* $\neq$ *None*) *candidates of*
[] $\Rightarrow$ *xs* |
(*c*#*cs*) $\Rightarrow$ *extend-until-conflict non-confl-set cs* (*c*#*xs*) *k*)

lemma *extend-until-conflict-retainment* :
assumes *x* $\in$ *set xs*
shows *x* $\in$ *set* (*extend-until-conflict non-confl-set candidates xs k*)
using *assms* **proof** (*induction k arbitrary*: *candidates xs*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
then show *?case* **proof** (*cases dropWhile* (λ *x* . *find* (λ *y* . (*x*,*y*) $\notin$ *non-confl-set*) *xs* $\neq$ *None*) *candidates*)
case *Nil*
then show *?thesis*
by (*metis Suc.prems extend-until-conflict.simps*(*2*) *list.simps*(*4*))
next
case (*Cons c cs*)
then show *?thesis*
by (*simp add*: *Suc.IH Suc.prems*)
qed
qed

lemma *extend-until-conflict-elem* :
assumes *x* $\in$ *set* (*extend-until-conflict non-confl-set candidates xs k*)
shows *x* $\in$ *set xs* $\vee$ *x* $\in$ *set candidates*
using *assms* **proof** (*induction k arbitrary*: *candidates xs*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)
then show *?case* **proof** (*cases dropWhile* (λ *x* . *find* (λ *y* . (*x*,*y*) $\notin$ *non-confl-set*) *xs* $\neq$ *None*) *candidates*)
case *Nil*
then show *?thesis*
by (*metis Suc.prems extend-until-conflict.simps*(*2*) *list.simps*(*4*))
next
case (*Cons c cs*)
then have *extend-until-conflict non-confl-set candidates xs* (*Suc k*) = *extend-until-conflict non-confl-set cs* (*c*#*xs*) *k*
by *auto*
then have *x* $\in$ *set* (*c* # *xs*) $\vee$ *x* $\in$ *set cs*
using *Suc.IH*[*of cs* (*c*#*xs*)] *Suc.prems* **by** *auto*

```
    moreover have set (c#cs) ⊆ set candidates
      using Cons by (metis set-dropWhileD subsetI)
    ultimately show ?thesis
      using set-ConsD by auto
  qed
qed


lemma extend-until-conflict-no-conflicts :
  assumes x ∈ set (extend-until-conflict non-confl-set candidates xs k)
  and     y ∈ set (extend-until-conflict non-confl-set candidates xs k)
  and     x ∈ set xs ⟹ y ∈ set xs ⟹ (x,y) ∈ non-confl-set ∨ (y,x) ∈ non-confl-set

  and     x ≠ y
shows (x,y) ∈ non-confl-set ∨ (y,x) ∈ non-confl-set
using assms proof (induction k arbitrary: candidates xs)
  case 0
  then show ?case by auto
next
  case (Suc k)
  then show ?case proof (cases dropWhile (λ x . find (λ y . (x,y) ∉ non-confl-set)
xs ≠ None) candidates)
    case Nil
    then have extend-until-conflict non-confl-set candidates xs (Suc k) = xs
      by (metis extend-until-conflict.simps(2) list.simps(4))
    then show ?thesis
      using Suc.prems by auto
  next
    case (Cons c cs)
     then have extend-until-conflict non-confl-set candidates xs (Suc k) = ex-
tend-until-conflict non-confl-set cs (c#xs) k
      by auto
    then have xk: x ∈ set (extend-until-conflict non-confl-set cs (c#xs) k)
        and  yk: y ∈ set (extend-until-conflict non-confl-set cs (c#xs) k)
      using Suc.prems by auto

    have **: x ∈ set (c#xs) ⟹ y ∈ set (c#xs) ⟹ (x,y) ∈ non-confl-set ∨ (y,x)
∈ non-confl-set
    proof −
      have scheme: ⋀ P xs x xs' . dropWhile P xs = (x#xs') ⟹ ¬ P x
        by (simp add: dropWhile-eq-Cons-conv)
      have find (λ y . (c,y) ∉ non-confl-set) xs = None
        using scheme[OF Cons] by simp
      then have *: ⋀ y . y ∈ set xs ⟹ (c,y) ∈ non-confl-set
        unfolding find-None-iff by blast

      assume x ∈ set (c#xs) and y ∈ set (c#xs)
      then consider (a1) x = c ∧ y ∈ set xs |
                    (a2) y = c ∧ x ∈ set xs |
```

(*a3*) $x \in$ *set xs* $\wedge$ $y \in$ *set xs*
using ‹$x \neq y$› **by** *auto*
then show *?thesis*
using $*$ *Suc.prems*(*3*) **by** (*cases*; *auto*)
qed

show *?thesis* **using** *Suc.IH*[*OF xk yk* $**$ *Suc.prems*(*4*)] **by** *blast*
qed
qed

definition *greedy-pairwise-r-distinguishable-state-sets-from-separators* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* $\Rightarrow$ '*a set list* **where**
greedy-pairwise-r-distinguishable-state-sets-from-separators M =
(*let pwrds* = *image fst* (*r-distinguishable-state-pairs-with-separators M*);
k = *size M*;
nL = *states-as-list M*
in map (λ*q* . *set* (*extend-until-conflict pwrds* (*remove1 q nL*) [*q*] *k*)) *nL*)

definition *maximal-repetition-sets-from-separators-list-greedy* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm* $\Rightarrow$ ('*a set* $\times$ '*a set*) *list* **where**
maximal-repetition-sets-from-separators-list-greedy M = (*let DR* = (*image fst* (*d-reachable-states-with-preambles M*))
in remdups (*map* (λ *S* . (*S*, *S* $\cap$ *DR*)) (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*)))

lemma *greedy-pairwise-r-distinguishable-state-sets-from-separators-cover* :
assumes $q \in$ *states M*
shows $\exists$ *S* $\in$ *set* (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*).
$q \in S$
using *assms extend-until-conflict-retainment*[*of q* [*q*]]
unfolding *states-as-list-set*[*symmetric*] *greedy-pairwise-r-distinguishable-state-sets-from-separators-def Let-def*
by *auto*

lemma *r-distinguishable-state-pairs-with-separators-sym* :
assumes (*q1*,*q2*) $\in$ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
shows (*q2*,*q1*) $\in$ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
using *assms*
unfolding *r-distinguishable-state-pairs-with-separators-def*
by *force*

lemma *greedy-pairwise-r-distinguishable-state-sets-from-separators-soundness* :
set (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*) $\subseteq$ (*pairwise-r-distinguishable-state-sets-fr*
M)
proof

fix S **assume** $S \in$ *set* (*greedy-pairwise-r-distinguishable-state-sets-from-separators* M)
then obtain q' **where** $q' \in$ *states* M
and $*$: $S =$ *set* (*extend-until-conflict* (*image fst* (*r-distinguishable-state-pairs-with-separators* M))
(*remove1* q' (*states-as-list* M))
[q']
(*size* M))
unfolding *greedy-pairwise-r-distinguishable-state-sets-from-separators-def Let-def states-as-list-set*[*symmetric*]
by *auto*

have $S \subseteq$ *states* M
proof
fix q **assume** $q \in S$
then have $q \in$ *set* (*extend-until-conflict* (*image fst* (*r-distinguishable-state-pairs-with-separators* M)) (*remove1* q' (*states-as-list* M)) [q'] (*size* M))
using $*$ **by** *auto*
then show $q \in$ *states* M
using *extend-until-conflict-elem*[*of q image fst* (*r-distinguishable-state-pairs-with-separators* M) (*remove1* q' (*states-as-list* M)) [q'] *size* M]
using *states-as-list-set* ‹$q' \in$ *states* M› **by** *auto*
qed

moreover have $\bigwedge$ *q1 q2* . $q1 \in S \Longrightarrow q2 \in S \Longrightarrow q1 \neq q2 \Longrightarrow (q1,q2) \in$ *image fst* (*r-distinguishable-state-pairs-with-separators* M)
proof −
fix *q1 q2* **assume** $q1 \in S$ **and** $q2 \in S$ **and** $q1 \neq q2$
then have *e1*: $q1 \in$ *set* (*extend-until-conflict* (*image fst* (*r-distinguishable-state-pairs-with-separators* M)) (*remove1* q' (*states-as-list* M)) [q'] (*size* M))
and *e2*: $q2 \in$ *set* (*extend-until-conflict* (*image fst* (*r-distinguishable-state-pairs-with-separators* M)) (*remove1* q' (*states-as-list* M)) [q'] (*size* M))
unfolding $*$ **by** *simp*+
have *e3*: ($q1 \in$ *set* [q'] $\Longrightarrow q2 \in$ *set* [q']
$\Longrightarrow$ ($q1$, $q2$) $\in$ *fst* ‘ *r-distinguishable-state-pairs-with-separators* M
$\vee$ ($q2$, $q1$) $\in$ *fst* ‘ *r-distinguishable-state-pairs-with-separators* M)
using ‹$q1 \neq q2$› **by** *auto*

show ($q1$,$q2$) $\in$ *image fst* (*r-distinguishable-state-pairs-with-separators* M)
using *extend-until-conflict-no-conflicts*[*OF e1 e2 e3* ‹$q1 \neq q2$›]
r-distinguishable-state-pairs-with-separators-sym[*of q2 q1 M*] **by** *blast*
qed

ultimately show $S \in$ (*pairwise-r-distinguishable-state-sets-from-separators* M)
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def* **by** *blast*
qed

end

38 Maximal Path Tries

Drastically reduced implementation of tries that consider only maximum length sequences as elements. Inserting a sequence that is prefix of some already contained sequence does not alter the trie. Intended to store IO-sequences to apply in testing, as in this use-case proper prefixes need not be applied separately.

theory *Maximal-Path-Trie*
imports *../Util*
begin

38.1 Utils for Updating Associative Lists

```
fun update-assoc-list-with-default :: 'a ⇒ ('b ⇒ 'b) ⇒ 'b ⇒ ('a × 'b) list ⇒ ('a × 'b) list where
  update-assoc-list-with-default k f d [] = [(k,f d)] |
  update-assoc-list-with-default k f d ((x,y)#xys) = (if k = x
    then ((x,f y)#xys)
    else (x,y) # (update-assoc-list-with-default k f d xys))

lemma update-assoc-list-with-default-key-found :
  assumes distinct (map fst xys)
  and     i < length xys
  and     fst (xys ! i) = k
shows update-assoc-list-with-default k f d xys =
        ((take i xys) @ [(k, f (snd (xys ! i)))] @ (drop (Suc i) xys))
using assms proof (induction xys arbitrary: i)
  case Nil
  then show ?case by auto
next
  case (Cons a xys)

  show ?case
  proof (cases i)
    case 0
    then have fst a = k using Cons.prems(3) by auto
    then have update-assoc-list-with-default k f d (a#xys) = (k, f (snd a)) # xys
      unfolding 0 by (metis prod.collapse update-assoc-list-with-default.simps(2))
    then show ?thesis unfolding 0 by auto
  next
    case (Suc j)
    then have fst a ≠ k
      using Cons.prems by auto

    have distinct (map fst xys)
    and  j < length xys
```

and *fst* (*xys* ! *j*) = *k*
using *Cons.prems* **unfolding** *Suc* **by** *auto*

then have *update-assoc-list-with-default k f d xys* = *take j xys* @ [(*k*, *f* (*snd* (*xys* ! *j*)))] @ *drop* (*Suc j*) *xys*
using *Cons.IH*[*of j*] **by** *auto*

then show *?thesis* **unfolding** *Suc* **using** ‹*fst a* ≠ *k*›
by (*metis append-Cons drop-Suc-Cons nth-Cons-Suc prod.collapse take-Suc-Cons update-assoc-list-with-default.simps*(*2*))
qed
qed

lemma *update-assoc-list-with-default-key-not-found* :
assumes *distinct* (*map fst xys*)
and *k* ∉ *set* (*map fst xys*)
shows *update-assoc-list-with-default k f d xys* = *xys* @ [(*k*,*f d*)]
using *assms* **by** (*induction xys*; *auto*)

lemma *update-assoc-list-with-default-key-distinct* :
assumes *distinct* (*map fst xys*)
shows *distinct* (*map fst* (*update-assoc-list-with-default k f d xys*))
proof (*cases k* ∈ *set* (*map fst xys*))
case *True*
then obtain *i* **where** *i* < *length xys* **and** *fst* (*xys* ! *i*) = *k*
by (*metis in-set-conv-nth length-map nth-map*)
then have ∗: (*map fst* (*take i xys* @ [(*k*, *f* (*snd* (*xys* ! *i*)))] @ *drop* (*Suc i*) *xys*)) = (*map fst xys*)
proof −
have *xys* ! *i* # *drop* (*Suc i*) *xys* = *drop i xys*
using *Cons-nth-drop-Suc* ‹*i* < *length xys*› **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) ‹*fst* (*xys* ! *i*) = *k*› *append-Cons append-self-conv2 append-take-drop-id fst-conv list.simps*(*9*) *map-append*)
qed
show *?thesis*
unfolding *update-assoc-list-with-default-key-found*[*OF assms* ‹*i* < *length xys*› ‹*fst* (*xys* ! *i*) = *k*›] ∗
using *assms* **by** *assumption*
next
case *False*
have ∗: (*map fst* (*xys* @ [(*k*, *f d*)])) = (*map fst xys*)@[*k*] **by** *auto*
show *?thesis*
using *assms False*
unfolding *update-assoc-list-with-default-key-not-found*[*OF assms False*] ∗ **by** *auto*
qed

38.2 Maximum Path Trie Implementation

datatype *'a mp-trie* = *MP-Trie* (*'a* × *'a mp-trie*) *list*

fun *mp-trie-invar* :: *'a mp-trie* ⇒ *bool* **where**
mp-trie-invar (*MP-Trie ts*) = (*distinct* (*map fst ts*) ∧ (∀ *t* ∈ *set* (*map snd ts*) . *mp-trie-invar t*))

definition *empty* :: *'a mp-trie* **where**
empty = *MP-Trie* []

lemma *empty-invar* : *mp-trie-invar empty* **unfolding** *empty-def* **by** *auto*

fun *height* :: *'a mp-trie* ⇒ *nat* **where**
height (*MP-Trie* []) = *0* |
height (*MP-Trie* (*xt*#*xts*)) = *Suc* (*foldr* (λ *t m* . *max* (*height t*) *m*) (*map snd* (*xt*#*xts*)) *0*)

lemma *height-0* :
assumes *height t* = *0*
shows *t* = *empty*
proof (*rule ccontr*)
assume *t* ≠ *empty*
then obtain *xt xts* **where** *t* = *MP-Trie* (*xt*#*xts*)
by (*metis empty-def height.cases*)
have *height t* > *0*
unfolding ‹*t* = *MP-Trie* (*xt*#*xts*)› *height.simps*
by *simp*
then show *False*
using *assms* **by** *simp*
qed

lemma *height-inc* :
assumes *t* ∈ *set* (*map snd ts*)
shows *height t* < *height* (*MP-Trie ts*)
proof −
obtain *xt xts* **where** *ts* = *xt*#*xts*
using *assms*
by (*metis list.set-cases list-map-source-elem*)

have *height t* < *Suc* (*foldr* (λ *t m* . *max* (*height t*) *m*) (*map snd* (*xt*#*xts*)) *0*)
using *assms* **unfolding** ‹*ts* = *xt*#*xts*› **using** *max-by-foldr*[*of t* (*map snd* (*xt*#*xts*)) *height*]
by *blast*

```
  then show ?thesis unfolding ‹ts = xt#xts› by auto
qed


fun insert :: 'a list ⇒ 'a mp-trie ⇒ 'a mp-trie where
  insert [] t = t |
  insert (x#xs) (MP-Trie ts) = (MP-Trie (update-assoc-list-with-default x (λ t .
insert xs t) empty ts))


lemma insert-invar : mp-trie-invar t ⟹ mp-trie-invar (insert xs t)
proof (induction xs arbitrary: t)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)

  obtain ts where t = MP-Trie ts
    using mp-trie-invar.cases by auto

  then have distinct (map fst ts)
      and ⋀ t . t ∈ set (map snd ts) ⟹ mp-trie-invar t
    using Cons.prems by auto


  show ?case proof (cases x ∈ set (map fst ts))
    case True
    then obtain i where i < length ts and fst (ts ! i) = x
      by (metis in-set-conv-nth length-map nth-map)
    have insert (x#xs) (MP-Trie ts) = (MP-Trie (take i ts @ [(x, insert xs (snd
(ts ! i)))] @ drop (Suc i) ts))
      unfolding insert.simps empty-def
      unfolding update-assoc-list-with-default-key-found[OF ‹distinct (map fst ts)›
‹i < length ts› ‹fst (ts ! i) = x›
                                                ,of (λ t . insert xs t) (MP-Trie [])]
      by simp

    have ⋀ t . t ∈ set (map snd (take i ts @ [(x, insert xs (snd (ts ! i)))] @ drop
(Suc i) ts)) ⟹ mp-trie-invar t
    proof −
      fix t assume t ∈ set (map snd (take i ts @ [(x, insert xs (snd (ts ! i)))] @
drop (Suc i) ts))
      then consider (a) t ∈ set (map snd (take i ts @ drop (Suc i) ts)) |
                    (b) t = insert xs (snd (ts ! i))
        by auto
      then show mp-trie-invar t proof cases
        case a
        then have t ∈ set (map snd ts)
       by (metis drop-map in-set-dropD in-set-takeD list-concat-non-elem map-append
```

take-map)
then show *?thesis* **using** ‹$\bigwedge$ *t* . *t* ∈ *set* (*map snd ts*) ⟹ *mp-trie-invar t*› **by** *blast*
next
case *b*
have (*snd* (*ts* ! *i*)) ∈ *set* (*map snd ts*)
using ‹*i* < *length ts*› **by** *auto*
then have *mp-trie-invar* (*snd* (*ts* ! *i*)) **using** ‹$\bigwedge$ *t* . *t* ∈ *set* (*map snd ts*) ⟹ *mp-trie-invar t*› **by** *blast*
then show *?thesis* **using** *Cons.IH* **unfolding** *b* **by** *blast*
qed
qed
moreover have *distinct* (*map fst* (*take i ts* @ [(*x*, *insert xs* (*snd* (*ts* ! *i*)))] @ *drop* (*Suc i*) *ts*))
using *update-assoc-list-with-default-key-distinct*[*OF* ‹*distinct* (*map fst ts*)›]
by (*metis* ‹*distinct* (*map fst ts*)› ‹*fst* (*ts* ! *i*) = *x*› ‹*i* < *length ts*› *update-assoc-list-with-default-key-found*)

ultimately show *?thesis*
unfolding ‹*t* = *MP-Trie ts*› ‹*insert* (*x*#*xs*) (*MP-Trie ts*) = (*MP-Trie* (*take i ts* @ [(*x*, *insert xs* (*snd* (*ts* ! *i*)))] @ *drop* (*Suc i*) *ts*))›
by *auto*
next
case *False*

have *mp-trie-invar* (*insert xs empty*)
by (*simp add*: *empty-invar Cons.IH*)

then show *?thesis*
using *Cons.prems update-assoc-list-with-default-key-distinct*[*OF* ‹*distinct* (*map fst ts*)›, *of x* (*insert xs*) (*MP-Trie* [])]
unfolding ‹*t* = *MP-Trie ts*› *insert.simps*
unfolding *update-assoc-list-with-default-key-not-found*[*OF* ‹*distinct* (*map fst ts*)› *False*]
by *auto*
qed
qed

fun *paths* :: *'a mp-trie* ⇒ *'a list list* **where**
paths (*MP-Trie* []) = [[]] |
paths (*MP-Trie* (*t*#*ts*)) = *concat* (*map* (λ (*x*,*t*) . *map* ((#) *x*) (*paths t*)) (*t*#*ts*))

lemma *paths-empty* :
assumes [] ∈ *set* (*paths t*)

```
  shows t = empty
proof (rule ccontr)
  assume t ≠ empty
  then obtain xt xts where t = MP-Trie (xt#xts)
    by (metis empty-def height.cases)
  then have [] ∈ set (concat (map (λ (x,t) . map ((#) x) (paths t)) (xt#xts)))
    using assms by auto
  then show False by auto
qed

lemma paths-nonempty :
  assumes [] ∉ set (paths t)
  shows set (paths t) ≠ {}
using assms proof (induction t rule: mp-trie-invar.induct)
  case (1 ts)
  have ts ≠ [] using 1.prems by auto
  then obtain x t xts where ts = ((x,t)#xts)
    using linear-order-from-list-position'.cases
    by (metis old.prod.exhaust)

  then have t ∈ set (map snd ts) by auto

  show ?case
  proof (cases [] ∈ set (paths t))
    case True
    then show ?thesis
      unfolding ‹ts = ((x,t)#xts)› paths.simps by auto
  next
    case False
    show ?thesis
      using 1.IH[OF ‹t ∈ set (map snd ts)› False]
      unfolding ‹ts = ((x,t)#xts)› paths.simps by auto
  qed
qed


lemma paths-maximal: mp-trie-invar t ⟹ xs' ∈ set (paths t) ⟹ ¬ (∃ xs'' . xs''
≠ [] ∧ xs'@xs'' ∈ set (paths t))
proof (induction xs' arbitrary: t)
  case Nil
  then have t = empty
    using paths-empty by blast
  then have paths t = [[]]
    by (simp add: empty-def)
  then show ?case by auto
next
  case (Cons x xs')

  then have t ≠ empty unfolding empty-def by auto
```

```
  then obtain xt xts where t = MP-Trie (xt#xts)
    by (metis empty-def height.cases)

  obtain t' where (x,t') ∈ set (xt#xts)
           and   xs' ∈ set (paths t')
    using Cons.prems(2)
    unfolding ‹t = MP-Trie (xt#xts)› paths.simps
    by force

  have mp-trie-invar t'
   using Cons.prems(1) ‹(x,t') ∈ set (xt#xts)› unfolding ‹t = MP-Trie (xt#xts)›
by auto

  show ?case
  proof
    assume ∃ xs''. xs'' ≠ [] ∧ (x # xs') @ xs'' ∈ set (paths t)
   then obtain xs'' where xs'' ≠ [] and (x # (xs' @ xs'')) ∈ set (paths (MP-Trie
(xt # xts)))
      unfolding ‹t = MP-Trie (xt#xts)› by force


    obtain t'' where (x,t'') ∈ set (xt#xts)
             and   (xs' @ xs'') ∈ set (paths t'')
      using ‹(x # (xs' @ xs'')) ∈ set (paths (MP-Trie (xt # xts)))›
      unfolding ‹t = MP-Trie (xt#xts)› paths.simps
      by force

    have distinct (map fst (xt#xts))
      using Cons.prems(1) unfolding ‹t = MP-Trie (xt#xts)› by simp
    then have t'' = t'
      using ‹(x,t') ∈ set (xt#xts)› ‹(x,t'') ∈ set (xt#xts)›
      by (meson eq-key-imp-eq-value)
    then have xs'@xs'' ∈ set (paths t')
      using ‹(xs' @ xs'') ∈ set (paths t'')› by auto
    then show False
     using ‹xs'' ≠ []› Cons.IH[OF ‹mp-trie-invar t'› ‹xs' ∈ set (paths t')›] by blast
  qed
qed



lemma paths-insert-empty :
  paths (insert xs empty) = [xs]
proof (induction xs)
  case Nil
  then show ?case unfolding empty-def by auto
next
  case (Cons x xs)
  then show ?case unfolding empty-def insert.simps by auto
```

qed

lemma *paths-order* :
  assumes *set ts* = *set ts′*
  and *length ts* = *length ts′*
shows *set* (*paths* (*MP-Trie ts*)) = *set* (*paths* (*MP-Trie ts′*))
  using *assms*(*2*,*1*) **proof** (*induction ts ts′ rule*: *list-induct2*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons x xs y ys*)

  have *scheme*: $\bigwedge$ *f xs ys* . *set xs* = *set ys* $\Longrightarrow$ *set* (*concat* (*map f xs*)) = *set* (*concat* (*map f ys*))
    by *auto*

  show *?case*
    using *scheme*[*OF Cons.prems*(*1*), *of* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*))] **by** *simp*
qed

lemma *paths-insert-maximal* :
  assumes *mp-trie-invar t*
  shows *set* (*paths* (*insert xs t*)) = (*if* ($\exists$ *xs′* . *xs*@*xs′* $\in$ *set* (*paths t*))
                then set (*paths t*)
                else Set.insert xs (*set* (*paths t*) − {*xs′* . $\exists$ *xs″* . *xs′*@*xs″* = *xs*}))
using *assms* **proof** (*induction xs arbitrary*: *t*)
  case *Nil*
  then show *?case*
    using *paths-nonempty* **by** *force*
next
  case (*Cons x xs*)
  show *?case* **proof** (*cases t* = *empty*)
    case *True*
    show *?thesis*
      unfolding *True*
      unfolding *paths-insert-empty*
      unfolding *empty-def paths.simps* **by** *auto*
  next
    case *False*

    then obtain *xt xts* **where** *t* = *MP-Trie* (*xt*#*xts*)
      by (*metis empty-def height.cases*)
    then have *t* = *MP-Trie* ((*fst xt*, *snd xt*)#*xts*)
      by *auto*

    have *distinct* (*map fst* (*xt*#*xts*))
      using *Cons.prems* ‹*t* = *MP-Trie* (*xt*#*xts*)› **by** *auto*

have (*paths t*) = *concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*xt* # *xts*))
unfolding ‹*t* = *MP-Trie* ((*fst xt*, *snd xt*)#*xts*)› **by** *simp*
then have *set* (*paths t*) = {*x*#*xs* | *x xs t* . (*x*,*t*) ∈ *set* (*xt*#*xts*) ∧ *xs* ∈ *set* (*paths t*)}
by *auto*
then have *Set.insert* (*x*#*xs*) (*set* (*paths t*)) = *Set.insert* (*x*#*xs*) {*x*#*xs* | *x xs t* . (*x*,*t*) ∈ *set* (*xt*#*xts*) ∧ *xs* ∈ *set* (*paths t*)}
by *blast*

show *?thesis* **proof** (*cases x* ∈ *set* (*map fst* (*xt*#*xts*)))
case *True*
case *True*
then obtain *i* **where** *i* < *length* (*xt*#*xts*) **and** *fst* ((*xt*#*xts*) ! *i*) = *x*
by (*metis in-set-conv-nth list-map-source-elem*)
then have ((*xt*#*xts*) ! *i*) = (*x*,*snd* ((*xt*#*xts*) ! *i*)) **by** *auto*

have *mp-trie-invar* (*snd* ((*xt* # *xts*) ! *i*))
using *Cons.prems* ‹*i* < *length* (*xt*#*xts*)› **unfolding** ‹*t* = *MP-Trie* (*xt*#*xts*)›
by (*metis* ‹(*xt* # *xts*) ! *i* = (*x*, *snd* ((*xt* # *xts*) ! *i*))› *in-set-zipE mp-trie-invar.simps nth-mem zip-map-fst-snd*)

have *insert* (*x*#*xs*) *t* = *MP-Trie* (*take i* (*xt* # *xts*) @ [(*x*, *insert xs* (*snd* ((*xt* # *xts*) ! *i*)))] @ *drop* (*Suc i*) (*xt* # *xts*))
unfolding ‹*t* = *MP-Trie* (*xt*#*xts*)› *insert.simps*
unfolding *update-assoc-list-with-default-key-found*[*OF* ‹*distinct* (*map fst* (*xt*#*xts*))› ‹*i* < *length* (*xt*#*xts*)› ‹*fst* ((*xt*#*xts*) ! *i*) = *x*›]
by *simp*

then have *set* (*paths* (*insert* (*x*#*xs*) *t*))
= *set* (*paths* (*MP-Trie* (*take i* (*xt* # *xts*) @ [(*x*, *insert xs* (*snd* ((*xt* # *xts*) ! *i*)))] @ *drop* (*Suc i*) (*xt* # *xts*))))
by *simp*
also have ... = *set* (*paths* (*MP-Trie* ((*x*, *insert xs* (*snd* ((*xt* # *xts*) ! *i*))) # (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))))
using *paths-order*[*of* (*take i* (*xt* # *xts*) @ [(*x*, *insert xs* (*snd* ((*xt* # *xts*) ! *i*)))] @ *drop* (*Suc i*) (*xt* # *xts*))
((*x*, *insert xs* (*snd* ((*xt* # *xts*) ! *i*))) # (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))]
by *force*
also have ... = *set* ((*map* ((#) *x*) (*paths* (*insert xs* (*snd* ((*xt* # *xts*) ! *i*))))) @ (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))))
unfolding *paths.simps* **by** *force*
finally have *set* (*paths* (*insert* (*x*#*xs*) *t*)) =
set (*map* ((#) *x*) (*paths* (*insert xs* (*snd* ((*xt* # *xts*) ! *i*)))))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))

by *force*
also have ... = (*image* ((#) *x*) (*set* (*paths* (*insert xs* (*snd* ((*xt* # *xts*) ! *i*))))))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
by *auto*
finally have *pi1*: *set* (*paths* (*insert* (*x*#*xs*) *t*)) =
image ((#) *x*) (*if* ∃ *xs'*. *xs* @ *xs'* ∈ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) *then set* (*paths* (*snd* ((*xt* # *xts*) ! *i*)))
else
Set.insert xs (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs'*. ∃ *xs''*. *xs'* @ *xs''* = *xs*}))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
unfolding *Cons.IH*[*OF* ‹*mp-trie-invar* (*snd* ((*xt* # *xts*) ! *i*))›] **by** *blast*

have *po1*: *set* (*xt*#*xts*) = *set* ((*x*,*snd* ((*xt*#*xts*) ! *i*)) # ((*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
using *list-index-split-set*[*OF* ‹*i* < *length* (*xt*#*xts*)›]
unfolding ‹((*xt*#*xts*) ! *i*) = (*x*,*snd* ((*xt*#*xts*) ! *i*))›[*symmetric*] **by** *assumption*
have *po2*: *length* (*xt*#*xts*) = *length* ((*x*,*snd* ((*xt*#*xts*) ! *i*)) # ((*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
using ‹*i* < *length* (*xt*#*xts*)› **by** *auto*

have *set* (*paths t*) = *set* (*paths* (*MP-Trie* ((*x*,*snd* ((*xt*#*xts*) ! *i*)) # ((*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))))
unfolding ‹*t* = *MP-Trie* (*xt*#*xts*)›
using *paths-order*[*OF po1 po2*] **by** *assumption*
also have ... = *set* ((*map* ((#) *x*) (*paths* (*snd* ((*xt* # *xts*) ! *i*)))) @ (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))))
unfolding *paths.simps* **by** *auto*
finally have *set* (*paths t*) =
set (*map* ((#) *x*) (*paths* (*snd* ((*xt* # *xts*) ! *i*))))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
by *force*

then have *pi2*: *set* (*paths t*) = (*image* ((#) *x*) (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*)))))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
by *auto*

show *?thesis* **proof** (*cases* ∃ *xs'*. *xs* @ *xs'* ∈ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))))
case *True*
then have *pi1'*: *set* (*paths* (*insert* (*x*#*xs*) *t*)) = *image* ((#) *x*) (*set* (*paths*

(*snd* ((*xt* # *xts*) ! *i*))))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
using *pi1* **by** *auto*

have *set* (*paths* (*insert* (*x* # *xs*) *t*)) = *set* (*paths t*)
unfolding *pi1′ pi2* **by** *simp*
moreover have ∃ *xs′*. (*x* # *xs*) @ *xs′* ∈ *set* (*paths t*)
using *True* **unfolding** *pi2* **by** *force*
ultimately show *?thesis* **by** *simp*
next
case *False*
then have *pi1′*: *set* (*paths* (*insert* (*x*#*xs*) *t*)) = *image* ((#) *x*) (*Set.insert xs* (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *xs*}))
∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
using *pi1* **by** *auto*

have *x1*: ((#) *x* ‘ *Set.insert xs* (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *xs*}))
= *Set.insert* (*x* # *xs*) ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *x* # *xs*})
proof −
have ⋀ *a* . *a* ∈ ((#) *x* ‘ *Set.insert xs* (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *xs*})) ⟹
a ∈ *Set.insert* (*x* # *xs*) ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *x* # *xs*})
by *fastforce*
moreover have ⋀ *a* . *a* ∈ *Set.insert* (*x* # *xs*) ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *x* # *xs*}) ⟹
a ∈ ((#) *x* ‘ *Set.insert xs* (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *xs*}))
proof −
fix *a* **assume** *a* ∈ *Set.insert* (*x* # *xs*) ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *x* # *xs*})
then consider (*a*) *a* = (*x*#*xs*) |
(*b*) *a* ∈ ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *x* # *xs*}) **by** *blast*
then show *a* ∈ ((#) *x* ‘ *Set.insert xs* (*set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) − {*xs′*. ∃ *xs″*. *xs′* @ *xs″* = *xs*}))
proof *cases*
case *a*
then show *?thesis* **by** *blast*
next
case *b*
then show *?thesis* **by** *force*
qed
qed
ultimately show *?thesis* **by** *blast*

qed

have *x2*: *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
= (*set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))) − {*xs*′. ∃ *xs*″. *xs*′ @ *xs*″ = *x* # *xs*})
and *x3*: ¬(∃ *xs*′. (*x* # *xs*) @ *xs*′ ∈ *set* (*paths t*))
proof −

have ⋀ *j* . *j* < *length* (*xt*#*xts*) ⟹ *j* ≠ *i* ⟹ *fst* ((*xt*#*xts*) ! *j*) ≠ *x*
using ‹*i* < *length* (*xt*#*xts*)› ‹*fst* ((*xt*#*xts*) ! *i*) = *x*› ‹*distinct* (*map fst* (*xt*#*xts*))›
by (*metis* (*no-types*, *lifting*) *length-map nth-eq-iff-index-eq nth-map*)

have ⋀ *xt*′ . *xt*′ ∈ *set* (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)) ⟹ *fst xt*′ ≠ *x*
proof −
fix *xt*′ **assume** *xt*′ ∈ *set* (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))
then consider (*a*) *xt*′ ∈ *set* (*take i* (*xt* # *xts*)) |
(*b*) *xt*′ ∈ *set* (*drop* (*Suc i*) (*xt* # *xts*))
by *auto*
then show *fst xt*′ ≠ *x* **proof** *cases*
case *a*
then obtain *j* **where** *j* < *length* (*take i* (*xt*#*xts*)) (*take i* (*xt*#*xts*)) ! *j* = *xt*′
by (*meson in-set-conv-nth*)

have *j* < *length* (*xt*#*xts*) **and** *j* < *i*
using ‹*j* < *length* (*take i* (*xt*#*xts*))› **by** *auto*
moreover have (*xt*#*xts*) ! *j* = *xt*′
using ‹(*take i* (*xt*#*xts*)) ! *j* = *xt*′› ‹*j* < *i*› **by** *auto*
ultimately show *?thesis* **using** ‹⋀ *j* . *j* < *length* (*xt*#*xts*) ⟹ *j* ≠ *i* ⟹ *fst* ((*xt*#*xts*) ! *j*) ≠ *x*› **by** *blast*
next
case *b*
then obtain *j* **where** *j* < *length* (*drop* (*Suc i*) (*xt*#*xts*)) (*drop* (*Suc i*) (*xt*#*xts*)) ! *j* = *xt*′
by (*meson in-set-conv-nth*)

have (*Suc i*) + *j* < *length* (*xt*#*xts*) **and** (*Suc i*) + *j* > *i*
using ‹*j* < *length* (*drop* (*Suc i*) (*xt*#*xts*))› **by** *auto*
moreover have (*xt*#*xts*) ! ((*Suc i*) + *j*) = *xt*′
using ‹(*drop* (*Suc i*) (*xt*#*xts*)) ! *j* = *xt*′›
using ‹*i* < *length* (*xt* # *xts*)› **by** *auto*
ultimately show *?thesis* **using** ‹⋀ *j* . *j* < *length* (*xt*#*xts*) ⟹ *j* ≠ *i* ⟹ *fst* ((*xt*#*xts*) ! *j*) ≠ *x*›[*of* (*Suc i*) + *j*]
by *auto*
qed

qed
then show *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
= (*set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))) − {*xs*′. ∃ *xs*″. *xs*′ @ *xs*″ = *x* # *xs*})
by *force*

show ¬(∃ *xs*′. (*x* # *xs*) @ *xs*′ ∈ *set* (*paths t*))
proof
assume ∃ *xs*′. (*x* # *xs*) @ *xs*′ ∈ *set* (*paths t*)
then obtain *xs*′ **where** (*x* # (*xs* @ *xs*′)) ∈ ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*)))) ∪
set (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
unfolding *pi2* **by** *force*
then consider (*a*) (*x* # (*xs* @ *xs*′)) ∈ ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*)))) |
(*b*) (*x* # (*xs* @ *xs*′)) ∈ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
by *blast*
then show *False* **proof** *cases*
case *a*
then show *?thesis* **using** *False* **by** *force*
next
case *b*
then show *?thesis* **using** ‹⋀ *xt*′ . *xt*′ ∈ *set* (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)) ⟹ *fst xt*′ ≠ *x*› **by** *force*
qed
qed
qed

have ∗: *Set.insert* (*x* # *xs*) ((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*))) ∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*)))) − {*xs*′. ∃ *xs*″. *xs*′ @ *xs*″ = *x* # *xs*})
= *Set.insert* (*x* # *xs*) (((#) *x* ‘ *set* (*paths* (*snd* ((*xt* # *xts*) ! *i*)))) − {*xs*′. ∃ *xs*″. *xs*′ @ *xs*″ = *x* # *xs*}) ∪ *set* (*concat* (*map* (λ(*x*, *t*). *map* ((#) *x*) (*paths t*)) (*take i* (*xt* # *xts*) @ *drop* (*Suc i*) (*xt* # *xts*))))
using *x2* **by** *blast*

have *set* (*paths* (*insert* (*x* # *xs*) *t*)) = *Set.insert* (*x* # *xs*) (*set* (*paths t*) − {*xs*′. ∃ *xs*″. *xs*′ @ *xs*″ = *x* # *xs*})
unfolding *pi1*′ *pi2 x1* ∗ **by** *blast*
then show *?thesis*
using *x3* **by** *simp*
qed
next
case *False*
have *insert* (*x*#*xs*) *t* = *MP-Trie* (*xt* # (*xts* @ [(*x*, *insert xs empty*)]))

unfolding ‹t = *MP-Trie* (xt#xts)› *insert.simps*
unfolding *update-assoc-list-with-default-key-not-found*[*OF* ‹*distinct* (*map fst* (xt#xts))› *False*]
by *simp*

have (*paths* (*MP-Trie* (xt # (xts @ [(x, *insert xs empty*)])))) = *concat* (*map* (λ(x, t). *map* ((#) x) (*paths* t)) (xt # xts @ [(x, *insert xs empty*)]))
unfolding *paths.simps empty-def* **by** *simp*
also have ... = (*concat* (*map* (λ(x, t). *map* ((#) x) (*paths* t)) (xt # xts))) @ (*map* ((#) x) (*paths* (*insert xs empty*)))
by *auto*
finally have *paths* (*insert* (x#xs) t) = (*paths* t) @ [x#xs]
unfolding ‹*insert* (x#xs) t = *MP-Trie* (xt # (xts @ [(x, *insert xs empty*)]))›
‹(*paths* t) = *concat* (*map* (λ(x, t). *map* ((#) x) (*paths* t)) (xt # xts))›[*symmetric*]
paths-insert-empty
by *auto*
then have *set* (*paths* (*insert* (x#xs) t)) = *Set.insert* (x#xs) (*set* (*paths* t))
by *force*

have ⋀ p . p ∈ *set* (*paths* t) ⟹ p ≠ [] ∧ *hd* p ≠ x
using *False*
unfolding ‹(*paths* t) = *concat* (*map* (λ(x, t). *map* ((#) x) (*paths* t)) (xt # xts))› **by** *force*
then have ⋀ xs' . xs' ∈ *set* (*paths* t) ⟹ ¬(∃ xs'' . xs'@xs'' = x#xs)
by (*metis hd-append2 list.sel*(*1*))
then have *set* (*paths* t) = (*set* (*paths* t) − {xs' . ∃ xs'' . xs'@xs'' = x#xs})
by *blast*
then show *?thesis*
unfolding ‹*set* (*paths* (*insert* (x#xs) t)) = *Set.insert* (x#xs) (*set* (*paths* t))›
using ‹⋀p. p ∈ *set* (*paths* t) ⟹ p ≠ [] ∧ *hd* p ≠ x› **by** *force*
qed
qed
qed

fun *from-list* :: *'a list list* ⇒ *'a mp-trie* **where**
from-list seqs = *foldr insert seqs empty*

lemma *from-list-invar* : *mp-trie-invar* (*from-list xs*)
using *empty-invar insert-invar* **by** (*induction xs*; *auto*)

lemma *from-list-paths* :
set (*paths* (*from-list* (x#xs))) = {y. y ∈ *set* (x#xs) ∧ ¬(∃ y' . y' ≠ [] ∧ y@y' ∈ *set* (x#xs))}

proof (*induction xs arbitrary*: *x*)
  case *Nil*
  have ∗: *paths* (*from-list* [*x*]) = *paths* (*insert x empty*) **by** *auto*

  show *?case*
    unfolding ∗
    unfolding *paths-insert-maximal*[*OF empty-invar*, *of x*]
    unfolding *empty-def*
    by (*cases x* ; *auto*)
next
  case (*Cons x′ xs*)

  have *from-list* (*x*#*x′*#*xs*) = *insert x* (*insert x′* (*from-list xs*)) **by** *auto*
  have *from-list* (*x*#*x′*#*xs*) = *insert x* (*from-list* (*x′*#*xs*)) **by** *auto*

  have *mp-trie-invar* (*insert x′* (*from-list xs*))
    using *from-list-invar insert-invar* **by** *metis*
  have (*insert x′* (*from-list xs*)) = *from-list* (*x′*#*xs*) **by** *auto*

  thm *paths-insert-maximal*[*OF* ‹*mp-trie-invar* (*insert x′* (*from-list xs*))›, *of x*]

  show *?case* **proof** (*cases* ∃ *xs′*. *x* @ *xs′* ∈ *set* (*paths* (*insert x′* (*from-list xs*))))
    case *True*
    then have *set* (*paths* (*insert x* (*insert x′* (*from-list xs*)))) = *set* (*paths* (*insert x′* (*from-list xs*)))
      using *paths-insert-maximal*[*OF* ‹*mp-trie-invar* (*insert x′* (*from-list xs*))›, *of x*] **by** *simp*
    then have *set* (*paths* (*insert x* (*from-list* (*x′* # *xs*)))) = *set* (*paths* (*from-list* (*x′* # *xs*)))
      unfolding ‹(*insert x′* (*from-list xs*)) = *from-list* (*x′*#*xs*)›
      by *assumption*
    then have *set* (*paths* (*from-list* (*x*#*x′*#*xs*))) = {*y* ∈ *set* (*x′* # *xs*). ∄ *y′*. *y′* ≠ [] ∧ *y* @ *y′* ∈ *set* (*x′* # *xs*)}
      unfolding *Cons* ‹*from-list* (*x*#*x′*#*xs*) = *insert x* (*from-list* (*x′*#*xs*))›
      by *assumption*

    show *?thesis* **proof** (*cases x* ∈ *set* (*paths* (*insert x′* (*from-list xs*))))
      case *True*
      then have *x* ∈ *set* (*x′*#*xs*)
        using ‹*set* (*paths* (*insert x* (*insert x′* (*from-list xs*)))) = *set* (*paths* (*insert x′* (*from-list xs*)))› ‹*set* (*paths* (*from-list* (*x* # *x′* # *xs*))) = {*y* ∈ *set* (*x′* # *xs*). ∄ *y′*. *y′* ≠ [] ∧ *y* @ *y′* ∈ *set* (*x′* # *xs*)}› **by** *auto*
      then show *?thesis*
        unfolding ‹*set* (*paths* (*from-list* (*x*#*x′*#*xs*))) = {*y* ∈ *set* (*x′* # *xs*). ∄ *y′*. *y′* ≠ [] ∧ *y* @ *y′* ∈ *set* (*x′* # *xs*)}› **by** *auto*
    next
      case *False*

have $\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\} = \{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs)\}$
proof −
obtain xs' **where** $xs' \neq []$ **and** $x @ xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\}$
using *True False*
by (*metis* ‹*from-list* $(x \# x' \# xs) = insert\ x\ (insert\ x'\ (from\text{-}list\ xs))$› ‹*set* $(paths\ (insert\ x\ (insert\ x'\ (from\text{-}list\ xs)))) = set\ (paths\ (insert\ x'\ (from\text{-}list\ xs)))$› ‹*set* $(paths\ (from\text{-}list\ (x \# x' \# xs))) = \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\}$› *append-Nil2*)
then have *s1*: $\{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs)\} = \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs)\}$
by *auto*

have $\bigwedge y\ .\ (\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)) \Longrightarrow (\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs))$
proof −
fix y **assume** $(\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs))$

show $(\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs))$
proof
assume $\exists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs)$
then have $\exists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs) - set\ (x' \# xs)$
using ‹$(\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs))$› **by** *auto*
then have $\exists y'\ .\ y' \neq [] \wedge y @ y' = x$
by *auto*
then show *False*
by (*metis* (*no-types*, *lifting*) *Nil-is-append-conv* ‹$\nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)$› ‹$x @ xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\}$› *append.assoc mem-Collect-eq*)
qed
qed
then have *s2*: $\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\} = \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x \# x' \# xs)\}$
by *auto*

show *?thesis*
unfolding *s1 s2* **by** *simp*
qed
then show *?thesis*
using ‹$set\ (paths\ (from\text{-}list\ (x \# x' \# xs))) = \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y @ y' \in set\ (x' \# xs)\}$› **by** *auto*
qed

next
case *False*

then have $*$: $set\ (paths\ (insert\ x\ (insert\ x'\ (from\text{-}list\ xs))))$

$= Set.insert\ x\ (set\ (paths\ (insert\ x'\ (from\text{-}list\ xs))) - \{xs'.\ \exists xs''.\ xs'\ @\ xs'' = x\})$
using *paths-insert-maximal*[*OF* ‹*mp-trie-invar* (*insert* x' (*from-list xs*))›, *of* x] **by** *simp*

have f: $\nexists xs'.\ x\ @\ xs' \in \{y \in set\ (x'\ \#\ xs).\ \nexists y'.\ y' \neq []\ \wedge\ y\ @\ y' \in set\ (x'\ \#\ xs)\}$
using *False*
unfolding ‹(*insert* x' (*from-list xs*)) = *from-list* (x'#*xs*)› *Cons*
by *assumption*
then have $x \notin \{y \in set\ (x'\ \#\ xs).\ \nexists y'.\ y' \neq []\ \wedge\ y\ @\ y' \in set\ (x'\ \#\ xs)\}$
by (*metis* (*no-types*, *lifting*) *append-Nil2*)

have $x \notin set\ (x'\ \#\ xs)$
proof
assume $x \in set\ (x'\ \#\ xs)$
then have $\exists y'.\ y' \neq []\ \wedge\ x\ @\ y' \in set\ (x'\ \#\ xs)$
using ‹$x \notin \{y \in set\ (x'\ \#\ xs).\ \nexists y'.\ y' \neq []\ \wedge\ y\ @\ y' \in set\ (x'\ \#\ xs)\}$› **by** *auto*

let *?xms* = $\{xs'\ .\ xs' \neq []\ \wedge\ x\ @\ xs' \in set\ (x'\ \#\ xs)\}$
have *?xms* $\neq \{\}$
using ‹$\exists y'.\ y' \neq []\ \wedge\ x\ @\ y' \in set\ (x'\ \#\ xs)$›
by *simp*
moreover have *finite ?xms*
proof −
have *?xms* $\subseteq$ *image* (*drop* (*length x*)) (*set* (x'#*xs*)) **by** *force*
then show *?thesis* **by** (*meson List.finite-set finite-surj*)
qed
ultimately have $\exists\ xs' \in$ *?xms* . $\forall\ xs'' \in$ *?xms* . *length* $xs'' \leq$ *length* xs'
by (*meson max-length-elem not-le-imp-less*)
then obtain xs' **where** $xs' \neq []$
and $x@xs' \in set\ (x'\#xs)$
and $\bigwedge xs''\ .\ xs'' \neq [] \Longrightarrow x@xs'' \in set\ (x'\#xs) \Longrightarrow length\ xs'' \leq length\ xs'$
by *blast*

have $\nexists y'.\ y' \neq []\ \wedge\ (x@xs')\ @\ y' \in set\ (x'\ \#\ xs)$
proof
assume $\exists y'.\ y' \neq []\ \wedge\ (x\ @\ xs')\ @\ y' \in set\ (x'\ \#\ xs)$
then obtain xs'' **where** $xs'' \neq []$ **and** $(x\ @\ xs')\ @\ xs'' \in set\ (x'\ \#\ xs)$
by *blast*
then have $xs'@xs'' \neq []$ **and** $x\ @\ (xs'\ @\ xs'') \in set\ (x'\ \#\ xs)$
by *auto*
then have *length* ($xs'@xs''$) $\leq$ *length* xs'
using ‹$\bigwedge xs''\ .\ xs'' \neq [] \Longrightarrow x@xs'' \in set\ (x'\#xs) \Longrightarrow length\ xs'' \leq length\ xs'$› **by** *blast*
then show *False*
using ‹$xs'' \neq []$› **by** *auto*

qed

then have x @ $xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\}$
using ‹$x@xs' \in set\ (x'\#xs)$› **by** *blast*
then show *False* **using** ‹$\nexists xs'.\ x$ @ $xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\}$›
by *blast*
qed

have $\nexists y'.\ y' \neq [] \wedge x$ @ $y' \in set\ (x \# x' \# xs)$
proof
assume $\exists y'.\ y' \neq [] \wedge x$ @ $y' \in set\ (x \# x' \# xs)$
then obtain y' **where** $y' \neq []$ **and** $x@y' \in set\ (x\#x'\#xs)$
by *blast*
then have $x@y' \in set\ (x'\#xs)$
by *auto*

let $?xms = \{xs'\ .\ xs' \neq [] \wedge x$ @ $xs' \in set\ (x \# x' \# xs)\}$
have $?xms \neq \{\}$
using ‹$\exists y'.\ y' \neq [] \wedge x$ @ $y' \in set\ (x \# x' \# xs)$›
by *simp*
moreover have *finite ?xms*
proof −
have $?xms \subseteq image\ (drop\ (length\ x))\ (set\ (x'\#xs))$ **by** *force*
then show *?thesis* **by** (*meson List.finite-set finite-surj*)
qed
ultimately have $\exists\ xs' \in ?xms\ .\ \forall\ xs'' \in ?xms\ .\ length\ xs'' \leq length\ xs'$
by (*meson max-length-elem not-le-imp-less*)
then obtain xs' **where** $xs' \neq []$
and $x@xs' \in set\ (x\#x'\#xs)$
and $\bigwedge xs''\ .\ xs'' \neq [] \Longrightarrow x@xs'' \in set\ (x\#x'\#xs) \Longrightarrow length\ xs'' \leq length\ xs'$
by *blast*

have $\nexists y'.\ y' \neq [] \wedge (x@xs')$ @ $y' \in set\ (x \# x' \# xs)$
proof
assume $\exists y'.\ y' \neq [] \wedge (x$ @ $xs')$ @ $y' \in set\ (x \# x' \# xs)$
then obtain xs'' **where** $xs'' \neq []$ **and** $(x$ @ $xs')$ @ $xs'' \in set\ (x \# x' \# xs)$
by *blast*
then have $xs'@xs'' \neq []$ **and** x @ $(xs'$ @ $xs'') \in set\ (x \# x' \# xs)$
by *auto*
then have $length\ (xs'@xs'') \leq length\ xs'$
using ‹$\bigwedge xs''\ .\ xs'' \neq [] \Longrightarrow x@xs'' \in set\ (x\#x'\#xs) \Longrightarrow length\ xs'' \leq length\ xs'$› **by** *blast*
then show *False*
using ‹$xs'' \neq []$› **by** *auto*

qed

then have x @ $xs' \in \{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x \# x' \# xs)\}$
using ‹$x@xs' \in set\ (x \# x'\#xs)$› **by** *blast*
then have x @ $xs' \in set\ (x' \# xs)$ **and** $\nexists y'.\ y' \neq [] \wedge (x@xs')$ @ $y' \in set\ (x' \# xs)$
using ‹$xs' \neq []$› **by** *auto*
then have x @ $xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\}$
by *blast*
then show *False* **using** ‹$\nexists xs'.\ x$ @ $xs' \in \{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\}$›
by *blast*
qed

have $Set.insert\ x\ (\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\} - \{xs'.\ \exists xs''.\ xs'$ @ $xs'' = x\})$
$= \{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x \# x' \# xs)\}$
proof −
have $\bigwedge y\ .\ y \in Set.insert\ x\ (\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\} - \{xs'.\ \exists xs''.\ xs'$ @ $xs'' = x\}) \Longrightarrow y \in \{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x \# x' \# xs)\}$
proof −
fix y **assume** $y \in Set.insert\ x\ (\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\} - \{xs'.\ \exists xs''.\ xs'$ @ $xs'' = x\})$
then consider $(a)\ y = x$ |
$(b)\ y \in (\{y \in set\ (x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)\} - \{xs'.\ \exists xs''.\ xs'$ @ $xs'' = x\})$
by *blast*
then show $y \in \{y \in set\ (x \# x' \# xs).\ \nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x \# x' \# xs)\}$
proof *cases*
case a
show *?thesis*
using ‹$\nexists y'.\ y' \neq [] \wedge x$ @ $y' \in set\ (x \# x' \# xs)$› **unfolding** a **by** *auto*
next
case b
then have $y \in set\ (x' \# xs)$ **and** $\nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)$ **and** $\neg(\exists xs''.\ y$ @ $xs'' = x)$
by *blast+*

have $y \in set\ (x\#x'\#xs)$
using ‹$y \in set\ (x' \# xs)$› **by** *auto*
moreover have $\nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x \# x' \# xs)$
using ‹$\nexists y'.\ y' \neq [] \wedge y$ @ $y' \in set\ (x' \# xs)$› ‹$\neg(\exists xs''.\ y$ @ $xs'' = x)$›
by *auto*
ultimately show *?thesis* **by** *blast*

qed
qed

moreover have $\bigwedge y\ .y \in \{y \in set\ (x\ \#\ x'\ \#\ xs).\ \nexists y'.\ y' \neq [] \wedge y\ @\ y' \in set\ (x\ \#\ x'\ \#\ xs)\} \Longrightarrow\ y \in Set.insert\ x\ (\{y \in set\ (x'\ \#\ xs).\ \nexists y'.\ y' \neq [] \wedge y\ @\ y' \in set\ (x'\ \#\ xs)\} - \{xs'.\ \exists xs''.\ xs'\ @\ xs'' = x\})$
by *auto*
ultimately show *?thesis* **by** *blast*
qed
then show *?thesis*
using $*$ *Cons.IH* **by** *auto*
qed
qed

38.2.1 New Code Generation for *remove-proper-prefixes*

lemma *remove-proper-prefixes-code-trie*[*code*] :
remove-proper-prefixes (*set xs*) = (*case xs of* [] ⇒ {} | (*x*#*xs*′) ⇒ *set* (*paths* (*from-list* (*x*#*xs*′))))
unfolding *from-list-paths remove-proper-prefixes-def* **by** (*cases xs*; *auto*)

end

39 R-Distinguishability

This theory defines the notion of r-distinguishability and relates it to state separators.

theory *R-Distinguishability*
imports *State-Separator*
begin

definition *r-compatible* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *bool* **where**
r-compatible M q1 q2 = ((∃ *S* . *completely-specified S* ∧ *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))))

abbreviation(*input*) *r-distinguishable M q1 q2* ≡ ¬ *r-compatible M q1 q2*

fun *r-distinguishable-k* :: (′*a*, ′*b*, ′*c*) *fsm* ⇒ ′*a* ⇒ ′*a* ⇒ *nat* ⇒ *bool* **where**
r-distinguishable-k M q1 q2 0 = (∃ *x* ∈ (*inputs M*) . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) |
r-distinguishable-k M q1 q2 (*Suc k*) = (*r-distinguishable-k M q1 q2 k*
∨ (∃ *x* ∈ (*inputs M*) . ∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧

t-input t2 = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))

39.1 R(k)-Distinguishability Properties

lemma *r-distinguishable-k-0-alt-def* :
r-distinguishable-k M q1 q2 0 = (∃ *x* ∈ (*inputs M*) . ¬(∃ *y q1′ q2′* . (*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q2*,*x*,*y*,*q2′*) ∈ *transitions M*))
by *fastforce*

lemma *r-distinguishable-k-Suc-k-alt-def* :
r-distinguishable-k M q1 q2 (*Suc k*) = (*r-distinguishable-k M q1 q2 k*
∨ (∃ *x* ∈ (*inputs M*) . ∀ *y q1′ q2′* . ((*q1*,*x*,*y*,*q1′*) ∈ *transitions M* ∧ (*q2*,*x*,*y*,*q2′*) ∈ *transitions M*) ⟶ *r-distinguishable-k M q1′ q2′ k*))
by *fastforce*

lemma *r-distinguishable-k-by-larger* :
assumes *r-distinguishable-k M q1 q2 k*
and $k \leq k'$
shows *r-distinguishable-k M q1 q2 k′*
using *assms nat-induct-at-least* **by** *fastforce*

lemma *r-distinguishable-k-0-not-completely-specified* :
assumes *r-distinguishable-k M q1 q2 0*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
proof −
let *?F1* = *from-FSM M q1*
let *?F2* = *from-FSM M q2*
let *?P* = *product ?F1 ?F2*

obtain *x* **where** *x* ∈ (*inputs M*)
and ¬ (∃ *t1 t2* . *t1* ∈ *transitions M* ∧ *t2* ∈ *transitions M* ∧ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)
using *assms*(*1*) **by** *fastforce*

then have ∗: ¬ (∃ *t1 t2* . *t1* ∈ *transitions ?F1* ∧ *t2* ∈ *transitions ?F2* ∧ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)
unfolding *from-FSM-simps*[*OF assms*(*2*)] *from-FSM-simps*[*OF assms*(*3*)] **by** *simp*

have ∗∗: ¬ (∃ *t* ∈ *transitions ?P* . *t-source t* = (*q1*,*q2*) ∧ *t-input t* = *x*)
proof (*rule ccontr*)

assume ¬ ¬ (∃ *t*∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)). *t-source t* = (*q1*, *q2*) ∧ *t-input t* = *x*)
then obtain *t* **where** *t* ∈ *transitions ?P* **and** *t-source t* = (*q1*,*q2*) **and** *t-input t* = *x*
by *blast*

have ∃ *t1 t2* . *t1* ∈ *transitions ?F1* ∧ *t2* ∈ *transitions ?F2* ∧ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*
using *product-transition-split*[*OF* ‹*t* ∈ *transitions ?P*›]
by (*metis* ‹*t-input t* = *x*› ‹*t-source t* = (*q1*, *q2*)› *fst-conv snd-conv*)
then show *False*
using ∗ **by** *auto*
qed

moreover have *x* ∈ (*inputs ?P*)
using ‹*x* ∈ (*inputs M*)›
by (*simp add*: *assms*(*3*))

ultimately have ¬ *completely-specified-state ?P* (*q1*,*q2*)
by (*meson completely-specified-state.elims*(*2*))

have (*q1*,*q2*) = *initial* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
by (*simp add*: *assms*(*2*) *assms*(*3*))

then show *?thesis*
using ‹¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*, *q2*)› **by** *presburger*
qed

lemma *r-0-distinguishable-from-not-completely-specified-initial* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *r-distinguishable-k M q1 q2 0*
proof −
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

from *assms* **obtain** *x* **where** *x* ∈ (*inputs ?P*)
and ¬ (∃ *t*∈ *transitions ?P*. *t-source t* = (*q1*, *q2*) ∧ *t-input t* = *x*)
unfolding *completely-specified-state.simps* **by** *blast*

then have *x* ∈ (*inputs M*)
by (*simp add*: *assms*(*2*) *assms*(*3*))

have $*$: $\neg$ ($\exists$ *t1 t2*.
t1 $\in$ *transitions* (*from-FSM M q1*) $\wedge$
t2 $\in$ *transitions* (*from-FSM M q2*) $\wedge$
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*)
proof

assume $\exists$ *t1 t2*.
t1 $\in$ *transitions* (*from-FSM M q1*) $\wedge$
t2 $\in$ *transitions* (*from-FSM M q2*) $\wedge$
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*
then obtain *t1 t2* **where** *t1* $\in$ *transitions* (*from-FSM M q1*)
and *t2* $\in$ *transitions* (*from-FSM M q2*)
and *t-source t1* = *q1*
and *t-source t2* = *q2*
and *t-input t1* = *x*
and *t-input t2* = *x*
and *t-output t1* = *t-output t2*
by *blast*

let *?t* = ((*t-source t1*, *t-source t2*),*t-input t1*,*t-output t1*,(*t-target t1*,*t-target t2*))
let *?t1* = (*fst* (*t-source ?t*), *t-input ?t*, *t-output ?t*, *fst* (*t-target ?t*))
let *?t2* = (*snd* (*t-source ?t*), *t-input ?t*, *t-output ?t*, *snd* (*t-target ?t*))

have *t1-alt* : *t1* = *?t1*
by *auto*
have *t-source t2* = *snd* (*t-source ?t*)
by *auto*
moreover have *t-input t2* = *t-input ?t*
using ‹*t-input t1* = *x*› ‹*t-input t2* = *x*› **by** *auto*
moreover have *t-output t2* = *t-output ?t*
using ‹*t-output t1* = *t-output t2*› **by** *auto*
moreover have *t-target t2* = *snd* (*t-target ?t*)
by *auto*
ultimately have (*t-source t2*, *t-input t2*, *t-output t2*, *t-target t2*) = *?t2*
by *auto*
then have *t2-alt* : *t2* = *?t2*
by *auto*

have *?t1* $\in$ *transitions* (*from-FSM M q1*)
using ‹*t1* $\in$ *transitions* (*from-FSM M q1*)› **by** *auto*
moreover have *?t2* $\in$ *transitions* (*from-FSM M q2*)
using ‹*t2* $\in$ *transitions* (*from-FSM M q2*)› *t2-alt* **by** *auto*

ultimately have *?t* $\in$ *transitions ?P*

unfolding *product-transitions-def* **by** *force*
moreover have *t-source ?t* = (*q1*,*q2*) **using** ‹*t-source t1* = *q1*› ‹*t-source t2* = *q2*› **by** *auto*
moreover have *t-input ?t* = *x* **using** ‹*t-input t1* = *x*› **by** *auto*
ultimately show *False*
using ‹¬ (∃ *t*∈ *transitions ?P*. *t-source t* = (*q1*, *q2*) ∧ *t-input t* = *x*)› **by** *blast*
qed

have ∗∗: ⋀*t1* . *t1* ∈ *transitions M* ⟹ *t-source t1* = *q1* ⟹ *t1* ∈ *transitions* (*from-FSM M q1*)
and ∗∗∗: ⋀*t2* . *t2* ∈ *transitions M* ⟹ *t-source t2* = *q2* ⟹ *t2* ∈ *transitions* (*from-FSM M q2*)
by (*simp add*: *assms*(*2*,*3*))+

then show *?thesis* **unfolding** *r-distinguishable-k.simps*
using ‹*x* ∈ (*inputs M*)› ∗ **by** *blast*
qed

lemma *r-0-distinguishable-from-not-completely-specified* :
assumes ¬ *completely-specified-state* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1′*,*q2′*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and (*q1′*,*q2′*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
shows *r-distinguishable-k M q1′ q2′ 0*
proof −
have *q1′* ∈ *states M*
using *assms*(*2*) *assms*(*4*) **by** *simp*
have *q2′* ∈ *states M*
using *assms*(*3*) *assms*(*4*) **by** *simp*
show *?thesis*
using *r-0-distinguishable-from-not-completely-specified-initial*[*OF* - ‹*q1′* ∈ *states M*› ‹*q2′* ∈ *states M*›]
assms(*1*)
unfolding *completely-specified-state.simps product-simps from-FSM-simps*[*OF assms*(*2*)] *from-FSM-simps*[*OF assms*(*3*)] *from-FSM-simps*[*OF* ‹*q1′* ∈ *states M*›] *from-FSM-simps*[*OF* ‹*q2′* ∈ *states M*›]
product-transitions-alt-def **by** *auto*
qed

lemma *r-distinguishable-k-intersection-path* :
assumes ¬ *r-distinguishable-k M q1 q2 k*
and *length xs* ≤ *Suc k*
and *set xs* ⊆ (*inputs M*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*

```
shows ∃ p . path (product (from-FSM M q1) (from-FSM M q2)) (q1,q2) p ∧ map
fst (p-io p) = xs
using assms proof (induction k arbitrary: q1 q2 xs)
  case 0
  let ?P = (product (from-FSM M q1) (from-FSM M q2))
  show ?case
  proof (cases length xs < Suc 0)
    case True
    then have xs = [] by auto
    moreover have path (product (from-FSM M q1) (from-FSM M q2)) (q1,q2) []
      by (simp add: 0.prems(4) 0.prems(5) nil)
    moreover have map fst (p-io []) = [] by auto
    ultimately show ?thesis
      by simp
  next
    case False

    have completely-specified-state ?P (q1,q2)
    proof (rule ccontr)
      assume ¬ completely-specified-state ?P (q1,q2)
      then have r-distinguishable-k M q1 q2 0
        using r-0-distinguishable-from-not-completely-specified-initial
        by (metis 0.prems(4) 0.prems(5))
      then show False
        using 0.prems by simp
    qed
     then have ∗: ∀ x ∈ (inputs ?P) . ∃ t . t ∈ transitions ?P ∧ t-source t =
(q1,q2) ∧ t-input t = x
      unfolding completely-specified-state.simps by blast

    let ?x = hd xs
    have xs = [?x]
      using 0.prems(2) False
      by (metis Suc-length-conv le-neq-implies-less length-0-conv list.sel(1))

    have ?x ∈ (inputs M)
      using 0.prems(3) False by auto
   then obtain t where t ∈ transitions ?P and t-source t = (q1,q2) and t-input
t = ?x
      using ∗ 0.prems(4) 0.prems(5) by auto

    then have path ?P (q1,q2) [t]
      using single-transition-path by metis
    moreover have map fst (p-io [t]) = xs
      using ‹t-input t = ?x› ‹xs = [hd xs]› by auto
    ultimately show ?thesis
      by (metis (no-types, lifting))
  qed
next
```

case (*Suc k*)
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

show *?case*
proof (*cases length xs* ≤ *Suc k*)
case *True*
have ¬ *r-distinguishable-k M q1 q2 k*
using *Suc.prems*(*1*) **by** *auto*
show *?thesis*
using *Suc.IH*[*OF* ‹¬ *r-distinguishable-k M q1 q2 k*› *True Suc.prems*(*3*,*4*,*5*)]
by *assumption*
next
case *False*
then have *length xs* = *Suc* (*Suc k*)
using *Suc.prems*(*2*) **by** *auto*

then have *hd xs* ∈ (*inputs M*)
by (*metis Suc.prems*(*3*) *contra-subsetD hd-in-set length-greater-0-conv zero-less-Suc*)

have *set* (*tl xs*) ⊆ (*inputs M*)
by (*metis* ‹*length xs* = *Suc* (*Suc k*)› *Suc.prems*(*3*) *dual-order.trans hd-Cons-tl length-0-conv nat.simps*(*3*) *set-subset-Cons*)
have *length* (*tl xs*) ≤ *Suc k*
by (*simp add*: ‹*length xs* = *Suc* (*Suc k*)›)

let *?x* = *hd xs*
let *?xs* = *tl xs*

have ∀ *x* ∈ (*inputs M*) . ∃ *t* ∈ *transitions ?P* . *t-source t* = (*q1*,*q2*) ∧ *t-input t* = *x* ∧ ¬ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
proof
fix *x* **assume** *x* ∈ (*inputs M*)

have ¬(∃ *x* ∈ (*inputs M*) . ∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
using *Suc.prems* **by** *auto*
then have ∀ *x* ∈ (*inputs M*) . ∃ *t1 t2* . (*t1* ∈ *transitions M* ∧ *t2* ∈ *transitions M* ∧ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2* ∧ ¬ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
by *blast*
then obtain *t1 t2* **where** *t1* ∈ *transitions M*
and *t2* ∈ *transitions M*
and *t-source t1* = *q1*
and *t-source t2* = *q2*
and *t-input t1* = *x*
and *t-input t2* = *x*
and *p4*: *t-output t1* = *t-output t2*
and ∗∗: ¬ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*

using ‹$x \in$ (*inputs M*)› **by** *auto*

have *p1*: *t1* $\in$ *transitions* (*from-FSM M q1*)
by (*simp add*: *Suc.prems*(*4*) ‹*t1* $\in$ *FSM.transitions M*›)
have *p2*: *t2* $\in$ *transitions* (*from-FSM M q2*)
by (*simp add*: *Suc.prems*(*5*) ‹*t2* $\in$ *FSM.transitions M*›)
have *p3*: *t-input t1* = *t-input t2*
using ‹*t-input t1* = *x*› ‹*t-input t2* = *x*› **by** *auto*

have ∗∗∗: ((*q1*,*q2*), *x*, *t-output t1*, (*t-target t1*, *t-target t2*)) $\in$ *transitions ?P*
using ‹*t-source t1* = *q1*› ‹*t-source t2* = *q2*› ‹*t-input t1* = *x*› *p1 p2 p3 p4*
unfolding *product-transitions-alt-def*
by *blast*

show $\exists$ *t* $\in$ *transitions ?P* . *t-source t* = (*q1*,*q2*) $\wedge$ *t-input t* = *x* $\wedge$ $\neg$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
by (*metis* ∗∗ ∗∗∗ *fst-conv snd-conv*)
qed

then obtain *t* **where** *t* $\in$ *transitions ?P* **and** *t-source t* = (*q1*,*q2*) **and** *t-input t* = *?x*
and $\neg$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
using ‹*?x* $\in$ (*inputs M*)› **by** *blast*

have *fst* (*t-target t*) $\in$ *FSM.states M* **and** *snd* (*t-target t*) $\in$ *FSM.states M*
using *fsm-transition-target*[*OF* ‹*t* $\in$ *transitions ?P*›] **unfolding** *product-simps from-FSM-simps*[*OF* ‹*q1* $\in$ *states M*›] *from-FSM-simps*[*OF* ‹*q2* $\in$ *states M*›] **by** *auto*

then obtain *p* **where** *p-def*: *path* (*product* (*from-FSM M* (*fst* (*t-target t*))) (*from-FSM M* (*snd* (*t-target t*)))) (*t-target t*) *p*
and *map fst* (*p-io p*) = *?xs*
using *Suc.IH*[*OF* ‹$\neg$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*› ‹*length* (*tl xs*) $\leq$ *Suc k*› ‹*set* (*tl xs*) $\subseteq$ (*inputs M*)›] **by** *auto*

have *path ?P* (*t-target t*) *p*
using *product-from-path-previous*[*OF p-def* ‹*t* $\in$ *transitions ?P*› *Suc.prems*(*4*,*5*)] **by** *assumption*

have *path ?P* (*q1*,*q2*) (*t*#*p*)
using *path.cons*[*OF*‹*t* $\in$ *transitions ?P*› ‹*path ?P* (*t-target t*) *p*›] ‹*t-source t* = (*q1*,*q2*)› **by** *metis*
moreover have *map fst* (*p-io* (*t*#*p*)) = *xs*
using ‹*t-input t* = *?x*› ‹*map fst* (*p-io p*) = *?xs*›
by (*metis* (*no-types*, *lifting*) ‹*length xs* = *Suc* (*Suc k*)› ‹*t-input t* = *hd xs*› *fst-conv hd-Cons-tl length-greater-0-conv list.simps*(*9*) *zero-less-Suc*)
ultimately show *?thesis*
by (*metis* (*no-types*, *lifting*))
qed

qed

lemma *r-distinguishable-k-intersection-paths* :
assumes ¬(∃ *k* . *r-distinguishable-k M q1 q2 k*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows ∀ *xs* . *set xs* ⊆ (*inputs M*) ⟶ (∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*)
proof (*rule ccontr*)
assume ¬ (∀ *xs* . *set xs* ⊆ (*inputs M*) ⟶ (∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*))
then obtain *xs* **where** *set xs* ⊆ (*inputs M*)
and ¬ (∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*)
by *blast*

have ¬ *r-distinguishable-k M q1 q2* (*length xs*)
using *assms* **by** *auto*

show *False*
using *r-distinguishable-k-intersection-path*[*OF* ‹¬ *r-distinguishable-k M q1 q2* (*length xs*)› - ‹*set xs* ⊆ (*inputs M*)› *assms*(*2*,*3*)]
‹¬ (∃ *p* . *path* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) (*q1*,*q2*) *p* ∧ *map fst* (*p-io p*) = *xs*)› **by** *fastforce*
qed

39.1.1 Equivalence of R-Distinguishability Definitions

lemma *r-distinguishable-alt-def* :
assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows *r-distinguishable M q1 q2* ⟷ (∃ *k* . *r-distinguishable-k M q1 q2 k*)
proof
show *r-distinguishable M q1 q2* ⟹ ∃ *k*. *r-distinguishable-k M q1 q2 k*
proof (*rule ccontr*)

assume *r-distinguishable M q1 q2*
assume *c-assm*: ¬ (∃ *k*. *r-distinguishable-k M q1 q2 k*)

let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))

let *?f* = λ *t* . ¬ *r-distinguishable-k M* (*fst* (*t-source t*)) (*snd* (*t-source t*)) *0* ∧ ¬ (∃ *k* . *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*) ∧ *t-source t* ∈ *reachable-states ?P*
let *?ft* = *Set.filter ?f* (*transitions ?P*)

let *?PC* = *filter-transitions ?P ?f*

let *?PCR* = *restrict-to-reachable-states ?PC*

have *h-ft* : *transitions ?PC* = { *t* ∈ *transitions ?P* . *?f t* }
by *auto*

have *states-non-r-d-k* : ⋀ *q* . *q* ∈ *reachable-states ?PC* ⟹ ¬ (∃ *k* . *r-distinguishable-k M* (*fst q*) (*snd q*) *k*)
proof −
fix *q* **assume** *q* ∈ *reachable-states ?PC*
have *q* = *initial ?PC* ∨ (∃ *t* ∈ *transitions ?PC* . *q* = *t-target t*)
by (*metis* (*no-types*, *lifting*) ‹*q* ∈ *reachable-states* (*FSM.filter-transitions* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*)) (λ*t*. ¬ *r-distinguishable-k M* (*fst* (*t-source t*)) (*snd* (*t-source t*)) *0* ∧ (∄ *k*. *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*) ∧ *t-source t* ∈ *reachable-states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))))› *reachable-states-initial-or-target*)
then have *q* = (*q1*,*q2*) ∨ (∃ *t* ∈ *transitions ?PC* . *q* = *t-target t*)
by (*simp add*: *assms*(*1*) *assms*(*2*))

show ¬ (∃ *k* . *r-distinguishable-k M* (*fst q*) (*snd q*) *k*)
proof (*cases q* = (*q1*,*q2*))
case *True*
then show *?thesis* **using** *c-assm* **by** *auto*
next
case *False*
then obtain *t* **where** *t* ∈ *transitions ?PC* **and** *q* = *t-target t* **using** ‹*q* = (*q1*,*q2*) ∨ (∃ *t* ∈ *transitions ?PC* . *q* = *t-target t*)› **by** *blast*
then show *?thesis*
using *h-ft* **by** *blast*
qed
qed
then have *states-non-r-d-k-PCR*: ⋀ *q* . *q* ∈ *states ?PCR* ⟹ ¬ (∃ *k* . *r-distinguishable-k M* (*fst q*) (*snd q*) *k*)
unfolding *restrict-to-reachable-states-simps* **by** *blast*

have ⋀ *q* . *q* ∈ *reachable-states ?PC* ⟹ *completely-specified-state ?PC q*
proof −
fix *q* **assume** *q* ∈ *reachable-states ?PC*
then have *q* ∈ *reachable-states ?P*
using *filter-transitions-reachable-states* **by** *fastforce*

show *completely-specified-state ?PC q*
proof (*rule ccontr*)
assume ¬ *completely-specified-state ?PC q*
then obtain *x* **where** *x* ∈ (*inputs ?PC*)
and ¬(∃ *t* ∈ *transitions ?PC* . *t-source t* = *q* ∧ *t-input t* = *x*)
unfolding *completely-specified-state.simps* **by** *blast*
then have ¬(∃ *t* ∈ *transitions ?P* . *t-source t* = *q* ∧ *t-input t* = *x* ∧ *?f t*)
using *h-ft* **by** *blast*

then have *not-f* : $\bigwedge$ *t* . *t* $\in$ *transitions ?P* $\Longrightarrow$ *t-source t* = *q* $\Longrightarrow$ *t-input t* = *x* $\Longrightarrow$ $\neg$ *?f t*
by *blast*

have $\exists$ *k* . *r-distinguishable-k M* (*fst q*) (*snd q*) *k*
proof (*cases r-distinguishable-k M* (*fst q*) (*snd q*) *0*)
case *True*
then show *?thesis* **by** *blast*
next
case *False*

let *?tp* = {*t* . *t* $\in$ *transitions ?P* $\wedge$ *t-source t* = *q* $\wedge$ *t-input t* = *x*}
have *finite ?tp* **using** *fsm-transitions-finite*[*of ?P*] **by** *force*

have *k-ex* : $\forall$ *t* $\in$ *?tp* . $\exists$ *k* . $\forall$ *k′* . *k′* $\geq$ *k* $\longrightarrow$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k′*
proof
fix *t* **assume** *t* $\in$ *?tp*
then have $\neg$ *?f t* **using** *not-f* **by** *blast*
then obtain *k* **where** *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
using *False* ‹*t* $\in$ *?tp*›
using ‹*q* $\in$ *reachable-states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))› **by** *blast*
then have $\forall$ *k′* . *k′* $\geq$ *k* $\longrightarrow$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k′*
using *nat-induct-at-least* **by** *fastforce*
then show $\exists$ *k* . $\forall$ *k′* . *k′* $\geq$ *k* $\longrightarrow$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k′* **by** *auto*
qed

obtain *k* **where** *k-def* : $\bigwedge$ *t* . *t* $\in$ *?tp* $\Longrightarrow$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
using *finite-set-min-param-ex*[*OF* ‹*finite ?tp*›, *of* λ *t k′* . *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k′*] *k-ex* **by** *blast*

then have $\forall$ *t* $\in$ *transitions ?P* . (*t-source t* = *q* $\wedge$ *t-input t* = *x*) $\longrightarrow$ *r-distinguishable-k M* (*fst* (*t-target t*)) (*snd* (*t-target t*)) *k*
by *blast*

have *r-distinguishable-k M* (*fst q*) (*snd q*) (*Suc k*)
proof −
have $\bigwedge$ *t1 t2* . *t1* $\in$ *transitions M* $\Longrightarrow$ *t2* $\in$ *transitions M* $\Longrightarrow$ *t-source t1* = *fst q* $\Longrightarrow$ *t-source t2* = *snd q* $\Longrightarrow$ *t-input t1* = *x* $\Longrightarrow$ *t-input t2* = *x* $\Longrightarrow$ *t-output t1* = *t-output t2* $\Longrightarrow$ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*
proof −
fix *t1 t2* **assume** *t1* $\in$ *transitions M*
and *t2* $\in$ *transitions M*

```
                    and t-source t1 = fst q
                    and t-source t2 = snd q
                    and t-input t1 = x
                    and t-input t2 = x
                    and t-output t1 = t-output t2
            then have t-input t1 = t-input t2
                 and t-output t1 = t-output t2 by auto


            have (fst q, snd q) ∈ reachable-states ?P using ‹q ∈ reachable-states
?P› by (metis prod.collapse)
            then have (fst q, snd q) ∈ states ?P using reachable-state-is-state by
metis
            then have fst q ∈ states (from-FSM M q1)
                 and snd q ∈ states (from-FSM M q2)
              unfolding product-simps by auto

            have t1 ∈ transitions (from-FSM M q1)
              by (simp add: ‹t1 ∈ FSM.transitions M› assms(1))
            moreover have t2 ∈ transitions (from-FSM M q2)
              by (simp add: ‹t2 ∈ FSM.transitions M› assms(2))
            moreover have t-source ((t-source t1, t-source t2),t-input t1,t-output
t1,(t-target t1,t-target t2)) = q
              using ‹t-source t1 = fst q› ‹t-source t2 = snd q› by auto
             moreover have t-input ((t-source t1, t-source t2),t-input t1,t-output
t1,(t-target t1,t-target t2)) = x
              using ‹t-input t1 = x› by auto
               ultimately have tt: ((t-source t1, t-source t2),t-input t1,t-output
t1,(t-target t1,t-target t2)) ∈ ?tp
              unfolding product-transitions-alt-def
               using ‹t-input t1 = x› ‹t-input t2 = x› ‹t-output t1 = t-output t2›
by fastforce


            show r-distinguishable-k M (t-target t1) (t-target t2) k
              using k-def[OF tt] by auto
          qed

          moreover have x ∈ (inputs M)
        using ‹x ∈ (inputs ?PC)› unfolding filter-transitions-simps product-simps
from-FSM-simps[OF ‹q1 ∈ states M›] from-FSM-simps[OF ‹q2 ∈ states M›]
            by blast
          ultimately show ?thesis
            unfolding r-distinguishable-k.simps by blast
        qed
        then show ?thesis by blast
      qed

      then show False
```

```
      using states-non-r-d-k[OF ‹q ∈ reachable-states ?PC›] by blast
    qed
  qed
  then have ⋀ q . q ∈ states ?PCR ⟹ completely-specified-state ?PCR q
    unfolding restrict-to-reachable-states-simps completely-specified-state.simps
    by blast
  then have completely-specified ?PCR
    using completely-specified-states by blast
  moreover have is-submachine ?PCR ?P
  proof −
    have is-submachine ?PC ?P
      unfolding is-submachine.simps filter-transitions-simps by blast
    moreover have is-submachine ?PCR ?PC
      unfolding is-submachine.simps restrict-to-reachable-states-simps
      using reachable-state-is-state by fastforce
    ultimately show ?thesis
      using submachine-transitive by blast
  qed
  ultimately have r-compatible M q1 q2
    unfolding r-compatible-def by blast
  then show False using ‹r-distinguishable M q1 q2›
    by blast
 qed


 show ∃ k. r-distinguishable-k M q1 q2 k ⟹ r-distinguishable M q1 q2
 proof (rule ccontr)
  assume ∗: ¬ r-distinguishable M q1 q2
  assume ∗∗: ∃ k. r-distinguishable-k M q1 q2 k
  then obtain k where r-distinguishable-k M q1 q2 k by auto
  then show False
  using ∗ assms proof (induction k arbitrary: q1 q2)
    case 0
   then obtain S where is-submachine S (product (from-FSM M q1) (from-FSM
M q2))
                  and completely-specified S
      by (meson r-compatible-def)
     then have completely-specified-state (product (from-FSM M q1) (from-FSM
M q2)) (initial (product (from-FSM M q1) (from-FSM M q2)))
      using complete-submachine-initial by metis
       then show False using r-distinguishable-k-0-not-completely-specified[OF
0.prems(1,3,4) ] by metis
  next
    case (Suc k)
    then show False
    proof (cases r-distinguishable-k M q1 q2 k)
      case True
      then show ?thesis
        using Suc.IH Suc.prems by blast
```

next
case *False*
then obtain x **where** $x \in$ (*inputs M*)
and $\forall$ *y q1′ q2′*. (*q1*, *x*, *y*, *q1′*) $\in$ *transitions M* $\wedge$ (*q2*, *x*, *y*, *q2′*) $\in$ *transitions M* $\longrightarrow$ *r-distinguishable-k M q1′ q2′ k*
using *Suc.prems*(*1*) **by** *fastforce*

from *Suc* **obtain** *S* **where** *is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
and *completely-specified S*
by (*meson r-compatible-def*)

have $x \in$ (*inputs* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))
by (*simp add*: *Suc.prems*(*4*) ‹*x* $\in$ *FSM.inputs M*›)
then have $x \in$ (*inputs S*)
using ‹*is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
by (*metis is-submachine.elims*(*2*))

moreover have *initial S* = (*q1*,*q2*)
using ‹*is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
by (*simp add*: *Suc.prems*(*3*) *Suc.prems*(*4*))
ultimately obtain *y q1′ q2′* **where** ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) $\in$ *transitions S*
using ‹*completely-specified S*› **using** *fsm-initial* **by** *fastforce*
then have ((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) $\in$ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using ‹*is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))›
by *auto*
then have (*q1*, *x*, *y*, *q1′*) $\in$ *transitions* (*from-FSM M q1*) **and** (*q2*, *x*, *y*, *q2′*) $\in$ *transitions* (*from-FSM M q2*)
unfolding *product-transitions-def* **by** *force*+
then have (*q1*, *x*, *y*, *q1′*) $\in$ *transitions M* **and** (*q2*, *x*, *y*, *q2′*) $\in$ *transitions M*
by (*simp add*: *Suc.prems*(*3*,*4*))+
then have *r-distinguishable-k M q1′ q2′ k*
using ‹$\forall$ *y q1′ q2′*. (*q1*, *x*, *y*, *q1′*) $\in$ *transitions M* $\wedge$ (*q2*, *x*, *y*, *q2′*) $\in$ *transitions M* $\longrightarrow$ *r-distinguishable-k M q1′ q2′ k*› **by** *blast*
have *r-distinguishable M q1′ q2′*
by (*metis* (*no-types*) *Suc.IH* ‹(*q1*, *x*, *y*, *q1′*) $\in$ *FSM.transitions M*› ‹(*q2*, *x*, *y*, *q2′*) $\in$ *FSM.transitions M*› ‹*r-distinguishable-k M q1′ q2′ k*› *fsm-transition-target snd-conv*)
moreover have $\exists$ *S′* . *completely-specified S′* $\wedge$ *is-submachine S′* (*product* (*from-FSM M q1′*) (*from-FSM M q2′*))
using *submachine-transition-complete-product-from*[*OF* ‹*is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))› ‹*completely-specified S*› ‹((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) $\in$ *transitions S*› *Suc.prems*(*3*,*4*)]
submachine-transition-product-from[*OF* ‹*is-submachine S* (*product* (*from-FSM M q1*) (*from-FSM M q2*))› ‹((*q1*,*q2*),*x*,*y*,(*q1′*,*q2′*)) $\in$ *transitions S*› *Suc.prems*(*3*,*4*)] **by** *blast*

ultimately show *False* **unfolding** *r-compatible-def* **by** *blast*
qed
qed
qed
qed

39.2 Bounds

inductive *is-least-r-d-k-path* :: ('a, 'b, 'c) *fsm* ⇒ 'a ⇒ 'a ⇒ (('a × 'a) × 'b × *nat*) *list* ⇒ *bool* **where**
immediate[*intro*!] : $x \in$ (*inputs M*) ⟹ ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟹ *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)] |
step[*intro*!] : *Suc k* = (*LEAST k'* . *r-distinguishable-k M q1 q2 k'*)
⟹ *x* ∈ (*inputs M*)
⟹ (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
⟹ *t1* ∈ *transitions M*
⟹ *t2* ∈ *transitions M*
⟹ *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*
⟹ *is-least-r-d-k-path M* (*t-target t1*) (*t-target t2*) *p*
⟹ *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)

inductive-cases *is-least-r-d-k-path-immediate-elim*[*elim*!]: *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)]
inductive-cases *is-least-r-d-k-path-step-elim*[*elim*!]: *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)

lemma *is-least-r-d-k-path-nonempty* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows *p* ≠ []
using *is-least-r-d-k-path.cases*[*OF assms*] **by** *blast*

lemma *is-least-r-d-k-path-0-extract* :
assumes *is-least-r-d-k-path M q1 q2* [*t*]
shows ∃ *x* . *t* = ((*q1*,*q2*),*x*,*0*)
using *is-least-r-d-k-path.cases*[*OF assms*]
by (*metis* (*no-types*, *lifting*) *list.inject is-least-r-d-k-path-nonempty*)

lemma *is-least-r-d-k-path-Suc-extract* :
assumes *is-least-r-d-k-path M q1 q2* (*t*#*t'*#*p*)
shows ∃ *x k* . *t* = ((*q1*,*q2*),*x*,*Suc k*)
using *is-least-r-d-k-path.cases*[*OF assms*]
by (*metis* (*no-types*, *lifting*) *list.distinct*(*1*) *list.inject*)

lemma *is-least-r-d-k-path-Suc-transitions* :

assumes *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)
shows (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
using *is-least-r-d-k-path-step-elim*[*OF assms*]
Suc-inject[*of - k*]
by *metis*

lemma *is-least-r-d-k-path-is-least* :
assumes *is-least-r-d-k-path M q1 q2* (*t*#*p*)
shows *r-distinguishable-k M q1 q2* (*snd* (*snd t*)) ∧ (*snd* (*snd t*)) = (*LEAST k′* . *r-distinguishable-k M q1 q2 k′*)
proof (*cases p*)
case *Nil*
then obtain *x* **where** *t* = ((*q1*,*q2*),*x*,*0*) **and** *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)]
using *assms is-least-r-d-k-path-0-extract* **by** *metis*
have ∗: *r-distinguishable-k M q1 q2 0*
using *is-least-r-d-k-path-immediate-elim*[*OF* ‹*is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)]›]
unfolding *r-distinguishable-k.simps* **by** *auto*
then have (∃ *k*. *r-distinguishable-k M q1 q2 k*)
by *blast*
then have *0* = (*LEAST k′* . *r-distinguishable-k M q1 q2 k′*)
using ‹*r-distinguishable-k M q1 q2 0*› **by** *auto*
moreover have *snd* (*snd t*) = *0*
using ‹*t* = ((*q1*,*q2*),*x*,*0*)› **by** *auto*
ultimately show *?thesis* **using** ∗ **by** *auto*
next
case (*Cons t′ p′*)
then obtain *x k* **where** *t* = ((*q1*,*q2*),*x*,*Suc k*) **and** *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*t′*#*p′*)
using *assms is-least-r-d-k-path-Suc-extract* **by** *metis*

have *x* ∈ (*inputs M*)
using *is-least-r-d-k-path-step-elim*[*OF* ‹*is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*t′*#*p′*)›] **by** *blast*
moreover have (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
using *is-least-r-d-k-path-Suc-transitions*[*OF* ‹*is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#(*t′*#*p′*))›] **by** *assumption*
ultimately have *r-distinguishable-k M q1 q2* (*Suc k*)
unfolding *r-distinguishable-k.simps* **by** *blast*
moreover have *Suc k* = (*LEAST k′* . *r-distinguishable-k M q1 q2 k′*)
using *is-least-r-d-k-path-step-elim*[*OF* ‹*is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*t′*#*p′*)›] **by** *blast*
ultimately show *?thesis*
by (*metis* ‹*t* = ((*q1*, *q2*), *x*, *Suc k*)› *snd-conv*)
qed

lemma *r-distinguishable-k-least-next* :
assumes $\exists$ *k* . *r-distinguishable-k M q1 q2 k*
and (*LEAST k* . *r-distinguishable-k M q1 q2 k*) = *Suc k*
and *x* $\in$ (*inputs M*)
and $\forall$ *t1*$\in$ *transitions M*. $\forall$ *t2*$\in$ *transitions M*.
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2* $\longrightarrow$
r-distinguishable-k M (*t-target t1*) (*t-target t2*) *k*
shows $\exists$ *t1* $\in$ *transitions M* . $\exists$ *t2* $\in$ *transitions M* . (*t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*) $\wedge$ (*LEAST k* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*) = *k*
proof −
have *r-distinguishable-k M q1 q2* (*Suc k*)
using *assms LeastI* **by** *metis*
moreover have $\neg$ *r-distinguishable-k M q1 q2 k*
using *assms*(*2*) **by** (*metis lessI not-less-Least*)

have $**$: ($\forall$ *t1*$\in$ *transitions M*.
$\forall$ *t2*$\in$ *transitions M*.
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2* $\longrightarrow$
(*LEAST k′* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*) $\leq$ *k*)
using *assms*(*3*,*4*) *Least-le* **by** *blast*

show *?thesis* **proof** (*rule ccontr*)
assume *assm* : $\neg$ ($\exists$ *t1*$\in$(*transitions M*).
$\exists$ *t2*$\in$(*transitions M*).
(*t-source t1* = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*) $\wedge$
(*LEAST k*. *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*) = *k*)

let *?hs* = {(*t1*,*t2*) | *t1 t2* . *t1* $\in$ *transitions M* $\wedge$ *t2* $\in$ *transitions M* $\wedge$ *t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*}
have *finite ?hs*
proof −
have *?hs* $\subseteq$ (*transitions M* $\times$ *transitions M*) **by** *blast*
moreover have *finite* (*transitions M* $\times$ *transitions M*) **using** *fsm-transitions-finite* **by** *blast*
ultimately show *?thesis*
by (*simp add*: *finite-subset*)
qed
have *fk-def* : $\bigwedge$ *tt* . *tt* $\in$ *?hs* $\Longrightarrow$ *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)

proof −
fix *tt* **assume** *tt* ∈ *?hs*
then have (*fst tt*) ∈ *transitions M* ∧ (*snd tt*) ∈ *transitions M* ∧ *t-source* (*fst tt*) = *q1* ∧ *t-source* (*snd tt*) = *q2* ∧ *t-input* (*fst tt*) = *x* ∧ *t-input* (*snd tt*) = *x* ∧ *t-output* (*fst tt*) = *t-output* (*snd tt*)
by *force*
then have ∃ *k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*
using *assms*(*4*) **by** *blast*
then show *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)
using *LeastI2-wellorder* **by** *blast*
qed

let *?k* = *Max* (*image* (λ *tt* . (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)) *?hs*)
have ⋀ *t1 t2* . *t1* ∈ *transitions M* ⟹ *t2* ∈ *transitions M* ⟹ *t-source t1* = *q1* ⟹ *t-source t2* = *q2* ⟹ *t-input t1* = *x* ⟹ *t-input t2* = *x* ⟹ *t-output t1* = *t-output t2* ⟹ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *?k*
proof −
fix *t1 t2* **assume** *t1* ∈ *transitions M*
and *t2* ∈ *transitions M*
and *t-source t1* = *q1*
and *t-source t2* = *q2*
and *t-input t1* = *x*
and *t-input t2* = *x*
and *t-output t1* = *t-output t2*
then have (*t1*,*t2*) ∈ *?hs* **by** *force*
then have *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) ((λ *tt* . (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)) (*t1*,*t2*))
using *fk-def* **by** *force*
have (λ *tt* . (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)) (*t1*,*t2*) ≤ *?k*
using ‹(*t1*,*t2*) ∈ *?hs*› ‹*finite ?hs*›
by (*meson Max.coboundedI finite-imageI image-iff*)
show *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *?k*
using *r-distinguishable-k-by-larger*[*OF* ‹*r-distinguishable-k M* (*t-target t1*) (*t-target t2*) ((λ *tt* . (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)) (*t1*,*t2*))› ‹(λ *tt* . (*LEAST k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*)) (*t1*,*t2*) ≤ *?k*›] **by** *assumption*
qed

then have *r-distinguishable-k M q1 q2* (*Suc ?k*)
unfolding *r-distinguishable-k.simps*
using ‹*x* ∈ (*inputs M*)› **by** *blast*

have *?hs* ≠ {}
proof
assume *?hs* = {}

then have *r-distinguishable-k M q1 q2 0*
unfolding *r-distinguishable-k.simps* **using** ‹$x \in$ (*inputs M*)› **by** *force*
then show *False*
using *assms*(*2*) **by** *auto*
qed

have $\bigwedge$ *t1 t2* . *t1*$\in$ *transitions M* $\Longrightarrow$
t2$\in$ *transitions M* $\Longrightarrow$
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2* $\Longrightarrow$
(*LEAST k′* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*) < *k*
proof −
fix *t1 t2* **assume** *t1*$\in$ *transitions M* **and** *t2*$\in$ *transitions M* **and** *t12-def* : *t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*
have (*LEAST k′* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*) $\leq$ *k* **using** ‹*t1*$\in$ *transitions M*› ‹*t2*$\in$ *transitions M*› *t12-def* ** **by** *blast*
moreover have (*LEAST k′* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*) $\neq$ *k* **using** ‹*t1*$\in$ *transitions M*› ‹*t2*$\in$ *transitions M*› *t12-def assm* **by** *blast*
ultimately show (*LEAST k′* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*) < *k* **by** *auto*
qed
moreover have $\bigwedge$ *tt* . *tt* $\in$ *?hs* $\Longrightarrow$ (*fst tt*) $\in$ *transitions M* $\wedge$ (*snd tt*) $\in$ *transitions M* $\wedge$ *t-source* (*fst tt*) = *q1* $\wedge$ *t-source* (*snd tt*) = *q2* $\wedge$ *t-input* (*fst tt*) = *x* $\wedge$ *t-input* (*snd tt*) = *x* $\wedge$ *t-output* (*fst tt*) = *t-output* (*snd tt*)
by *force*
ultimately have $\bigwedge$ *tt* . *tt* $\in$ *?hs* $\Longrightarrow$ (*LEAST k′* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k′*) < *k* **by** *blast*
moreover obtain *tt* **where** *tt* $\in$ *?hs* **and** *?k* = (*LEAST k′* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k′*)
using *Max-elem*[*OF* ‹*finite ?hs*› ‹*?hs* $\neq$ {}›, *of* λ *tt* . (*LEAST k′* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k′*)] **by** *blast*
ultimately have *?k* < *k*
using ‹*finite ?hs*› **by** *presburger*

then show *False*
using *assms*(*2*) ‹*r-distinguishable-k M q1 q2* (*Suc ?k*)›
by (*metis* (*no-types*, *lifting*) *Suc-mono not-less-Least*)
qed
qed

lemma *is-least-r-d-k-path-length-from-r-d* :
assumes $\exists$ *k* . *r-distinguishable-k M q1 q2 k*
shows $\exists$ *t p* . *is-least-r-d-k-path M q1 q2* (*t*#*p*) $\wedge$ *length* (*t*#*p*) = *Suc* (*LEAST k* . *r-distinguishable-k M q1 q2 k*)
proof −
let *?k* = *LEAST k* . *r-distinguishable-k M q1 q2 k*

have *r-distinguishable-k M q1 q2 ?k*
using *assms LeastI* **by** *blast*

then show *?thesis* **using** *assms* **proof** (*induction ?k arbitrary*: *q1 q2*)
case *0*
then have *r-distinguishable-k M q1 q2 0* **by** *auto*
then obtain *x* **where** $x \in$ (*inputs M*) **and** $\neg$ ($\exists$ *t1* $\in$ *transitions M* . $\exists$ *t2* $\in$ *transitions M* . *t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*)
unfolding *r-distinguishable-k.simps* **by** *blast*
then have *is-least-r-d-k-path M q1 q2* [((*q1*,*q2*),*x*,*0*)]
by *auto*
then show *?case* **using** *0.hyps*
by (*metis length-Cons list.size*(*3*))
next
case (*Suc k*)
then have *r-distinguishable-k M q1 q2* (*Suc k*) **by** *auto*
moreover have $\neg$ *r-distinguishable-k M q1 q2 k*
using *Suc* **by** (*metis lessI not-less-Least*)
ultimately obtain *x* **where** $x \in$ (*inputs M*) **and** $*$: ($\forall$ *t1*$\in$(*transitions M*).
$\forall$ *t2*$\in$(*transitions M*).
t-source t1 = *q1* $\wedge$
t-source t2 = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2* $\longrightarrow$
r-distinguishable-k M (*t-target t1*) (*t-target t2*) *k*)
unfolding *r-distinguishable-k.simps* **by** *blast*

obtain *t1 t2* **where** *t1* $\in$ *transitions M* **and** *t2* $\in$ *transitions M*
and *t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*
and *k* = (*LEAST k. r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
using *r-distinguishable-k-least-next*[*OF Suc.prems*(*2*) - ‹*x* $\in$ (*inputs M*)› $*$] *Suc.hyps*(*2*) **by** *metis*
then have *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) (*LEAST k. r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)
using $*$ **by** *metis*

then obtain *t′ p′* **where** *is-least-r-d-k-path M* (*t-target t1*) (*t-target t2*) (*t′* # *p′*)
and *length* (*t′* # *p′*) = *Suc* (*Least* (*r-distinguishable-k M* (*t-target t1*) (*t-target t2*)))
using *Suc.hyps*(*1*)[*OF* ‹*k* = (*LEAST k. r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)›] **by** *blast*

then have *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*t′*#*p′*)
using *is-least-r-d-k-path.step*[*OF Suc.hyps*(*2*) ‹*x* $\in$ (*inputs M*)› $*$ ‹*t1* $\in$ *transitions M*› ‹*t2* $\in$ *transitions M*› ‹*t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*›]

by *auto*

show *?case*
by (*metis* (*no-types*) *Suc.hyps*(*2*) ‹*is-least-r-d-k-path M q1 q2* (((*q1*, *q2*), *x*, *Suc k*) # *t′* # *p′*)› ‹*k* = (*LEAST k*. *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)› ‹*length* (*t′* # *p′*) = *Suc* (*Least* (*r-distinguishable-k M* (*t-target t1*) (*t-target t2*)))› *length-Cons*)
qed
qed

lemma *is-least-r-d-k-path-states* :
assumes *is-least-r-d-k-path M q1 q2 p*
and *q1* ∈ *states M*
and *q2* ∈ *states M*
shows *set* (*map fst p*) ⊆ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *assms* **proof** (*induction p*)
case (*immediate x M q1 q2*)
then show *?case* **by** *auto*
next
case (*step k M q1 q2 x t1 t2 p*)
then have *t-target t1* ∈ *states M* **and** *t-target t2* ∈ *states M* **by** *blast+*
have *t-source t1* = *q1* **and** *t-source t2* = *q2*
using *step* **by** *metis+*

have *t-target t1* ∈ *states* (*from-FSM M q1*)
by (*simp add*: ‹*t-target t1* ∈ *FSM.states M*› *step.prems*(*1*))
have *t-target t2* ∈ *states* (*from-FSM M q2*)
by (*simp add*: ‹*t-target t2* ∈ *FSM.states M*› *step.prems*(*2*))

have *t1* ∈ *transitions* (*from-FSM M q1*)
by (*simp add*: *step.hyps*(*4*) *step.prems*(*1*))
have *t2* ∈ *transitions* (*from-FSM M q2*)
by (*simp add*: *step.hyps*(*5*) *step.prems*(*2*))
have *t-input t1* = *t-input t2* **using** *step.hyps*(*6*) **by** *auto*
have *t-output t1* = *t-output t2* **using** *step.hyps*(*6*) **by** *auto*

have ((*q1*,*q2*),*t-input t1*, *t-output t1*, (*t-target t1*, *t-target t2*)) ∈ *transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using ‹*t1* ∈ *transitions* (*from-FSM M q1*)› ‹*t2* ∈ *transitions* (*from-FSM M q2*)› ‹*t-input t1* = *t-input t2*› ‹*t-output t1* = *t-output t2*› ‹*t-source t1* = *q1*› ‹*t-source t2* = *q2*›
unfolding *product-transitions-alt-def* **by** *blast*

then have (*t-target t1*, *t-target t2*) ∈ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *fsm-transition-target*

by (*metis snd-conv*)

moreover have *states* (*product* (*from-FSM M* (*t-target t1*)) (*from-FSM M* (*t-target t2*))) ⊆ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *calculation step.prems*(*1*) *step.prems*(*2*) **by** *auto*

moreover have *set* (*map fst p*) ⊆ *states* (*product* (*from-FSM M* (*t-target t1*)) (*from-FSM M* (*t-target t2*)))
using *step.IH* ‹*t-target t1* ∈ *states* (*from-FSM M q1*)› ‹*t-target t2* ∈ *states* (*from-FSM M q2*)›
using *step.prems* **by** *auto*
ultimately have *set* (*map fst p*) ⊆ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
by *blast*

moreover have *set* (*map fst* [((*q1*,*q2*),*x*,*Suc k*)]) ⊆ *states* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *fsm-transition-source*[*OF* ‹((*q1*, *q2*), *t-input t1*, *t-output t1*, *t-target t1*, *t-target t2*) ∈ (*transitions* (*product* (*from-FSM M q1*) (*from-FSM M q2*)))›]
by *auto*

ultimately show *?case*
by *auto*
qed

lemma *is-least-r-d-k-path-decreasing* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows ∀ *t′* ∈ *set* (*tl p*) . *snd* (*snd t′*) < *snd* (*snd* (*hd p*))
using *assms* **proof**(*induction p*)
case (*immediate x M q1 q2*)
then show *?case* **by** *auto*
next
case (*step k M q1 q2 x t1 t2 p*)
then show *?case* **proof** (*cases p*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*Cons t′ p′*)
then have *is-least-r-d-k-path M* (*t-target t1*) (*t-target t2*) (*t′*#*p′*) **using** *step.hyps*(*7*) **by** *auto*

have *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) (*snd* (*snd t′*))
and *snd* (*snd t′*) = (*LEAST k′*. *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*)
using *is-least-r-d-k-path-is-least*[*OF* ‹*is-least-r-d-k-path M* (*t-target t1*) (*t-target t2*) (*t′*#*p′*)›] **by** *auto*

have *snd* (*snd t′*) < *Suc k*
by (*metis* ‹*snd* (*snd t′*) = (*LEAST k′*. *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k′*)› *local.step*(*3*) *local.step*(*4*) *local.step*(*5*) *local.step*(*6*) *not-less-Least not-less-eq*)
moreover have ∀ *t′′*∈*set p*. *snd* (*snd t′′*) ≤ *snd* (*snd t′*)
using *Cons step.IH* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *is-least-r-d-k-path-suffix* :
assumes *is-least-r-d-k-path M q1 q2 p*
and *i* < *length p*
shows *is-least-r-d-k-path M* (*fst* (*fst* (*hd* (*drop i p*)))) (*snd* (*fst* (*hd* (*drop i p*)))) (*drop i p*)
using *assms* **proof**(*induction p arbitrary*: *i*)
case (*immediate x M q1 q2*)
then show *?case* **by** *auto*
next
case (*step k M q1 q2 x t1 t2 p*)
then have *is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)
by *blast*

have ⋀ *i* . *i* < *length p* ⟹ *is-least-r-d-k-path M* (*fst* (*fst* (*hd* (*drop* (*Suc i*) (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*snd* (*fst* (*hd* (*drop* (*Suc i*) (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*drop* (*Suc i*) (((*q1*, *q2*), *x*, *Suc k*) # *p*))
using *step.IH* **by** *simp*
then have ⋀ *i* . *i* < *length* (((*q1*, *q2*), *x*, *Suc k*) # *p*) ⟹ *i* > *0* ⟹ *is-least-r-d-k-path M* (*fst* (*fst* (*hd* (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*snd* (*fst* (*hd* (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))
by (*metis Suc-less-eq gr0-implies-Suc length-Cons*)
moreover have ⋀ *i* . *i* < *length* (((*q1*, *q2*), *x*, *Suc k*) # *p*) ⟹ *i* = *0* ⟹ *is-least-r-d-k-path M* (*fst* (*fst* (*hd* (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*snd* (*fst* (*hd* (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))))) (*drop i* (((*q1*, *q2*), *x*, *Suc k*) # *p*))
using ‹*is-least-r-d-k-path M q1 q2* (((*q1*,*q2*),*x*,*Suc k*)#*p*)› **by** *auto*
ultimately show *?case*
using *step.prems* **by** *blast*
qed

lemma *is-least-r-d-k-path-distinct* :
assumes *is-least-r-d-k-path M q1 q2 p*
shows *distinct* (*map fst p*)
using *assms* **proof**(*induction p*)
case (*immediate x M q1 q2*)
then show *?case* **by** *auto*
next
case (*step k M q1 q2 x t1 t2 p*)

```
  then have is-least-r-d-k-path M q1 q2 (((q1,q2),x,Suc k)#p)
    by blast

  show ?case proof (rule ccontr)
    assume ¬ distinct (map fst (((q1, q2), x, Suc k) # p))
    then have (q1,q2) ∈ set (map fst p)
      using step.IH by simp
    then obtain i where i < length p and (map fst p) ! i = (q1,q2)
      by (metis distinct-Ex1 length-map step.IH)
    then obtain x′ k′ where hd (drop i p) = ((q1,q2),x′,k′)
      by (metis fst-conv hd-drop-conv-nth nth-map old.prod.exhaust)

    have is-least-r-d-k-path M q1 q2 (drop i p)
     using is-least-r-d-k-path-suffix[OF ‹is-least-r-d-k-path M q1 q2 (((q1,q2),x,Suc
k)#p)›] ‹i < length p›
    proof −
      have snd (fst (hd (drop i p))) = q2
        using ‹hd (drop i p) = ((q1, q2), x′, k′)› by auto
      then show ?thesis
          by (metis (no-types) ‹hd (drop i p) = ((q1, q2), x′, k′)› ‹i < length p›
fst-conv is-least-r-d-k-path-suffix step.hyps(7))
    qed

    have k′ < Suc k
     using is-least-r-d-k-path-decreasing[OF ‹is-least-r-d-k-path M q1 q2 (((q1,q2),x,Suc
k)#p)›]
      by (metis Cons-nth-drop-Suc ‹hd (drop i p) = ((q1, q2), x′, k′)› ‹i < length
p› hd-in-set in-set-dropD list.sel(1) list.sel(3) list.simps(3) snd-conv)
    moreover have k′ = (LEAST k′. r-distinguishable-k M q1 q2 k′)
        using is-least-r-d-k-path-is-least ‹is-least-r-d-k-path M q1 q2 (drop i p)›
is-least-r-d-k-path-is-least
      by (metis Cons-nth-drop-Suc ‹hd (drop i p) = ((q1, q2), x′, k′)› ‹i < length
p› hd-drop-conv-nth snd-conv)
    ultimately show False
      using step.hyps(1) dual-order.strict-implies-not-eq by blast
  qed
qed


lemma r-distinguishable-k-least-bound :
  assumes ∃ k . r-distinguishable-k M q1 q2 k
      and q1 ∈ states M
      and q2 ∈ states M
  shows (LEAST k . r-distinguishable-k M q1 q2 k) ≤ (size (product (from-FSM
M q1) (from-FSM M q2)))
proof (rule ccontr)
  assume ¬ (LEAST k. r-distinguishable-k M q1 q2 k) ≤ (size (product (from-FSM
M q1) (from-FSM M q2)))
   then have c-assm : (size (product (from-FSM M q1) (from-FSM M q2))) <
```

(*LEAST k. r-distinguishable-k M q1 q2 k*)
by *linarith*

obtain *t p* **where** *is-least-r-d-k-path M q1 q2* (*t* # *p*)
and *length* (*t* # *p*) = *Suc* (*LEAST k. r-distinguishable-k M q1 q2 k*)
using *is-least-r-d-k-path-length-from-r-d*[*OF assms*(*1*)] **by** *blast*
then have (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) < *length* (*t* # *p*)
using *c-assm* **by** *linarith*

have *distinct* (*map fst* (*t* # *p*))
using *is-least-r-d-k-path-distinct*[*OF* ‹*is-least-r-d-k-path M q1 q2* (*t* # *p*)›] **by** *assumption*
then have *card* (*set* (*map fst* (*t* # *p*))) = *length* (*t* # *p*)
using *distinct-card* **by** *fastforce*
moreover have *card* (*set* (*map fst* (*t* # *p*))) ≤ *size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
using *is-least-r-d-k-path-states*[*OF* ‹*is-least-r-d-k-path M q1 q2* (*t* # *p*)› *assms*(*2*,*3*)] *fsm-states-finite card-mono* **unfolding** *size-def* **by** *blast*
ultimately have *length* (*t* # *p*) ≤ *size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))
by (*metis*)
then show *False*
using ‹*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*)) < *length* (*t* # *p*)› **by** *linarith*
qed

39.3 Deciding R-Distinguishability

fun *r-distinguishable-k-least* :: (*'a*, *'b*::*linorder*, *'c*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ *nat* ⇒ (*nat* × *'b*) *option* **where**
r-distinguishable-k-least M q1 q2 0 = (*case find* (λ *x* . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) (*sort* (*inputs-as-list M*)) *of*
Some x ⇒ *Some* (*0*,*x*) |
None ⇒ *None*) |
r-distinguishable-k-least M q1 q2 (*Suc n*) = (*case r-distinguishable-k-least M q1 q2 n of*
Some k ⇒ *Some k* |
None ⇒ (*case find* (λ *x* . ∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *n*) (*sort* (*inputs-as-list M*)) *of*
Some x ⇒ *Some* (*Suc n*,*x*) |
None ⇒ *None*))

lemma *r-distinguishable-k-least-ex* :
assumes *r-distinguishable-k-least M q1 q2 k* = *None*

shows ¬ *r-distinguishable-k M q1 q2 k*
using *assms* **proof** (*induction k*)
case *0*
show *?case* **proof** (*rule ccontr*)
assume ¬ ¬ *r-distinguishable-k M q1 q2 0*
then have (∃ *x*∈*set* (*sort* (*inputs-as-list M*)).
¬ (∃ *t1*∈(*transitions M*).
∃ *t2*∈(*transitions M*).
t-source t1 = *q1* ∧
t-source t2 = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*))
unfolding *r-distinguishable-k.simps*
using *inputs-as-list-set* **by** *auto*
then obtain *x* **where** *find* (λ *x* . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) (*sort* (*inputs-as-list M*)) = *Some x*
unfolding *r-distinguishable-k.simps* **using** *find-None-iff*[*of* λ *x* . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) *sort* (*inputs-as-list M*)] **by** *blast*
then have *r-distinguishable-k-least M q1 q2 0* = *Some* (*0*,*x*)
unfolding *r-distinguishable-k-least.simps* **by** *auto*
then show *False* **using** *0* **by** *simp*
qed
next
case (*Suc k*)

have *r-distinguishable-k-least M q1 q2 k* = *None*
using *Suc.prems* **unfolding** *r-distinguishable-k-least.simps*
using *option.disc-eq-case*(*2*) **by** *force*
then have ∗: ¬ *r-distinguishable-k M q1 q2 k*
using *Suc.IH* **by** *auto*

have *find*
(λ*x*. ∀ *t1*∈(*transitions M*).
∀ *t2*∈(*transitions M*).
t-source t1 = *q1* ∧
t-source t2 = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2* ⟶
r-distinguishable-k M (*t-target t1*) (*t-target t2*) *k*)
(*sort* (*inputs-as-list M*)) = *None*
using *Suc.prems* ‹*r-distinguishable-k-least M q1 q2 k* = *None*› **unfolding** *r-distinguishable-k-least.simps*
using *option.disc-eq-case*(*2*) **by** *force*

then have ∗∗: ¬(∃ *x* ∈ *set* (*sort* (*inputs-as-list M*)) . (∀ *t1*∈(*transitions M*).
∀ *t2*∈(*transitions M*).
t-source t1 = *q1* ∧
t-source t2 = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output*

t1 = *t-output t2* ⟶
r-distinguishable-k M (*t-target t1*) (*t-target t2*) *k*))
using *find-None-iff*[*of* (λ*x*. ∀ *t1*∈(*transitions M*).
∀ *t2*∈(*transitions M*).
t-source t1 = *q1* ∧
t-source t2 = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2* ⟶
r-distinguishable-k M (*t-target t1*) (*t-target t2*) *k*) (*sort* (*inputs-as-list M*))] **by** *auto*

show *?case* **using** ∗ ∗∗ **unfolding** *r-distinguishable-k.simps*
using *inputs-as-list-set* **by** *fastforce*
qed

lemma *r-distinguishable-k-least-0-correctness* :
assumes *r-distinguishable-k-least M q1 q2 n* = *Some* (*0*,*x*)
shows *r-distinguishable-k M q1 q2 0* ∧ *0* =
(*LEAST k* . *r-distinguishable-k M q1 q2 k*)
∧ (*x* ∈ (*inputs M*) ∧ ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*))
∧ (∀ *x′* ∈ (*inputs M*) . *x′* < *x* ⟶ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x′* ∧ *t-input t2* = *x′* ∧ *t-output t1* = *t-output t2*))
using *assms* **proof** (*induction n*)
case *0*
then obtain *x′* **where** *x′-def* : *find* (λ *x* . ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*)) (*sort* (*inputs-as-list M*)) = *Some x′*
unfolding *r-distinguishable-k-least.simps* **by** *fastforce*
then have *x* = *x′* **using** *0* **unfolding** *r-distinguishable-k-least.simps* **by** *fastforce*
then have *x* ∈ *set* (*sort* (*inputs-as-list M*)) ∧ ¬ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) **using** *0* **unfolding** *r-distinguishable-k-least.simps r-distinguishable-k.simps*
using *find-condition*[*OF x′-def*] *find-set*[*OF x′-def*] **by** *blast*
moreover have *r-distinguishable-k M q1 q2 0*
using *calculation List.linorder-class.set-sort* **unfolding** *r-distinguishable-k.simps*
using *inputs-as-list-set* **by** *auto*
moreover have *0* = (*LEAST k* . *r-distinguishable-k M q1 q2 k*)
using *calculation*(*2*) **by** *auto*
moreover have (∀ *x′* ∈ (*inputs M*) . *x′* < *x* ⟶ (∃ *t1* ∈ *transitions M* . ∃ *t2* ∈ *transitions M* . *t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x′* ∧ *t-input t2* = *x′* ∧ *t-output t1* = *t-output t2*))
using *find-sort-least*(*1*)[*OF x′-def*] ‹*x* = *x′*› *inputs-as-list-set*
using *leD* **by** *blast*
ultimately show *?case* **unfolding** *inputs-as-list-set set-sort* **by** *force*

next
  case (*Suc n*)
  then show *?case* **proof** (*cases r-distinguishable-k-least M q1 q2 n*)
    case *None*
    have *r-distinguishable-k-least M q1 q2* (*Suc n*) $\neq$ *Some* (*0*, *x*)
      using *Suc.prems* **unfolding** *r-distinguishable-k-least.simps None*
    by (*metis* (*no-types*, *lifting*) *Zero-not-Suc fst-conv option.case-eq-if option.distinct*(*1*) *option.sel*)
    then show *?thesis* **using** *Suc.prems* **by** *auto*
  next
    case (*Some a*)
    then have *r-distinguishable-k-least M q1 q2 n* = *Some* (*0*, *x*)
      using *Suc.prems* **by** *auto*
    then show *?thesis* **using** *Suc.IH* **by** *blast*
  qed
qed

lemma *r-distinguishable-k-least-Suc-correctness* :
  assumes *r-distinguishable-k-least M q1 q2 n* = *Some* (*Suc k*,*x*)
  shows *r-distinguishable-k M q1 q2* (*Suc k*) $\wedge$ (*Suc k*) =
        (*LEAST k* . *r-distinguishable-k M q1 q2 k*)
          $\wedge$ (*x* $\in$ (*inputs M*) $\wedge$ ($\forall$ *t1* $\in$ *transitions M* . $\forall$ *t2* $\in$ *transitions M* . (*t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*) $\longrightarrow$ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))
          $\wedge$ ($\forall$ *x'* $\in$ (*inputs M*) . *x'* < *x* $\longrightarrow$ $\neg$($\forall$ *t1* $\in$ *transitions M* . $\forall$ *t2* $\in$ *transitions M* . (*t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x'* $\wedge$ *t-input t2* = *x'* $\wedge$ *t-output t1* = *t-output t2*) $\longrightarrow$ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))
using *assms* **proof** (*induction n*)
  case *0*
  then show *?case* **by** (*cases find*
        (λ*x*. $\neg$ ($\exists$ *t1*$\in$(*transitions M*).
                $\exists$ *t2*$\in$(*transitions M*).
                  t-source t1 = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*))
        (*sort* (*inputs-as-list M*)); *auto*)
next
  case (*Suc n*)
  then show *?case* **proof** (*cases r-distinguishable-k-least M q1 q2 n*)
    case *None*
    then have $*$: (*case find* (λ *x* . $\forall$ *t1* $\in$ *transitions M* . $\forall$ *t2* $\in$ *transitions M* . (*t-source t1* = *q1* $\wedge$ *t-source t2* = *q2* $\wedge$ *t-input t1* = *x* $\wedge$ *t-input t2* = *x* $\wedge$ *t-output t1* = *t-output t2*) $\longrightarrow$ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *n*) (*sort* (*inputs-as-list M*)) *of*
      Some x $\Rightarrow$ *Some* (*Suc n*,*x*) |
      None $\Rightarrow$ *None*) = *Some* (*Suc k*,*x*)
      using *Suc.prems* **unfolding** *r-distinguishable-k-least.simps* **by** *auto*
    then obtain *x'* **where** *x'-def* : *find* (λ *x* . $\forall$ *t1* $\in$ *transitions M* . $\forall$ *t2* $\in$

transitions M . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *n*) (*sort* (*inputs-as-list M*)) = *Some x′*
by *fastforce*
then have *x* = *x′* **using** ∗ **by** *fastforce*
then have *p3*: *x* ∈ (*inputs M*) ∧ (∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x* ∧ *t-input t2* = *x* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *n*)
using *find-condition*[*OF x′-def*] *find-set*[*OF x′-def*] *set-sort inputs-as-list-set* **by** *metis*
then have *p1*: *r-distinguishable-k M q1 q2* (*Suc n*)
unfolding *r-distinguishable-k.simps* **by** *blast*
moreover have ¬ *r-distinguishable-k M q1 q2 n*
using *r-distinguishable-k-least-ex*[*OF None*] **by** *assumption*
ultimately have *p2*: (*Suc n*) = (*LEAST k* . *r-distinguishable-k M q1 q2 k*)
by (*metis LeastI Least-le le-SucE r-distinguishable-k-by-larger*)

from ∗ **have** *k* = *n* **using** *x′-def* **by** *auto*
then have (∀ *x′* ∈ (*inputs M*) . *x′* < *x* ⟶ ¬(∀ *t1* ∈ *transitions M* . ∀ *t2* ∈ *transitions M* . (*t-source t1* = *q1* ∧ *t-source t2* = *q2* ∧ *t-input t1* = *x′* ∧ *t-input t2* = *x′* ∧ *t-output t1* = *t-output t2*) ⟶ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*))
using *find-sort-least*(*1*)[*OF x′-def*] ‹*x* = *x′*› *inputs-as-list-set*
using *leD* **by** *blast*
then show *?thesis* **using** *p1 p2 p3* ‹*k* = *n*› **by** *blast*
next
case (*Some a*)
then have *r-distinguishable-k-least M q1 q2 n* = *Some* (*Suc k*, *x*)
using *Suc.prems* **by** *auto*
then show *?thesis* **using** *Suc.IH*
by (*meson r-distinguishable-k.simps*(*2*))
qed
qed

lemma *r-distinguishable-k-least-is-least* :
assumes *r-distinguishable-k-least M q1 q2 n* = *Some* (*k*,*x*)
shows (∃ *k* . *r-distinguishable-k M q1 q2 k*) ∧ (*k* = (*LEAST k* . *r-distinguishable-k M q1 q2 k*))
proof (*cases k*)
case *0*
then show *?thesis* **using** *assms r-distinguishable-k-least-0-correctness* **by** *metis*
next
case (*Suc n*)
then show *?thesis* **using** *assms r-distinguishable-k-least-Suc-correctness* **by** *metis*
qed

lemma *r-distinguishable-k-from-r-distinguishable-k-least* :

assumes *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows (∃ *k* . *r-distinguishable-k M q1 q2 k*) = (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) ≠ *None*)
(**is** *?P1* = *?P2*)
proof
show *?P1* ⟹ *?P2*
using *r-distinguishable-k-least-ex r-distinguishable-k-least-bound*[*OF - assms*] *r-distinguishable-k-by-larger*
by (*metis LeastI*)
show *?P2* ⟹ *?P1*
proof −
assume *?P2*
then obtain *a* **where** (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some a*)
by *blast*
then obtain *x k* **where** *kx-def* : (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some* (*k*,*x*))
using *prod.collapse* **by** *metis*
then show *?P1*
proof (*cases k*)
case *0*
then have (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some* (*0*,*x*))
using *kx-def* **by** *presburger*
show *?thesis* **using** *r-distinguishable-k-least-0-correctness*[*OF* ‹(*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some* (*0*,*x*))›] **by** *blast*
next
case (*Suc n*)
then have (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some* ((*Suc n*),*x*))
using *kx-def* **by** *presburger*
show *?thesis* **using** *r-distinguishable-k-least-Suc-correctness*[*OF* ‹(*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) = *Some* ((*Suc n*),*x*))›] **by** *blast*
qed
qed
qed

definition *is-r-distinguishable* :: (*'a*, *'b*, *'c*) *fsm* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
is-r-distinguishable M q1 q2 = (∃ *k* . *r-distinguishable-k M q1 q2 k*)

lemma *is-r-distinguishable-contained-code*[*code*] :
is-r-distinguishable M q1 q2 = (*if* (*q1* ∈ *states M* ∧ *q2* ∈ *states M*) *then* (*r-distinguishable-k-least M q1 q2* (*size* (*product* (*from-FSM M q1*) (*from-FSM M q2*))) ≠ *None*)
else ¬(*inputs M* = {}))

proof (*cases q1* ∈ *states M* ∧ *q2* ∈ *states M*)
case *True*
then show *?thesis*
unfolding *is-r-distinguishable-def* **using** *r-distinguishable-k-from-r-distinguishable-k-least* **by** *metis*
next
case *False*
then have ∗: (¬ (∃ *t* ∈ *transitions M* . *t-source t* = *q1*)) ∨ (¬ (∃ *t* ∈ *transitions M* . *t-source t* = *q2*))
using *fsm-transition-source* **by** *auto*
show *?thesis* **proof** (*cases inputs M* = {})
case *True*
moreover have ⋀ *k* . *r-distinguishable-k M q1 q2 k* ⟹ *inputs M* ≠ {}
proof −
fix *k* **assume** *r-distinguishable-k M q1 q2 k*
then show *inputs M* ≠ {} **by** (*induction k*; *auto*)
qed
ultimately have *is-r-distinguishable M q1 q2* = *False*
by (*meson is-r-distinguishable-def*)
then show *?thesis* **using** *False True* **by** *auto*
next
case *False*
then show *?thesis*
by (*meson* ∗ *equals0I fst-conv is-r-distinguishable-def r-distinguishable-k-0-alt-def r-distinguishable-k-from-r-distinguishable-k-least*)
qed
qed

39.4 State Separators and R-Distinguishability

lemma *state-separator-r-distinguishes-k* :
assumes *is-state-separator-from-canonical-separator* (*canonical-separator M q1 q2*) *q1 q2 S*
and *q1* ∈ *states M* **and** *q2* ∈ *states M*
shows ∃ *k* . *r-distinguishable-k M q1 q2 k*
proof −
let *?P* = (*product* (*from-FSM M q1*) (*from-FSM M q2*))
let *?C* = (*canonical-separator M q1 q2*)

have *is-submachine S ?C*
and *single-input S*
and *acyclic S*
and *deadlock-state S* (*Inr q1*)
and *deadlock-state S* (*Inr q2*)
and *Inr q1* ∈ *reachable-states S*
and *Inr q2* ∈ *reachable-states S*
and (∀ *q*∈*reachable-states S*. *q* ≠ *Inr q1* ∧ *q* ≠ *Inr q2* ⟶ *isl q* ∧ ¬ *deadlock-state S q*)
and *tc*: (∀ *q*∈*reachable-states S*.

$\forall x \in (inputs\ ?C).$
$(\exists t \in transitions\ S.\ t\text{-}source\ t = q \wedge t\text{-}input\ t = x) \longrightarrow$
$(\forall t' \in transitions\ ?C.\ t\text{-}source\ t' = q \wedge t\text{-}input\ t' = x \longrightarrow t' \in transitions\ S))$
using *assms*(*1*) **unfolding** *is-state-separator-from-canonical-separator-def* **by** *linarith+*

let *?Prop* = (λ *q* . *case q of*
(*Inl* (*q1'*,*q2'*)) $\Rightarrow$ ($\exists$ *k* . *r-distinguishable-k M q1' q2' k*) |
(*Inr qr*) $\Rightarrow$ *True*)
have *rprop*: $\forall$ *q* $\in$ *reachable-states S* . *?Prop q*
using ‹*acyclic S*› **proof** (*induction rule*: *acyclic-induction*)
case (*reachable-state q*)
then show *?case* **proof** (*cases* $\neg$ *isl q*)
case *True*
then have *q* = *Inr q1* $\vee$ *q* = *Inr q2*
using ‹($\forall$ *q*$\in$*reachable-states S*. *q* $\neq$ *Inr q1* $\wedge$ *q* $\neq$ *Inr q2* $\longrightarrow$ *isl q* $\wedge$ $\neg$ *deadlock-state S q*)› *reachable-state*(*1*) **by** *blast*
then show *?thesis* **by** *auto*
next
case *False*
then obtain *q1' q2'* **where** *q* = *Inl* (*q1'*,*q2'*)
using *isl-def prod.collapse* **by** *metis*
then have $\neg$ *deadlock-state S q*
using ‹($\forall$ *q*$\in$*reachable-states S*. *q* $\neq$ *Inr q1* $\wedge$ *q* $\neq$ *Inr q2* $\longrightarrow$ *isl q* $\wedge$ $\neg$ *deadlock-state S q*)› *reachable-state*(*1*) **by** *blast*

then obtain *t* **where** *t* $\in$ *transitions S* **and** *t-source t* = *q*
unfolding *deadlock-state.simps* **by** *blast*
then have ($\forall$ *t'*$\in$ *transitions ?C*. *t-source t'* = *q* $\wedge$ *t-input t'* = *t-input t* $\longrightarrow$ *t'* $\in$ *transitions S*)
using *reachable-state*(*1*) *tc*
using *fsm-transition-input* **by** *fastforce*

have *Inl* (*q1'*,*q2'*) $\in$ *reachable-states ?C*
using *reachable-state*(*1*) **unfolding** ‹*q* = *Inl* (*q1'*,*q2'*)› *reachable-states-def*
using *submachine-path-initial*[*OF* ‹*is-submachine S* (*canonical-separator M q1 q2*)›]
unfolding *canonical-separator-simps*[*OF assms*(*2*,*3*)] *is-state-separator-from-canonical-separator-initial*[*O*
assms(*1*−*3*)] **by** *fast*
then obtain *p* **where** *path ?C* (*initial ?C*) *p*
and *target* (*initial ?C*) *p* = *Inl* (*q1'*,*q2'*)
unfolding *reachable-states-def* **by** *auto*
then have *isl* (*target* (*initial ?C*) *p*) **by** *auto*
then obtain *p'* **where** *path ?P* (*initial ?P*) *p'*
and *p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p'*
using *canonical-separator-path-from-shift*[*OF* ‹*path ?C* (*initial ?C*) *p*›]

using *assms*(*2*) *assms*(*3*) **by** *blast*

have (*q1'*,*q2'*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))
using *reachable-state-is-state*[*OF* ‹*Inl* (*q1'*,*q2'*) ∈ *reachable-states ?C*›] **unfolding** *canonical-separator-simps*[*OF assms*(*2*,*3*)]
by *auto*

have *path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) (*left-path p'*)
and *path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) (*right-path p'*)
using *product-path*[*of from-FSM M q1 from-FSM M q2 q1 q2 p'*] ‹*path ?P* (*initial ?P*) *p'*›
by (*simp add*: *paths-from-product-path*)+
moreover have *target* (*initial* (*from-FSM M q1*)) (*left-path p'*) = *q1'*
using ‹*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p'*› ‹*target* (*initial ?C*) *p* = *Inl* (*q1'*,*q2'*)› *canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)] *assms*(*2*)
by (*cases p' rule*: *rev-cases*; *auto*)
moreover have *target* (*initial* (*from-FSM M q2*)) (*right-path p'*) = *q2'*
using ‹*p* = *map* (λ*t*. (*Inl* (*t-source t*), *t-input t*, *t-output t*, *Inl* (*t-target t*))) *p'*› ‹*target* (*initial ?C*) *p* = *Inl* (*q1'*,*q2'*)› *canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)] *assms*(*3*)
by (*cases p' rule*: *rev-cases*; *auto*)
moreover have *p-io* (*left-path p'*) = *p-io* (*right-path p'*) **by** *auto*
ultimately have *p12'* : ∃ *p1 p2*.
path (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) *p1* ∧
path (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) *p2* ∧
target (*initial* (*from-FSM M q1*)) *p1* = *q1'* ∧
target (*initial* (*from-FSM M q2*)) *p2* = *q2'* ∧ *p-io p1* = *p-io p2*
by *blast*

have *q1'* ∈ *states* (*from-FSM M q1*)
using *path-target-is-state*[*OF* ‹*path* (*from-FSM M q1*) (*initial* (*from-FSM M q1*)) (*left-path p'*)›] ‹*target* (*initial* (*from-FSM M q1*)) (*left-path p'*) = *q1'*› **by** *auto*
have *q2'* ∈ *states* (*from-FSM M q2*)
using *path-target-is-state*[*OF* ‹*path* (*from-FSM M q2*) (*initial* (*from-FSM M q2*)) (*right-path p'*)›] ‹*target* (*initial* (*from-FSM M q2*)) (*right-path p'*) = *q2'*› **by** *auto*

have *t-input t* ∈ (*inputs S*)
using ‹*t* ∈ *transitions S*› **by** *auto*
then have *t-input t* ∈ (*inputs ?C*)
using ‹*is-submachine S ?C*› **by** *auto*
then have *t-input t* ∈ (*inputs M*)
using *canonical-separator-simps*(*3*)[*OF assms*(*2*,*3*)] **by** *metis*

have ∗: ⋀ *t1 t2* . *t1* ∈ *transitions M* ⟹ *t2* ∈ *transitions M* ⟹ *t-source t1* = *q1'* ⟹ *t-source t2* = *q2'* ⟹ *t-input t1* = *t-input t* ⟹ *t-input t2* = *t-input t* ⟹

t-output t1 = *t-output t2* ⟹ (∃ *k* . *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*)

proof −

fix *t1 t2* **assume** *t1* ∈ *transitions M*
and *t2* ∈ *transitions M*
and *t-source t1* = *q1′*
and *t-source t2* = *q2′*
and *t-input t1* = *t-input t*
and *t-input t2* = *t-input t*
and *t-output t1* = *t-output t2*

then have *t-input t1* = *t-input t2* **by** *auto*

have *t1* ∈ *transitions* (*from-FSM M q1*)
using ‹*t-source t1* = *q1′*› ‹*q1′* ∈ *states* (*from-FSM M q1*)› ‹*t1* ∈ *transitions M*› **by** (*simp add*: *assms*(*2*))

have *t2* ∈ *transitions* (*from-FSM M q2*)
using ‹*t-source t2* = *q2′*› ‹*q2′* ∈ *states* (*from-FSM M q2*)› ‹*t2* ∈ *transitions M*› **by** (*simp add*: *assms*(*3*))

let *?t* = ((*t-source t1*, *t-source t2*), *t-input t1*, *t-output t1*, *t-target t1*, *t-target t2*)

have *?t* ∈ *transitions ?P*
using ‹*t1* ∈ *transitions* (*from-FSM M q1*)› ‹*t2* ∈ *transitions* (*from-FSM M q2*)› ‹*t-input t1* = *t-input t2*› ‹*t-output t1* = *t-output t2*›
unfolding *product-transitions-alt-def*
by *blast*

then have *shift-Inl ?t* ∈ *transitions ?C*
using ‹(*q1′*,*q2′*) ∈ *states* (*Product-FSM.product* (*FSM.from-FSM M q1*) (*FSM.from-FSM M q2*))›
unfolding ‹*t-source t1* = *q1′*› ‹*t-source t2* = *q2′*› *canonical-separator-transitions-def*[*OF assms*(*2*,*3*)] **by** *fastforce*

moreover have *t-source* (*shift-Inl ?t*) = *q*
using ‹*t-source t1* = *q1′*› ‹*t-source t2* = *q2′*› ‹*q* = *Inl* (*q1′*,*q2′*)› **by** *auto*

ultimately have *shift-Inl ?t* ∈ *transitions S*
using ‹(∀ *t′*∈ *transitions ?C*. *t-source t′* = *q* ∧ *t-input t′* = *t-input t* ⟶ *t′* ∈ *transitions S*)› ‹*t-input t1* = *t-input t*› **by** *auto*

have *case t-target* (*shift-Inl ?t*) *of Inl* (*q1′*, *q2′*) ⇒ ∃ *k*. *r-distinguishable-k M q1′ q2′ k* | *Inr qr* ⇒ *True*
using *reachable-state.IH*(*2*)[*OF* ‹*shift-Inl ?t* ∈ *transitions S*› ‹*t-source* (*shift-Inl ?t*) = *q*›] **by** (*cases q*; *auto*)

moreover have *t-target* (*shift-Inl ?t*) = *Inl* (*t-target t1*, *t-target t2*)
by *auto*

ultimately show ∃ *k*. *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *k*
by *auto*

qed

let *?hs* = {(*t1*,*t2*) | *t1 t2* . *t1* ∈ *transitions M* ∧ *t2* ∈ *transitions M* ∧ *t-source t1* = *q1′* ∧ *t-source t2* = *q2′* ∧ *t-input t1* = *t-input t* ∧ *t-input t2* = *t-input t* ∧ *t-output t1* = *t-output t2*}
have *finite ?hs*
proof −
have *?hs* ⊆ (*transitions M* × *transitions M*) **by** *blast*
moreover have *finite* (*transitions M* × *transitions M*) **using** *fsm-transitions-finite* **by** *blast*
ultimately show *?thesis*
by (*simp add*: *finite-subset*)
qed
obtain *fk* **where** *fk-def* : ⋀ *tt* . *tt* ∈ *?hs* ⟹ *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) (*fk tt*)
proof
let *?fk* = λ *tt* . *SOME k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*
show ⋀ *tt* . *tt* ∈ *?hs* ⟹ *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) (*?fk tt*)
proof −
fix *tt* **assume** *tt* ∈ *?hs*
then have (*fst tt*) ∈ *transitions M* ∧ (*snd tt*) ∈ *transitions M* ∧ *t-source* (*fst tt*) = *q1′* ∧ *t-source* (*snd tt*) = *q2′* ∧ *t-input* (*fst tt*) = *t-input t* ∧ *t-input* (*snd tt*) = *t-input t* ∧ *t-output* (*fst tt*) = *t-output* (*snd tt*)
by *force*
then have ∃ *k* . *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) *k*
using ∗ **by** *blast*
then show *r-distinguishable-k M* (*t-target* (*fst tt*)) (*t-target* (*snd tt*)) (*?fk tt*)
by (*simp add*: *someI-ex*)
qed
qed

let *?k* = *Max* (*image fk ?hs*)
have ⋀ *t1 t2* . *t1* ∈ *transitions M* ⟹ *t2* ∈ *transitions M* ⟹ *t-source t1* = *q1′* ⟹ *t-source t2* = *q2′* ⟹ *t-input t1* = *t-input t* ⟹ *t-input t2* = *t-input t* ⟹ *t-output t1* = *t-output t2* ⟹ *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *?k*
proof −
fix *t1 t2* **assume** *t1* ∈ *transitions M*
and *t2* ∈ *transitions M*
and *t-source t1* = *q1′*
and *t-source t2* = *q2′*
and *t-input t1* = *t-input t*
and *t-input t2* = *t-input t*
and *t-output t1* = *t-output t2*
then have (*t1*,*t2*) ∈ *?hs*
by *force*

then have *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) (*fk* (*t1*,*t2*))
using *fk-def* **by** *force*
have *fk* (*t1*,*t2*) ≤ *?k*
using ‹(*t1*,*t2*) ∈ *?hs*› ‹*finite ?hs*› **by** *auto*
show *r-distinguishable-k M* (*t-target t1*) (*t-target t2*) *?k*
using *r-distinguishable-k-by-larger*[*OF* ‹*r-distinguishable-k M* (*t-target t1*) (*t-target t2*) (*fk* (*t1*,*t2*))› ‹*fk* (*t1*,*t2*) ≤ *?k*›] **by** *assumption*
qed

then have *r-distinguishable-k M q1′ q2′* (*Suc ?k*)
unfolding *r-distinguishable-k.simps*
using ‹*t-input t* ∈ (*inputs M*)› **by** *blast*
then show *?Prop q*
using ‹*q* = *Inl* (*q1′*,*q2′*)›
by (*metis* (*no-types*, *lifting*) *case-prodI old.sum.simps*(*5*))
qed
qed

moreover have *Inl* (*q1*,*q2*) ∈ *states S*
using ‹*is-submachine S ?C*› *canonical-separator-simps*(*1*)[*OF assms*(*2*,*3*)] *fsm-initial*[*of S*] **by** *auto*
ultimately show ∃ *k*. *r-distinguishable-k M q1 q2 k*
using *reachable-states-initial*[*of S*] **using** *is-state-separator-from-canonical-separator-initial*[*OF assms*(*1*−*3*)] **by** *auto*
qed

end

40 Traversal Set

This theory defines the calculation of m-traversal paths. These are paths extended from some state until they visit pairwise r-distinguishable states a number of times dependent on m.

theory *Traversal-Set*
imports *Helper-Algorithms*
begin

definition *m-traversal-paths-with-witness-up-to-length* ::
(*′a*,*′b*,*′c*) *fsm* ⇒ *′a* ⇒ (*′a set* × *′a set*) *list* ⇒ *nat* ⇒ *nat* ⇒ ((*′a*×*′b*×*′c*×*′a*) *list* × (*′a set* × *′a set*)) *set*
where
m-traversal-paths-with-witness-up-to-length M q D m k
= *paths-up-to-length-or-condition-with-witness M* (λ *p* . *find* (λ *d* . *length* (*filter* (λ*t* . *t-target t* ∈ *fst d*) *p*) ≥ *Suc* (*m* − (*card* (*snd d*)))) *D*) *k q*

definition *m-traversal-paths-with-witness* ::

$('a,'b,'c)$ *fsm* $\Rightarrow$ $'a$ $\Rightarrow$ $('a$ *set* $\times$ $'a$ *set*) *list* $\Rightarrow$ *nat* $\Rightarrow$ $(('a\times'b\times'c\times'a)$ *list* $\times$ $('a$ *set* $\times$ $'a$ *set*)) *set*
where
m-traversal-paths-with-witness M q D m = *m-traversal-paths-with-witness-up-to-length M q D m* (*Suc* (*size M* $*$ *m*))

lemma *m-traversal-paths-with-witness-finite* : *finite* (*m-traversal-paths-with-witness M q D m*)
unfolding *m-traversal-paths-with-witness-def m-traversal-paths-with-witness-up-to-length-def*
by (*simp add*: *paths-up-to-length-or-condition-with-witness-finite*)

lemma *m-traversal-paths-with-witness-up-to-length-max-length* :
assumes $\bigwedge$ *q* . *q* $\in$ *states M* $\Longrightarrow$ $\exists$ *d* $\in$ *set D* . *q* $\in$ *fst d*
and $\bigwedge$ *d* . *d* $\in$ *set D* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
and *q* $\in$ *states M*
and (*p*,*d*) $\in$ (*m-traversal-paths-with-witness-up-to-length M q D m k*)
shows *length p* $\leq$ *Suc* ((*size M*) $*$ *m*)
proof (*rule ccontr*)
assume $\neg$ *length p* $\leq$ *Suc* (*FSM.size M* $*$ *m*)

let *?f* = (λ *p* . *find* (λ *d* . *length* (*filter* (λ*t* . *t-target t* $\in$ *fst d*) *p*) $\geq$ *Suc* (*m* $-$ (*card* (*snd d*)))) *D*)

have *path M q* [] **using** *assms*(*3*) **by** *auto*

have *path M q p*
and *length p* $\leq$ *k*
and *?f p* = *Some d*
and $\forall$ *p′ p″*. *p* = *p′* @ *p″* $\wedge$ *p″* $\neq$ [] $\longrightarrow$ *?f p′* = *None*
using *assms*(*4*)
unfolding *m-traversal-paths-with-witness-up-to-length-def paths-up-to-length-or-condition-with-witness-def*

by *auto*

let *?p* = *take* (*Suc* (*m* $*$ *size M*)) *p*
let *?p′* = *drop* (*Suc* (*m* $*$ *size M*)) *p*
have *path M q ?p*
using ‹*path M q p*› **using** *path-prefix*[*of M q ?p drop* (*Suc* (*m* $*$ *size M*)) *p*]
by *simp*
have *?p′* $\neq$ []
using ‹$\neg$ *length p* $\leq$ *Suc* (*FSM.size M* $*$ *m*)›
by (*simp add*: *mult.commute*)

have $\exists$ *q* $\in$ *states M* . *length* (*filter* (λ*t* . *t-target t* = *q*) *?p*) $\geq$ *Suc m*
proof (*rule ccontr*)
assume $\neg$ ($\exists$ *q*$\in$*states M*. *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*))
then have $\forall$ *q* $\in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) $<$ *Suc m*

by *auto*
then have $\forall\ q \in states\ M\ .\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ ?p) \leq m$
by *auto*

have $(\sum q{\in}states\ M.\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ ?p)) \leq (\sum q{\in}states\ M\ .\ m)$
using ‹$\forall\ q \in states\ M\ .\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ ?p) \leq m$› **by** (*meson sum-mono*)
then have $length\ ?p \leq m * (size\ M)$
using *path-length-sum*[*OF* ‹*path M q ?p*›]
using *fsm-states-finite*[*of M*]
by (*simp add*: *mult.commute*)

then show *False*
using ‹$\neg\ length\ p \leq Suc\ (FSM.size\ M * m)$›
by (*simp add*: *mult.commute*)
qed

then obtain q **where** $q \in states\ M$
and $length\ (filter\ (\lambda\ t\ .\ t\text{-}target\ t = q)\ ?p) \geq Suc\ m$
by *blast*
then obtain d **where** $d \in set\ D$
and $q \in fst\ d$
using *assms*(*1*) **by** *blast*
then have $\bigwedge\ t\ .\ t\text{-}target\ t = q \Longrightarrow t\text{-}target\ t \in fst\ d$ **by** *auto*
then have $length\ (filter\ (\lambda\ t\ .\ t\text{-}target\ t = q)\ ?p) \leq length\ (filter\ (\lambda\ t\ .\ t\text{-}target\ t \in fst\ d)\ ?p)$
using *filter-length-weakening*[*of* $\lambda\ t\ .\ t\text{-}target\ t = q\ \lambda\ t\ .\ t\text{-}target\ t \in fst\ d$] **by** *auto*
then have $Suc\ m \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ ?p)$
using ‹$length\ (filter\ (\lambda\ t\ .\ t\text{-}target\ t = q)\ ?p) \geq Suc\ m$› **by** *auto*
then have $?f\ ?p \neq None$
using *assms*(*2*)
proof −
have $\forall p.\ find\ p\ D \neq None \vee \neg\ p\ d$
by (*metis* ‹$d \in set\ D$› *find-from*)
then show *?thesis*
using ‹$Suc\ m \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ (Suc\ (m * FSM.size\ M))\ p))$›
diff-le-self le-trans not-less-eq-eq
by *blast*
qed
then obtain d' **where** $?f\ ?p = Some\ d'$
by *blast*

then have $p = ?p@?p' \wedge\ ?p' \neq []\ \wedge\ ?f\ ?p = Some\ d'$
using ‹$?p' \neq []$› **by** *auto*

then show *False*

using ‹$\forall p'\ p''.\ p = p' @ p'' \wedge p'' \neq [] \longrightarrow (?f\ p') = None$›
by (*metis* (*no-types*) *option.distinct*(*1*))
qed

lemma *m-traversal-paths-with-witness-set* :
assumes $\bigwedge q\ .\ q \in states\ M \Longrightarrow \exists\ d \in set\ D\ .\ q \in fst\ d$
and $\bigwedge d\ .\ d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
and $q \in states\ M$
shows (*m-traversal-paths-with-witness M q D m*)
$= \{(p,d) \mid p\ d\ .\ path\ M\ q\ p$
$\wedge\ find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.$
t-target $t \in fst\ d)\ p))\ D = Some\ d$
$\wedge\ (\forall p'\ p''.\ p = p' @ p'' \wedge p'' \neq [] \longrightarrow find\ (\lambda d.\ Suc\ (m -$
$card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D = None)\}$
(**is** $?MTP = ?P$)
proof −
let $?f = (\lambda\ p\ .\ find\ (\lambda\ d\ .\ length\ (filter\ (\lambda t\ .\ t\text{-}target\ t \in fst\ d)\ p) \geq Suc\ (m -$
$(card\ (snd\ d))))\ D)$

have *path M q* []
using *assms*(*3*) **by** *auto*

have $\bigwedge p\ .\ p \in ?MTP \Longrightarrow p \in ?P$
unfolding *m-traversal-paths-with-witness-def m-traversal-paths-with-witness-up-to-length-def*

paths-up-to-length-or-condition-with-witness-def
by *force*
moreover have $\bigwedge p\ .\ p \in ?P \Longrightarrow p \in ?MTP$
proof −
fix *px* **assume** $px \in ?P$
then obtain *p x* **where** $px = (p,x)$
and *p1*: *path M q p*
and $**$: $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t$
$\in fst\ d)\ p))\ D = Some\ x$
and $***$:$(\forall p'\ p''.$
$p = p' @ p'' \wedge p'' \neq [] \longrightarrow$
$find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t$
$\in fst\ d)\ p'))\ D = None)$
using *prod.collapse* **by** *force*

then have $(p,x) \in$ (*m-traversal-paths-with-witness-up-to-length M q D m* (*length*
p))
unfolding *m-traversal-paths-with-witness-up-to-length-def paths-up-to-length-or-condition-with-witness-def*
by *force*
then have $length\ p \leq Suc\ (size\ M * m)$
using *m-traversal-paths-with-witness-up-to-length-max-length*[*OF assms*] **by**
blast

have $(p,x) \in$ *?MTP*
using ‹*path M q p*› ‹*length p* $\leq$ *Suc* (*size M* $*$ *m*)› ‹*?f p* = *Some x*›
‹$\forall$ *p′ p″*. *p* = *p′* @ *p″* $\wedge$ *p″* $\neq$ [] $\longrightarrow$ (*?f p′*) = *None*›
unfolding *m-traversal-paths-with-witness-def m-traversal-paths-with-witness-up-to-length-def*

paths-up-to-length-or-condition-with-witness-def
by *force*
then show *px* $\in$ *?MTP*
using ‹*px* = (*p*,*x*)› **by** *simp*
qed
ultimately show *?thesis*
by (*meson subsetI subset-antisym*)
qed

lemma *maximal-repetition-sets-from-separators-cover* :
assumes *q* $\in$ *states M*
shows $\exists$ *d* $\in$ (*maximal-repetition-sets-from-separators M*) . *q* $\in$ *fst d*
unfolding *maximal-repetition-sets-from-separators-def*
using *maximal-pairwise-r-distinguishable-state-sets-from-separators-cover*[*OF assms*]
by *auto*

lemma *maximal-repetition-sets-from-separators-d-reachable-subset* :
shows $\bigwedge$ *d* . *d* $\in$ (*maximal-repetition-sets-from-separators M*) $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
unfolding *maximal-repetition-sets-from-separators-def*
by *auto*

lemma *m-traversal-paths-with-witness-set-containment* :
assumes *q* $\in$ *states M*
and *path M q p*
and *d* $\in$ *set repSets*
and *Suc* (*m* $-$ *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p*)
and $\bigwedge$ *p′ p″*.
p = *p′* @ *p″* $\Longrightarrow$ *p″* $\neq$ [] $\Longrightarrow$
$\neg$ ($\exists$ *d*$\in$*set repSets*.
Suc (*m* $-$ *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*)
p′))
and $\bigwedge$ *q* . *q*$\in$*states M* $\Longrightarrow$ $\exists$ *d*$\in$*set repSets*. *q* $\in$ *fst d*
and $\bigwedge$ *d* . *d*$\in$*set repSets* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
shows $\exists$ *d′* . (*p*,*d′*) $\in$ (*m-traversal-paths-with-witness M q repSets m*)
proof $-$
obtain *d′* **where** *find* (λ*d*. *Suc* (*m* $-$ *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target*
t $\in$ *fst d*) *p*)) *repSets* = *Some d′*
using *assms*(*3*,*4*) *find-None-iff*[*of* (λ*d*. *Suc* (*m* $-$ *card* (*snd d*)) $\leq$ *length* (*filter*

$(\lambda t.\ t\text{-}target\ t \in fst\ d)\ p))\ repSets]$
by *auto*
moreover have $(\bigwedge p'\ p''.\ p = p'\ @\ p'' \Longrightarrow p'' \neq []$
$\Longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ repSets = None)$
using *assms*(*5*) *find-None-iff*[*of - repSets*] **by** *force*
ultimately show *?thesis*
using *m-traversal-paths-with-witness-set*[*of M repSets q m, OF assms*(*6*,*7*,*1*)]
using *assms*(*2*) **by** *blast*
qed

lemma *m-traversal-path-exist* :
assumes *completely-specified M*
and $q \in states\ M$
and $inputs\ M \neq \{\}$
and $\bigwedge q\ .\ q \in states\ M \Longrightarrow \exists\, d \in set\ D.\ q \in fst\ d$
and $\bigwedge d\ .\ d \in set\ D \Longrightarrow snd\ d \subseteq fst\ d$
shows $\exists\ p'\ d'\ .\ (p',d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
proof −

obtain p **where** $path\ M\ q\ p$ **and** $length\ p = Suc\ ((size\ M) * m)$
using *path-of-length-ex*[*OF assms*(*1*−*3*)] **by** *blast*

let $?f = (\lambda\ p\ .\ find\ (\lambda\ d\ .\ length\ (filter\ (\lambda t\ .\ t\text{-}target\ t \in fst\ d)\ p) \geq Suc\ (m - (card\ (snd\ d))))\ D)$

have $\exists\ q \in states\ M\ .\ length\ (filter\ (\lambda t\ .\ t\text{-}target\ t = q)\ p) \geq Suc\ m$
proof (*rule ccontr*)
assume $\neg\ (\exists\, q \in states\ M.\ Suc\ m \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ p))$
then have $\forall\ q \in states\ M\ .\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ p) < Suc\ m$
by *auto*
then have $\forall\ q \in states\ M\ .\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ p) \leq m$
by *auto*

have $(\sum q \in states\ M.\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ p)) \leq (\sum q \in states\ M\ .\ m)$
using ‹$\forall\ q \in states\ M\ .\ length\ (filter\ (\lambda t.\ t\text{-}target\ t = q)\ p) \leq m$› **by** (*meson sum-mono*)
then have $length\ p \leq m * (size\ M)$
using *path-length-sum*[*OF* ‹*path M q p*›]
using *fsm-states-finite*[*of M*]
by (*simp add*: *mult.commute*)

then show *False*
using ‹$length\ p = Suc\ ((size\ M) * m)$›
by (*simp add*: *mult.commute*)
qed

then obtain q' **where** $q' \in$ *states M*
and *length* (*filter* (λ *t* . *t-target t* = q') *p*) $\geq$ *Suc m*
by *blast*
then obtain *d* **where** $d \in$ *set D*
and $q' \in$ *fst d*
using *assms*(*4*) **by** *blast*
then have $\bigwedge$ *t* . *t-target t* = $q' \Longrightarrow$ *t-target t* $\in$ *fst d* **by** *auto*
then have *length* (*filter* (λ *t* . *t-target t* = q') *p*) $\leq$ *length* (*filter* (λ *t* . *t-target t* $\in$ *fst d*) *p*)
using *filter-length-weakening*[*of* λ *t* . *t-target t* = q' λ *t* . *t-target t* $\in$ *fst d*] **by** *auto*
then have *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p*)
using ‹*length* (*filter* (λ *t* . *t-target t* = q') *p*) $\geq$ *Suc m*› **by** *auto*
then have *?f p* $\neq$ *None*
using *assms*(*2*)
proof −
have $\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*
by (*metis* ‹*d* $\in$ *set D*› *find-from*)
have *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*p*. *t-target p* $\in$ *fst d*) *p*)
using ‹*Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p*)› **by** *linarith*
then show *?thesis*
using ‹$\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*› **by** *blast*
qed
then obtain d' **where** *?f p* = *Some* d'
by *blast*

show *?thesis* **proof** (*cases* ($\forall$ p' p''. *p* = p' @ p'' $\wedge$ p'' $\neq$ [] $\longrightarrow$ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) p')) *D* = *None*))
case *True*
then show *?thesis*
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] ‹*path M q p*› ‹*?f p* = *Some* d'›
by *blast*
next
case *False*

define *ps* **where** *ps-def*: *ps* = {p' . $\exists$ p''. *p* = p' @ p'' $\wedge$ p'' $\neq$ []
$\wedge$ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) p')) *D* $\neq$ *None*}
have *ps* $\neq$ {}
using *False ps-def*
by *blast*
moreover have *finite ps*
proof −
have *ps* $\subseteq$ *set* (*prefixes p*)
unfolding *prefixes-set ps-def*
by *blast*
then show *?thesis*

by (*meson List.finite-set rev-finite-subset*)
qed
ultimately obtain p' **where** $p' \in ps$ **and** $\bigwedge p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$
by (*meson leI min-length-elem*)
then have $\bigwedge p''\ p'''\ .\ p' = p'' @ p''' \Longrightarrow p''' \neq []$
$\Longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'')) \ D = None$
proof −
fix $p''\ p'''$ **assume** $p' = p'' @ p'''$ **and** $p''' \neq []$
show $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'')) \ D = None$
proof (*rule ccontr*)
assume $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'')) \ D \neq None$
moreover have $\exists p'''.\ p = p'' @ p''' \wedge p''' \neq []$
using ‹$p' \in ps$› ‹$p' = p'' @ p'''$› **unfolding** *ps-def* **by** *auto*
ultimately have $p'' \in ps$
unfolding *ps-def* **by** *auto*
moreover have $length\ p'' < length\ p'$
using ‹$p''' \neq []$› ‹$p' = p'' @ p'''$› **by** *auto*
ultimately show *False*
using ‹$\bigwedge p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$›
using *leD* **by** *auto*
qed
qed

have $path\ M\ q\ p'$ **and** $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p')) \ D \neq None$
using ‹$path\ M\ q\ p$› ‹$p' \in ps$› **unfolding** *ps-def* **by** *auto*
then obtain d' **where** $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p')) \ D = Some\ d'$
by *auto*

then have $path\ M\ q\ p' \wedge$
$find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p')) \ D = Some\ d' \wedge$
$(\forall p''\ p'''.\ p' = p'' @ p''' \wedge p''' \neq [] \longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'')) \ D = None)$
using ‹$\bigwedge p''\ p'''\ .\ p' = p'' @ p''' \Longrightarrow p''' \neq [] \Longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'')) \ D = None$›
‹$path\ M\ q\ p'$›
by *blast*
then have $(p',d') \in (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ D\ m)$
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] **by** *blast*
then show *?thesis* **by** *blast*
qed
qed

lemma *m-traversal-path-extension-exist* :
assumes *completely-specified M*
and $q \in$ *states M*
and *inputs M* $\neq \{\}$
and $\bigwedge q$. $q \in$*states M* $\Longrightarrow \exists\, d \in$*set D*. $q \in$ *fst d*
and $\bigwedge d$. $d \in$ *set D* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
and *path M q p1*
and *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p1*)) *D* = *None*
shows $\exists$ *p2 d′* . (*p1*@*p2*,*d′*) $\in$ (*m-traversal-paths-with-witness M q D m*)
proof −
obtain *p2* **where** *path M* (*target q p1*) *p2* **and** *length p2* = (*Suc* ((*size M*) ∗ *m*)) − *length p1*
using *path-of-length-ex*[*OF assms*(*1*) *path-target-is-state*[*OF assms*(*6*)] *assms*(*3*)]

by *blast*

have *path M q* (*p1*@*p2*)
using *assms*(*6*) ‹*path M* (*target q p1*) *p2*› **by** *auto*

let *?f* = (λ *p* . *find* (λ *d* . *length* (*filter* (λ*t* . *t-target t* $\in$ *fst d*) *p*) $\geq$ *Suc* (*m* − (*card* (*snd d*)))) *D*)

have *length p1* < *Suc* ((*size M*) ∗ *m*)
proof (*rule ccontr*)
assume ¬ *length p1* < *Suc* (*FSM.size M* ∗ *m*)
then have *length* (*take* (*Suc* (*FSM.size M* ∗ *m*)) *p1*) = *Suc* (*FSM.size M* ∗ *m*)
by *auto*
let *?p* = (*take* (*Suc* (*FSM.size M* ∗ *m*)) *p1*)

have *path M q ?p*
using ‹*path M q p1*›
by (*metis append-take-drop-id path-append-elim*)

have $\exists$ *q* $\in$ *states M* . *length* (*filter* (λ*t* . *t-target t* = *q*) *?p*) $\geq$ *Suc m*
proof (*rule ccontr*)
assume ¬ ($\exists$ *q*$\in$*states M*. *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*))
then have $\forall$ *q* $\in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) < *Suc m*
by *auto*
then have $\forall$ *q* $\in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) $\leq$ *m*
by *auto*

have ($\sum$ *q*$\in$*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*)) $\leq$ ($\sum$ *q*$\in$*states M* . *m*)
using ‹$\forall$ *q* $\in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) $\leq$ *m*› **by**

```
(meson sum-mono)
      then have length ?p ≤ m ∗ (size M)
        using path-length-sum[OF ‹path M q ?p›]
        using fsm-states-finite[of M]
        by (simp add: mult.commute)

      then show False
        using ‹length ?p = Suc ((size M) ∗ m)›
        by (simp add: mult.commute)
    qed
    then obtain q′ where q′ ∈ states M
                  and length (filter (λ t . t-target t = q′) ?p) ≥ Suc m
      by blast
    then obtain d where d ∈ set D
                   and q′ ∈ fst d
      using assms(4) by blast
    then have ⋀ t . t-target t = q′ ⟹ t-target t ∈ fst d by auto
    then have length (filter (λ t . t-target t = q′) ?p) ≤ length (filter (λ t . t-target
t ∈ fst d) ?p)
      using filter-length-weakening[of λ t . t-target t = q′ λ t . t-target t ∈ fst d]
by auto
    then have Suc m ≤ length (filter (λt. t-target t ∈ fst d) ?p)
      using ‹length (filter (λ t . t-target t = q′) ?p) ≥ Suc m› by auto
    moreover have length (filter (λt. t-target t ∈ fst d) ?p) ≤ length (filter (λt.
t-target t ∈ fst d) p1)
    proof −
      have ⋀ xs P n . length (filter P (take n xs)) ≤ length (filter P xs)
      by (metis append-take-drop-id filter-append le0 le-add-same-cancel1 length-append)
      then show ?thesis by auto
    qed
    ultimately have Suc m ≤ length (filter (λt. t-target t ∈ fst d) p1)
      by auto
    then have ?f p1 ≠ None
      using assms(2)
    proof −
      have ∀ p. find p D ≠ None ∨ ¬ p d
        by (metis ‹d ∈ set D› find-from)
      have Suc (m − card (snd d)) ≤ length (filter (λp. t-target p ∈ fst d) p1)
        using ‹Suc m ≤ length (filter (λt. t-target t ∈ fst d) p1)› by linarith
      then show ?thesis
        using ‹∀ p. find p D ≠ None ∨ ¬ p d› by blast
    qed
    then obtain d′ where ?f p1 = Some d′
      by blast
    then show False
      using assms(7) by simp
  qed

  have length (p1@p2) = (Suc ((size M) ∗ m))
```

using ‹*length p2* = (*Suc* ((*size M*) ∗ *m*)) − *length p1*›
‹*length p1* < *Suc* ((*size M*) ∗ *m*)›
by *simp*

have ∃ *q* ∈ *states M* . *length* (*filter* (λ*t* . *t-target t* = *q*) (*p1*@*p2*)) ≥ *Suc m*
proof (*rule ccontr*)
assume ¬ (∃ *q*∈*states M*. *Suc m* ≤ *length* (*filter* (λ*t*. *t-target t* = *q*) (*p1*@*p2*)))
then have ∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*p1*@*p2*)) < *Suc m*
by *auto*
then have ∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*p1*@*p2*)) ≤ *m*
by *auto*

have (∑ *q*∈*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) (*p1*@*p2*))) ≤ (∑ *q*∈*states M* . *m*)
using ‹∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*p1*@*p2*)) ≤ *m*› **by** (*meson sum-mono*)
then have *length* (*p1*@*p2*) ≤ *m* ∗ (*size M*)
using *path-length-sum*[*OF* ‹*path M q* (*p1*@*p2*)›]
using *fsm-states-finite*[*of M*]
by (*simp add*: *mult.commute*)

then show *False*
using ‹*length* (*p1*@*p2*) = *Suc* ((*size M*) ∗ *m*)›
by (*simp add*: *mult.commute*)
qed
then obtain *q′* **where** *q′* ∈ *states M*
and *length* (*filter* (λ *t* . *t-target t* = *q′*) (*p1*@*p2*)) ≥ *Suc m*
by *blast*
then obtain *d* **where** *d* ∈ *set D*
and *q′* ∈ *fst d*
using *assms*(*4*) **by** *blast*
then have ⋀ *t* . *t-target t* = *q′* ⟹ *t-target t* ∈ *fst d* **by** *auto*
then have *length* (*filter* (λ *t* . *t-target t* = *q′*) (*p1*@*p2*)) ≤ *length* (*filter* (λ *t* . *t-target t* ∈ *fst d*) (*p1*@*p2*))
using *filter-length-weakening*[*of* λ *t* . *t-target t* = *q′* λ *t* . *t-target t* ∈ *fst d*]
by *blast*
then have *Suc m* ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) (*p1*@*p2*))
using ‹*length* (*filter* (λ *t* . *t-target t* = *q′*) (*p1*@*p2*)) ≥ *Suc m*› **by** *auto*
then have *?f* (*p1*@*p2*) ≠ *None*
using *assms*(*2*)
proof −
have ∀ *p*. *find p D* ≠ *None* ∨ ¬ *p d*
by (*metis* ‹*d* ∈ *set D*› *find-from*)
have *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*p*. *t-target p* ∈ *fst d*) (*p1*@*p2*))
using ‹*Suc m* ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) (*p1*@*p2*))› **by** *linarith*
then show *?thesis*
using ‹∀ *p*. *find p D* ≠ *None* ∨ ¬ *p d*› **by** *blast*
qed

then obtain d' **where** *?f* $(p1@p2) = Some\ d'$
by *blast*

show *?thesis* **proof** (*cases* $(\forall p'\ p''.\ (p1@p2) = p' @ p'' \wedge p'' \neq [] \longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D = None)$)
case *True*
then show *?thesis*
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] ‹*path M q* $(p1@p2)$› ‹*?f* $(p1@p2) = Some\ d'$›
by *blast*
next
case *False*

define *ps* **where** *ps-def*: $ps = \{p'\ .\ \exists\ p''.\ (p1@p2) = p' @ p'' \wedge p'' \neq [] \wedge find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D \neq None\}$
have $ps \neq \{\}$ **using** *False ps-def* **by** *blast*
moreover have *finite ps*
proof −
have $ps \subseteq set\ (prefixes\ (p1@p2))$
unfolding *prefixes-set ps-def*
by *auto*
then show *?thesis*
by (*meson List.finite-set rev-finite-subset*)
qed
ultimately obtain p' **where** $p' \in ps$ **and** $\bigwedge p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$
by (*meson leI min-length-elem*)
then have $\bigwedge p''\ p'''\ .\ p' = p'' @ p''' \Longrightarrow p''' \neq [] \Longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D = None$
proof −
fix $p''\ p'''$ **assume** $p' = p'' @ p'''$ **and** $p''' \neq []$
show $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D = None$
proof (*rule ccontr*)
assume $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D \neq None$
moreover have $\exists p'''.\ (p1@p2) = p'' @ p''' \wedge p''' \neq []$
using ‹$p' \in ps$› ‹$p' = p'' @ p'''$› **unfolding** *ps-def* **by** *auto*
ultimately have $p'' \in ps$
unfolding *ps-def* **by** *auto*
moreover have $length\ p'' < length\ p'$
using ‹$p''' \neq []$› ‹$p' = p'' @ p'''$› **by** *auto*
ultimately show *False*
using ‹$\bigwedge p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$›
using *leD* **by** *auto*

qed
qed

obtain p'' **where** $(p1@p2) = p' @ p''$
and $p'' \neq []$
and *find* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D \neq None$
using ‹$p' \in ps$› **unfolding** *ps-def* **by** *blast*
then obtain d' **where** *find* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D = Some\ d'$
by *auto*

have *path* $M\ q\ p'$
using ‹*path* $M\ q\ (p1@p2)$› **unfolding** ‹$(p1@p2) = p' @ p''$› **by** *auto*

have $length\ p' > length\ p1$
proof (*rule ccontr*)
assume $\neg\ length\ p1 < length\ p'$
then obtain i **where** $p' = take\ i\ p1$
by (*metis* ‹$p1 @ p2 = p' @ p''$› *append-eq-append-conv-if less-le*)

have $\bigwedge i\ .\ find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1)))\ D = None$
proof −
fix i
show *find* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1)))\ D = None$
proof (*rule ccontr*)
assume *find* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1)))\ D \neq None$
then obtain d **where** $d \in set\ D$
and $Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1))$
using *find-None-iff*[*of* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1)))\ D$]
by *meson*

moreover have $length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ (take\ i\ p1)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p1)$
using *filter-take-length* **by** *metis*
ultimately have $Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p1)$
using *le-trans* **by** *blast*
then show *False*
using ‹$d \in set\ D$› *assms*(*7*) **unfolding** *find-None-iff*
by *blast*
qed
qed

then have *find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p')) D = *None*
unfolding ‹p' = *take* i *p1*› **by** *blast*
then show *False*
using ‹*find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p')) D ≠ *None*›
by *auto*
qed

moreover have p' = *take* (*length* p') (*p1*@*p2*)
using ‹(*p1*@*p2*) = p' @ p''› **by** *auto*

ultimately obtain p **where** p' = *p1* @ p
by *auto*

have *path* M q p' ∧
find (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p')) D = *Some* d' ∧
(∀ p'' p'''. p' = p'' @ p''' ∧ p''' ≠ [] ⟶ *find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p'')) D = *None*)
using ‹⋀p'' p''' . p' = p'' @ p''' ⟹ p''' ≠ [] ⟹ *find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p'')) D = *None*›
‹*path* M q p'› ‹*find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) p')) D = *Some* d'›
by *blast*
then have (p',d') ∈ (*m-traversal-paths-with-witness* M q D m)
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] **by** *blast*
then show *?thesis* **unfolding** ‹p' = *p1* @ p› **by** *blast*
qed
qed

lemma *m-traversal-path-extension-exist-for-transition* :
assumes *completely-specified* M
and q ∈ *states* M
and *inputs* M ≠ {}
and ⋀ q . q∈*states* M ⟹ ∃ d∈*set* D. q ∈ *fst* d
and ⋀ d . d ∈ *set* D ⟹ *snd* d ⊆ *fst* d
and *path* M q *p1*
and *find* (λd. *Suc* (m − *card* (*snd* d)) ≤ *length* (*filter* (λt. *t-target* t ∈ *fst* d) *p1*)) D = *None*
and t ∈ *transitions* M
and *t-source* t = *target* q *p1*
shows ∃ *p2* d' . (*p1*@[t]@*p2*,d') ∈ (*m-traversal-paths-with-witness* M q D m)
proof −
let *?q* = (*target* q (*p1* @ [t]))
let *?p* = *p1* @ [t]

have *path M q ?p*
using ‹*path M q p1*› ‹*t* ∈ *transitions M*› ‹*t-source t* = *target q p1*› *path-append-transition*
by *simp*

obtain *p2* **where** *path M ?q p2* **and** *length p2* = (*Suc* ((*size M*) ∗ *m*)) − (*length ?p*)
using *path-of-length-ex*[*OF assms*(*1*) *path-target-is-state*[*OF* ‹*path M q* (*p1*@[*t*])›] *assms*(*3*)]
by *blast*

have *path M q* (*?p*@*p2*)
using ‹*path M q ?p*› ‹*path M ?q p2*› **by** *auto*

let *?f* = (λ *p* . *find* (λ *d* . *length* (*filter* (λ*t* . *t-target t* ∈ *fst d*) *p*) ≥ *Suc* (*m* − (*card* (*snd d*)))) *D*)

have *length p1* < *Suc* ((*size M*) ∗ *m*)
proof (*rule ccontr*)
assume ¬ *length p1* < *Suc* (*FSM.size M* ∗ *m*)
then have *length* (*take* (*Suc* (*FSM.size M* ∗ *m*)) *p1*) = *Suc* (*FSM.size M* ∗ *m*)
by *auto*
let *?p* = (*take* (*Suc* (*FSM.size M* ∗ *m*)) *p1*)

have *path M q ?p*
using ‹*path M q p1*›
by (*metis append-take-drop-id path-append-elim*)

have ∃ *q* ∈ *states M* . *length* (*filter* (λ*t* . *t-target t* = *q*) *?p*) ≥ *Suc m*
proof (*rule ccontr*)
assume ¬ (∃ *q*∈*states M*. *Suc m* ≤ *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*))
then have ∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) < *Suc m*
by *auto*
then have ∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) ≤ *m*
by *auto*

have (∑ *q*∈*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*)) ≤ (∑ *q*∈*states M* . *m*)
using ‹∀ *q* ∈ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) *?p*) ≤ *m*› **by** (*meson sum-mono*)
then have *length ?p* ≤ *m* ∗ (*size M*)
using *path-length-sum*[*OF* ‹*path M q ?p*›]
using *fsm-states-finite*[*of M*]
by (*simp add*: *mult.commute*)

then show *False*
using ‹*length ?p* = *Suc* ((*size M*) ∗ *m*)›
by (*simp add*: *mult.commute*)

qed
then obtain q' **where** $q' \in$ *states M*
and *length* (*filter* (λ *t* . *t-target t* = q') *?p*) $\geq$ *Suc m*
by *blast*
then obtain *d* **where** $d \in$ *set D*
and $q' \in$ *fst d*
using *assms*(*4*) **by** *blast*
then have $\bigwedge$ *t* . *t-target t* = $q' \Longrightarrow$ *t-target t* $\in$ *fst d* **by** *auto*
then have *length* (*filter* (λ *t* . *t-target t* = q') *?p*) $\leq$ *length* (*filter* (λ *t* . *t-target t* $\in$ *fst d*) *?p*)
using *filter-length-weakening*[*of* λ *t* . *t-target t* = q' λ *t* . *t-target t* $\in$ *fst d*]
by *auto*
then have *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *?p*)
using ‹*length* (*filter* (λ *t* . *t-target t* = q') *?p*) $\geq$ *Suc m*› **by** *auto*
moreover have *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *?p*) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p1*)
proof −
have $\bigwedge$ *xs P n* . *length* (*filter P* (*take n xs*)) $\leq$ *length* (*filter P xs*)
by (*metis append-take-drop-id filter-append le0 le-add-same-cancel1 length-append*)
then show *?thesis* **by** *auto*
qed
ultimately have *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p1*)
by *auto*
then have *?f p1* $\neq$ *None*
using *assms*(*2*)
proof −
have $\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*
by (*metis* ‹*d* $\in$ *set D*› *find-from*)
have *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*p*. *t-target p* $\in$ *fst d*) *p1*)
using ‹*Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) *p1*)› **by** *linarith*
then show *?thesis*
using ‹$\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*› **by** *blast*
qed
then obtain d' **where** *?f p1* = *Some* d'
by *blast*
then show *False*
using *assms*(*7*) **by** *simp*
qed

have *length* (*?p@p2*) = (*Suc* ((*size M*) $*$ *m*))
using ‹*length p2* = (*Suc* ((*size M*) $*$ *m*)) − *length ?p*›
‹*length p1* < *Suc* ((*size M*) $*$ *m*)›
by *simp*

have $\exists$ $q \in$ *states M* . *length* (*filter* (λ*t* . *t-target t* = *q*) (*?p@p2*)) $\geq$ *Suc m*
proof (*rule ccontr*)
assume $\neg$ ($\exists$ *q*$\in$*states M*. *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* = *q*) (*?p@p2*)))

then have $\forall\ q \in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*?p@p2*)) < *Suc m*
by *auto*
then have $\forall\ q \in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*?p@p2*)) $\leq$ *m*
by *auto*

have ($\sum$ *q*$\in$*states M*. *length* (*filter* (λ*t*. *t-target t* = *q*) (*?p@p2*))) $\leq$ ($\sum$ *q*$\in$*states M* . *m*)
using ‹$\forall\ q \in$ *states M* . *length* (*filter* (λ*t*. *t-target t* = *q*) (*?p@p2*)) $\leq$ *m*› **by** (*meson sum-mono*)
then have *length* (*?p@p2*) $\leq$ *m* $*$ (*size M*)
using *path-length-sum*[*OF* ‹*path M q* (*?p@p2*)›]
using *fsm-states-finite*[*of M*]
by (*simp add*: *mult.commute*)

then show *False*
using ‹*length* (*?p@p2*) = *Suc* ((*size M*) $*$ *m*)›
by (*simp add*: *mult.commute*)
qed

then obtain *q′* **where** *q′* $\in$ *states M*
and *length* (*filter* (λ *t* . *t-target t* = *q′*) (*?p@p2*)) $\geq$ *Suc m*
by *blast*
then obtain *d* **where** *d* $\in$ *set D*
and *q′* $\in$ *fst d*
using *assms*(*4*) **by** *blast*
then have $\bigwedge$ *t* . *t-target t* = *q′* $\Longrightarrow$ *t-target t* $\in$ *fst d* **by** *auto*
then have *length* (*filter* (λ *t* . *t-target t* = *q′*) (*?p@p2*)) $\leq$ *length* (*filter* (λ *t* . *t-target t* $\in$ *fst d*) (*?p@p2*))
using *filter-length-weakening*[*of* λ *t* . *t-target t* = *q′* λ *t* . *t-target t* $\in$ *fst d*]
by *blast*
then have *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) (*?p@p2*))
using ‹*length* (*filter* (λ *t* . *t-target t* = *q′*) (*?p@p2*)) $\geq$ *Suc m*› **by** *auto*
then have *?f* (*?p@p2*) $\neq$ *None*
using *assms*(*2*)
proof −
have $\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*
by (*metis* ‹*d* $\in$ *set D*› *find-from*)
have *Suc* (*m* − *card* (*snd d*)) $\leq$ *length* (*filter* (λ*p*. *t-target p* $\in$ *fst d*) (*?p@p2*))
using ‹*Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst d*) (*?p@p2*))› **by** *linarith*
then show *?thesis*
using ‹$\forall$ *p*. *find p D* $\neq$ *None* $\vee$ $\neg$ *p d*› **by** *blast*
qed
then obtain *d′* **where** *?f* (*?p@p2*) = *Some d′*
by *blast*

show *?thesis* **proof** (*cases* ($\forall p'\ p''$. (*?p@p2*) = p' @ p'' $\wedge$ $p'' \neq []$ $\longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) D = *None*))
case *True*
obtain d' **where** ((*?p@p2*), d') $\in$ *m-traversal-paths-with-witness M q D m*
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] ‹*path M q* (*?p@p2*)› ‹*?f* (*?p@p2*) = *Some* d'› *True* **by** *force*
then show *?thesis*
unfolding *append.assoc*[*symmetric*] **by** *blast*
next
case *False*

show *?thesis* **proof** (*cases find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) *?p*)) D)
case (*Some a*)

have $*$: ($\forall p'\ p''$. *?p* = p' @ p'' $\wedge$ $p'' \neq []$ $\longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) D = *None*)
proof −
have $\bigwedge p'\ p''$. *?p* = p' @ $p'' \Longrightarrow p'' \neq [] \Longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) D = *None*
proof −
fix $p'\ p''$ **assume** *?p* = p' @ p'' **and** $p'' \neq []$
then have *length* p' $\leq$ *length p1* **by** (*induction* p'' *rule*: *rev-induct*; *auto*)
moreover have p' = *take* (*length* p') *?p*
unfolding ‹*?p* = p' @ p''› **by** *auto*
ultimately have p' = *take* (*length* p') *p1*
by *auto*

show *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) D = *None*
proof (*rule ccontr*)
assume *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) $D \neq$ *None*
moreover have $\bigwedge x$. *length* (*filter* (λt. *t-target t* $\in$ *fst x*) p') $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst x*) *p1*)
using ‹p' = *take* (*length* p') *p1*›
by (*metis filter-take-length*)
ultimately have *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) *p1*)) $D \neq$ *None*
unfolding *find-None-iff*
using *le-trans* **by** *blast*
then show *False*
using *assms*(*7*) **by** *simp*
qed
qed
then show *?thesis* **by** *blast*
qed

obtain d' **where** $(?p,\ d') \in$ *m-traversal-paths-with-witness M q D m*
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] ‹*path M q ?p*› *Some* $*$ **by** *force*
then show *?thesis*
by *fastforce*
next
case *None*

define *ps* **where** *ps-def*: $ps = \{p' \,.\, \exists\ p''.\ (?p@p2) = p' @ p''$
$\wedge\ p'' \neq []$
$\wedge$ *find* $(\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p'))\ D \neq None\}$
have $ps \neq \{\}$ **using** *False ps-def* **by** *blast*
moreover have *finite ps*
proof $-$
have $ps \subseteq set\ (prefixes\ (?p@p2))$
unfolding *prefixes-set ps-def*
by *auto*
then show *?thesis*
by (*meson List.finite-set rev-finite-subset*)
qed
ultimately obtain p' **where** $p' \in ps$ **and** $\bigwedge\ p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$
by (*meson leI min-length-elem*)
then have $\bigwedge p''\ p'''\ .\ p' = p'' @ p''' \Longrightarrow p''' \neq [] \Longrightarrow find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D = None$
proof $-$
fix $p''\ p'''$ **assume** $p' = p'' @ p'''$ **and** $p''' \neq []$
show $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D = None$
proof (*rule ccontr*)
assume $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in fst\ d)\ p''))\ D \neq None$
moreover have $\exists p'''.\ (?p@p2) = p'' @ p''' \wedge p''' \neq []$
using ‹$p' \in ps$› ‹$p' = p'' @ p'''$› **unfolding** *ps-def* **by** *auto*
ultimately have $p'' \in ps$
unfolding *ps-def* **by** *auto*
moreover have $length\ p'' < length\ p'$
using ‹$p''' \neq []$› ‹$p' = p'' @ p'''$› **by** *auto*
ultimately show *False*
using ‹$\bigwedge\ p''\ .\ p'' \in ps \Longrightarrow length\ p' \leq length\ p''$›
using *leD* **by** *auto*
qed
qed

obtain p'' **where** $(?p@p2) = p' @ p''$
and $p'' \neq []$
and $find\ (\lambda d.\ Suc\ (m - card\ (snd\ d)) \leq length\ (filter\ (\lambda t.\ t\text{-}target$

$t \in$ *fst d*) p')) $D \neq$ *None*
using ‹$p' \in ps$› **unfolding** *ps-def* **by** *blast*
then obtain d' **where** *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) D = *Some* d'
by *auto*

have *path M q* p'
using ‹*path M q* (*?p@p2*)› **unfolding** ‹(*?p@p2*) = p' @ p''› **by** *auto*

have *length* p' > *length ?p*
proof (*rule ccontr*)

assume ¬ *length ?p* < *length* p'
then obtain i **where** p' = *take i ?p*
by (*metis* ‹*?p* @ *p2* = p' @ p''› *append-eq-append-conv-if less-le*)

have ⋀ i . *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*))) D = *None*
proof −
fix i
show *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*))) D = *None*
proof (*rule ccontr*)
assume *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*))) $D \neq$ *None*
then obtain d **where** $d \in$ *set D*
and *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*))
using *find-None-iff*[*of* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*))) D]
by *meson*

moreover have *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*take i ?p*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) *?p*)
using *filter-take-length* **by** *metis*
ultimately have *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) *?p*)
using *le-trans* **by** *blast*
then show *False*
using ‹$d \in$ *set D*› *None* **unfolding** *find-None-iff*
by *blast*
qed
qed

then have *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) D = *None*
unfolding ‹p' = *take i ?p*› **by** *blast*
then show *False*
using ‹*find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈

fst d) *p*′)) *D* ≠ *None*›
by *auto*
qed

moreover have *p*′ = *take* (*length p*′) (*?p@p2*)
using ‹(*?p@p2*) = *p*′ @ *p*′′› **by** *auto*

ultimately obtain *p* **where** *p*′ = *?p* @ *p*
by (*metis dual-order.strict-implies-order take-all take-append*)

have *path M q p*′ ∧
find (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p*′)) *D* = *Some d*′ ∧
(∀ *p*′′ *p*′′′. *p*′ = *p*′′ @ *p*′′′ ∧ *p*′′′ ≠ [] ⟶ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p*′′)) *D* = *None*)
using ‹⋀*p*′′ *p*′′′ . *p*′ = *p*′′ @ *p*′′′ ⟹ *p*′′′ ≠ [] ⟹ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p*′′)) *D* = *None*› ‹*path M q p*′› ‹*find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p*′)) *D* = *Some d*′›
by *blast*
then have (*p*′,*d*′) ∈ (*m-traversal-paths-with-witness M q D m*)
using *m-traversal-paths-with-witness-set*[*OF assms*(*4*,*5*,*2*), *of m*] **by** *blast*
then show *?thesis* **unfolding** ‹*p*′ = *?p* @ *p*› **by** *fastforce*
qed
qed
qed

end

41 Test Suites

This theory introduces a predicate *implies-completeness* and proves that any test suite satisfying this predicate is sufficient to check the reduction conformance relation between two (possibly nondeterministic FSMs)

theory *Test-Suite*
imports *Helper-Algorithms Adaptive-Test-Case Traversal-Set*
begin

41.1 Preliminary Definitions

type-synonym (′*a*,′*b*,′*c*) *preamble* = (′*a*,′*b*,′*c*) *fsm*
type-synonym (′*a*,′*b*,′*c*) *traversal-path* = (′*a* × ′*b* × ′*c* × ′*a*) *list*
type-synonym (′*a*,′*b*,′*c*) *separator* = (′*a*,′*b*,′*c*) *fsm*

A test suite contains of 1) a set of d-reachable states with their associated preambles 2) a map from d-reachable states to their associated m-traversal paths 3) a map from d-reachable states and associated m-traversal paths to

the set of states to r-distinguish the targets of those paths from 4) a map from pairs of r-distinguishable states to a separator

datatype $('a,'b,'c,'d)$ *test-suite* = *Test-Suite* $('a \times ('a,'b,'c)$ *preamble*$)$ *set*
$'a \Rightarrow ('a,'b,'c)$ *traversal-path set*
$('a \times ('a,'b,'c)$ *traversal-path*$) \Rightarrow 'a$ *set*
$('a \times 'a) \Rightarrow (('d,'b,'c)$ *separator* $\times\ 'd \times 'd)$ *set*

41.2 A Sufficiency Criterion for Reduction Testing

fun *implies-completeness-for-repetition-sets* :: $('a,'b,'c,'d)$ *test-suite* $\Rightarrow$ $('a,'b,'c)$ *fsm* $\Rightarrow$ *nat* $\Rightarrow$ $('a$ *set* $\times\ 'a$ *set*$)$ *list* $\Rightarrow$ *bool* **where**
implies-completeness-for-repetition-sets (*Test-Suite prs tps rd-targets separators*) *M m repetition-sets* =
((*initial M*,*initial-preamble M*) $\in$ *prs*
$\wedge$ ($\forall$ *q P* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ (*is-preamble P M q*) $\wedge$ (*tps q*) $\neq$ {})
$\wedge$ ($\forall$ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) $\in$ *separators* (*q1*,*q2*) $\longrightarrow$ (*A*,*d2*,*d1*) $\in$ *separators* (*q2*,*q1*) $\wedge$ *is-separator M q1 q2 A d1 d2*)
$\wedge$ ($\forall$ *q* . *q* $\in$ *states M* $\longrightarrow$ ($\exists$ *d* $\in$ *set repetition-sets*. *q* $\in$ *fst d*))
$\wedge$ ($\forall$ *d* . *d* $\in$ *set repetition-sets* $\longrightarrow$ ((*fst d* $\subseteq$ *states M*) $\wedge$ (*snd d* = *fst d* $\cap$ *fst* ‘ *prs*) $\wedge$ ($\forall$ *q1 q2* . *q1* $\in$ *fst d* $\longrightarrow$ *q2* $\in$ *fst d* $\longrightarrow$ *q1* $\neq$ *q2* $\longrightarrow$ *separators* (*q1*,*q2*) $\neq$ {})))
$\wedge$ ($\forall$ *q* . *q* $\in$ *image fst prs* $\longrightarrow$ *tps q* $\subseteq$ {*p1* . $\exists$ *p2 d* . (*p1*@*p2*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*} $\wedge$ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) $\subseteq$ *tps q*)
$\wedge$ ($\forall$ *q p d* . *q* $\in$ *image fst prs* $\longrightarrow$ (*p*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\longrightarrow$
(($\forall$ *p1 p2 p3* . *p*=*p1*@*p2*@*p3* $\longrightarrow$ *p2* $\neq$ [] $\longrightarrow$ *target q p1* $\in$ *fst d* $\longrightarrow$ *target q* (*p1*@*p2*) $\in$ *fst d* $\longrightarrow$ *target q p1* $\neq$ *target q* (*p1*@*p2*) $\longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ (*p1*@*p2*) $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*,(*p1*@*p2*)) $\wedge$ *target q* (*p1*@*p2*) $\in$ *rd-targets* (*q*,*p1*)))
$\wedge$ ($\forall$ *p1 p2 q'* . *p*=*p1*@*p2* $\longrightarrow$ *q'* $\in$ *image fst prs* $\longrightarrow$ *target q p1* $\in$ *fst d* $\longrightarrow$ *q'* $\in$ *fst d* $\longrightarrow$ *target q p1* $\neq$ *q'* $\longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*,[]) $\wedge$ *q'* $\in$ *rd-targets* (*q*,*p1*))))
$\wedge$ ($\forall$ *q1 q2* . *q1* $\neq$ *q2* $\longrightarrow$ *q1* $\in$ *snd d* $\longrightarrow$ *q2* $\in$ *snd d* $\longrightarrow$ ([] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*,[]) $\wedge$ *q2* $\in$ *rd-targets* (*q1*,[]))))
)

definition *implies-completeness* :: $('a,'b,'c,'d)$ *test-suite* $\Rightarrow$ $('a,'b,'c)$ *fsm* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
implies-completeness T M m = ($\exists$ *repetition-sets* . *implies-completeness-for-repetition-sets T M m repetition-sets*)

lemma *implies-completeness-for-repetition-sets-simps* :
assumes *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets separators*) *M m repetition-sets*
shows (*initial M*,*initial-preamble M*) $\in$ *prs*
and $\bigwedge$ *q P* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$ (*is-preamble P M q*) $\wedge$ (*tps q*) $\neq$ {}

and $\bigwedge$ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) ∈ *separators* (*q1*,*q2*) ⟹ (*A*,*d2*,*d1*) ∈ *separators* (*q2*,*q1*) ∧ *is-separator M q1 q2 A d1 d2*
and $\bigwedge$ *q* . *q* ∈ *states M* ⟹ (∃ *d* ∈ *set repetition-sets*. *q* ∈ *fst d*)
and $\bigwedge$ *d* . *d* ∈ *set repetition-sets* ⟹ (*fst d* ⊆ *states M*) ∧ (*snd d* = *fst d* ∩ *fst* ‘ *prs*)
and $\bigwedge$ *d q1 q2* . *d* ∈ *set repetition-sets* ⟹ *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *separators* (*q1*,*q2*) ≠ {}
and $\bigwedge$ *q* . *q* ∈ *image fst prs* ⟹ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*} ∧ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) ⊆ *tps q*
and $\bigwedge$ *q p d p1 p2 p3* . *q* ∈ *image fst prs* ⟹ (*p*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹ *p*=*p1*@*p2*@*p3* ⟹ *p2* ≠ [] ⟹ *target q p1* ∈ *fst d* ⟹ *target q* (*p1*@*p2*) ∈ *fst d* ⟹ *target q p1* ≠ *target q* (*p1*@*p2*) ⟹ (*p1* ∈ *tps q* ∧ (*p1*@*p2*) ∈ *tps q* ∧ *target q p1* ∈ *rd-targets* (*q*,(*p1*@*p2*)) ∧ *target q* (*p1*@*p2*) ∈ *rd-targets* (*q*,*p1*))
and $\bigwedge$ *q p d p1 p2 q′* . *q* ∈ *image fst prs* ⟹ (*p*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹ *p*=*p1*@*p2* ⟹ *q′* ∈ *image fst prs* ⟹ *target q p1* ∈ *fst d* ⟹ *q′* ∈ *fst d* ⟹ *target q p1* ≠ *q′* ⟹ (*p1* ∈ *tps q* ∧ [] ∈ *tps q′* ∧ *target q p1* ∈ *rd-targets* (*q′*,[]) ∧ *q′* ∈ *rd-targets* (*q*,*p1*))
and $\bigwedge$ *q p d q1 q2* . *q* ∈ *image fst prs* ⟹ (*p*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹ *q1* ≠ *q2* ⟹ *q1* ∈ *snd d* ⟹ *q2* ∈ *snd d* ⟹ ([] ∈ *tps q1* ∧ [] ∈ *tps q2* ∧ *q1* ∈ *rd-targets* (*q2*,[]) ∧ *q2* ∈ *rd-targets* (*q1*,[]))
proof−

show (*initial M*,*initial-preamble M*) ∈ *prs*
and $\bigwedge$ *q* . *q* ∈ *image fst prs* ⟹ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*} ∧ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) ⊆ *tps q*
using *assms* **unfolding** *implies-completeness-for-repetition-sets.simps* **by** *blast*+

show $\bigwedge$ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) ∈ *separators* (*q1*,*q2*) ⟹ (*A*,*d2*,*d1*) ∈ *separators* (*q2*,*q1*) ∧ *is-separator M q1 q2 A d1 d2*
and $\bigwedge$ *q P* . (*q*,*P*) ∈ *prs* ⟹ (*is-preamble P M q*) ∧ (*tps q*) ≠ {}
and $\bigwedge$ *q* . *q* ∈ *states M* ⟹ (∃ *d* ∈ *set repetition-sets*. *q* ∈ *fst d*)
and $\bigwedge$ *d* . *d* ∈ *set repetition-sets* ⟹ (*fst d* ⊆ *states M*) ∧ (*snd d* = *fst d* ∩ *fst* ‘ *prs*)
and $\bigwedge$ *d q1 q2* . *d* ∈ *set repetition-sets* ⟹ *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *separators* (*q1*,*q2*) ≠ {}
using *assms* **unfolding** *implies-completeness-for-repetition-sets.simps* **by** *force*+

show $\bigwedge$ *q p d p1 p2 p3* . *q* ∈ *image fst prs* ⟹ (*p*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹ *p*=*p1*@*p2*@*p3* ⟹ *p2* ≠ [] ⟹ *target q p1* ∈ *fst d* ⟹ *target q* (*p1*@*p2*) ∈ *fst d* ⟹ *target q p1* ≠ *target q* (*p1*@*p2*) ⟹ (*p1* ∈ *tps q* ∧ (*p1*@*p2*) ∈ *tps q* ∧ *target q p1* ∈ *rd-targets* (*q*,(*p1*@*p2*)) ∧ *target q* (*p1*@*p2*) ∈ *rd-targets* (*q*,*p1*))
using *assms* **unfolding** *implies-completeness-for-repetition-sets.simps* **by** (*metis* (*no-types*, *lifting*))

show $\bigwedge$ *q p d p1 p2 q′* . *q* ∈ *image fst prs* ⟹ (*p*,*d*) ∈ *m-traversal-paths-with-witness*

M q repetition-sets m $\Longrightarrow$ *p=p1@p2* $\Longrightarrow$ *q′* $\in$ *image fst prs* $\Longrightarrow$ *target q p1* $\in$ *fst d* $\Longrightarrow$ *q′* $\in$ *fst d* $\Longrightarrow$ *target q p1* $\neq$ *q′* $\Longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ [] $\in$ *tps q′* $\wedge$ *target q p1* $\in$ *rd-targets* (*q′*,[]) $\wedge$ *q′* $\in$ *rd-targets* (*q*,*p1*))
using *assms* **unfolding** *implies-completeness-for-repetition-sets.simps* **by** (*metis* (*no-types*, *lifting*))

show $\bigwedge$ *q p d q1 q2* . *q* $\in$ *image fst prs* $\Longrightarrow$ (*p*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *q1* $\in$ *snd d* $\Longrightarrow$ *q2* $\in$ *snd d* $\Longrightarrow$ ([] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*,[]) $\wedge$ *q2* $\in$ *rd-targets* (*q1*,[]))
using *assms* **unfolding** *implies-completeness-for-repetition-sets.simps* **by** (*metis* (*no-types*, *lifting*))
qed

41.3 A Pass Relation for Test Suites and Reduction Testing

fun *passes-test-suite* :: (*′a*,*′b*,*′c*) *fsm* $\Rightarrow$ (*′a*,*′b*,*′c*,*′d*) *test-suite* $\Rightarrow$ (*′e*,*′b*,*′c*) *fsm* $\Rightarrow$ *bool* **where**
passes-test-suite M (*Test-Suite prs tps rd-targets separators*) *M′* = (
— Reduction on preambles: as the preambles contain all responses of M to their chosen inputs, M' must not exhibit any other response
($\forall$ *q P io x y y′* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\longrightarrow$ *io*@[(*x*,*y′*)] $\in$ *L M′* $\longrightarrow$ *io*@[(*x*,*y′*)] $\in$ *L P*)
— Reduction on traversal-paths applied after preambles (i.e., completed paths in preambles) - note that tps q is not necessarily prefix-complete
$\wedge$ ($\forall$ *q P pP ioT pT x y y′* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *path P* (*initial P*) *pP* $\longrightarrow$ *target* (*initial P*) *pP* = *q* $\longrightarrow$ *pT* $\in$ *tps q* $\longrightarrow$ *ioT*@[(*x*,*y*)] $\in$ *set* (*prefixes* (*p-io pT*)) $\longrightarrow$ (*p-io pP*)@*ioT*@[(*x*,*y′*)] $\in$ *L M′* $\longrightarrow$ ($\exists$ *pT′* . *pT′* $\in$ *tps q* $\wedge$ *ioT*@[(*x*,*y′*)] $\in$ *set* (*prefixes* (*p-io pT′*))))
— Passing separators: if M' contains an IO-sequence that in the test suite leads through a preamble and an m-traversal path and the target of the latter is to be r-distinguished from some other state, then M' passes the corresponding ATC
$\wedge$ ($\forall$ *q P pP pT* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *path P* (*initial P*) *pP* $\longrightarrow$ *target* (*initial P*) *pP* = *q* $\longrightarrow$ *pT* $\in$ *tps q* $\longrightarrow$ (*p-io pP*)@(*p-io pT*) $\in$ *L M′* $\longrightarrow$ ($\forall$ *q′ A d1 d2 qT* . *q′* $\in$ *rd-targets* (*q*,*pT*) $\longrightarrow$ (*A*,*d1*,*d2*) $\in$ *separators* (*target q pT*, *q′*) $\longrightarrow$ *qT* $\in$ *io-targets M′* ((*p-io pP*)@(*p-io pT*)) (*initial M′*) $\longrightarrow$ *pass-separator-ATC M′ A qT d2*))
)

41.4 Soundness of Sufficient Test Suites

lemma *passes-test-suite-soundness-helper-1* :
assumes *is-preamble P M q*
and *observable M*
and *io*@[(*x*,*y*)] $\in$ *L P*
and *io*@[(*x*,*y′*)] $\in$ *L M*
shows *io*@[(*x*,*y′*)] $\in$ *L P*
proof −
have *is-submachine P M*
and $*$: $\bigwedge$ *q′ x t t′* . *q′*$\in$*reachable-states P* $\Longrightarrow$ *x*$\in$*FSM.inputs M* $\Longrightarrow$
t$\in$*FSM.transitions P* $\Longrightarrow$ *t-source t* = *q′* $\Longrightarrow$ *t-input t* = *x* $\Longrightarrow$

$t' \in FSM.transitions\ M \Longrightarrow$ *t-source* $t' = q' \Longrightarrow$ *t-input* $t' = x \Longrightarrow t' \in FSM.transitions\ P$
using *assms*(*1*) **unfolding** *is-preamble-def* **by** *blast+*

have *initial P* = *initial M*
unfolding *submachine-simps*[*OF* ‹*is-submachine P M*›]
by *simp*

obtain *p* **where** *path M* (*initial M*) *p* **and** *p-io p* = *io* @ [(*x*,*y*′)]
using *assms*(*4*) **unfolding** *submachine-simps*[*OF* ‹*is-submachine P M*›] **by** *auto*

obtain *p*′ *t* **where** *p* = *p*′@[*t*]
using ‹*p-io p* = *io* @ [(*x*,*y*′)]› **by** (*induction p rule*: *rev-induct*; *auto*)

have *path M* (*initial M*) *p*′ **and** *t* ∈ *transitions M* **and** *t-source t* = *target* (*initial M*) *p*′
using ‹*path M* (*initial M*) *p*› *path-append-transition-elim*
unfolding ‹*p* = *p*′@[*t*]› **by** *force+*

have *p-io p*′ = *io* **and** *t-input t* = *x* **and** *t-output t* = *y*′
using ‹*p-io p* = *io* @ [(*x*,*y*′)]› **unfolding** ‹*p* = *p*′@[*t*]› **by** *force+*

have *p-io p*′ ∈ *LS P* (*FSM.initial M*)
using *assms*(*3*) **unfolding** ‹*p-io p*′ = *io*› ‹*initial P* = *initial M*›
by (*meson language-prefix*)

have *FSM.initial M* ∈ *reachable-states P*
unfolding *submachine-simps*(*1*)[*OF* ‹*is-submachine P M*›, *symmetric*]
using *reachable-states-initial* **by** *blast*

obtain *pp* **where** *path P* (*initial P*) *pp* **and** *p-io pp* = *io* @ [(*x*,*y*)]
using *assms*(*3*) **by** *auto*
then obtain *pp*′ *t*′ **where** *pp* = *pp*′@[*t*′]
proof −
assume *a1*: ⋀*pp*′ *t*′. *pp* = *pp*′ @ [*t*′] ⟹ *thesis*
have *pp* ≠ []
using ‹*p-io pp* = *io* @ [(*x*, *y*)]› **by** *auto*
then show *?thesis*
using *a1* **by** (*metis* (*no-types*) *rev-exhaust*)
qed

have *path P* (*initial P*) *pp*′ **and** *t*′ ∈ *transitions P* **and** *t-source t*′ = *target* (*initial P*) *pp*′
using ‹*path P* (*initial P*) *pp*› *path-append-transition-elim*
unfolding ‹*pp* = *pp*′@[*t*′]› **by** *force+*
have *p-io pp*′ = *io* **and** *t-input t*′ = *x*
using ‹*p-io pp* = *io* @ [(*x*,*y*)]› **unfolding** ‹*pp* = *pp*′@[*t*′]› **by** *force+*

have *path M (initial M) pp′*
using ‹*path P (initial P) pp′*› *submachine-path-initial*[*OF* ‹*is-submachine P M*›] **by** *blast*

have *pp′ = p′*
using *observable-path-unique*[*OF assms*(*2*) ‹*path M (initial M) pp′*› ‹*path M (initial M) p′*›]
unfolding ‹*p-io pp′ = io*› ‹*p-io p′ = io*›
by *blast*
then have *t-source t′ = target (initial M) p′*
using ‹*t-source t′ = target (initial P) pp′*› **unfolding** ‹*initial P = initial M*› **by** *blast*

have *path P (FSM.initial M) p′*
using *observable-preamble-paths*[*OF assms*(*1*,*2*) ‹*path M (initial M) p′*›
‹*p-io p′ ∈ LS P (FSM.initial M)*›
‹*FSM.initial M ∈ reachable-states P*›]
by *assumption*
then have *target (initial M) p′ ∈ reachable-states P*
using *reachable-states-intro* **unfolding** ‹*initial P = initial M*›[*symmetric*] **by** *blast*
moreover have *x ∈ inputs M*
using ‹*t ∈ transitions M*› ‹*t-input t = x*› *fsm-transition-input* **by** *blast*

have *t ∈ transitions P*
using ∗[*OF* ‹*target (initial M) p′ ∈ reachable-states P*› ‹*x ∈ inputs M*› ‹*t′ ∈ transitions P*›
‹*t-source t′ = target (initial M) p′*› ‹*t-input t′ = x*› ‹*t ∈ transitions M*›
‹*t-source t = target (FSM.initial M) p′*› ‹*t-input t = x*›]
by *assumption*

then have *path P (initial P) (p′@[t])*
using ‹*path P (initial P) pp′*› ‹*t-source t = target (initial M) p′*›
unfolding ‹*pp′ = p′*› ‹*initial P = initial M*›
using *path-append-transition* **by** *simp*
then show *?thesis*
unfolding ‹*p = p′@[t]*›[*symmetric*] *LS.simps*
using ‹*p-io p = io@[(x,y′)]*›
by *force*
qed

lemma *passes-test-suite-soundness* :
assumes *implies-completeness* (*Test-Suite prs tps rd-targets separators*) *M m*

and *observable M*
and *observable M′*
and *inputs M′ = inputs M*
and *completely-specified M*
and *L M′ ⊆ L M*
shows *passes-test-suite M (Test-Suite prs tps rd-targets separators) M′*
proof −
obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets (Test-Suite prs tps rd-targets separators) M m repetition-sets*
using *assms*(*1*) **unfolding** *implies-completeness-def* **by** *blast*

have *t1*: (*initial M*, *initial-preamble M*) ∈ *prs*
using *implies-completeness-for-repetition-sets-simps*(*1*)[*OF repetition-sets-def*] **by** *assumption*
have *t2*: ⋀ *q P*. (*q*, *P*) ∈ *prs* ⟹ *is-preamble P M q* ∧ *tps q* ≠ {}
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*] **by** *assumption*
have *t3*: ⋀ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) ∈ *separators* (*q1*, *q2*) ⟹ (*A*, *d2*, *d1*) ∈ *separators* (*q2*, *q1*) ∧ *is-separator M q1 q2 A d1 d2*
using *implies-completeness-for-repetition-sets-simps*(*3*)[*OF repetition-sets-def*] **by** *assumption*

have *t5*: ⋀*q*. *q* ∈ *FSM.states M* ⟹ (∃ *d*∈*set repetition-sets*. *q* ∈ *fst d*)
using *implies-completeness-for-repetition-sets-simps*(*4*)[*OF repetition-sets-def*] **by** *assumption*

have *t6*: ⋀ *q*. *q* ∈ *fst ‘ prs* ⟹ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*}
∧ *fst ‘* (*m-traversal-paths-with-witness M q repetition-sets m*) ⊆ *tps q*
using *implies-completeness-for-repetition-sets-simps*(*7*)[*OF repetition-sets-def*] **by** *assumption*

have *t7*: ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *fst d* ⊆ *FSM.states M*
and *t8*: ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *snd d* ⊆ *fst d*
and *t9*: ⋀ *d q1 q2*. *d* ∈ *set repetition-sets* ⟹ *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *separators* (*q1*, *q2*) ≠ {}
using *implies-completeness-for-repetition-sets-simps*(*5*,*6*)[*OF repetition-sets-def*]

by *blast*+

have *t10*: ⋀ *q p d p1 p2 p3*.
q ∈ *fst ‘ prs* ⟹
(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹
p = *p1* @ *p2* @ *p3* ⟹
p2 ≠ [] ⟹
target q p1 ∈ *fst d* ⟹
target q (*p1* @ *p2*) ∈ *fst d* ⟹

target q p1 $\neq$ *target q* (*p1* @ *p2*) $\Longrightarrow$
p1 $\in$ *tps q* $\wedge$ *p1* @ *p2* $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*, *p1* @ *p2*) $\wedge$ *target q* (*p1* @ *p2*) $\in$ *rd-targets* (*q*, *p1*)
using *implies-completeness-for-repetition-sets-simps*(*8*)[*OF repetition-sets-def*] **by** *assumption*

have *t11*: $\bigwedge$ *q p d p1 p2 q'*.
q $\in$ *fst* ‘ *prs* $\Longrightarrow$
(*p*, *d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$
p = *p1* @ *p2* $\Longrightarrow$
q' $\in$ *fst* ‘ *prs* $\Longrightarrow$
target q p1 $\in$ *fst d* $\Longrightarrow$
q' $\in$ *fst d* $\Longrightarrow$
target q p1 $\neq$ *q'* $\Longrightarrow$
p1 $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*, []) $\wedge$ *q'* $\in$ *rd-targets* (*q*, *p1*)
using *implies-completeness-for-repetition-sets-simps*(*9*)[*OF repetition-sets-def*] **by** *assumption*

have $\bigwedge$ *q P io x y y'* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L M'* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L P*
proof –
fix *q P io x y y'* **assume** (*q*,*P*) $\in$ *prs* **and** *io*@[(*x*,*y*)] $\in$ *L P* **and** *io*@[(*x*,*y'*)] $\in$ *L M'*

have *is-preamble P M q*
using ‹(*q*,*P*) $\in$ *prs*› ‹$\bigwedge$ *q P*. (*q*, *P*) $\in$ *prs* $\Longrightarrow$ *is-preamble P M q* $\wedge$ *tps q* $\neq$ {}› **by** *blast*

have *io*@[(*x*,*y'*)] $\in$ *L M*
using ‹*io*@[(*x*,*y'*)] $\in$ *L M'*› *assms*(*6*) **by** *blast*

show *io*@[(*x*,*y'*)] $\in$ *L P*
using *passes-test-suite-soundness-helper-1*[*OF* ‹*is-preamble P M q*› *assms*(*2*) ‹*io*@[(*x*,*y*)] $\in$ *L P*› ‹*io*@[(*x*,*y'*)] $\in$ *L M*›]
by *assumption*
qed
then have *p1*: ($\forall$ *q P io x y y'* . (*q*,*P*) $\in$ *prs* $\longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L M'* $\longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L P*)
by *blast*

have $\bigwedge$ *q P pP ioT pT x x' y y'* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$
path P (*initial P*) *pP* $\Longrightarrow$
target (*initial P*) *pP* = *q* $\Longrightarrow$
pT $\in$ *tps q* $\Longrightarrow$
ioT @ [(*x*, *y*)] $\in$ *set* (*prefixes* (*p-io pT*)) $\Longrightarrow$

$(p\text{-}io\ pP)@ioT@[(x',y')] \in L\ M' \Longrightarrow$
$(\exists\ pT'\ .\ pT' \in tps\ q \wedge ioT\ @\ [(x',\ y')] \in set\ (prefixes$
$(p\text{-}io\ pT')))$
proof −
fix $q\ P\ pP\ ioT\ pT\ x\ x'\ y\ y'$
assume $(q,P) \in prs$
and $path\ P\ (initial\ P)\ pP$
and $target\ (initial\ P)\ pP = q$
and $pT \in tps\ q$
and $ioT\ @\ [(x,\ y)] \in set\ (prefixes\ (p\text{-}io\ pT))$
and $(p\text{-}io\ pP)@ioT@[(x',y')] \in L\ M'$

have $is\text{-}preamble\ P\ M\ q$
using ‹$(q,P) \in prs$› ‹$\bigwedge\ q\ P.\ (q,\ P) \in prs \Longrightarrow is\text{-}preamble\ P\ M\ q \wedge tps\ q \neq$
$\{\}$› **by** *blast*
then have $q \in states\ M$
unfolding *is-preamble-def*
by (*metis* ‹$path\ P\ (FSM.initial\ P)\ pP$› ‹$target\ (FSM.initial\ P)\ pP = q$›
path-target-is-state submachine-path)

have $initial\ P = initial\ M$
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have $path\ M\ (initial\ M)\ pP$
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹$path\ P\ (FSM.initial\ P)\ pP$› **by** *blast*

have $(p\text{-}io\ pP)@ioT@[(x',y')] \in L\ M$
using ‹$(p\text{-}io\ pP)@ioT@[(x',y')] \in L\ M'$› *assms*(*6*) **by** *blast*
then obtain pM' **where** $path\ M\ (initial\ M)\ pM'$ **and** $p\text{-}io\ pM' = (p\text{-}io$
$pP)@ioT@[(x',y')]$
by *auto*

let $?pP = take\ (length\ pP)\ pM'$
let $?pT = take\ (length\ ioT)\ (drop\ (length\ pP)\ pM')$
let $?t\ = last\ pM'$

have $pM' = ?pP\ @\ ?pT\ @\ [?t]$
proof −
have $length\ pM' = (length\ pP) + (length\ ioT) + 1$
using ‹$p\text{-}io\ pM' = (p\text{-}io\ pP)@ioT@[(x',y')]$›
unfolding *length-map*[*of* $(\lambda\ t\ .\ (t\text{-}input\ t,\ t\text{-}output\ t))$, *of* pM', *symmetric*]
length-map[*of* $(\lambda\ t\ .\ (t\text{-}input\ t,\ t\text{-}output\ t))$, *of* pP, *symmetric*]
by *auto*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *add-diff-cancel-right′ antisym-conv antisym-conv2*

append-butlast-last-id append-eq-append-conv2 butlast-conv-take drop-Nil drop-eq-Nil
le-add1 less-add-one take-add)
qed

have *p-io ?pP* = *p-io pP*
using ‹*p-io pM′* = (*p-io pP*)@*ioT*@[(*x′*,*y′*)]›
by (*metis* (*no-types*, *lifting*) *append-eq-conv-conj length-map take-map*)

have *p-io ?pT* = *ioT*
using ‹*p-io pM′* = (*p-io pP*)@*ioT*@[(*x′*,*y′*)]›
using ‹*pM′* = *?pP* @ *?pT* @ [*?t*]›
by (*metis* (*no-types*, *lifting*) *append-eq-conv-conj length-map map-append take-map*)

have *p-io* [*?t*] = [(*x′*,*y′*)]
using ‹*p-io pM′* = (*p-io pP*)@*ioT*@[(*x′*,*y′*)]›
using ‹*pM′* = *?pP* @ *?pT* @ [*?t*]›
by (*metis* (*no-types*, *lifting*) *append-is-Nil-conv last-appendR last-map last-snoc list.simps*(*8*) *list.simps*(*9*))

have *path M* (*initial M*) *?pP*
using ‹*path M* (*initial M*) *pM′*› ‹*pM′* = *?pP* @ *?pT* @ [*?t*]›
by (*meson path-prefix-take*)

have *?pP* = *pP*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*initial M*) *?pP*› ‹*path M* (*initial M*) *pP*› ‹*p-io ?pP* = *p-io pP*›]
by *assumption*
then have *path M q* (*?pT*@[*?t*])
by (*metis* ‹*FSM.initial P* = *FSM.initial M*› ‹*pM′* = *take* (*length pP*) *pM′* @ *take* (*length ioT*) (*drop* (*length pP*) *pM′*) @ [*last pM′*]› ‹*path M* (*FSM.initial M*) *pM′*› ‹*target* (*FSM.initial P*) *pP* = *q*› *path-suffix*)
then have *path M q ?pT*
and *?t* ∈ *transitions M*
and *t-source ?t* = *target q ?pT*
by *auto*

have *inputs M* ≠ {}
using *language-io*(*1*)[*OF* ‹(*p-io pP*)@*ioT*@[(*x′*,*y′*)] ∈ *L M*›, *of x′ y′*]
by *auto*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*,*P*) ∈ *prs*›
using *image-iff* **by** *fastforce*

obtain *ioT′* **where** *p-io pT* = (*ioT* @ [(*x*, *y*)]) @ *ioT′*
using ‹*ioT* @ [(*x*, *y*)] ∈ *set* (*prefixes* (*p-io pT*))›
unfolding *prefixes-set mem-Collect-eq* **by** *metis*

then have *length pT* > *length ioT*
using *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*)) *pT*]
by *auto*

obtain *pT′ d′* **where** (*pT* @ *pT′*, *d′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹*pT* ∈ *tps q*›
by *blast*

let *?p* = *pT* @ *pT′*

have *path M q ?p*
and *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *?p*)) *repetition-sets* = *Some d′*
and ⋀ *p′ p″*. *?p* = *p′* @ *p″* ⟹ *p″* ≠ [] ⟹ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p′*)) *repetition-sets* = *None*
using ‹(*pT* @ *pT′*, *d′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
m-traversal-paths-with-witness-set[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*]
by *blast+*

let *?pIO* = *take* (*length ioT*) *pT*
have *?pIO* = *take* (*length ioT*) *?p*
using ‹*length pT* > *length ioT*› **by** *auto*
then have *?p* = *?pIO*@(*drop* (*length ioT*) *?p*)
by *auto*
have (*drop* (*length ioT*) *?p*) ≠ []
using ‹*length pT* > *length ioT*› **by** *auto*

have *p-io ?pIO* = *ioT*
proof −
have *p-io ?pIO* = *take* (*length ioT*) (*p-io pT*)
by (*simp add*: *take-map*)
moreover have *take* (*length ioT*) (*p-io pT*) = *ioT*
using ‹*p-io pT* = (*ioT* @ [(*x*, *y*)]) @ *ioT′*› **by** *auto*
ultimately show *?thesis* **by** *simp*
qed
then have *p-io ?pIO* = *p-io ?pT*
using ‹*p-io ?pT* = *ioT*› **by** *simp*

have *path M q ?pIO*
using ‹*path M q ?p*› **unfolding** ‹*?pIO* = *take* (*length ioT*) *?p*›
using *path-prefix-take* **by** *metis*

have *?pT* = *?pIO*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M q ?pIO*› ‹*path M q ?pT*› ‹*p-io ?pIO* = *p-io ?pT*›]
by *simp*

show ($\exists$ pT' . $pT' \in$ *tps q* $\wedge$ *ioT* @ [(x', y')] $\in$ *set* (*prefixes* (*p-io* pT')))
proof (*cases find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) (*?pT*@[*?t*]))) *repetition-sets* = *None*)
case *True*

obtain pT' d' **where** (*?pT* @ [*?t*] @ pT', d') $\in$ *m-traversal-paths-with-witness M q repetition-sets m*
using *m-traversal-path-extension-exist*[*OF* ‹*completely-specified M*› ‹*q* $\in$ *states M*› ‹*inputs M* $\neq$ {}› *t5 t8* ‹*path M q* (*?pT*@[*?t*])› *True*]
by *auto*
then have *?pT* @ [*?t*] @ pT' $\in$ *tps q*
using *t6*[*OF* ‹*q* $\in$ *fst* ‘ *prs*›] **by** *force*

moreover have *ioT* @ [(x', y')] $\in$ *set* (*prefixes* (*p-io* (*?pT* @ [*?t*] @ pT')))
using ‹*p-io ?pIO* = *ioT*› ‹*p-io* [*?t*] = [(x',y')]›
unfolding ‹*?pT* = *?pIO*› *prefixes-set* **by** *force*

ultimately show *?thesis*
by *blast*
next
case *False*

note ‹*path M q* (*?pT* @ [*?t*])›
moreover obtain d' **where** *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) (*?pT*@[*?t*]))) *repetition-sets* = *Some* d'
using *False* **by** *blast*

moreover have $\forall$ p' p''. (*?pT* @ [*?t*]) = p' @ p'' $\wedge$ $p'' \neq$ [] $\longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) *repetition-sets* = *None*
proof −
have $\bigwedge$ p' p''. (*?pT* @ [*?t*]) = p' @ p'' $\Longrightarrow$ $p'' \neq$ [] $\Longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) *repetition-sets* = *None*
proof −
fix p' p'' **assume** (*?pT* @ [*?t*]) = p' @ p'' **and** $p'' \neq$ []
then obtain pIO' **where** *?pIO* = p' @ pIO'
unfolding ‹*?pT* = *?pIO*›
by (*metis butlast-append butlast-snoc*)
then have *?p* = p'@pIO'@(*drop* (*length ioT*) *?p*)
using ‹*?p* = *?pIO*@((*drop* (*length ioT*) *?p*))›
by (*metis append.assoc*)

have pIO' @ *drop* (*length ioT*) (pT @ pT') $\neq$ []
using ‹(*drop* (*length ioT*) *?p*) $\neq$ []› **by** *auto*

show *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) *repetition-sets* = *None*
using ‹$\bigwedge$ p' p''. *?p* = p' @ p'' $\Longrightarrow$ $p'' \neq$ [] $\Longrightarrow$ *find* (λd. *Suc* (m − *card* (*snd d*)) $\leq$ *length* (*filter* (λt. *t-target t* $\in$ *fst d*) p')) *repetition-sets* = *None*›

[*of* p' pIO'@(*drop* (*length* ioT) $?p$), *OF* ‹$?p = p'$@pIO'@(*drop* (*length* ioT) $?p$)› ‹pIO' @ *drop* (*length* ioT) (pT @ pT') $\neq$ []›]
by *assumption*
qed
then show *?thesis* **by** *blast*
qed

ultimately have (($?pT$ @ [$?t$]),d') $\in$ *m-traversal-paths-with-witness* M q *repetition-sets* m
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹$q \in$ *states* M›, *of* m]
by *auto*
then have ($?pT$ @ [$?t$]) $\in$ *tps* q
using *t6*[*OF* ‹$q \in$ *fst* ‘ *prs*›] **by** *force*
moreover have ioT @ [(x', y')] $\in$ *set* (*prefixes* (*p-io* ($?pT$ @ [$?t$])))
using ‹*p-io* $?pT = ioT$› ‹*p-io* [$?t$] = [(x',y')]›
unfolding *prefixes-set* **by** *force*

ultimately show *?thesis*
by *blast*
qed
qed
then have *p2*: ($\forall$ q P pP ioT pT x y y' . (q,P) $\in$ *prs* $\longrightarrow$
path P (*initial* P) pP $\longrightarrow$
target (*initial* P) $pP = q$ $\longrightarrow$
$pT \in$ *tps* q $\longrightarrow$
ioT @ [(x, y)] $\in$ *set* (*prefixes* (*p-io* pT)) $\longrightarrow$
(*p-io* pP)@ioT@[(x,y')] $\in$ L M' $\longrightarrow$
($\exists$ pT' . $pT' \in$ *tps* q $\wedge$ ioT @ [(x, y')] $\in$ *set* (*prefixes* (*p-io* pT'))))
by *blast*

have $\bigwedge$ q P pP pT q' A $d1$ $d2$ qT . (q,P) $\in$ *prs* $\Longrightarrow$
path P (*initial* P) pP $\Longrightarrow$
target (*initial* P) $pP = q$ $\Longrightarrow$
$pT \in$ *tps* q $\Longrightarrow$
$q' \in$ *rd-targets* (q,pT) $\Longrightarrow$
(A,$d1$,$d2$) $\in$ *separators* (*target* q pT, q') $\Longrightarrow$
$qT \in$ *io-targets* M' ((*p-io* pP)@(*p-io* pT)) (*initial* M') $\Longrightarrow$
pass-separator-ATC M' A qT $d2$
proof −
fix q P pP pT q' A $d1$ $d2$ qT
assume (q,P) $\in$ *prs*
and *path* P (*initial* P) pP
and *target* (*initial* P) $pP = q$
and $pT \in$ *tps* q
and $q' \in$ *rd-targets* (q,pT)
and (A,$d1$,$d2$) $\in$ *separators* (*target* q pT, q')

and $qT \in$ *io-targets* M' ((*p-io* pP)@(*p-io* pT)) (*initial* M')

have $q \in$ *fst* ‘ *prs*
using ‹$(q,P) \in$ *prs*› **by** *force*
have *is-preamble* P M q
using ‹$(q,P) \in$ *prs*› ‹$\bigwedge$ q P. $(q,\ P) \in$ *prs* $\Longrightarrow$ *is-preamble* P M q $\wedge$ *tps* $q \neq \{\}$› **by** *blast*
then have $q \in$ *states* M
unfolding *is-preamble-def*
by (*metis* ‹*path* P (*FSM.initial* P) pP› ‹*target* (*FSM.initial* P) $pP = q$› *path-target-is-state submachine-path*)

have *initial* P = *initial* M
using ‹*is-preamble* P M q› **unfolding** *is-preamble-def* **by** *auto*
have *path* M (*initial* M) pP
using ‹*is-preamble* P M q› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path* P (*FSM.initial* P) pP› **by** *blast*

have *is-separator* M (*target* q pT) q' A $d1$ $d2$
using *t3*[*OF* ‹$(A,d1,d2) \in$ *separators* (*target* q pT, q')›]
by *blast*

have $qT \in$ *states* M'
using ‹$qT \in$ *io-targets* M' ((*p-io* pP)@(*p-io* pT)) (*initial* M')›
io-targets-states
by (*metis* (*no-types*, *lifting*) *subsetD*)

obtain pT' d' **where** (pT @ pT', d') $\in$ *m-traversal-paths-with-witness* M q *repetition-sets* m
using *t6*[*OF* ‹$q \in$ *fst* ‘ *prs*›] ‹$pT \in$ *tps* q›
by *blast*
then have *path* M q pT
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹$q \in$ *states* M›, *of* m]
by *auto*
then have *target* q $pT \in$ *FSM.states* M
using *path-target-is-state* **by** *metis*

have $q' \in$ *FSM.states* M
using *is-separator-separated-state-is-state*[*OF* ‹*is-separator* M (*target* q pT) q' A $d1$ $d2$›] **by** *simp*

have $\neg$ *pass-separator-ATC* M' A qT $d2$ $\Longrightarrow$ $\neg$ *LS* M' $qT \subseteq$ *LS* M (*target* q pT)
using *pass-separator-ATC-fail-no-reduction*[*OF* ‹*observable* M'› ‹*observable* M› ‹$qT \in$ *states* M'›
‹*target* q $pT \in$ *FSM.states* M› ‹$q' \in$ *FSM.states* M›
‹*is-separator* M (*target* q pT) q' A $d1$ $d2$› ‹*inputs* M' = *inputs* M›]

by *assumption*

moreover have $LS\ M'\ qT \subseteq LS\ M\ (target\ q\ pT)$
proof −
have $(target\ q\ pT) = target\ (initial\ M)\ (pP@pT)$
using ‹$target\ (initial\ P)\ pP = q$› **unfolding** ‹$initial\ P = initial\ M$› **by** *auto*

have $path\ M\ (initial\ M)\ (pP@pT)$
using ‹$path\ M\ (initial\ M)\ pP$› ‹$target\ (initial\ P)\ pP = q$› ‹$path\ M\ q\ pT$› **unfolding** ‹$initial\ P = initial\ M$› **by** *auto*

then have $(target\ q\ pT) \in io\text{-}targets\ M\ (p\text{-}io\ pP\ @\ p\text{-}io\ pT)\ (FSM.initial\ M)$
unfolding *io-targets.simps* ‹$(target\ q\ pT) = target\ (initial\ M)\ (pP@pT)$›
using *map-append* **by** *blast*

show *?thesis*
using *observable-language-target*[*OF* ‹$observable\ M$› ‹$(target\ q\ pT) \in io\text{-}targets\ M\ (p\text{-}io\ pP\ @\ p\text{-}io\ pT)\ (FSM.initial\ M)$› ‹$qT \in io\text{-}targets\ M'\ ((p\text{-}io\ pP)@(p\text{-}io\ pT))\ (initial\ M')$› ‹$L\ M' \subseteq L\ M$›]
by *assumption*
qed

ultimately show $pass\text{-}separator\text{-}ATC\ M'\ A\ qT\ d2$
by *blast*
qed
then have *p3*: $(\forall\ q\ P\ pP\ pT\ .\ (q,P) \in prs \longrightarrow$
$path\ P\ (initial\ P)\ pP \longrightarrow$
$target\ (initial\ P)\ pP = q \longrightarrow$
$pT \in tps\ q \longrightarrow$
$(p\text{-}io\ pP)@(p\text{-}io\ pT) \in L\ M' \longrightarrow$
$(\forall\ q'\ A\ d1\ d2\ qT\ .\ q' \in rd\text{-}targets\ (q,pT) \longrightarrow$
$(A,d1,d2) \in separators\ (target\ q\ pT,\ q') \longrightarrow$
$qT \in io\text{-}targets\ M'\ ((p\text{-}io\ pP)@(p\text{-}io\ pT))\ (initial\ M') \longrightarrow$
$pass\text{-}separator\text{-}ATC\ M'\ A\ qT\ d2))$
by *blast*

show *?thesis*
using *p1 p2 p3*
unfolding *passes-test-suite.simps*
by *blast*
qed

41.5 Exhaustiveness of Sufficient Test Suites

This subsection shows that test suites satisfying the sufficiency criterion are exhaustive. That is, for a System Under Test with at most m states that contains an error (i.e.: is not a reduction) a test suite sufficient for m will not pass.

41.5.1 R Functions

definition $R :: ('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'b \times 'c \times 'a)\ list \Rightarrow ('a \times 'b \times 'c \times 'a)\ list \Rightarrow ('a \times 'b \times 'c \times 'a)\ list\ set$ **where**
$R\ M\ q\ q'\ pP\ p = \{pP\ @\ p' \mid p' .\ p' \neq [] \wedge target\ q\ p' = q' \wedge (\exists\ p'' .\ p = p'@p'')\}$

definition $RP :: ('a,'b,'c)\ fsm \Rightarrow 'a \Rightarrow 'a \Rightarrow ('a \times 'b \times 'c \times 'a)\ list \Rightarrow ('a \times 'b \times 'c \times 'a)\ list \Rightarrow ('a \times ('a,'b,'c)\ preamble)\ set \Rightarrow ('d,'b,'c)\ fsm \Rightarrow ('a \times 'b \times 'c \times 'a)\ list\ set$ **where**
$RP\ M\ q\ q'\ pP\ p\ PS\ M' = (if\ \exists\ P' .\ (q',P') \in PS\ then\ insert\ (SOME\ pP' .\ \exists\ P' .\ (q',P') \in PS \wedge path\ P'\ (initial\ P')\ pP' \wedge target\ (initial\ P')\ pP' = q' \wedge p\text{-}io\ pP' \in L\ M')\ (R\ M\ q\ q'\ pP\ p)\ else\ (R\ M\ q\ q'\ pP\ p))$

lemma *RP-from-R* :
assumes $\bigwedge q\ P .\ (q,P) \in PS \Longrightarrow is\text{-}preamble\ P\ M\ q$
and $\bigwedge q\ P\ io\ x\ y\ y' .\ (q,P) \in PS \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
and $completely\text{-}specified\ M'$
and $inputs\ M' = inputs\ M$
shows $(RP\ M\ q\ q'\ pP\ p\ PS\ M' = R\ M\ q\ q'\ pP\ p)$
$\vee (\exists\ P'\ pP' .\ (q',P') \in PS \wedge$
$path\ P'\ (initial\ P')\ pP' \wedge$
$target\ (initial\ P')\ pP' = q' \wedge$
$path\ M\ (initial\ M)\ pP' \wedge$
$target\ (initial\ M)\ pP' = q' \wedge$
$p\text{-}io\ pP' \in L\ M' \wedge$
$RP\ M\ q\ q'\ pP\ p\ PS\ M' =$
$insert\ pP'\ (R\ M\ q\ q'\ pP\ p))$
proof (*rule ccontr*)
assume $\neg\ (RP\ M\ q\ q'\ pP\ p\ PS\ M' = R\ M\ q\ q'\ pP\ p \vee (\exists\ P'\ pP' .\ (q',P') \in PS \wedge path\ P'\ (initial\ P')\ pP' \wedge target\ (initial\ P')\ pP' = q' \wedge path\ M\ (initial\ M)\ pP' \wedge target\ (initial\ M)\ pP' = q' \wedge p\text{-}io\ pP' \in L\ M' \wedge RP\ M\ q\ q'\ pP\ p\ PS\ M' = insert\ pP'\ (R\ M\ q\ q'\ pP\ p)))$
then have $(RP\ M\ q\ q'\ pP\ p\ PS\ M' \neq R\ M\ q\ q'\ pP\ p)$
and $\neg\ (\exists\ P'\ pP' .\ (q',P') \in PS \wedge$
$path\ P'\ (initial\ P')\ pP' \wedge$
$target\ (initial\ P')\ pP' = q' \wedge$
$path\ M\ (initial\ M)\ pP' \wedge$

target (*initial M*) *pP′* = *q′* ∧
p-io pP′ ∈ *L M′* ∧
RP M q q′ pP p PS M′ = *insert pP′* (*R M q q′ pP p*))
by *blast+*

let *?p* = *SOME pP′* . ∃ *P′* . (*q′*,*P′*) ∈ *PS* ∧ *path P′* (*initial P′*) *pP′* ∧ *target* (*initial P′*) *pP′* = *q′* ∧ *p-io pP′* ∈ *L M′*

have ∃ *P′* . (*q′*,*P′*) ∈ *PS*
using ‹(*RP M q q′ pP p PS M′* ≠ *R M q q′ pP p*)› **unfolding** *RP-def* **by** *auto*
then obtain *P′* **where** (*q′*,*P′*) ∈ *PS*
by *auto*
then have *is-preamble P′ M q′*
using *assms* **by** *blast*

obtain *pP′* **where** *path P′* (*initial P′*) *pP′* **and** *target* (*initial P′*) *pP′* = *q′* **and** *p-io pP′* ∈ *L M′*
using *preamble-pass-path*[*OF* ‹*is-preamble P′ M q′*›
assms(*2*)[*OF* ‹(*q′*,*P′*) ∈ *PS*›] *assms*(*3*,*4*)]
by *force*
then have ∃ *pP′* . ∃ *P′* . (*q′*,*P′*) ∈ *PS* ∧ *path P′* (*initial P′*) *pP′* ∧ *target* (*initial P′*) *pP′* = *q′* ∧ *p-io pP′* ∈ *L M′*
using ‹(*q′*,*P′*) ∈ *PS*› **by** *blast*
have ∃ *P′* . (*q′*,*P′*) ∈ *PS* ∧ *path P′* (*initial P′*) *?p* ∧ *target* (*initial P′*) *?p* = *q′* ∧ *p-io ?p* ∈ *L M′*
using *someI-ex*[*OF* ‹∃ *pP′* . ∃ *P′* . (*q′*,*P′*) ∈ *PS* ∧ *path P′* (*initial P′*) *pP′* ∧ *target* (*initial P′*) *pP′* = *q′* ∧ *p-io pP′* ∈ *L M′*›]
by *blast*

then obtain *P″* **where** (*q′*,*P″*) ∈ *PS* **and** *path P″* (*initial P″*) *?p* **and** *target* (*initial P″*) *?p* = *q′* **and** *p-io ?p* ∈ *L M′*
by *auto*
then have *is-preamble P″ M q′*
using *assms* **by** *blast*

have *initial P″* = *initial M*
using ‹*is-preamble P″ M q′*› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *?p*
using ‹*is-preamble P″ M q′*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P″* (*FSM.initial P″*) *?p*› **by** *blast*
have *target* (*initial M*) *?p* = *q′*
using ‹*target* (*initial P″*) *?p* = *q′*› **unfolding** ‹*initial P″* = *initial M*› **by** *assumption*

have *RP M q q′ pP p PS M′* = *insert ?p* (*R M q q′ pP p*)
using ‹∃ *P′* . (*q′*,*P′*) ∈ *PS*› **unfolding** *RP-def* **by** *auto*

then have (∃ *P′ pP′* . (*q′*,*P′*) ∈ *PS* ∧

path P′ (initial P′) pP′ ∧
target (initial P′) pP′ = q′ ∧
path M (initial M) pP′ ∧
target (initial M) pP′ = q′ ∧
p-io pP′ ∈ L M′ ∧
RP M q q′ pP p PS M′ = insert pP′ (R M q q′ pP p))
using ‹(*q′*,*P′′*) ∈ *PS*› ‹*path P′′ (initial P′′) ?p*› ‹*target (initial P′′) ?p = q′*› ‹*path M (initial M) ?p*› ‹*target (initial M) ?p = q′*› ‹*p-io ?p ∈ L M′*› **by** *blast*
then show *False*
using ‹¬ (∃ *P′ pP′* . (*q′*,*P′*) ∈ *PS* ∧ *path P′ (initial P′) pP′* ∧ *target (initial P′) pP′ = q′* ∧ *path M (initial M) pP′* ∧ *target (initial M) pP′ = q′* ∧ *p-io pP′* ∈ *L M′* ∧ *RP M q q′ pP p PS M′ = insert pP′ (R M q q′ pP p))*›
by *blast*
qed

lemma *RP-from-R-inserted* :
assumes ⋀ *q P* . (*q*,*P*) ∈ *PS* ⟹ *is-preamble P M q*
and ⋀ *q P io x y y′* . (*q*,*P*) ∈ *PS* ⟹ *io*@[(*x*,*y*)] ∈ *L P* ⟹ *io*@[(*x*,*y′*)] ∈ *L M′* ⟹ *io*@[(*x*,*y′*)] ∈ *L P*
and *completely-specified M′*
and *inputs M′ = inputs M*
and *pP′* ∈ *RP M q q′ pP p PS M′*
and *pP′* ∉ *R M q q′ pP p*
obtains *P′* **where** (*q′*,*P′*) ∈ *PS*
path P′ (initial P′) pP′
target (initial P′) pP′ = q′
path M (initial M) pP′
target (initial M) pP′ = q′
p-io pP′ ∈ *L M′*
RP M q q′ pP p PS M′ = insert pP′ (R M q q′ pP p)
proof −
have (*RP M q q′ pP p PS M′* ≠ *R M q q′ pP p*)
using *assms*(*5*,*6*) **by** *blast*

then have (∃ *P′ pP′*.
(*q′*, *P′*) ∈ *PS* ∧
path P′ (FSM.initial P′) pP′ ∧
target (FSM.initial P′) pP′ = q′ ∧
path M (FSM.initial M) pP′ ∧ *target (FSM.initial M) pP′ = q′* ∧ *p-io pP′* ∈ *L M′* ∧ *RP M q q′ pP p PS M′ = insert pP′ (R M q q′ pP p))*
using *RP-from-R*[*OF assms*(*1*−*4*), *of PS - - q q′ pP p*] **by** *force*
then obtain *P′ pP′′* **where** (*q′*, *P′*) ∈ *PS*
path P′ (FSM.initial P′) pP′′
target (FSM.initial P′) pP′′ = q′
path M (FSM.initial M) pP′′
target (FSM.initial M) pP′′ = q′
p-io pP′′ ∈ *L M′*

$RP\ M\ q\ q'\ pP\ p\ PS\ M' = insert\ pP''\ (R\ M\ q\ q'\ pP\ p)$
by *blast*

moreover have $pP'' = pP'$ **using** ‹$RP\ M\ q\ q'\ pP\ p\ PS\ M' = insert\ pP''\ (R\ M\ q\ q'\ pP\ p)$› *assms*(*5*,*6*) **by** *simp*
ultimately show *?thesis* **using** *that*[*of* P'] **unfolding** ‹$pP'' = pP'$› **by** *blast*
qed

lemma *finite-R* :
assumes *path* $M\ q\ p$
shows *finite* $(R\ M\ q\ q'\ pP\ p)$
proof −
have $\bigwedge p'\ .\ p' \in (R\ M\ q\ q'\ pP\ p) \Longrightarrow p' \in set\ (prefixes\ (pP@p))$
proof −
fix p' **assume** $p' \in (R\ M\ q\ q'\ pP\ p)$
then obtain p'' **where** $p' = pP\ @\ p''$
unfolding *R-def* **by** *blast*
then obtain p''' **where** $p = p''\ @\ p'''$
using ‹$p' \in (R\ M\ q\ q'\ pP\ p)$› **unfolding** *R-def* **by** *blast*

show $p' \in set\ (prefixes\ (pP@p))$
unfolding *prefixes-set* ‹$p' = pP\ @\ p''$› ‹$p = p''\ @\ p'''$› **by** *auto*
qed
then have $(R\ M\ q\ q'\ pP\ p) \subseteq set\ (prefixes\ (pP@p))$
by *blast*
then show *?thesis*
using *rev-finite-subset* **by** *auto*
qed

lemma *finite-RP* :
assumes *path* $M\ q\ p$
and $\bigwedge q\ P\ .\ (q,P) \in PS \Longrightarrow is\text{-}preamble\ P\ M\ q$
and $\bigwedge q\ P\ io\ x\ y\ y'\ .\ (q,P) \in PS \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
and *completely-specified* M'
and $inputs\ M' = inputs\ M$
shows *finite* $(RP\ M\ q\ q'\ pP\ p\ PS\ M')$
using *finite-R*[*OF assms*(*1*), *of* $q'\ pP$]
RP-from-R[*OF assms*(*2*,*3*,*4*,*5*), *of PS - -* $q\ q'\ pP\ p$] **by** *force*

lemma *R-component-ob* :
assumes $pR' \in R\ M\ q\ q'\ pP\ p$
obtains pR **where** $pR' = pP@pR$
using *assms* **unfolding** *R-def* **by** *blast*

lemma *R-component* :

assumes $(pP@pR) \in R\ M\ q\ q'\ pP\ p$
shows $pR = take\ (length\ pR)\ p$
and $length\ pR \leq length\ p$
and $t\text{-}target\ (p\ !\ (length\ pR - 1)) = q'$
and $pR \neq []$
proof −
let $?R = R\ M\ q\ q'\ p$

have $pR \neq []$ **and** $target\ q\ pR = q'$ **and** $\exists\ p''\ .\ p = pR@p''$
using ‹$pP@pR \in R\ M\ q\ q'\ pP\ p$› **unfolding** *R-def* **by** *blast+*
then obtain pR' **where** $p = pR@pR'$
by *blast*

then show $pR = take\ (length\ pR)\ p$ **and** $length\ pR \leq length\ p$
by *auto*

show $t\text{-}target\ (p\ !\ (length\ pR - 1)) = q'$
using ‹$pR \neq []$› ‹$target\ q\ pR = q'$› **unfolding** *target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) *Suc-diff-1* ‹$pR = take\ (length\ pR)\ p$›
append-butlast-last-id last.simps last-map length-butlast lessI list.map-disc-iff

not-gr-zero nth-append-length nth-take take-eq-Nil)

show $pR \neq []$
using ‹$pR \neq []$›
by *assumption*
qed

lemma *R-component-observable* :
assumes $pP@pR \in R\ M\ (target\ (initial\ M)\ pP)\ q'\ pP\ p$
and $observable\ M$
and $path\ M\ (initial\ M)\ pP$
and $path\ M\ (target\ (initial\ M)\ pP)\ p$
shows $io\text{-}targets\ M\ (p\text{-}io\ pP\ @\ p\text{-}io\ pR)\ (initial\ M) = \{target\ (target\ (initial\ M)\ pP)\ (take\ (length\ pR)\ p)\}$
proof −
have $pR = take\ (length\ pR)\ p$
and $length\ pR \leq length\ p$
and $t\text{-}target\ (p\ !\ (length\ pR - 1)) = q'$
using *R-component*[*OF assms*(*1*)] **by** *blast+*

let $?q = (target\ (initial\ M)\ pP)$
have $path\ M\ ?q\ (take\ (length\ pR)\ p)$
using *assms*(*4*) **by** (*simp add*: *path-prefix-take*)
have $p\text{-}io\ (take\ (length\ pR)\ p) = p\text{-}io\ pR$
using ‹$pR = take\ (length\ pR)\ p$› **by** *auto*

have $*$:*path M* (*initial M*) (*pP* @ (*take* (*length pR*) *p*))
using ‹*path M* (*initial M*) *pP*› ‹*path M ?q* (*take* (*length pR*) *p*)› **by** *auto*
have $**$:*p-io* (*pP* @ (*take* (*length pR*) *p*)) = (*p-io pP* @ *p-io pR*)
using ‹*p-io* (*take* (*length pR*) *p*) = *p-io pR*› **by** *auto*

have *target* (*initial M*) (*pP* @ (*take* (*length pR*) *p*)) = *target ?q* (*take* (*length pR*) *p*)
by *auto*
then have *target ?q* (*take* (*length pR*) *p*) ∈ *io-targets M* (*p-io pP* @ *p-io pR*) (*initial M*)
unfolding *io-targets.simps* **using** $*$ $**$
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

show *io-targets M* (*p-io pP* @ *p-io pR*) (*initial M*) = {*target ?q* (*take* (*length pR*) *p*)}
using *observable-io-targets*[*OF* ‹*observable M*› *language-state-containment*[*OF* $*$ $**$]]
by (*metis* (*no-types*) ‹*target* (*target* (*FSM.initial M*) *pP*) (*take* (*length pR*) *p*) ∈ *io-targets M* (*p-io pP* @ *p-io pR*) (*FSM.initial M*)› *singleton-iff*)
qed

lemma *R-count* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *observable M′*
and ⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*
and *path M* (*target* (*initial M*) *pP*) *p*
and *butlast io* = *p-io p* @ *ioX*
shows *card* (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) (*R M* (*target* (*initial M*) *pP*) *q′ pP p*))) = *card* (*R M* (*target* (*initial M*) *pP*) *q′ pP p*)
(**is** *card ?Tgts* = *card ?R*)
and ⋀ *pR* . *pR* ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹ ∃ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
and ⋀ *pR1 pR2* . *pR1* ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹
pR2 ∈ (*R M* (*target* (*initial M*) *pP*) *q′ pP p*) ⟹
pR1 ≠ *pR2* ⟹
io-targets M′ (*p-io pR1*) (*initial M′*) ∩ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
proof −

have *sequence-to-failure-extending-preamble-path M M′ PS pP io*
and ⋀ *p′ io′* . *sequence-to-failure-extending-preamble-path M M′ PS p′ io′* ⟹ *length io* ≤ *length io′*
using ‹*minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast*+

obtain *q P* **where** (*q*,*P*) ∈ *PS*

and *path P (initial P) pP*
and *target (initial P) pP = q*
and *((p-io pP) @ butlast io) ∈ L M*
and *((p-io pP) @ io) ∉ L M*
and *((p-io pP) @ io) ∈ L M′*

using ‹*sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *sequence-to-failure-extending-preamble-path-def*
by *blast*

have *is-preamble P M q*
using ‹*(q,P) ∈ PS*› ‹*⋀ q P. (q, P) ∈ PS ⟹ is-preamble P M q*› **by** *blast*
then have *q ∈ states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P (FSM.initial P) pP*› ‹*target (FSM.initial P) pP = q*› *path-target-is-state submachine-path*)

have *initial P = initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have *path M (initial M) pP*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P (FSM.initial P) pP*› **by** *blast*
have *target (initial M) pP = q*
using ‹*target (initial P) pP = q*› **unfolding** ‹*initial P = initial M*› **by** *assumption*

then have *path M q p*
using ‹*path M (target (initial M) pP) p*› **by** *auto*

have *io ≠ []*
using ‹*((p-io pP) @ butlast io) ∈ L M*› ‹*((p-io pP) @ io) ∉ L M*› **by** *auto*

obtain *pX* **where** *path M (target (initial M) pP) (p@pX)* **and** *p-io (p@pX) = butlast io*
proof −
have *p-io pP @ p-io p @ ioX ∈ L M*
using ‹*((p-io pP) @ butlast io) ∈ L M*›
unfolding ‹*butlast io = p-io p @ ioX*›
by *assumption*

obtain *p1 p23* **where** *path M (FSM.initial M) p1*
and *path M (target (FSM.initial M) p1) p23*
and *p-io p1 = p-io pP*
and *p-io p23 = p-io p @ ioX*
using *language-state-split*[*OF* ‹*p-io pP @ p-io p @ ioX ∈ L M*›]
by *blast*

have *p1* = *pP*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*FSM.initial M*) *p1*› ‹*path M* (*FSM.initial M*) *pP*› ‹*p-io p1* = *p-io pP*›]
by *assumption*
then have *path M* (*target* (*FSM.initial M*) *pP*) *p23*
using ‹*path M* (*target* (*FSM.initial M*) *p1*) *p23*› **by** *auto*
then have *p-io p* @ *ioX* ∈ *LS M* (*target* (*initial M*) *pP*)
using ‹*p-io p23* = *p-io p* @ *ioX*› *language-state-containment* **by** *auto*

obtain *p2 p3* **where** *path M* (*target* (*FSM.initial M*) *pP*) *p2*
and *path M* (*target* (*target* (*FSM.initial M*) *pP*) *p2*) *p3*
and *p-io p2* = *p-io p*
and *p-io p3* = *ioX*
using *language-state-split*[*OF* ‹*p-io p* @ *ioX* ∈ *LS M* (*target* (*initial M*) *pP*)›]

by *blast*

have *p2* = *p*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*target* (*FSM.initial M*) *pP*) *p2*› ‹*path M* (*target* (*FSM.initial M*) *pP*) *p*› ‹*p-io p2* = *p-io p*›]
by *assumption*
then have *path M* (*target* (*FSM.initial M*) *pP*) (*p*@*p3*)
using ‹*path M* (*target* (*FSM.initial M*) *pP*) *p*› ‹*path M* (*target* (*target* (*FSM.initial M*) *pP*) *p2*) *p3*›
by *auto*
moreover have *p-io* (*p*@*p3*) = *butlast io*
unfolding ‹*butlast io* = *p-io p* @ *ioX*› **using** ‹*p-io p3* = *ioX*›
by *auto*
ultimately show *?thesis*
using *that*[*of p3*]
by *simp*
qed

have *finite ?R*
using *finite-R*[*OF* ‹*path M* (*target* (*initial M*) *pP*) *p*›]
by *assumption*
moreover have ⋀ *pR* . *pR* ∈ *?R* ⟹ *finite* (*io-targets M′* (*p-io pR*) (*initial M′*))
using *io-targets-finite* **by** *metis*
ultimately have *finite ?Tgts*
by *blast*

obtain *pP′ p′* **where** *path M′* (*FSM.initial M′*) *pP′*

and *path M′ (target (FSM.initial M′) pP′) p′*
and *p-io pP′ = p-io pP*
and *p-io p′ = io*
using *language-state-split*[*OF* ‹((*p-io pP*) @ *io*) ∈ *L M′*›]
by *blast*

have *length p* ≤ *length* (*butlast io*)
using ‹*butlast io = p-io p @ ioX*› **by** *auto*
moreover have *length* (*butlast io*) < *length io*
using ‹*io* ≠ []› **by** *auto*
ultimately have *length p* < *length p′*
unfolding ‹*p-io p′ = io*› *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*)), *symmetric*]
by *simp*

let *?q* = (*target* (*FSM.initial M′*) *pP′*)

have ⋀ *pR* . *pP@pR* ∈ *?R* ⟹ *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
proof −
fix *pR* **assume** *pP@pR* ∈ *?R*
then have *pR = take* (*length pR*) *p* ∧ *length pR* ≤ *length p*
using *R-component*(*1*,*2*) **by** *metis*
then have *p-io pR = take* (*length pR*) (*butlast io*)
unfolding ‹*butlast io = p-io p @ ioX*›
by (*metis* (*no-types*, *lifting*) *length-map take-le take-map*)
moreover have *p-io* (*take* (*length pR*) *p′*) = *take* (*length pR*) *io*
by (*metis* (*full-types*) ‹*p-io p′ = io*› *take-map*)
moreover have *take* (*length pR*) (*butlast io*) = *take* (*length pR*) *io*
by (*meson* ‹*length* (*butlast io*) < *length io*› ‹*length p* ≤ *length* (*butlast io*)›
‹*pR = take* (*length pR*) *p* ∧ *length pR* ≤ *length p*› *dual-order.strict-trans2*
take-butlast)
ultimately have *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
by *simp*
moreover have *path M′ ?q* (*take* (*length pR*) *p′*)
using ‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *p′*›
by (*simp add*: *path-prefix-take*)
ultimately show *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*)
= *p-io pR*
by *blast*
qed

have *singleton-prop′*: ⋀ *pR* . *pP@pR* ∈ *?R* ⟹ *io-targets M′* (*p-io* (*pP@pR*))
(*initial M′*) = {*target ?q* (*take* (*length pR*) *p′*)}
proof −
fix *pR* **assume** *pP@pR* ∈ *?R*

then have *path M′ ?q* (*take* (*length pR*) *p′*) **and** *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
using ‹⋀ *pR* . *pP*@*pR* ∈ *?R* ⟹ *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*) = *p-io pR*› **by** *blast*+

have ∗:*path M′* (*initial M′*) (*pP′* @ (*take* (*length pR*) *p′*))
using ‹*path M′* (*initial M′*) *pP′*› ‹*path M′ ?q* (*take* (*length pR*) *p′*)› **by** *auto*
have ∗∗:*p-io* (*pP′* @ (*take* (*length pR*) *p′*)) = (*p-io* (*pP*@*pR*))
using ‹*p-io pP′* = *p-io pP*› ‹*p-io* (*take* (*length pR*) *p′*) = *p-io pR*› **by** *auto*

have *target* (*initial M′*) (*pP′* @ (*take* (*length pR*) *p′*)) = *target ?q* (*take* (*length pR*) *p′*)
by *auto*
then have *target ?q* (*take* (*length pR*) *p′*) ∈ *io-targets M′* (*p-io* (*pP*@*pR*)) (*initial M′*)
unfolding *io-targets.simps* **using** ∗ ∗∗
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

show *io-targets M′* (*p-io* (*pP*@*pR*)) (*initial M′*) = {*target ?q* (*take* (*length pR*) *p′*)}
using *observable-io-targets*[*OF* ‹*observable M′*› *language-state-containment*[*OF* ∗ ∗∗]]
by (*metis* (*no-types*) ‹*target* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR*) *p′*) ∈ *io-targets M′* (*p-io* (*pP*@*pR*)) (*FSM.initial M′*)› *singleton-iff*)
qed

have *singleton-prop*: ⋀ *pR* . *pR* ∈ *?R* ⟹ *io-targets M′* (*p-io pR*) (*initial M′*) = {*target ?q* (*take* (*length pR* − *length pP*) *p′*)}
proof −
fix *pR* **assume** *pR* ∈ *?R*
then obtain *pR′* **where** *pR* = *pP*@*pR′*
using *R-component-ob*[*of - M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*] **by** *blast*
have ∗∗: (*length* (*pP* @ *pR′*) − *length pP*) = *length pR′*
by *auto*

show *io-targets M′* (*p-io pR*) (*initial M′*) = {*target ?q* (*take* (*length pR* − *length pP*) *p′*)}
using *singleton-prop′*[*of pR′*] ‹*pR* ∈ *?R*› **unfolding** ‹*pR* = *pP*@*pR′*› ∗∗ **by** *blast*
qed
then show ⋀ *pR* . *pR* ∈ *?R* ⟹ ∃ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
by *blast*

have *pairwise-dist-prop′*: ⋀ *pR1 pR2* . *pP*@*pR1* ∈ *?R* ⟹ *pP*@*pR2* ∈ *?R* ⟹ *pR1* ≠ *pR2* ⟹ *io-targets M′* (*p-io* (*pP*@*pR1*)) (*initial M′*) ∩ *io-targets M′* (*p-io* (*pP*@*pR2*)) (*initial M′*) = {}
proof −

have *diff-prop*: $\bigwedge$ *pR1 pR2* . *pP@pR1* $\in$ *?R* $\Longrightarrow$ *pP@pR2* $\in$ *?R* $\Longrightarrow$ *length pR1* $<$ *length pR2* $\Longrightarrow$ *io-targets M′* (*p-io* (*pP@pR1*)) (*initial M′*) $\cap$ *io-targets M′* (*p-io* (*pP@pR2*)) (*initial M′*) = {}
proof −
fix *pR1 pR2* **assume** *pP@pR1* $\in$ *?R* **and** *pP@pR2* $\in$ *?R* **and** *length pR1* $<$ *length pR2*

let *?i* = *length pR1* − *1*
let *?j* = *length pR2* − *1*

have *pR1* = *take* (*length pR1*) *p* **and** ‹*length pR1* $\leq$ *length p*› **and** *t-target* (*p* ! *?i*) = *q′*
using *R-component*[*OF* ‹*pP@pR1* $\in$ *?R*›]
by *simp+*
have *length pR1* $\neq$ *0*
using ‹*pP@pR1* $\in$ *?R*› **unfolding** *R-def*
by *simp*
then have *?i* $<$ *?j*
using ‹*length pR1* $<$ *length pR2*›
by (*simp add*: *less-diff-conv*)

have *pR2* = *take* (*length pR2*) *p* **and** ‹*length pR2* $\leq$ *length p*› **and** *t-target* (*p* ! *?j*) = *q′*
using *R-component*[*OF* ‹*pP@pR2* $\in$ *?R*›]
by *simp+*
then have *?j* $<$ *length* (*butlast io*)
using ‹*length p* $\leq$ *length* (*butlast io*)› ‹*length pR1* $<$ *length pR2*› **by** *linarith*

have *?q* $\in$ *io-targets M′* (*p-io pP*) (*FSM.initial M′*)
unfolding ‹*p-io pP′* = *p-io pP*›[*symmetric*] *io-targets.simps*
using ‹*path M′* (*initial M′*) *pP′*› **by** *auto*

have *t-target* (*p* ! *?i*) = *t-target* (*p* ! *?j*)
using ‹*t-target* (*p* ! *?i*) = *q′*› ‹*t-target* (*p* ! *?j*) = *q′*› **by** *simp*
moreover have (*p* @ *pX*) ! *?i* = *p* ! *?i*
by (*meson* ‹*length pR1* $<$ *length pR2*› ‹*length pR2* $\leq$ *length p*› *less-imp-diff-less less-le-trans nth-append*)
moreover have (*p* @ *pX*) ! *?j* = *p* ! *?j*
by (*metis* (*no-types*) ‹*length pR1* $<$ *length pR2*› ‹*pR2* = *take* (*length pR2*) *p*› *diff-less less-imp-diff-less less-nat-zero-code less-numeral-extra*(*1*) *not-le-imp-less not-less-iff-gr-or-eq nth-append take-all*)
ultimately have *t-target* (*p′* ! *?i*) $\neq$ *t-target* (*p′* ! *?j*)
using *minimal-sequence-to-failure-extending-preamble-no-repetitions-along-path*[*OF assms*(*1*,*2*) ‹*path M* (*target* (*initial M*) *pP*) (*p@pX*)› ‹*p-io* (*p* @ *pX*) = *butlast io*› ‹*?q* $\in$ *io-targets M′* (*p-io pP*) (*FSM.initial M′*)› ‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *p′*› ‹*p-io p′* = *io*› ‹*?i* $<$ *?j*› ‹*?j* $<$ *length* (*butlast io*)› *assms*(*4*)]

by *auto*

have *t1*: *io-targets M′* (*p-io* (*pP@pR1*)) (*initial M′*) = {*t-target* (*p′* ! *?i*)}
proof −
have (*p′* ! *?i*) = *last* (*take* (*length pR1*) *p′*)
using ‹*length pR1* ≤ *length p*› ‹*length p* < *length p′*›
by (*metis Suc-diff-1* ‹*length pR1* ≠ *0*› *dual-order.strict-trans2 length-0-conv length-greater-0-conv less-imp-diff-less take-last-index*)
then have ∗: *target* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR1*) *p′*) = *t-target* (*p′* ! *?i*)
unfolding *target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) ‹*length p* < *length p′*› ‹*length pR1* ≠ *0*› *gr-implies-not-zero last.simps last-map length-0-conv map-is-Nil-conv take-eq-Nil*)
have ∗∗: (*length* (*pP* @ *pR1*) − *length pP*) = *length pR1*
by *auto*
show *?thesis*
using *singleton-prop*[*OF* ‹*pP@pR1* ∈ *?R*›]
unfolding ∗ ∗∗ **by** *assumption*
qed

have *t2*: *io-targets M′* (*p-io* (*pP@pR2*)) (*initial M′*) = {*t-target* (*p′* ! *?j*)}
proof −
have (*p′* ! *?j*) = *last* (*take* (*length pR2*) *p′*)
using ‹*length pR2* ≤ *length p*› ‹*length p* < *length p′*›
by (*metis Suc-diff-1* ‹*length pR1* − *1* < *length pR2* − *1*› *le-less-trans less-imp-diff-less*
linorder-neqE-nat not-less-zero take-last-index zero-less-diff)
then have ∗: *target* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR2*) *p′*) = *t-target* (*p′* ! *?j*)
unfolding *target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) *Nil-is-map-conv* ‹*length p* < *length p′*› ‹*length pR1* < *length pR2*›
last.simps last-map list.size(*3*) *not-less-zero take-eq-Nil*)
have ∗∗: (*length* (*pP* @ *pR2*) − *length pP*) = *length pR2*
by *auto*
show *?thesis*
using *singleton-prop′*[*OF* ‹*pP@pR2* ∈ *?R*›]
unfolding ∗ ∗∗ **by** *assumption*
qed

show *io-targets M′* (*p-io* (*pP@pR1*)) (*initial M′*) ∩ *io-targets M′* (*p-io* (*pP@pR2*)) (*initial M′*) = {}
using ‹*t-target* (*p′* ! *?i*) ≠ *t-target* (*p′* ! *?j*)›
unfolding *t1 t2* **by** *simp*
qed

fix *pR1 pR2* **assume** *pP@pR1* ∈ *?R* **and** *pP@pR2* ∈ *?R* **and** *pR1* ≠ *pR2*
then have *length pR1* ≠ *length pR2*

unfolding *R-def*
by *auto*

then consider (*a*) *length pR1* < *length pR2* | (*b*) *length pR2* < *length pR1*
using *nat-neq-iff* **by** *blast*
then show *io-targets M′* (*p-io* (*pP@pR1*)) (*initial M′*) ∩ *io-targets M′* (*p-io* (*pP@pR2*)) (*initial M′*) = {}
proof *cases*
case *a*
show *?thesis* **using** *diff-prop*[*OF* ‹*pP@pR1* ∈ *?R*› ‹*pP@pR2* ∈ *?R*› *a*] **by** *blast*
next
case *b*
show *?thesis* **using** *diff-prop*[*OF* ‹*pP@pR2* ∈ *?R*› ‹*pP@pR1* ∈ *?R*› *b*] **by** *blast*
qed
qed

then show *pairwise-dist-prop*: ⋀ *pR1 pR2* . *pR1* ∈ *?R* ⟹ *pR2* ∈ *?R* ⟹ *pR1* ≠ *pR2* ⟹ *io-targets M′* (*p-io pR1*) (*initial M′*) ∩ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
using *R-component-ob*
by (*metis* (*no-types*, *lifting*))

let *?f* = (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*))

have *p1*: (⋀*S1 S2*. *S1* ∈ *?R* ⟹ *S2* ∈ *?R* ⟹ *S1* = *S2* ∨ *?f S1* ∩ *?f S2* = {})
using *pairwise-dist-prop* **by** *blast*
have *p2*: (⋀*S*. *S* ∈ *R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p* ⟹ *io-targets M′* (*p-io S*) (*FSM.initial M′*) ≠ {})
using *singleton-prop* **by** *blast*
have *c1*: *card* (*R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*) = *card* ((λ*S*. *io-targets M′* (*p-io S*) (*FSM.initial M′*)) ‘ *R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*)
using *card-union-of-distinct*[*of ?R*, *OF p1* ‹*finite ?R*› *p2*] **by** *force*

have *p3*: (⋀*S*. *S* ∈ (λ*S*. *io-targets M′* (*p-io S*) (*FSM.initial M′*)) ‘ *R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p* ⟹ ∃ *t*. *S* = {*t*})
using *singleton-prop* **by** *blast*
have *c2*:*card* ((λ*S*. *io-targets M′* (*p-io S*) (*FSM.initial M′*)) ‘ *R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*) = *card* (⋃*S*∈*R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*. *io-targets M′* (*p-io S*) (*FSM.initial M′*))
using *card-union-of-singletons*[*of* ((λ*S*. *io-targets M′* (*p-io S*) (*FSM.initial M′*)) ‘ *R M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*), *OF p3*] **by** *force*

show *card ?Tgts* = *card ?R*

```
    unfolding c1 c2 by blast
qed


lemma R-update :
  R M q q' pP (p@[t]) = (if (target q (p@[t]) = q')
                         then insert (pP@p@[t]) (R M q q' pP p)
                         else (R M q q' pP p))
  (is ?R1 = ?R2)
proof (cases (target q (p@[t]) = q'))
  case True
  then have *: ?R2 = insert (pP@p@[t]) (R M q q' pP p)
    by auto

  have ⋀ p' . p' ∈ R M q q' pP (p@[t]) ⟹ p' ∈ insert (pP@p@[t]) (R M q q' pP
p)
  proof -
    fix p' assume p' ∈ R M q q' pP (p@[t])

    obtain p'' where p' = pP @ p''
      using R-component-ob[OF ‹p' ∈ R M q q' pP (p@[t])›] by blast

    obtain p''' where p'' ≠ [] and target q p'' = q' and p @ [t] = p'' @ p'''
      using ‹p' ∈ R M q q' pP (p@[t])› unfolding R-def ‹p' = pP @ p''›
      by auto

    show p' ∈ insert (pP@p@[t]) (R M q q' pP p)
    proof (cases p''' rule: rev-cases)
      case Nil
      then have p' = pP@(p@[t]) using ‹p' = pP @ p''› ‹p @ [t] = p'' @ p'''› by
auto
      then show ?thesis by blast
    next
      case (snoc p'''' t')
      then have p = p'' @ p'''' using ‹p @ [t] = p'' @ p'''› by auto
      then show ?thesis
        unfolding R-def using ‹p'' ≠ []› ‹target q p'' = q'›
        by (simp add: ‹p' = pP @ p''›)
    qed
  qed
  moreover have ⋀ p' . p' ∈ insert (pP@p@[t]) (R M q q' pP p) ⟹ p' ∈ R M
q q' pP (p@[t])
  proof -
    fix p' assume p' ∈ insert (pP@p@[t]) (R M q q' pP p)
    then consider (a) p' = (pP@p@[t]) | (b) p' ∈ (R M q q' pP p) by blast
    then show p' ∈ R M q q' pP (p@[t]) proof cases
      case a
      then show ?thesis using True unfolding R-def
        by simp
```

```
    next
      case b
      then show ?thesis unfolding R-def
        using append.assoc by blast
    qed
  qed
  ultimately show ?thesis
    unfolding ∗ by blast
next
  case False
  then have ∗: ?R2 = (R M q q′ pP p)
    by auto

  have ⋀ p′ . p′ ∈ R M q q′ pP (p@[t]) ⟹ p′ ∈ (R M q q′ pP p)
  proof −
    fix p′ assume p′ ∈ R M q q′ pP (p@[t])

    obtain p′′ where p′ = pP @ p′′
      using R-component-ob[OF ‹p′ ∈ R M q q′ pP (p@[t])›] by blast

    obtain p′′′ where p′′ ≠ [] and target q p′′ = q′ and p @ [t] = p′′ @ p′′′
      using ‹p′ ∈ R M q q′ pP (p@[t])› unfolding R-def ‹p′ = pP @ p′′› by blast

    show p′ ∈ (R M q q′ pP p)
    proof (cases p′′′ rule: rev-cases)
      case Nil
      then have p′ = pP@(p@[t]) using ‹p′ = pP @ p′′› ‹p @ [t] = p′′ @ p′′′› by
auto
      then show ?thesis
        using False ‹p @ [t] = p′′ @ p′′′› ‹target q p′′ = q′› local.Nil by auto
    next
      case (snoc p′′′′ t′)
      then have p = p′′ @ p′′′′ using ‹p @ [t] = p′′ @ p′′′› by auto
      then show ?thesis
        unfolding R-def using ‹p′′ ≠ []› ‹target q p′′ = q′›
        by (simp add: ‹p′ = pP @ p′′›)
    qed
  qed
  moreover have ⋀ p′ . p′ ∈ (R M q q′ pP p) ⟹ p′ ∈ R M q q′ pP (p@[t])
  proof −
    fix p′ assume p′ ∈ (R M q q′ pP p)
    then show p′ ∈ R M q q′ pP (p@[t]) unfolding R-def
      using append.assoc by blast
  qed
  ultimately show ?thesis
    unfolding ∗ by blast
qed
```

lemma *R-union-card-is-suffix-length* :
  assumes *path M* (*initial M*) *pP*
  and *path M* (*target* (*initial M*) *pP*) *p*
shows ($\sum$ *q* ∈ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*)) = *length p*
using *assms*(*2*) **proof** (*induction p rule*: *rev-induct*)
  case *Nil*
  have ⋀ *q′* . *R M* (*target* (*initial M*) *pP*) *q′ pP* [] = {}
    unfolding *R-def* **by** *auto*
  then show *?case*
    by *simp*
next
  case (*snoc t p*)
  then have *path M* (*target* (*initial M*) *pP*) *p*
    by *auto*

  let *?q* = (*target* (*initial M*) *pP*)
  let *?q′* = *target ?q* (*p* @ [*t*])

  have ⋀ *q* . *q* ≠ *?q′* ⟹ *R M ?q q pP* (*p*@[*t*]) = *R M ?q q pP p*
    using *R-update*[*of M ?q - pP p t*] **by** *force*
  then have ∗: ($\sum$ *q* ∈ *states M* − {*?q′*} . *card* (*R M* (*target* (*initial M*) *pP*) *q pP* (*p*@[*t*])))
                = ($\sum$ *q* ∈ *states M* − {*?q′*} . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*))
    by *force*

  have *R M ?q ?q′ pP* (*p*@[*t*]) = *insert* (*pP*@*p*@[*t*]) (*R M ?q ?q′ pP p*)
    using *R-update*[*of M ?q ?q′ pP p t*] **by** *force*
  moreover have (*pP*@*p*@[*t*]) ∉ (*R M ?q ?q′ pP p*)
    unfolding *R-def* **by** *simp*
  ultimately have ∗∗: *card* (*R M* (*target* (*initial M*) *pP*) *?q′ pP* (*p*@[*t*])) = *Suc* (*card* (*R M* (*target* (*initial M*) *pP*) *?q′ pP p*))
    using *finite-R*[*OF* ‹*path M* (*target* (*initial M*) *pP*) (*p*@[*t*])›] *finite-R*[*OF* ‹*path M* (*target* (*initial M*) *pP*) *p*›]
    by *simp*

  have *?q′* ∈ *states M*
    using *path-target-is-state*[*OF* ‹*path M* (*target* (*FSM.initial M*) *pP*) (*p* @ [*t*])›] **by** *assumption*
  then have ∗∗∗: ($\sum$ *q* ∈ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP* (*p*@[*t*])))
                = ($\sum$ *q* ∈ *states M* − {*?q′*} . *card* (*R M* (*target* (*initial M*) *pP*) *q pP* (*p*@[*t*]))) + (*card* (*R M* (*target* (*initial M*) *pP*) *?q′ pP* (*p*@[*t*])))
      and ∗∗∗∗: ($\sum$ *q* ∈ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*))
                = ($\sum$ *q* ∈ *states M* − {*?q′*} . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*)) + (*card* (*R M* (*target* (*initial M*) *pP*) *?q′ pP p*))

by (*metis* (*no-types*, *lifting*) *Diff-insert-absorb add.commute finite-Diff fsm-states-finite mk-disjoint-insert sum.insert*)+

have ($\sum$ $q \in$ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP* (*p*@[*t*]))) = *Suc* ($\sum$ $q \in$ *states M* . *card* (*R M* (*target* (*initial M*) *pP*) *q pP p*))
unfolding **** *** ** * **by** *simp*

then show *?case*
unfolding *snoc.IH*[*OF* ‹*path M* (*target* (*initial M*) *pP*) *p*›] **by** *auto*
qed

lemma *RP-count* :
assumes *minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*
and *observable M*
and *observable M′*
and $\bigwedge$ *q P*. (*q*, *P*) $\in$ *PS* $\Longrightarrow$ *is-preamble P M q*
and *path M* (*target* (*initial M*) *pP*) *p*
and *butlast io* = *p-io p* @ *ioX*
and $\bigwedge$ *q P io x y y′* . (*q*,*P*) $\in$ *PS* $\Longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\Longrightarrow$ *io*@[(*x*,*y′*)] $\in$ *L M′* $\Longrightarrow$ *io*@[(*x*,*y′*)] $\in$ *L P*
and *completely-specified M′*
and *inputs M′* = *inputs M*
shows *card* ($\bigcup$ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*)))
= *card* (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*)
(**is** *card ?Tgts* = *card ?RP*)
and $\bigwedge$ *pR* . *pR* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\Longrightarrow$ $\exists$ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
and $\bigwedge$ *pR1 pR2* . *pR1* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\Longrightarrow$ *pR2* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\Longrightarrow$ *pR1* $\neq$ *pR2* $\Longrightarrow$ *io-targets M′* (*p-io pR1*) (*initial M′*) $\cap$ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
proof −
let *?P1* = *card* ($\bigcup$ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*))) = *card* (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*)
let *?P2* = $\forall$ *pR* . *pR* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\longrightarrow$ ($\exists$ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*})
let *?P3* = $\forall$ *pR1 pR2* . *pR1* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\longrightarrow$ *pR2* $\in$ (*RP M* (*target* (*initial M*) *pP*) *q′ pP p PS M′*) $\longrightarrow$ *pR1* $\neq$ *pR2* $\longrightarrow$ *io-targets M′* (*p-io pR1*) (*initial M′*) $\cap$ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
let *?combined-goals* = *?P1* $\wedge$ *?P2* $\wedge$ *?P3*

let *?q* = (*target* (*initial M*) *pP*)
let *?R* = *R M ?q q′ pP p*

consider (*a*) (*?RP* = *?R*) |
(*b*) ($\exists$ *P′ pP′* . (*q′*,*P′*) $\in$ *PS* $\wedge$

path P′ (initial P′) pP′ ∧
target (initial P′) pP′ = q′ ∧
path M (initial M) pP′ ∧
target (initial M) pP′ = q′ ∧
p-io pP′ ∈ L M′ ∧
?RP = insert pP′ ?R)
using *RP-from-R*[*OF assms*(*4*,*7*,*8*,*9*), *of PS - - ?q q′ pP p*] **by** *force*

then have *?combined-goals* **proof** *cases*
case *a*
show *?thesis* **unfolding** *a* **using** *R-count*[*OF assms*(*1*−*6*)] **by** *blast*
next
case *b*

have *sequence-to-failure-extending-preamble-path M M′ PS pP io*
and ⋀ *p′ io′ . sequence-to-failure-extending-preamble-path M M′ PS p′ io′* ⟹ *length io* ≤ *length io′*
using ‹*minimal-sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast*+

obtain *q P* **where** (*q*,*P*) ∈ *PS*
and *path P* (*initial P*) *pP*
and *target* (*initial P*) *pP* = *q*
and ((*p-io pP*) @ *butlast io*) ∈ *L M*
and ((*p-io pP*) @ *io*) ∉ *L M*
and ((*p-io pP*) @ *io*) ∈ *L M′*

using ‹*sequence-to-failure-extending-preamble-path M M′ PS pP io*›
unfolding *sequence-to-failure-extending-preamble-path-def*
by *blast*

have *is-preamble P M q*
using ‹(*q*,*P*) ∈ *PS*› ‹⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*› **by** *blast*
then have *q* ∈ *states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› *path-target-is-state submachine-path*)

have *initial P* = *initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *pP*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P* (*FSM.initial P*) *pP*› **by** *blast*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*› **unfolding** ‹*initial P* = *initial M*› **by**

assumption

then have *path M q p*
using ‹*path M* (*target* (*initial M*) *pP*) *p*› **by** *auto*

have *io* ≠ []
using ‹((*p-io pP*) @ *butlast io*) ∈ *L M*› ‹((*p-io pP*) @ *io*) ∉ *L M*› **by** *auto*

have *finite ?RP*
using *finite-RP*[*OF* ‹*path M* (*target* (*initial M*) *pP*) *p*› *assms*(*4*,*7*,*8*,*9*)] **by** *force*
moreover have ⋀ *pR* . *pR* ∈ *?RP* ⟹ *finite* (*io-targets M′* (*p-io pR*) (*initial M′*))
using *io-targets-finite* **by** *metis*
ultimately have *finite ?Tgts*
by *blast*

obtain *pP′ p′* **where** *path M′* (*FSM.initial M′*) *pP′*
and *path M′* (*target* (*FSM.initial M′*) *pP′*) *p′*
and *p-io pP′* = *p-io pP*
and *p-io p′* = *io*
using *language-state-split*[*OF* ‹((*p-io pP*) @ *io*) ∈ *L M′*›]
by *blast*

have *length p* ≤ *length* (*butlast io*)
using ‹*butlast io* = *p-io p* @ *ioX*› **by** *auto*
moreover have *length* (*butlast io*) < *length io*
using ‹*io* ≠ []› **by** *auto*
ultimately have *length p* < *length p′*
unfolding ‹*p-io p′* = *io*› *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*)), *symmetric*] **by** *simp*

let *?q* = (*target* (*FSM.initial M′*) *pP′*)

have ⋀ *pR* . *pP*@*pR* ∈ *?R* ⟹ *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
proof −
fix *pR* **assume** *pP*@*pR* ∈ *?R*
then have *pR* = *take* (*length pR*) *p* ∧ *length pR* ≤ *length p*
using *R-component*(*1*,*2*) **by** *metis*
then have *p-io pR* = *take* (*length pR*) (*butlast io*)
by (*metis* (*no-types*, *lifting*) *assms*(*6*) *length-map take-le take-map*)
moreover have *p-io* (*take* (*length pR*) *p′*) = *take* (*length pR*) *io*

by (*metis* (*full-types*) ‹*p-io p′ = io*› *take-map*)
moreover have *take* (*length pR*) (*butlast io*) = *take* (*length pR*) *io*
using ‹*length p* ≤ *length* (*butlast io*)› ‹*pR* = *take* (*length pR*) *p* ∧ *length pR* ≤ *length p*›
butlast-take-le dual-order.trans
by *blast*
ultimately have *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
by *simp*
moreover have *path M′ ?q* (*take* (*length pR*) *p′*)
using ‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *p′*›
by (*simp add*: *path-prefix-take*)
ultimately show *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
by *blast*
qed

have *singleton-prop′-R*: ⋀ *pR* . *pP@pR* ∈ *?R* ⟹ *io-targets M′* (*p-io* (*pP@pR*)) (*initial M′*) = {*target ?q* (*take* (*length pR*) *p′*)}
proof −
fix *pR* **assume** *pP@pR* ∈ *?R*
then have *path M′ ?q* (*take* (*length pR*) *p′*) **and** *p-io* (*take* (*length pR*) *p′*) = *p-io pR*
using ‹⋀ *pR* . *pP@pR* ∈ *?R* ⟹ *path M′ ?q* (*take* (*length pR*) *p′*) ∧ *p-io* (*take* (*length pR*) *p′*) = *p-io pR*› **by** *blast*+

have ∗:*path M′* (*initial M′*) (*pP′* @ (*take* (*length pR*) *p′*))
using ‹*path M′* (*initial M′*) *pP′*› ‹*path M′ ?q* (*take* (*length pR*) *p′*)› **by** *auto*
have ∗∗:*p-io* (*pP′* @ (*take* (*length pR*) *p′*)) = (*p-io* (*pP@pR*))
using ‹*p-io pP′* = *p-io pP*› ‹*p-io* (*take* (*length pR*) *p′*) = *p-io pR*› **by** *auto*

have *target* (*initial M′*) (*pP′* @ (*take* (*length pR*) *p′*)) = *target ?q* (*take* (*length pR*) *p′*)
by *auto*
then have *target ?q* (*take* (*length pR*) *p′*) ∈ *io-targets M′* (*p-io* (*pP@pR*)) (*initial M′*)
unfolding *io-targets.simps* **using** ∗ ∗∗
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)

show *io-targets M′* (*p-io* (*pP@pR*)) (*initial M′*) = {*target ?q* (*take* (*length pR*) *p′*)}
using *observable-io-targets*[*OF* ‹*observable M′*› *language-state-containment*[*OF* ∗ ∗∗]]
by (*metis* (*no-types*) ‹*target* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR*) *p′*) ∈ *io-targets M′* (*p-io* (*pP@pR*)) (*FSM.initial M′*)› *singleton-iff*)
qed

have *singleton-prop-R*: $\bigwedge$ *pR* . *pR* $\in$ *?R* $\Longrightarrow$ *io-targets M′* (*p-io pR*) (*initial M′*) = {*target ?q* (*take* (*length pR* − *length pP*) *p′*)}
proof −
fix *pR* **assume** *pR* $\in$ *?R*
then obtain *pR′* **where** *pR* = *pP*@*pR′*
using *R-component-ob*[*of - M* (*target* (*FSM.initial M*) *pP*) *q′ pP p*] **by** *blast*
have ∗∗: (*length* (*pP* @ *pR′*) − *length pP*) = *length pR′*
by *auto*

show *io-targets M′* (*p-io pR*) (*initial M′*) = {*target ?q* (*take* (*length pR* − *length pP*) *p′*)}
using *singleton-prop′-R*[*of pR′*] ‹*pR* $\in$ *?R*› **unfolding** ‹*pR* = *pP*@*pR′*› ∗∗ **by** *blast*
qed

from *b* **obtain** *P′ pP″* **where** (*q′*,*P′*) $\in$ *PS*
and *path P′* (*initial P′*) *pP″*
and *target* (*initial P′*) *pP″* = *q′*
and *path M* (*initial M*) *pP″*
and *target* (*initial M*) *pP″* = *q′*
and *p-io pP″* $\in$ *L M′*
and *?RP* = *insert pP″ ?R*
by *blast*
have *initial P′* = *initial M*
using *assms*(*4*)[*OF* ‹(*q′*,*P′*) $\in$ *PS*›] **unfolding** *is-preamble-def* **by** *auto*

have $\bigwedge$ *pR* . *pR* $\in$ *?RP* $\Longrightarrow$ *pR* $\in$ *?R* $\vee$ *pR* = *pP″*
using ‹*?RP* = *insert pP″ ?R*› **by** *blast*
then have *rp-cases*[*consumes 1*, *case-names in-R inserted*]: $\bigwedge$ *pR P* . (*pR* $\in$ *?RP*) $\Longrightarrow$ (*pR* $\in$ *?R* $\Longrightarrow$ *P*) $\Longrightarrow$ (*pR* = *pP″* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by *force*

have *singleton-prop-RP*: $\bigwedge$ *pR* . *pR* $\in$ *?RP* $\Longrightarrow$ $\exists$ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
proof −
fix *pR* **assume** *pR* $\in$ *?RP*
then show $\exists$ *q* . *io-targets M′* (*p-io pR*) (*initial M′*) = {*q*}
proof (*cases rule*: *rp-cases*)
case *in-R*
then show *?thesis* **using** *singleton-prop-R* **by** *blast*
next
case *inserted*

show *?thesis*
using *observable-io-targets*[*OF* ‹*observable M'*› ‹*p-io pP''* ∈ *L M'*›]
unfolding *inserted*
by *meson*
qed
qed
then have *?P2* **by** *blast*

have *pairwise-dist-prop-RP*: ⋀ *pR1 pR2* . *pR1* ∈ *?RP* ⟹ *pR2* ∈ *?RP* ⟹ *pR1* ≠ *pR2* ⟹ *io-targets M'* (*p-io pR1*) (*initial M'*) ∩ *io-targets M'* (*p-io pR2*) (*initial M'*) = {}
proof −

have *pairwise-dist-prop-R*: ⋀ *pR1 pR2* . *pR1* ∈ *?R* ⟹ *pR2* ∈ *?R* ⟹ *pR1* ≠ *pR2* ⟹ *io-targets M'* (*p-io pR1*) (*initial M'*) ∩ *io-targets M'* (*p-io pR2*) (*initial M'*) = {}
using *R-count*(*3*)[*OF assms*(*1*−*6*)] **by** *force*

have *pairwise-dist-prop-PS*: ⋀ *pR1* . *pR1* ∈ *?RP* ⟹ *pR1* ≠ *pP''* ⟹ *io-targets M'* (*p-io pR1*) (*initial M'*) ∩ *io-targets M'* (*p-io pP''*) (*initial M'*) = {}
proof −
fix *pR1* **assume** *pR1* ∈ *?RP* **and** *pR1* ≠ *pP''*
then have *pR1* ∈ *?R*
using ‹⋀ *pR* . *pR* ∈ *?RP* ⟹ *pR* ∈ *?R* ∨ *pR* = *pP''*› **by** *blast*

obtain *pR'* **where** *pR1* = *pP*@*pR'*
using *R-component-ob*[*OF* ‹*pR1* ∈ *?R*›] **by** *blast*
then have *pP*@*pR'* ∈ *?R*
using ‹*pR1* ∈ *?R*› **by** *blast*

have *pR'* = *take* (*length pR'*) *p*
and *length pR'* ≤ *length p*
and *t-target* (*p* ! (*length pR'* − *1*)) = *q'*
and *pR'* ≠ []
using *R-component*[*OF* ‹*pP*@*pR'* ∈ *?R*›] **by** *blast*+

let *?i* = (*length pR'*) − *1*
have *?i* < *length p*
using ‹*length pR'* ≤ *length p*› ‹*pR'* ≠ []›
using *diff-less dual-order.strict-trans1 less-numeral-extra*(*1*) **by** *blast*
then have *?i* < *length* (*butlast io*)
using ‹*length p* ≤ *length* (*butlast io*)› *less-le-trans* **by** *blast*

have *io-targets M'* (*p-io pR1*) (*initial M'*) = {*t-target* (*p'* ! *?i*)}
proof −

have $(p' ! ?i) = last\ (take\ (length\ pR')\ p')$
using ‹$length\ pR' \leq length\ p$› ‹$length\ p < length\ p'$›
by (*metis Suc-diff-1* ‹$pR' \neq []$› *dual-order.strict-trans2 length-greater-0-conv less-imp-diff-less take-last-index*)
then have $*$: $target\ ?q\ (take\ (length\ pR')\ p') = t\text{-}target\ (p' ! ?i)$
unfolding *target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) ‹$length\ p < length\ p'$› ‹$pR' \neq []$› *gr-implies-not-zero last.simps*
last-map length-0-conv map-is-Nil-conv take-eq-Nil)
moreover have $io\text{-}targets\ M'\ (p\text{-}io\ pR1)\ (initial\ M') = \{target\ ?q\ (take\ (length\ pR')\ p')\}$
using *singleton-prop'-R* ‹$pR1 \in ?R$› **unfolding** ‹$pR1 = pP@pR'$› **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

have $t\text{-}target\ (p'\ !\ (length\ pR' - 1)) \notin io\text{-}targets\ M'\ (p\text{-}io\ pP'')\ (FSM.initial\ M')$
proof −

obtain pX **where** $path\ M\ (target\ (initial\ M)\ pP)\ (p@pX)$ **and** $p\text{-}io\ (p@pX) = butlast\ io$
proof −
have $p\text{-}io\ pP\ @\ p\text{-}io\ p\ @\ ioX \in L\ M$
using ‹$((p\text{-}io\ pP)\ @\ butlast\ io) \in L\ M$›
unfolding ‹$butlast\ io = p\text{-}io\ p\ @\ ioX$›
by *assumption*

obtain $p1\ p23$ **where** $path\ M\ (FSM.initial\ M)\ p1$ **and** $path\ M\ (target\ (FSM.initial\ M)\ p1)\ p23$
and $p\text{-}io\ p1 = p\text{-}io\ pP$ **and** $p\text{-}io\ p23 = p\text{-}io\ p\ @\ ioX$
using *language-state-split*[*OF* ‹$p\text{-}io\ pP\ @\ p\text{-}io\ p\ @\ ioX \in L\ M$›] **by** *blast*

have $p1 = pP$
using *observable-path-unique*[*OF* ‹$observable\ M$› ‹$path\ M\ (FSM.initial\ M)\ p1$› ‹$path\ M\ (FSM.initial\ M)\ pP$› ‹$p\text{-}io\ p1 = p\text{-}io\ pP$›]
by *assumption*
then have $path\ M\ (target\ (FSM.initial\ M)\ pP)\ p23$
using ‹$path\ M\ (target\ (FSM.initial\ M)\ p1)\ p23$› **by** *auto*
then have $p\text{-}io\ p\ @\ ioX \in LS\ M\ (target\ (initial\ M)\ pP)$
using ‹$p\text{-}io\ p23 = p\text{-}io\ p\ @\ ioX$› *language-state-containment* **by** *auto*

obtain $p2\ p3$ **where** $path\ M\ (target\ (FSM.initial\ M)\ pP)\ p2$
and $path\ M\ (target\ (target\ (FSM.initial\ M)\ pP)\ p2)\ p3$
and $p\text{-}io\ p2 = p\text{-}io\ p$
and $p\text{-}io\ p3 = ioX$
using *language-state-split*[*OF* ‹$p\text{-}io\ p\ @\ ioX \in LS\ M\ (target\ (initial$

M) *pP*)›]
by *blast*

have *p2* = *p*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*target* (*FSM.initial M*) *pP*) *p2*› ‹*path M* (*target* (*FSM.initial M*) *pP*) *p*› ‹*p-io p2* = *p-io p*›]
by *assumption*
then have *path M* (*target* (*FSM.initial M*) *pP*) (*p@p3*)
using ‹*path M* (*target* (*FSM.initial M*) *pP*) *p*› ‹*path M* (*target* (*target* (*FSM.initial M*) *pP*) *p2*) *p3*›
by *auto*
moreover have *p-io* (*p@p3*) = *butlast io*
unfolding ‹*butlast io* = *p-io p* @ *ioX*›
using ‹*p-io p3* = *ioX*›
by *auto*
ultimately show *?thesis*
using *that*[*of p3*]
by *simp*
qed

have *target* (*FSM.initial M′*) *pP′* ∈ *io-targets M′* (*p-io pP*) (*FSM.initial M′*)
using ‹*p-io pP′* = *p-io pP*› ‹*path M′* (*FSM.initial M′*) *pP′*› *observable-path-io-target* **by** *auto*

have (*t-target* (*p* ! (*length pR′* − *1*)), *P′*) ∈ *PS*
using ‹(*q′*,*P′*) ∈ *PS*› **unfolding** ‹*t-target* (*p* ! (*length pR′* − *1*)) = *q′*› **by** *assumption*
then have (*t-target* ((*p* @ *pX*) ! *?i*), *P′*) ∈ *PS*
by (*metis* ‹*length pR′* − *1* < *length p*› *nth-append*)

have *target* (*FSM.initial P′*) *pP″* = *t-target* (*p* ! (*length pR′* − *1*))
unfolding ‹*target* (*initial M*) *pP″* = *q′*› ‹*t-target* (*p* ! (*length pR′* − *1*)) = *q′*› ‹*initial P′* = *initial M*› **by** *simp*
then have *target* (*FSM.initial P′*) *pP″* = *t-target* ((*p* @ *pX*) ! *?i*)
by (*metis* ‹*length pR′* − *1* < *length p*› *nth-append*)

show *?thesis*
using *minimal-sequence-to-failure-extending-preamble-no-repetitions-with-other-preambles*
[*OF assms*(*1*,*2*) ‹*path M* (*target* (*initial M*) *pP*) (*p@pX*)› ‹*p-io* (*p@pX*) = *butlast io*›
‹*target* (*FSM.initial M′*) *pP′* ∈ *io-targets M′* (*p-io pP*) (*FSM.initial M′*)›
‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *p′*› ‹*p-io p′* = *io*› *assms*(*4*)

‹?i < length (butlast io)› ‹(t-target ((p @ pX) ! ?i), P′) ∈ PS›
‹path P′ (initial P′) pP″› ‹target (FSM.initial P′) pP″ =
t-target ((p @ pX) ! ?i)›]
by *blast*
qed
then show *io-targets M′ (p-io pR1) (initial M′) ∩ io-targets M′ (p-io pP″)*
(initial M′) = {}
unfolding ‹*io-targets M′ (p-io pR1) (initial M′) = {t-target (p′ ! ?i)}*›
by *blast*
qed

fix *pR1 pR2* **assume** *pR1 ∈ ?RP* **and** *pR2 ∈ ?RP* **and** *pR1 ≠ pR2*

then consider (*a*) *pR1 ∈ ?R ∧ pR2 ∈ ?R* |
(*b*) *pR1 = pP″* |
(*c*) *pR2 = pP″*
using ‹⋀ *pR . pR ∈ ?RP ⟹ pR ∈ ?R ∨ pR = pP″*› ‹*pR1 ≠ pR2*› **by**
blast
then show *io-targets M′ (p-io pR1) (initial M′) ∩ io-targets M′ (p-io pR2)*
(initial M′) = {}
proof *cases*
case *a*
then show *?thesis* **using** *pairwise-dist-prop-R*[*of pR1 pR2, OF - -* ‹*pR1 ≠*
pR2›] **by** *blast*
next
case *b*
then show *?thesis* **using** *pairwise-dist-prop-PS*[*OF* ‹*pR2 ∈ ?RP*›] ‹*pR1 ≠*
pR2› **by** *blast*
next
case *c*
then show *?thesis* **using** *pairwise-dist-prop-PS*[*OF* ‹*pR1 ∈ ?RP*›] ‹*pR1 ≠*
pR2› **by** *blast*
qed
qed
then have *?P3* **by** *blast*

let *?f = (λ pR . io-targets M′ (p-io pR) (initial M′))*

have *p1*: (⋀*S1 S2. S1 ∈ ?RP ⟹ S2 ∈ ?RP ⟹ S1 = S2 ∨ ?f S1 ∩ ?f S2*
= {})
using *pairwise-dist-prop-RP* **by** *blast*
have *p2*: (⋀*S. S ∈ ?RP ⟹ io-targets M′ (p-io S) (FSM.initial M′) ≠ {}*)
using *singleton-prop-RP* **by** *blast*
have *c1*: *card ?RP = card ((λS. io-targets M′ (p-io S) (FSM.initial M′)) ‘*
?RP)
using *card-union-of-distinct*[*of ?RP, OF p1* ‹*finite ?RP*› *p2*] **by** *force*

have *p3*: ($\bigwedge S.\ S \in (\lambda S.\ \textit{io-targets}\ M'\ (\textit{p-io}\ S)\ (\textit{FSM.initial}\ M')) \text{ `} ?RP \Longrightarrow \exists\, t.\ S = \{t\}$)
using *singleton-prop-RP* **by** *blast*
have *c2*:*card* $((\lambda S.\ \textit{io-targets}\ M'\ (\textit{p-io}\ S)\ (\textit{FSM.initial}\ M')) \text{ `} ?RP) = \textit{card}\ (\bigcup S \in ?RP.\ \textit{io-targets}\ M'\ (\textit{p-io}\ S)\ (\textit{FSM.initial}\ M'))$
using *card-union-of-singletons*[*of* $((\lambda S.\ \textit{io-targets}\ M'\ (\textit{p-io}\ S)\ (\textit{FSM.initial}\ M')) \text{ `} ?RP)$, *OF p3*] **by** *force*

have *?P1*
unfolding *c1 c2* **by** *blast*

show *?combined-goals*
using ‹*?P1*› ‹*?P2*› ‹*?P3*›
by *blast*
qed

then show $\textit{card}\ (\bigcup\ (\textit{image}\ (\lambda\ pR\ .\ \textit{io-targets}\ M'\ (\textit{p-io}\ pR)\ (\textit{initial}\ M'))\ (RP\ M\ (\textit{target}\ (\textit{initial}\ M)\ pP)\ q'\ pP\ p\ PS\ M'))) = \textit{card}\ (RP\ M\ (\textit{target}\ (\textit{initial}\ M)\ pP)\ q'\ pP\ p\ PS\ M')$
and $\bigwedge\ pR\ .\ pR \in (RP\ M\ (\textit{target}\ (\textit{initial}\ M)\ pP)\ q'\ pP\ p\ PS\ M') \Longrightarrow \exists\ q\ .\ \textit{io-targets}\ M'\ (\textit{p-io}\ pR)\ (\textit{initial}\ M') = \{q\}$
and $\bigwedge\ pR1\ pR2\ .\ pR1 \in (RP\ M\ (\textit{target}\ (\textit{initial}\ M)\ pP)\ q'\ pP\ p\ PS\ M') \Longrightarrow pR2 \in (RP\ M\ (\textit{target}\ (\textit{initial}\ M)\ pP)\ q'\ pP\ p\ PS\ M') \Longrightarrow pR1 \neq pR2 \Longrightarrow \textit{io-targets}\ M'\ (\textit{p-io}\ pR1)\ (\textit{initial}\ M') \cap \textit{io-targets}\ M'\ (\textit{p-io}\ pR2)\ (\textit{initial}\ M') = \{\}$
by *blast+*
qed

lemma *RP-target*:
assumes $pR \in (RP\ M\ q\ q'\ pP\ p\ PS\ M')$
assumes $\bigwedge\ q\ P\ .\ (q,P) \in PS \Longrightarrow \textit{is-preamble}\ P\ M\ q$
and $\bigwedge\ q\ P\ io\ x\ y\ y'\ .\ (q,P) \in PS \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
and $\textit{completely-specified}\ M'$
and $\textit{inputs}\ M' = \textit{inputs}\ M$
shows $\textit{target}\ (\textit{initial}\ M)\ pR = q'$
proof −
show $\textit{target}\ (\textit{initial}\ M)\ pR = q'$
proof ($\textit{cases}\ pR \in R\ M\ q\ q'\ pP\ p$)
case *True*
then show *?thesis* **unfolding** *R-def* **by** *force*
next
case *False*
then have $RP\ M\ q\ q'\ pP\ p\ PS\ M' \neq R\ M\ q\ q'\ pP\ p$
using *assms*(*1*) **by** *blast*
then have ($\exists\, P'\ pP'$.

$(q', P') \in PS \wedge$
path P' (*FSM.initial* P') $pP' \wedge$
target (*FSM.initial* P') $pP' = q' \wedge$
path M (*FSM.initial* M) $pP' \wedge$ *target* (*FSM.initial* M) $pP' = q' \wedge$ *p-io* $pP' \in L\ M' \wedge$ *RP* $M\ q\ q'\ pP\ p\ PS\ M' =$ *insert* pP' ($R\ M\ q\ q'\ pP\ p$))
using *RP-from-R*[*OF assms*(*2*−*5*), *of PS - - q q' pP p*] **by** *force*
then obtain pP' **where** *target* (*FSM.initial* M) $pP' = q'$ **and** *RP* $M\ q\ q'\ pP\ p\ PS\ M' =$ *insert* pP' ($R\ M\ q\ q'\ pP\ p$)
by *blast*

have $pR = pP'$
using ‹*RP* $M\ q\ q'\ pP\ p\ PS\ M' =$ *insert* pP' ($R\ M\ q\ q'\ pP\ p$)› ‹$pR \in$ (*RP* $M\ q\ q'\ pP\ p\ PS\ M'$)› *False* **by** *blast*

show *?thesis* **using** ‹*target* (*FSM.initial* M) $pP' = q'$› **unfolding** ‹$pR = pP'$› **by** *assumption*
qed
qed

41.5.2 Proof of Exhaustiveness

lemma *passes-test-suite-exhaustiveness-helper-1* :
assumes *completely-specified* M'
and *inputs* M' = *inputs* M
and *observable* M
and *observable* M'
and $(q,P) \in PS$
and *path* P (*initial* P) pP
and *target* (*initial* P) $pP = q$
and *p-io* pP @ *p-io* $p \in L\ M'$
and $(p,\ d) \in$ *m-traversal-paths-with-witness* $M\ q$ *repetition-sets* m
and *implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) $M\ m$ *repetition-sets*
and *passes-test-suite* M (*Test-Suite PS tps rd-targets separators*) M'
and $q' \neq q''$
and $q' \in$ *fst* d
and $q'' \in$ *fst* d
and $pR1 \in$ (*RP* $M\ q\ q'\ pP\ p\ PS\ M'$)
and $pR2 \in$ (*RP* $M\ q\ q''\ pP\ p\ PS\ M'$)
shows *io-targets* M' (*p-io* $pR1$) (*initial* M') $\cap$ *io-targets* M' (*p-io* $pR2$) (*initial* M') = {}
proof −

let *?RP1* = (*RP* $M\ q\ q'\ pP\ p\ PS\ M'$)
let *?RP2* = (*RP* $M\ q\ q''\ pP\ p\ PS\ M'$)
let *?R1* = ($R\ M\ q\ q'\ pP\ p$)
let *?R2* = ($R\ M\ q\ q''\ pP\ p$)

have *t1*: (*initial M*, *initial-preamble M*) ∈ *PS*
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *blast*
have *t2*: ⋀ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *force*
have *t3*: ⋀ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) ∈ *separators* (*q1*, *q2*) ⟹ (*A*, *d2*, *d1*) ∈ *separators* (*q2*, *q1*) ∧ *is-separator M q1 q2 A d1 d2*
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *force*

have *t5*: ⋀*q*. *q* ∈ *FSM.states M* ⟹ (∃ *d*∈*set repetition-sets*. *q* ∈ *fst d*)
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *force*

have *t6*: ⋀ *q*. *q* ∈ *fst* ‘ *PS* ⟹ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*} ∧ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) ⊆ *tps q*
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *auto*

have ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *fst d* ⊆ *FSM.states M* ∧ *snd d* = *fst d* ∩ *fst* ‘ *PS* ∧ (∀ *q1 q2*. *q1* ∈ *fst d* ⟶ *q2* ∈ *fst d* ⟶ *q1* ≠ *q2* ⟶ *separators* (*q1*, *q2*) ≠ {})
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps* **by** *force*
then have *t7*: ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *fst d* ⊆ *FSM.states M*
and *t8*: ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *snd d* ⊆ *fst d*
and *t8'*: ⋀ *d*. *d* ∈ *set repetition-sets* ⟹ *snd d* = *fst d* ∩ *fst* ‘ *PS*
and *t9*: ⋀ *d q1 q2*. *d* ∈ *set repetition-sets* ⟹ *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *separators* (*q1*, *q2*) ≠ {}
by *blast*+

have *t10*: ⋀ *q p d p1 p2 p3*.
q ∈ *fst* ‘ *PS* ⟹
(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m* ⟹
p = *p1* @ *p2* @ *p3* ⟹
p2 ≠ [] ⟹
target q p1 ∈ *fst d* ⟹
target q (*p1* @ *p2*) ∈ *fst d* ⟹

target q p1 $\neq$ *target q* (*p1* @ *p2*) $\Longrightarrow$
p1 $\in$ *tps q* $\wedge$ *p1* @ *p2* $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*, *p1* @ *p2*) $\wedge$ *target q* (*p1* @ *p2*) $\in$ *rd-targets* (*q*, *p1*)
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps*
by (*metis* (*no-types*, *lifting*))

have *t11*: $\bigwedge$ *q p d p1 p2 q'*.
q $\in$ *fst* ‘ *PS* $\Longrightarrow$
(*p*, *d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$
p = *p1* @ *p2* $\Longrightarrow$
q' $\in$ *fst* ‘ *PS* $\Longrightarrow$
target q p1 $\in$ *fst d* $\Longrightarrow$
q' $\in$ *fst d* $\Longrightarrow$
target q p1 $\neq$ *q'* $\Longrightarrow$
p1 $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*, []) $\wedge$ *q'* $\in$ *rd-targets* (*q*, *p1*)
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps*
by (*metis* (*no-types*, *lifting*))

have *t12*: $\bigwedge$ *q p d q1 q2*.
q $\in$ *fst* ‘ *PS* $\Longrightarrow$
(*p*, *d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m* $\Longrightarrow$
q1 $\neq$ *q2* $\Longrightarrow$
q1 $\in$ *snd d* $\Longrightarrow$
q2 $\in$ *snd d* $\Longrightarrow$
[] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*, []) $\wedge$ *q2* $\in$ *rd-targets* (*q1*, [])
using ‹*implies-completeness-for-repetition-sets* (*Test-Suite PS tps rd-targets separators*) *M m repetition-sets*›
unfolding *implies-completeness-for-repetition-sets.simps*
by (*metis* (*no-types*, *lifting*))

have *pass1*: $\bigwedge$ *q P io x y y'* . (*q*,*P*) $\in$ *PS* $\Longrightarrow$ *io*@[(*x*,*y*)] $\in$ *L P* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L M'* $\Longrightarrow$ *io*@[(*x*,*y'*)] $\in$ *L P*
using ‹*passes-test-suite M* (*Test-Suite PS tps rd-targets separators*) *M'*›
unfolding *passes-test-suite.simps*
by *meson*

have *pass2*: $\bigwedge$ *q P pP ioT pT x y y'* . (*q*,*P*) $\in$ *PS* $\Longrightarrow$ *path P* (*initial P*) *pP* $\Longrightarrow$ *target* (*initial P*) *pP* = *q* $\Longrightarrow$ *pT* $\in$ *tps q* $\Longrightarrow$ *ioT*@[(*x*,*y*)] $\in$ *set* (*prefixes* (*p-io pT*)) $\Longrightarrow$ (*p-io pP*)@*ioT*@[(*x*,*y'*)] $\in$ *L M'* $\Longrightarrow$ ($\exists$ *pT'* . *pT'* $\in$ *tps q* $\wedge$ *ioT*@[(*x*,*y'*)] $\in$ *set* (*prefixes* (*p-io pT'*)))

using ‹*passes-test-suite M* (*Test-Suite PS tps rd-targets separators*) *M′*›
unfolding *passes-test-suite.simps* **by** *blast*

have *pass3*: $\bigwedge$ *q P pP pT q′ A d1 d2 qT* . (*q*,*P*) ∈ *PS* ⟹ *path P* (*initial P*) *pP* ⟹ *target* (*initial P*) *pP* = *q* ⟹ *pT* ∈ *tps q* ⟹ (*p-io pP*)@(*p-io pT*) ∈ *L M′* ⟹ *q′* ∈ *rd-targets* (*q*,*pT*) ⟹ (*A*,*d1*,*d2*) ∈ *separators* (*target q pT*, *q′*) ⟹ *qT* ∈ *io-targets M′* ((*p-io pP*)@(*p-io pT*)) (*initial M′*) ⟹ *pass-separator-ATC M′ A qT d2*
using ‹*passes-test-suite M* (*Test-Suite PS tps rd-targets separators*) *M′*›
unfolding *passes-test-suite.simps* **by** *blast*

have *is-preamble P M q*
using ‹(*q*,*P*) ∈ *PS*› ‹$\bigwedge$ *q P*. (*q*, *P*) ∈ *PS* ⟹ *is-preamble P M q*›
by *blast*
then have *q* ∈ *states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› *path-target-is-state submachine-path*)

have *initial P* = *initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def*
by *auto*
have *path M* (*initial M*) *pP*
using ‹*is-preamble P M q*› *submachine-path-initial* ‹*path P* (*FSM.initial P*) *pP*›
unfolding *is-preamble-def*
by *blast*
moreover have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*›
unfolding ‹*initial P* = *initial M*›
by *assumption*
ultimately have *q* ∈ *states M*
using *path-target-is-state*
by *metis*

have *q* ∈ *fst* ‘ *PS*
using ‹(*q*,*P*) ∈ *PS*› **by** *force*

have *d* ∈ *set repetition-sets*
using ‹(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*]
using *find-set* **by** *force*

have *q′* ∈ *states M*
by (*meson* ‹*d* ∈ *set repetition-sets*› *assms*(*13*) *subset-iff t7*)

have $q'' \in$ *states M*
by (*meson* ‹*d* ∈ *set repetition-sets*› *assms*(*14*) *subset-iff t7*)

have *target* (*initial M*) *pR1* = *q'*
using *RP-target*[*OF* ‹*pR1* ∈ *?RP1*› *t2 pass1* ‹*completely-specified M'*› ‹*inputs M'* = *inputs M*›] **by** *force*
then have *target* (*initial M*) *pR1* ∈ *fst d*
using ‹*q'* ∈ *fst d*› **by** *blast*

have *target* (*initial M*) *pR2* = *q''*
using *RP-target*[*OF* ‹*pR2* ∈ *?RP2*› *t2 pass1* ‹*completely-specified M'*› ‹*inputs M'* = *inputs M*›] **by** *force*
then have *target* (*initial M*) *pR2* ∈ *fst d*
using ‹*q''* ∈ *fst d*› **by** *blast*

have *pR1* ≠ *pR2*
using ‹*target* (*initial M*) *pR1* = *q'*› ‹*target* (*initial M*) *pR2* = *q''*› ‹*q'* ≠ *q''*› **by** *auto*

obtain *A t1 t2* **where** (*A*,*t1*,*t2*) ∈ *separators* (*q'*,*q''*)
using *t9*[*OF* ‹*d* ∈ *set repetition-sets*› ‹*q'* ∈ *fst d*› ‹*q''* ∈ *fst d*› ‹*q'* ≠ *q''*›]
by *auto*
have (*A*,*t2*,*t1*) ∈ *separators* (*q''*,*q'*) **and** *is-separator M q' q'' A t1 t2*
using *t3*[*OF* ‹(*A*,*t1*,*t2*) ∈ *separators* (*q'*,*q''*)›] **by** *simp+*
then have *is-separator M q'' q' A t2 t1*
using *is-separator-sym* **by** *force*

show *io-targets M'* (*p-io pR1*) (*initial M'*) ∩ *io-targets M'* (*p-io pR2*) (*initial M'*) = {}
proof (*rule ccontr*)
assume *io-targets M'* (*p-io pR1*) (*FSM.initial M'*) ∩ *io-targets M'* (*p-io pR2*) (*FSM.initial M'*) ≠ {}
then obtain *qT* **where** *qT* ∈ *io-targets M'* (*p-io pR1*) (*FSM.initial M'*)
and *qT* ∈ *io-targets M'* (*p-io pR2*) (*FSM.initial M'*)
by *blast*

then have *qT* ∈ *states M'*
using *path-target-is-state* **unfolding** *io-targets.simps* **by** *force*

consider (*a*) *pR1* ∈ *?R1* ∧ *pR2* ∈ *?R2* |
(*b*) *pR1* ∈ *?R1* ∧ *pR2* ∉ *?R2* |
(*c*) *pR1* ∉ *?R1* ∧ *pR2* ∈ *?R2* |
(*d*) *pR1* ∉ *?R1* ∧ *pR2* ∉ *?R2*
by *blast*

then show *False* **proof** *cases*

case *a*
then have *pR1* ∈ *?R1* **and** *pR2* ∈ *?R2* **by** *auto*

obtain *pR1′* **where** *pR1* = *pP@pR1′* **using** *R-component-ob*[*OF* ‹*pR1* ∈ *?R1*›] **by** *blast*
obtain *pR2′* **where** *pR2* = *pP@pR2′* **using** *R-component-ob*[*OF* ‹*pR2* ∈ *?R2*›] **by** *blast*

have *pR1′* = *take* (*length pR1′*) *p* **and** *length pR1′* ≤ *length p* **and** *t-target* (*p* ! (*length pR1′* − *1*)) = *q′* **and** *pR1′* ≠ []
using *R-component*[*of pP pR1′ M q q′ p*] ‹*pR1* ∈ *?R1*› **unfolding** ‹*pR1* = *pP@pR1′*› **by** *blast+*

have *pR2′* = *take* (*length pR2′*) *p* **and** *length pR2′* ≤ *length p* **and** *t-target* (*p* ! (*length pR2′* − *1*)) = *q″* **and** *pR2′* ≠ []
using *R-component*[*of pP pR2′ M q q″ p*] ‹*pR2* ∈ *?R2*› **unfolding** ‹*pR2* = *pP@pR2′*› **by** *blast+*

have *target q pR1′* = *q′*
using ‹*target* (*initial M*) *pR1* = *q′*› ‹*pR1′* ≠ []› **unfolding** *target.simps visited-states.simps* ‹*pR1* = *pP@pR1′*› **by** *simp*
then have *target q pR1′* ∈ *fst d*
using ‹*q′* ∈ *fst d*› **by** *blast*

have *target q pR2′* = *q″*
using ‹*target* (*initial M*) *pR2* = *q″*› ‹*pR2′* ≠ []› **unfolding** *target.simps visited-states.simps* ‹*pR2* = *pP@pR2′*› **by** *simp*
then have *target q pR2′* ∈ *fst d*
using ‹*q″* ∈ *fst d*› **by** *blast*

have *pR1′* ≠ *pR2′*
using ‹*pR1* ≠ *pR2*› **unfolding** ‹*pR1* = *pP@pR1′*› ‹*pR2* = *pP@pR2′*› **by** *simp*
then have *length pR1′* ≠ *length pR2′*
using ‹*pR1′* = *take* (*length pR1′*) *p*› ‹*pR2′* = *take* (*length pR2′*) *p*› **by** *auto*
then consider (*a1*) *length pR1′* < *length pR2′* | (*a2*) *length pR2′* < *length pR1′*
using *nat-neq-iff* **by** *blast*
then have *pR1′* ∈ *tps q* ∧ *pR2′* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*, *pR2′*) ∧ *q″* ∈ *rd-targets* (*q*, *pR1′*)
proof *cases*
case *a1*
then have *pR2′* = *pR1′* @ (*drop* (*length pR1′*) *pR2′*)
using ‹*pR1′* = *take* (*length pR1′*) *p*› ‹*pR2′* = *take* (*length pR2′*) *p*›
by (*metis append-take-drop-id less-imp-le-nat take-le*)
then have *p* = *pR1′* @ (*drop* (*length pR1′*) *pR2′*) @ (*drop* (*length pR2′*) *p*)
using ‹*pR2′* = *take* (*length pR2′*) *p*›
by (*metis append.assoc append-take-drop-id*)

have (*drop* (*length pR1'*) *pR2'*) ≠ []
using *a1* ‹*pR2'* = *take* (*length pR2'*) *p*› **by** *auto*
have *target q* (*pR1'* @ *drop* (*length pR1'*) *pR2'*) ∈ *fst d*
using ‹*pR2'* = *pR1'* @ (*drop* (*length pR1'*) *pR2'*)›[*symmetric*] ‹*target q pR2'* ∈ *fst d*› **by** *auto*

show *?thesis*
using *t10*[*OF* ‹*q* ∈ *fst* ‘ *PS*› ‹(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
‹*p* = *pR1'* @ (*drop* (*length pR1'*) *pR2'*) @ (*drop* (*length pR2'*) *p*)›
‹(*drop* (*length pR1'*) *pR2'*) ≠ []› ‹*target q pR1'* ∈ *fst d*›
‹*target q* (*pR1'* @ *drop* (*length pR1'*) *pR2'*) ∈ *fst d*›]
unfolding ‹*pR2'* = *pR1'* @ (*drop* (*length pR1'*) *pR2'*)›[*symmetric*] ‹*target q pR1'* = *q'*› ‹*target q pR2'* = *q''*›
using ‹*q'* ≠ *q''*›
by *blast*
next
case *a2*
then have *pR1'* = *pR2'* @ (*drop* (*length pR2'*) *pR1'*)
using ‹*pR1'* = *take* (*length pR1'*) *p*› ‹*pR2'* = *take* (*length pR2'*) *p*›
by (*metis append-take-drop-id less-imp-le-nat take-le*)
then have *p* = *pR2'* @ (*drop* (*length pR2'*) *pR1'*) @ (*drop* (*length pR1'*) *p*)
using ‹*pR1'* = *take* (*length pR1'*) *p*›
by (*metis append.assoc append-take-drop-id*)

have (*drop* (*length pR2'*) *pR1'*) ≠ []
using *a2* ‹*pR1'* = *take* (*length pR1'*) *p*› **by** *auto*
have *target q* (*pR2'* @ *drop* (*length pR2'*) *pR1'*) ∈ *fst d*
using ‹*pR1'* = *pR2'* @ (*drop* (*length pR2'*) *pR1'*)›[*symmetric*] ‹*target q pR1'* ∈ *fst d*› **by** *auto*

show *?thesis*
using *t10*[*OF* ‹*q* ∈ *fst* ‘ *PS*› ‹(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
‹*p* = *pR2'* @ (*drop* (*length pR2'*) *pR1'*) @ (*drop* (*length pR1'*) *p*)›
‹(*drop* (*length pR2'*) *pR1'*) ≠ []› ‹*target q pR2'* ∈ *fst d*›
‹*target q* (*pR2'* @ *drop* (*length pR2'*) *pR1'*) ∈ *fst d*›]
unfolding ‹*pR1'* = *pR2'* @ (*drop* (*length pR2'*) *pR1'*)›[*symmetric*] ‹*target q pR1'* = *q'*› ‹*target q pR2'* = *q''*›
using ‹*q'* ≠ *q''*›
by *blast*
qed
then have *pR1'* ∈ *tps q* **and** *pR2'* ∈ *tps q* **and** *q'* ∈ *rd-targets* (*q*, *pR2'*) **and** *q''* ∈ *rd-targets* (*q*, *pR1'*)
by *simp+*

have *p-io pP @ p-io pR1′ ∈ L M′*
using *language-prefix-append*[*OF* ‹*p-io pP @ p-io p ∈ L M′*›, *of length pR1′*]
using ‹*pR1′ = take* (*length pR1′*) *p*› **by** *simp*
have *pass-separator-ATC M′ A qT t2*
using *pass3*[*OF* ‹(*q*, *P*) ∈ *PS*› ‹*path P* (*initial P*) *pP*› ‹*target* (*initial P*) *pP* = *q*› ‹*pR1′ ∈ tps q*›
‹*p-io pP @ p-io pR1′ ∈ L M′*› ‹*q″ ∈ rd-targets* (*q*, *pR1′*)›, *of A t1 t2*]
‹(*A*, *t1*, *t2*) ∈ *separators* (*q′*, *q″*)› ‹*qT ∈ io-targets M′* (*p-io pR1*) (*FSM.initial M′*)›
unfolding ‹*target q pR1′ = q′*› ‹*pR1 = pP @ pR1′*› **by** *auto*

have *p-io pP @ p-io pR2′ ∈ L M′*
using *language-prefix-append*[*OF* ‹*p-io pP @ p-io p ∈ L M′*›, *of length pR2′*]
using ‹*pR2′ = take* (*length pR2′*) *p*› **by** *simp*
have *pass-separator-ATC M′ A qT t1*
using *pass3*[*OF* ‹(*q*, *P*) ∈ *PS*› ‹*path P* (*initial P*) *pP*› ‹*target* (*initial P*) *pP* = *q*› ‹*pR2′ ∈ tps q*›
‹*p-io pP @ p-io pR2′ ∈ L M′*› ‹*q′ ∈ rd-targets* (*q*, *pR2′*)›, *of A t2 t1*]
‹(*A*, *t2*, *t1*) ∈ *separators* (*q″*, *q′*)› ‹*qT ∈ io-targets M′* (*p-io pR2*) (*FSM.initial M′*)›
unfolding ‹*target q pR2′ = q″*› ‹*pR2 = pP @ pR2′*› **by** *auto*

have *qT ≠ qT*
using *pass-separator-ATC-reduction-distinction*[*OF* ‹*observable M*› ‹*observable M′*› ‹*inputs M′ = inputs M*› ‹*pass-separator-ATC M′ A qT t2*› ‹*pass-separator-ATC M′ A qT t1*› ‹*q′ ∈ states M*› ‹*q″ ∈ states M*› ‹*q′ ≠ q″*› ‹*qT ∈ states M′*› ‹*qT ∈ states M′*› ‹*is-separator M q′ q″ A t1 t2*› ‹*completely-specified M′*›]
by *assumption*
then show *False*
by *simp*

next
case *b*

then have *pR1 ∈ ?R1* **and** *pR2 ∉ ?R2*
using ‹*pR1 ∈ ?RP1*› **by** *auto*

obtain *pR1′* **where** *pR1 = pP@pR1′* **using** *R-component-ob*[*OF* ‹*pR1 ∈ ?R1*›] **by** *blast*

have *pR1′ = take* (*length pR1′*) *p* **and** *length pR1′ ≤ length p* **and** *t-target* (*p* ! (*length pR1′ − 1*)) = *q′* **and** *pR1′ ≠* []

using *R-component*[*of pP pR1′ M q q′ p*] ‹*pR1* ∈ *?R1*› **unfolding** ‹*pR1* = *pP@pR1′*› **by** *blast+*

have *target q pR1′* = *q′*
using ‹*target* (*initial M*) *pR1* = *q′*› ‹*pR1′* ≠ []› **unfolding** *target.simps visited-states.simps* ‹*pR1* = *pP@pR1′*› **by** *simp*
then have *target q pR1′* ∈ *fst d* **and** *target q pR1′* ≠ *q″*
using ‹*q′* ∈ *fst d*› ‹*q′* ≠ *q″*› **by** *blast+*

obtain *P′* **where** (*q″*, *P′*) ∈ *PS*
path P′ (*FSM.initial P′*) *pR2*
target (*FSM.initial P′*) *pR2* = *q″*
path M (*FSM.initial M*) *pR2*
target (*FSM.initial M*) *pR2* = *q″*
p-io pR2 ∈ *L M′*
RP M q q″ pP p PS M′ = *insert pR2* (*R M q q″ pP p*)
using *RP-from-R-inserted*[*OF t2 pass1* ‹*completely-specified M′*› ‹*inputs M′* = *inputs M*› ‹*pR2* ∈ *?RP2*› ‹*pR2* ∉ *?R2*›,
of λ *q P io x y y′* . *q* λ *q P io x y y′* . *y*]
by *blast*

have *q″* ∈ *fst ‘ PS* **using** ‹(*q″*,*P′*) ∈ *PS*› **by** *force*
have *p* = *pR1′* @ (*drop* (*length pR1′*) *p*) **using** ‹*pR1′* = *take* (*length pR1′*) *p*›
by (*metis append-take-drop-id*)

have *pR1′* ∈ *tps q* **and** [] ∈ *tps q″* **and** *target q pR1′* ∈ *rd-targets* (*q″*, []) **and** *q″* ∈ *rd-targets* (*q*, *pR1′*)
using *t11*[*OF* ‹*q* ∈ *fst ‘ PS*› ‹(*p*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
‹*p* = *pR1′* @ (*drop* (*length pR1′*) *p*)› ‹*q″* ∈ *fst ‘ PS*›
‹*target q pR1′* ∈ *fst d*› ‹*q″* ∈ *fst d*› ‹*target q pR1′* ≠ *q″*›]
by *simp+*

have *p-io pP* @ *p-io pR1′* ∈ *L M′*
using *language-prefix-append*[*OF* ‹*p-io pP* @ *p-io p* ∈ *L M′*›, *of length pR1′*]
using ‹*pR1′* = *take* (*length pR1′*) *p*› **by** *simp*
have *pass-separator-ATC M′ A qT t2*
using *pass3*[*OF* ‹(*q*, *P*) ∈ *PS*› ‹*path P* (*initial P*) *pP*› ‹*target* (*initial P*) *pP* = *q*› ‹*pR1′* ∈ *tps q*›
‹*p-io pP* @ *p-io pR1′* ∈ *L M′*› ‹*q″* ∈ *rd-targets* (*q*, *pR1′*)›, *of A t1 t2*]
‹(*A*, *t1*, *t2*) ∈ *separators* (*q′*, *q″*)› ‹*qT* ∈ *io-targets M′* (*p-io pR1*) (*FSM.initial M′*)›
unfolding ‹*target q pR1′* = *q′*› ‹*pR1* = *pP* @ *pR1′*› **by** *auto*

have *pass-separator-ATC M′ A qT t1*
using *pass3*[*OF* ‹(*q*′′, *P*′) ∈ *PS*› ‹*path P*′ (*FSM.initial P*′) *pR2*› ‹*target* (*FSM.initial P*′) *pR2* = *q*′′›
‹[] ∈ *tps q*′′› - ‹*target q pR1*′ ∈ *rd-targets* (*q*′′, [])›, *of A t2 t1 qT*]
‹(*A*, *t2*, *t1*) ∈ *separators* (*q*′′, *q*′)› ‹*qT* ∈ *io-targets M*′ (*p-io pR2*) (*FSM.initial M*′)› ‹*p-io pR2* ∈ *L M*′›
unfolding ‹*target q pR1*′ = *q*′› **by** *auto*

have *qT* ≠ *qT*
using *pass-separator-ATC-reduction-distinction*[*OF* ‹*observable M*› ‹*observable M*′› ‹*inputs M*′ = *inputs M*›
‹*pass-separator-ATC M*′ *A qT t2*›
‹*pass-separator-ATC M*′ *A qT t1*› ‹*q*′ ∈ *states M*›
‹*q*′′ ∈ *states M*› ‹*q*′ ≠ *q*′′› ‹*qT* ∈ *states M*′›
‹*qT* ∈ *states M*′› ‹*is-separator M q*′ *q*′′ *A t1 t2*›
‹*completely-specified M*′›]
by *assumption*
then show *False*
by *simp*
next
case *c*
then have *pR2* ∈ *?R2* **and** *pR1* ∉ *?R1*
using ‹*pR2* ∈ *?RP2*› **by** *auto*

obtain *pR2*′ **where** *pR2* = *pP*@*pR2*′ **using** *R-component-ob*[*OF* ‹*pR2* ∈ *?R2*›] **by** *blast*

have *pR2*′ = *take* (*length pR2*′) *p*
and *length pR2*′ ≤ *length p*
and *t-target* (*p* ! (*length pR2*′ − *1*)) = *q*′′
and *pR2*′ ≠ []
using *R-component*[*of pP pR2*′ *M q q*′′ *p*] ‹*pR2* ∈ *?R2*›
unfolding ‹*pR2* = *pP*@*pR2*′›
by *blast*+

have *target q pR2*′ = *q*′′
using ‹*target* (*initial M*) *pR2* = *q*′′› ‹*pR2*′ ≠ []›
unfolding *target.simps visited-states.simps* ‹*pR2* = *pP*@*pR2*′›
by *simp*
then have *target q pR2*′ ∈ *fst d* **and** *target q pR2*′ ≠ *q*′
using ‹*q*′′ ∈ *fst d*› ‹*q*′ ≠ *q*′′› **by** *blast*+

obtain *P*′ **where** (*q*′, *P*′) ∈ *PS*

path P′ (FSM.initial P′) pR1
target (FSM.initial P′) pR1 = q′
path M (FSM.initial M) pR1
target (FSM.initial M) pR1 = q′
p-io pR1 ∈ L M′
RP M q q′ pP p PS M′ = insert pR1 (R M q q′ pP p)
using *RP-from-R-inserted*[*OF t2 pass1* ‹*completely-specified M′*› ‹*inputs M′ = inputs M*› ‹*pR1 ∈ ?RP1*› ‹*pR1 ∉ ?R1*›,
of λ q P io x y y′ . q λ q P io x y y′ . y]
by *blast*

have *q′ ∈ fst ‘ PS* **using** ‹*(q′,P′) ∈ PS*› **by** *force*
have *p = pR2′ @ (drop (length pR2′) p)* **using** ‹*pR2′ = take (length pR2′) p*›
by (*metis append-take-drop-id*)

have *pR2′ ∈ tps q* **and** *[] ∈ tps q′* **and** *target q pR2′ ∈ rd-targets (q′, []) and q′ ∈ rd-targets (q, pR2′)*
using *t11*[*OF* ‹*q ∈ fst ‘ PS*› ‹*(p, d) ∈ m-traversal-paths-with-witness M q repetition-sets m*›
‹*p = pR2′ @ (drop (length pR2′) p)*› ‹*q′ ∈ fst ‘ PS*› ‹*target q pR2′ ∈ fst d*›
‹*q′ ∈ fst d*› ‹*target q pR2′ ≠ q′*›]
by *simp+*

have *p-io pP @ p-io pR2′ ∈ L M′*
using *language-prefix-append*[*OF* ‹*p-io pP @ p-io p ∈ L M′*›, *of length pR2′*]
using ‹*pR2′ = take (length pR2′) p*› **by** *simp*
have *pass-separator-ATC M′ A qT t1*
using *pass3*[*OF* ‹*(q, P) ∈ PS*› ‹*path P (initial P) pP*› ‹*target (initial P) pP = q*› ‹*pR2′ ∈ tps q*›
‹*p-io pP @ p-io pR2′ ∈ L M′*› ‹*q′ ∈ rd-targets (q, pR2′)*›, *of A t2 t1*]
‹*(A, t2, t1) ∈ separators (q″, q′)*› ‹*qT ∈ io-targets M′ (p-io pR2) (FSM.initial M′)*›
unfolding ‹*target q pR2′ = q″*› ‹*pR2 = pP @ pR2′*› **by** *auto*

have *pass-separator-ATC M′ A qT t2*
using *pass3*[*OF* ‹*(q′, P′) ∈ PS*› ‹*path P′ (FSM.initial P′) pR1*› ‹*target (FSM.initial P′) pR1 = q′*›
‹*[] ∈ tps q′*› - ‹*target q pR2′ ∈ rd-targets (q′, [])*›, *of A t1 t2 qT*]
‹*(A, t1, t2) ∈ separators (q′, q″)*› ‹*qT ∈ io-targets M′ (p-io pR1) (FSM.initial M′)*› ‹*p-io pR1 ∈ L M′*›
unfolding ‹*target q pR2′ = q″*› **by** *auto*

have *qT ≠ qT*
using *pass-separator-ATC-reduction-distinction*[*OF* ‹*observable M*› ‹*observ-*

able M′› ‹*inputs M′ = inputs M*›
‹*pass-separator-ATC M′ A qT t1*›
‹*pass-separator-ATC M′ A qT t2*›
‹*q″ ∈ states M*› ‹*q′ ∈ states M*› -
‹*qT ∈ states M′*› ‹*qT ∈ states M′*›
‹*is-separator M q″ q′ A t2 t1*›
‹*completely-specified M′*›]
‹*q′ ≠ q″*› **by** *simp*
then show *False*
by *simp*
next
case *d*

then have *pR1 ∉ ?R1* **and** *pR2 ∉ ?R2*
by *auto*

obtain *P′* **where** (*q′*, *P′*) ∈ *PS*
path P′ (*FSM.initial P′*) *pR1*
target (*FSM.initial P′*) *pR1* = *q′*
path M (*FSM.initial M*) *pR1*
target (*FSM.initial M*) *pR1* = *q′*
p-io pR1 ∈ *L M′*
RP M q q′ pP p PS M′ = *insert pR1* (*R M q q′ pP p*)
using *RP-from-R-inserted*[*OF t2 pass1* ‹*completely-specified M′*› ‹*inputs M′ = inputs M*›
‹*pR1 ∈ ?RP1*› ‹*pR1 ∉ ?R1*›, *of* λ *q P io x y y′* . *q* λ *q P io x y y′* . *y*]
by *blast*

have *q′ ∈ snd d*
by (*metis IntI* ‹(*q′*, *P′*) ∈ *PS*› ‹*d ∈ set repetition-sets*› *assms*(*13*) *fst-eqD image-eqI t8′*)

obtain *P″* **where** (*q″*, *P″*) ∈ *PS*
path P″ (*FSM.initial P″*) *pR2*
target (*FSM.initial P″*) *pR2* = *q″*
path M (*FSM.initial M*) *pR2*
target (*FSM.initial M*) *pR2* = *q″*
p-io pR2 ∈ *L M′*
RP M q q″ pP p PS M′ = *insert pR2* (*R M q q″ pP p*)
using *RP-from-R-inserted*[*OF t2 pass1* ‹*completely-specified M′*› ‹*inputs M′ = inputs M*› ‹*pR2 ∈ ?RP2*› ‹*pR2 ∉ ?R2*›,
of λ *q P io x y y′* . *q* λ *q P io x y y′* . *y*]
by *blast*

have *q″ ∈ snd d*
by (*metis IntI* ‹(*q″*, *P″*) ∈ *PS*› ‹*d ∈ set repetition-sets*› *assms*(*14*) *fst-eqD image-eqI t8′*)

have $[] \in tps\ q'$ **and** $[] \in tps\ q''$ **and** $q' \in rd\text{-}targets\ (q'', [])$ **and** $q'' \in rd\text{-}targets\ (q', [])$
using *t12*[*OF* ‹$q \in fst$ ‘ PS› ‹$(p, d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m$› ‹$q' \neq q''$› ‹$q' \in snd\ d$› ‹$q'' \in snd\ d$›]
by *simp*+

have *pass-separator-ATC* $M'\ A\ qT\ t1$
using *pass3*[*OF* ‹$(q'', P'') \in PS$› ‹$path\ P''\ (initial\ P'')\ pR2$› ‹$target\ (initial\ P'')\ pR2 = q''$›
‹$[] \in tps\ q''$› - ‹$q' \in rd\text{-}targets\ (q'', [])$›, *of* $A\ t2\ t1\ qT$]
‹$p\text{-}io\ pR2 \in L\ M'$› ‹$(A, t2, t1) \in separators\ (q'', q')$› ‹$qT \in io\text{-}targets\ M'\ (p\text{-}io\ pR2)\ (FSM.initial\ M')$›
by *auto*

have *pass-separator-ATC* $M'\ A\ qT\ t2$
using *pass3*[*OF* ‹$(q', P') \in PS$› ‹$path\ P'\ (initial\ P')\ pR1$› ‹$target\ (initial\ P')\ pR1 = q'$›
‹$[] \in tps\ q'$› - ‹$q'' \in rd\text{-}targets\ (q', [])$›, *of* $A\ t1\ t2\ qT$]
‹$p\text{-}io\ pR1 \in L\ M'$› ‹$(A, t1, t2) \in separators\ (q', q'')$› ‹$qT \in io\text{-}targets\ M'\ (p\text{-}io\ pR1)\ (FSM.initial\ M')$›
by *auto*

have $qT \neq qT$
using *pass-separator-ATC-reduction-distinction*[*OF* ‹*observable* M› ‹*observable* M'›
‹$inputs\ M' = inputs\ M$›
‹*pass-separator-ATC* $M'\ A\ qT\ t1$›
‹*pass-separator-ATC* $M'\ A\ qT\ t2$›
‹$q'' \in states\ M$›
‹$q' \in states\ M$› - ‹$qT \in states\ M'$›
‹$qT \in states\ M'$›
‹*is-separator* $M\ q''\ q'\ A\ t2\ t1$›
‹*completely-specified* M'›]
‹$q' \neq q''$› **by** *simp*
then show *False*
by *simp*
qed
qed
qed

lemma *passes-test-suite-exhaustiveness* :
assumes *passes-test-suite* M (*Test-Suite prs tps rd-targets separators*) M'
and *implies-completeness* (*Test-Suite prs tps rd-targets separators*) $M\ m$
and *observable* M
and *observable* M'

and *inputs* $M' = $ *inputs* M
and *inputs* $M \neq \{\}$
and *completely-specified* M
and *completely-specified* M'
and *size* $M' \leq m$
shows $L\ M' \subseteq L\ M$
proof (*rule ccontr*)
assume $\neg\ L\ M' \subseteq L\ M$

obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets separators*) *M m repetition-sets*
using *assms*(*2*) **unfolding** *implies-completeness-def* **by** *blast*

have *t1*: (*initial M*, *initial-preamble M*) $\in$ *prs*
using *implies-completeness-for-repetition-sets-simps*(*1*)[*OF repetition-sets-def*]

by *assumption*
have *t2*: $\bigwedge$ *q P*. (*q*, *P*) $\in$ *prs* $\Longrightarrow$ *is-preamble P M q*
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*]

by *blast*
have *t3*: $\bigwedge$ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) $\in$ *separators* (*q1*, *q2*) $\Longrightarrow$ (*A*, *d2*, *d1*) $\in$ *separators* (*q2*, *q1*) $\wedge$ *is-separator M q1 q2 A d1 d2*
using *implies-completeness-for-repetition-sets-simps*(*3*)[*OF repetition-sets-def*]

by *assumption*
have *t5*: $\bigwedge$*q*. *q* $\in$ *FSM.states M* $\Longrightarrow$ ($\exists$ *d*$\in$*set repetition-sets*. *q* $\in$ *fst d*)
using *implies-completeness-for-repetition-sets-simps*(*4*)[*OF repetition-sets-def*]
by *assumption*
have *t6*: $\bigwedge$ *q*. *q* $\in$ *fst* ‘ *prs* $\Longrightarrow$ *tps q* $\subseteq$ {*p1* . $\exists$ *p2 d* . (*p1*@*p2*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*} $\wedge$ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) $\subseteq$ *tps q*
using *implies-completeness-for-repetition-sets-simps*(*7*)[*OF repetition-sets-def*]

by *assumption*

have *t7*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *fst d* $\subseteq$ *FSM.states M*
and *t8*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
and *t8'*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *snd d* = *fst d* $\cap$ *fst* ‘ *prs*
and *t9*: $\bigwedge$ *d q1 q2*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *q1* $\in$ *fst d* $\Longrightarrow$ *q2* $\in$ *fst d* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *separators* (*q1*, *q2*) $\neq$ {}
using *implies-completeness-for-repetition-sets-simps*(*5*,*6*)[*OF repetition-sets-def*]

by *blast*+

have *t10*: $\bigwedge$ *q p d p1 p2 p3*.

$q \in fst$ ‘ $prs \Longrightarrow$
$(p,\ d) \in$ *m-traversal-paths-with-witness* $M\ q$ *repetition-sets* $m \Longrightarrow$
$p = p1$ @ $p2$ @ $p3 \Longrightarrow$
$p2 \neq [] \Longrightarrow$
target $q\ p1 \in fst\ d \Longrightarrow$
target $q\ (p1$ @ $p2) \in fst\ d \Longrightarrow$
target $q\ p1 \neq$ *target* $q\ (p1$ @ $p2) \Longrightarrow$
$p1 \in tps\ q \wedge p1$ @ $p2 \in tps\ q \wedge$ *target* $q\ p1 \in$ *rd-targets* $(q,\ p1$ @ $p2)$ $\wedge$ *target* $q\ (p1$ @ $p2) \in$ *rd-targets* $(q,\ p1)$
using *implies-completeness-for-repetition-sets-simps*(*8*)[*OF repetition-sets-def*]
by *assumption*

have *t11*: $\bigwedge\ q\ p\ d\ p1\ p2\ q'$.
$q \in fst$ ‘ $prs \Longrightarrow$
$(p,\ d) \in$ *m-traversal-paths-with-witness* $M\ q$ *repetition-sets* $m \Longrightarrow$
$p = p1$ @ $p2 \Longrightarrow$
$q' \in fst$ ‘ $prs \Longrightarrow$
target $q\ p1 \in fst\ d \Longrightarrow$
$q' \in fst\ d \Longrightarrow$
target $q\ p1 \neq q' \Longrightarrow$
$p1 \in tps\ q \wedge [] \in tps\ q' \wedge$ *target* $q\ p1 \in$ *rd-targets* $(q',\ []) \wedge q' \in$ *rd-targets* $(q,\ p1)$
using *implies-completeness-for-repetition-sets-simps*(*9*)[*OF repetition-sets-def*]
by *assumption*

have *t12*: $\bigwedge\ q\ p\ d\ q1\ q2$.
$q \in fst$ ‘ $prs \Longrightarrow$
$(p,\ d) \in$ *m-traversal-paths-with-witness* $M\ q$ *repetition-sets* $m \Longrightarrow$
$q1 \neq q2 \Longrightarrow$
$q1 \in snd\ d \Longrightarrow$
$q2 \in snd\ d \Longrightarrow$
$[] \in tps\ q1 \wedge [] \in tps\ q2 \wedge q1 \in$ *rd-targets* $(q2,\ []) \wedge q2 \in$ *rd-targets* $(q1,\ [])$
using *implies-completeness-for-repetition-sets-simps*(*10*)[*OF repetition-sets-def*]
by *assumption*

have *pass1*: $\bigwedge\ q\ P\ io\ x\ y\ y'$. $(q,P) \in prs \Longrightarrow io@[(x,y)] \in L\ P \Longrightarrow io@[(x,y')] \in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
using ‹*passes-test-suite* M (*Test-Suite prs tps rd-targets separators*) M'›
unfolding *passes-test-suite.simps*
by *meson*

have *pass2*: $\bigwedge\ q\ P\ pP\ ioT\ pT\ x\ y\ y'$. $(q,P) \in prs \Longrightarrow$ *path* P (*initial* P) $pP \Longrightarrow$ *target* (*initial* P) $pP = q \Longrightarrow pT \in tps\ q \Longrightarrow ioT@[(x,y)] \in$ *set* (*prefixes* (*p-io* pT)) $\Longrightarrow$ (*p-io* pP)$@ioT@[(x,y')] \in L\ M' \Longrightarrow (\exists\ pT'$. $pT' \in tps\ q \wedge ioT@[(x,y')]$

$\in$ *set* (*prefixes* (*p-io pT'*)))
using ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets separators*) *M'*›
unfolding *passes-test-suite.simps*
by *blast*

have *pass3*: $\bigwedge$ *q P pP pT q' A d1 d2 qT* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$ *path P* (*initial P*) *pP* $\Longrightarrow$ *target* (*initial P*) *pP* = *q* $\Longrightarrow$ *pT* $\in$ *tps q* $\Longrightarrow$ (*p-io pP*)@(*p-io pT*) $\in$ *L M'* $\Longrightarrow$ *q'* $\in$ *rd-targets* (*q*,*pT*) $\Longrightarrow$ (*A*,*d1*,*d2*) $\in$ *separators* (*target q pT*, *q'*) $\Longrightarrow$ *qT* $\in$ *io-targets M'* ((*p-io pP*)@(*p-io pT*)) (*initial M'*) $\Longrightarrow$ *pass-separator-ATC M' A qT d2*
using ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets separators*) *M'*›
unfolding *passes-test-suite.simps*
by *blast*

obtain *pP io* **where** *minimal-sequence-to-failure-extending-preamble-path M M' prs pP io*
using *minimal-sequence-to-failure-extending-preamble-ex*[*OF t1* ‹$\neg$ *L M'* $\subseteq$ *L M*›]
by *blast*

then have *sequence-to-failure-extending-preamble-path M M' prs pP io*
$\bigwedge$ *io'*. *sequence-to-failure-extending-preamble-path M M' prs pP io'* $\Longrightarrow$ *length io* $\leq$ *length io'*
unfolding *minimal-sequence-to-failure-extending-preamble-path-def*
by *blast+*

obtain *q P* **where** *q* $\in$ *states M*
and (*q*,*P*) $\in$ *prs*
and *path P* (*initial P*) *pP*
and *target* (*initial P*) *pP* = *q*
and ((*p-io pP*) @ *butlast io*) $\in$ *L M*
and ((*p-io pP*) @ *io*) $\notin$ *L M*
and ((*p-io pP*) @ *io*) $\in$ *L M'*
using ‹*sequence-to-failure-extending-preamble-path M M' prs pP io*›
unfolding *sequence-to-failure-extending-preamble-path-def*
by *blast*

let *?xF* = *fst* (*last io*)
let *?yF* = *snd* (*last io*)
let *?xyF* = (*?xF*,*?yF*)
let *?ioF* = *butlast io*
have *io* $\neq$ []
using ‹((*p-io pP*) @ *io*) $\notin$ *L M*› ‹((*p-io pP*) @ *butlast io*) $\in$ *L M*› **by** *auto*
then have *io* = *?ioF*@[*?xyF*]
by *auto*

have *?xF* ∈ *inputs M′*
using *language-io(1)*[*OF* ‹((*p-io pP*) @ *io*) ∈ *L M′*›, *of ?xF ?yF*] ‹*io* ≠ []› **by** *auto*
then have *?xF* ∈ *inputs M*
using ‹*inputs M′* = *inputs M*› **by** *simp*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*,*P*) ∈ *prs*› **by** *force*
have *is-preamble P M q*
using ‹(*q*,*P*) ∈ *prs*› *t2* **by** *blast*
then have *q* ∈ *states M*
unfolding *is-preamble-def*
by (*metis* ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› *path-target-is-state submachine-path*)

have *initial P* = *initial M*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **by** *auto*
have *path M* (*initial M*) *pP*
using ‹*is-preamble P M q*› **unfolding** *is-preamble-def* **using** *submachine-path-initial*
using ‹*path P* (*FSM.initial P*) *pP*› **by** *blast*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*› **unfolding** ‹*initial P* = *initial M*› **by** *assumption*

obtain *pM dM ioEx* **where** (*pM*,*dM*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
and *io* = (*p-io pM*)@*ioEx*
and *ioEx* ≠ []
proof −

obtain *pF* **where** *path M q pF* **and** *p-io pF* = *?ioF*
using *observable-path-suffix*[*OF* ‹((*p-io pP*) @ *?ioF*) ∈ *L M*› ‹*path M* (*initial M*) *pP*› ‹*observable M*›]
unfolding ‹*target* (*initial M*) *pP* = *q*›
by *blast*

obtain *tM* **where** *tM* ∈ *transitions M* **and** *t-source tM* = *target q pF* **and** *t-input tM* = *?xF*
using ‹*?xF* ∈ *inputs M*› *path-target-is-state*[*OF* ‹*path M q pF*›]
‹*completely-specified M*›
unfolding *completely-specified.simps*
by *blast*

then have *path M q* (*pF*@[*tM*])
using ‹*path M q pF*› *path-append-transition* **by** *simp*

show *?thesis* **proof** (*cases find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) (*pF*@[*tM*]))) *repetition-sets*)
case *None*

obtain *pF′ d′* **where** ((*pF*@[*tM*]) @ *pF′*, *d′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *m-traversal-path-extension-exist*[*OF* ‹*completely-specified M*› ‹*q* ∈ *states M*› ‹*inputs M* ≠ {}› *t5 t8* ‹*path M q* (*pF*@[*tM*])› *None*]
by *blast*
then have (*pF*@[*tM*]) @ *pF′* ∈ *tps q*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] **by** *force*

have (*p-io pF*) @ [(*?xF*,*t-output tM*)] ∈ *set* (*prefixes* (*p-io* ((*pF*@[*tM*])@*pF′*)))
using ‹*t-input tM* = *?xF*›
unfolding *prefixes-set* **by** *auto*

have *p-io pP* @ *p-io pF* @ [*?xyF*] ∈ *L M′*
using ‹((*p-io pP*) @ *io*) ∈ *L M′*› **unfolding** ‹*p-io pF* = *?ioF*› ‹*io* = *?ioF*@[*?xyF*]›[*symmetric*] **by** *assumption*

obtain *pT′* **where** *pT′* ∈ *tps q*
and *p-io pF* @ [(*fst* (*last io*), *snd* (*last io*))] ∈ *set* (*prefixes* (*p-io pT′*))
using *pass2*[*OF* ‹(*q*,*P*) ∈ *prs*› ‹*path P* (*initial P*) *pP*› ‹*target* (*initial P*) *pP* = *q*› ‹(*pF*@[*tM*]) @ *pF′* ∈ *tps q*›
‹(*p-io pF*) @ [(*?xF*,*t-output tM*)] ∈ *set* (*prefixes* (*p-io* ((*pF*@[*tM*])@*pF′*)))› ‹*p-io pP* @ *p-io pF* @ [*?xyF*] ∈ *L M′*›]
by *blast*

have *path M q pT′*
proof −
obtain *pT″ d″* **where** (*pT′*@*pT″*, *d″*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using ‹*pT′* ∈ *tps q*› *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›]
by *blast*
then have *path M q* (*pT′*@*pT″*)
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›]
by *force*
then show *?thesis*
by *auto*
qed
then have *path M* (*initial M*) (*pP*@*pT′*)
using ‹*path M* (*initial M*) *pP*› ‹*target* (*initial M*) *pP* = *q*› **by** *auto*
then have (*p-io* (*pP*@*pT′*)) ∈ *L M*
unfolding *LS.simps* **by** *blast*
then have (*p-io pP*)@(*p-io pT′*) ∈ *L M*
by *auto*

have *io* ∈ *set* (*prefixes* (*p-io pT′*))
using ‹*p-io pF* @ [(*fst* (*last io*), *snd* (*last io*))] ∈ *set* (*prefixes* (*p-io pT′*))›
unfolding ‹*p-io pF* = *?ioF*› ‹*io* = *?ioF*@[*?xyF*]›[*symmetric*] **by** *assumption*
then obtain *io′* **where** *p-io pT′* = *io* @ *io′*
unfolding *prefixes-set mem-Collect-eq* **by** *metis*

have *p-io pP* @ *io* ∈ *L M*
using ‹(*p-io pP*)@(*p-io pT′*) ∈ *L M*›
unfolding ‹*p-io pT′* = *io* @ *io′*›
unfolding *append.assoc*[*symmetric*]
using *language-prefix*[*of p-io pP* @ *io io′*, *of M initial M*]
by *blast*

then show *?thesis*
using ‹(*p-io pP*) @ *io* ∉ *L M*› **by** *simp*
next
case (*Some d*)

let *?ps* = { *p1* . ∃ *p2* . (*pF*@[*tM*]) = *p1* @ *p2* ∧ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p1*)) *repetition-sets* ≠ *None*}

have *finite ?ps*
proof −
have *?ps* ⊆ *set* (*prefixes* (*pF*@[*tM*]))
unfolding *prefixes-set* **by** *force*
moreover have *finite* (*set* (*prefixes* (*pF*@[*tM*])))
by *simp*
ultimately show *?thesis*
by (*simp add*: *finite-subset*)
qed
moreover have *?ps* ≠ {}
proof −
have *pF* @ [*tM*] = (*pF* @ [*tM*]) @ [] ∧ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) (*pF* @ [*tM*]))) *repetition-sets* ≠ *None*
using *Some* **by** *auto*
then have (*pF*@[*tM*]) ∈ *?ps*
by *blast*
then show *?thesis* **by** *blast*
qed
ultimately obtain *pMin* **where** *pMin* ∈ *?ps* **and** ⋀ *p′* . *p′* ∈ *?ps* ⟹ *length pMin* ≤ *length p′*
by (*meson leI min-length-elem*)

obtain *pMin′ dMin* **where** (*pF*@[*tM*]) = *pMin* @ *pMin′*

and *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) *pMin*)) *repetition-sets* = *Some dMin*
using ‹*pMin* ∈ *?ps*› **by** *blast*
then have *path M q pMin*
using ‹*path M q* (*pF*@[*tM*])› **by** *auto*

moreover have (∀ p' p''. *pMin* = p' @ p'' ∧ p'' ≠ [] ⟶ *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) *repetition-sets* = *None*)
proof −
have ⋀ p' p''. *pMin* = p' @ p'' ⟹ p'' ≠ [] ⟹ *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) *repetition-sets* = *None*
proof −
fix p' p'' **assume** *pMin* = p' @ p'' **and** p'' ≠ []
show *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) *repetition-sets* = *None*
proof (*rule ccontr*)
assume *find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) p')) *repetition-sets* ≠ *None*
then have p' ∈ *?ps*
using ‹(*pF*@[*tM*]) = *pMin* @ *pMin'*› **unfolding** ‹*pMin* = p' @ p''› *append.assoc* **by** *blast*

have *length* p' < *length pMin*
using ‹*pMin* = p' @ p''› ‹p'' ≠ []› **by** *auto*
then show *False*
using ‹⋀ p' . p' ∈ *?ps* ⟹ *length pMin* ≤ *length* p'›[*OF* ‹p' ∈ *?ps*›] **by** *simp*
qed
qed
then show *?thesis* **by** *blast*
qed

ultimately have (*pMin*,*dMin*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using ‹*find* (λd. *Suc* (m − *card* (*snd d*)) ≤ *length* (*filter* (λt. *t-target t* ∈ *fst d*) *pMin*)) *repetition-sets* = *Some dMin*›
m-traversal-paths-with-witness-set[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*]
by *blast*
then have *pMin* ∈ *tps q*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›]
by *force*

show *?thesis* **proof** (*cases pMin* = (*pF*@[*tM*]))

case *True*
then have *?ioF* @ [(*?xF*, *t-output tM*)] ∈ *set* (*prefixes* (*p-io pMin*))
using ‹*p-io pF* = *?ioF*› ‹*t-input tM* = *?xF*› **unfolding** *prefixes-set* **by** *force*

obtain *pMinF* **where** *pMinF* ∈ *tps q* **and** *io* ∈ *set* (*prefixes* (*p-io pMinF*))
using *pass2*[*OF* ‹(*q*,*P*) ∈ *prs*› ‹*path P* (*initial P*) *pP*› ‹*target* (*initial P*) *pP* = *q*› ‹*pMin* ∈ *tps q*› ‹*?ioF* @ [(*?xF*, *t-output tM*)] ∈ *set* (*prefixes* (*p-io pMin*))›, *of ?yF*]
using ‹*p-io pP* @ *io* ∈ *L M′*›
unfolding ‹*io* = *?ioF*@[*?xyF*]›[*symmetric*]
by *blast*

have *path M q pMinF*
proof −
obtain *pT″ d″* **where** (*pMinF*@*pT″*, *d″*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using ‹*pMinF* ∈ *tps q*› *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] **by** *blast*
then have *path M q* (*pMinF*@*pT″*)
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›]
by *force*
then show *?thesis* **by** *auto*
qed
then have *path M* (*initial M*) (*pP*@*pMinF*)
using ‹*path M* (*initial M*) *pP*› ‹*target* (*initial M*) *pP* = *q*› **by** *auto*
then have (*p-io* (*pP*@*pMinF*)) ∈ *L M*
unfolding *LS.simps* **by** *blast*
then have (*p-io pP*)@(*p-io pMinF*) ∈ *L M*
by *auto*

obtain *io′* **where** *p-io pMinF* = *io* @ *io′*
using ‹*io* ∈ *set* (*prefixes* (*p-io pMinF*))›
unfolding *prefixes-set mem-Collect-eq* **by** *metis*

have *p-io pP* @ *io* ∈ *L M*
using ‹(*p-io pP*)@(*p-io pMinF*) ∈ *L M*›
unfolding ‹*p-io pMinF* = *io* @ *io′*›
unfolding *append.assoc*[*symmetric*]
using *language-prefix*[*of p-io pP* @ *io io′*, *of M initial M*]
by *blast*
then show *?thesis*
using ‹(*p-io pP*) @ *io* ∉ *L M*› **by** *simp*
next
case *False*
then obtain *pMin″* **where** *pF* = *pMin* @ *pMin″*
using ‹(*pF*@[*tM*]) = *pMin* @ *pMin′*›
by (*metis butlast-append butlast-snoc*)
then have *io* = (*p-io pMin*) @ (*p-io pMin″*) @ [*?xyF*]
using ‹*io* = *?ioF* @ [*?xyF*]› ‹*p-io pF* = *?ioF*›
by (*metis* (*no-types*, *lifting*) *append-assoc map-append*)
then show *?thesis*
using *that*[*OF* ‹(*pMin*,*dMin*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›, *of* (*p-io pMin″*) @ [*?xyF*]]
by *auto*

qed
qed
qed

have *p-io pP @ p-io pM ∈ L M′*
using ‹((*p-io pP*) @ *io*) ∈ *L M′*› **unfolding** ‹*io* = (*p-io pM*)@*ioEx*› *append.assoc*[*symmetric*]
using *language-prefix*[*of p-io pP @ p-io pM ioEx M′ initial M′*] **by** *blast*

have *no-shared-targets-for-distinct-states* : ⋀ *q′ q″ pR1 pR2. q′ ≠ q″* ⟹
q′ ∈ fst dM ⟹
q″ ∈ fst dM ⟹
pR1 ∈ RP M q q′ pP pM prs M′ ⟹
pR2 ∈ RP M q q″ pP pM prs M′ ⟹
io-targets M′ (*p-io pR1*) (*initial M′*) ∩ *io-targets M′* (*p-io pR2*) (*initial M′*) = {}
using *passes-test-suite-exhaustiveness-helper-1*[*OF* ‹*completely-specified M′*› ‹*inputs M′ = inputs M*› ‹*observable M*› ‹*observable M′*› ‹(*q*,*P*) ∈ *prs*› ‹*path P* (*initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› ‹*p-io pP @ p-io pM ∈ L M′*› ‹(*pM*, *dM*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*› *repetition-sets-def* ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets separators*) *M′*›]
by *blast*

have *path M q pM*
and *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *pM*)) *repetition-sets* = *Some dM*
using ‹(*pM*,*dM*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*] **by** *force*+
then have *path M* (*target* (*FSM.initial M*) *pP*) *pM*
unfolding ‹(*target* (*FSM.initial M*) *pP*) = *q*› **by** *simp*

have *dM* ∈ *set repetition-sets*
using *find-set*[*OF* ‹*find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *pM*)) *repetition-sets* = *Some dM*›] **by** *assumption*
have *Suc* (*m* − *card* (*snd dM*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst dM*) *pM*)
using *find-condition*[*OF* ‹*find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *pM*)) *repetition-sets* = *Some dM*›] **by** *assumption*

obtain *ioX* **where** *butlast io* = (*p-io pM*)@*ioX*
using ‹*io* = (*p-io pM*)@*ioEx*›
by (*simp add*: ‹*ioEx* ≠ []› *butlast-append*)

have *RP-card* : $\bigwedge$ *q'* . *card* ($\bigcup$*pR*∈*RP M* (*target* (*FSM.initial M*) *pP*) *q' pP pM prs M'*. *io-targets M'* (*p-io pR*) (*FSM.initial M'*)) = *card* (*RP M* (*target* (*FSM.initial M*) *pP*) *q' pP pM prs M'*)
and *RP-targets*: $\bigwedge$ *q' pR* . *pR* ∈ *RP M* (*target* (*FSM.initial M*) *pP*) *q' pP pM prs M'* ⟹ ∃ *q*. *io-targets M'* (*p-io pR*) (*FSM.initial M'*) = {*q*}
and *no-shared-targets-for-identical-states*: $\bigwedge$ *q' pR1 pR2* . *pR1* ∈ *RP M* (*target* (*FSM.initial M*) *pP*) *q' pP pM prs M'* ⟹ *pR2* ∈ *RP M* (*target* (*FSM.initial M*) *pP*) *q' pP pM prs M'* ⟹ *pR1* ≠ *pR2* ⟹ *io-targets M'* (*p-io pR1*) (*FSM.initial M'*) ∩ *io-targets M'* (*p-io pR2*) (*FSM.initial M'*) = {}
using *RP-count*[*OF* ‹*minimal-sequence-to-failure-extending-preamble-path M M' prs pP io*› ‹*observable M*› ‹*observable M'*› *t2* ‹*path M* (*target* (*FSM.initial M*) *pP*) *pM*› ‹*butlast io* = (*p-io pM*)@*ioX*› *pass1* ‹*completely-specified M'*› ‹*inputs M'* = *inputs M*›, *of* λ *q P io x y y'* . *q* λ *q P io x y y'* . *y*]
by *blast+*

have *snd-dM-prop*: $\bigwedge$ *q'* . *q'* ∈ *snd dM* ⟹ (⋃ *pR* ∈ (*RP M q q' pP pM prs M'*) . *io-targets M'* (*p-io pR*) (*initial M'*)) ≠ (⋃ *pR* ∈ (*R M q q' pP pM*) . *io-targets M'* (*p-io pR*) (*initial M'*))
proof −
fix *q'* **assume** *q'* ∈ *snd dM*

let *?RP* = (*RP M q q' pP pM prs M'*)
let *?R* = (*R M q q' pP pM*)
let *?P* = λ *pP'* . ∃ *P'*. (*q'*, *P'*) ∈ *prs* ∧ *path P'* (*FSM.initial P'*) *pP'* ∧ *target* (*FSM.initial P'*) *pP'* = *q'* ∧ *p-io pP'* ∈ *L M'*

obtain *PQ* **where** (*q'*,*PQ*) ∈ *prs*
using ‹*q'* ∈ *snd dM*› *t8'*[*OF* ‹*dM* ∈ *set repetition-sets*›] **by** *auto*
then have *is-preamble PQ M q'* **and** ∃ *P'*. (*q'*, *P'*) ∈ *prs*
using *t2* **by** *blast+*

obtain *pq* **where** *path PQ* (*initial PQ*) *pq* **and** *target* (*initial PQ*) *pq* = *q'* **and** *p-io pq* ∈ *L M'*
using *preamble-pass-path*[*OF* ‹*is-preamble PQ M q'*› *pass1*[*OF* ‹(*q'*,*PQ*) ∈ *prs*›] ‹*completely-specified M'*› ‹*inputs M'* = *inputs M*›]
by *force*
then have ∃ *pP'* . *?P pP'*
using ‹(*q'*,*PQ*) ∈ *prs*› **by** *blast*

define *pPQ* **where** *pPQ-def*: *pPQ* = (*SOME pP'*. *?P pP'*)

have *?P pPQ*
unfolding *pPQ-def* **using** *someI-ex*[*OF* ‹∃ *pP'* . *?P pP'*›] **by** *assumption*
then obtain *PQ'* **where** (*q'*,*PQ'*) ∈ *prs*
and *path PQ'* (*initial PQ'*) *pPQ*
and *target* (*initial PQ'*) *pPQ* = *q'*
and *p-io pPQ* ∈ *L M'*
by *blast*

have *?RP* = *insert pPQ* (*R M q q′ pP pM*)
unfolding *RP-def pPQ-def*
using ‹∃ *P′*. (*q′*, *P′*) ∈ *prs*› **by** *auto*

obtain *pPQ′* **where** *path M′* (*initial M′*) *pPQ′* **and** *p-io pPQ′* = *p-io pPQ*
using ‹*p-io pPQ* ∈ *L M′*› **by** *auto*

then have *io-targets M′* (*p-io pPQ*) (*initial M′*) = {*target* (*initial M′*) *pPQ′*}
using ‹*observable M′*› **by** (*metis* (*mono-tags*, *lifting*) *observable-path-io-target*)

moreover have *target* (*initial M′*) *pPQ′* ∉ (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) *?R*))
proof
assume *target* (*initial M′*) *pPQ′* ∈ (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) *?R*))
then obtain *pR* **where** *pR* ∈ *?R* **and** *target* (*initial M′*) *pPQ′* ∈ *io-targets M′* (*p-io pR*) (*initial M′*)
by *blast*

obtain *pR′* **where** *pR* = *pP@pR′*
using *R-component-ob*[*OF* ‹*pR* ∈ *?R*›] **by** *blast*
then have *pP@pR′* ∈ *?R*
using ‹*pR* ∈ *?R*› **by** *simp*
have *pR′* = *take* (*length pR′*) *pM*
and *length pR′* ≤ *length pM*
and *t-target* (*pM* ! (*length pR′* − *1*)) = *q′*
and *pR′* ≠ []
using *R-component*[*OF* ‹*pP@pR′* ∈ *?R*›] **by** *auto*

obtain *pX* **where** *path M* (*target* (*initial M*) *pP*) (*pM@pX*) **and** *p-io* (*pM@pX*) = *butlast io*
proof −
have *p-io pP* @ *p-io pM* @ *ioX* ∈ *L M*
using ‹((*p-io pP*) @ *butlast io*) ∈ *L M*›
unfolding ‹*butlast io* = *p-io pM* @ *ioX*›
by *assumption*

obtain *p1 p23* **where** *path M* (*FSM.initial M*) *p1*
and *path M* (*target* (*FSM.initial M*) *p1*) *p23*
and *p-io p1* = *p-io pP*
and *p-io p23* = *p-io pM* @ *ioX*
using *language-state-split*[*OF* ‹*p-io pP* @ *p-io pM* @ *ioX* ∈ *L M*›]
by *blast*

```
      have p1 = pP
       using observable-path-unique[OF ‹observable M› ‹path M (FSM.initial M)
p1› ‹path M (FSM.initial M) pP› ‹p-io p1 = p-io pP›]
        by assumption
      then have path M (target (FSM.initial M) pP) p23
        using ‹path M (target (FSM.initial M) p1) p23› by auto
      then have p-io pM @ ioX ∈ LS M (target (initial M) pP)
        using ‹p-io p23 = p-io pM @ ioX› language-state-containment by auto

      obtain p2 p3 where path M (target (FSM.initial M) pP) p2
                    and path M (target (target (FSM.initial M) pP) p2) p3
                    and p-io p2 = p-io pM
                    and p-io p3 = ioX
       using language-state-split[OF ‹p-io pM @ ioX ∈ LS M (target (initial M)
pP)›]
        by blast

      have p2 = pM
     using observable-path-unique[OF ‹observable M› ‹path M (target (FSM.initial
M) pP) p2›
                                    ‹path M (target (FSM.initial M) pP) pM› ‹p-io
p2 = p-io pM›]
        by assumption
      then have path M (target (FSM.initial M) pP) (pM@p3)
       using ‹path M (target (FSM.initial M) pP) pM› ‹path M (target (target
(FSM.initial M) pP) p2) p3›
        by auto
      moreover have p-io (pM@p3) = butlast io
        unfolding ‹butlast io = p-io pM @ ioX› using ‹p-io p3 = ioX›
        by auto
      ultimately show ?thesis
        using that[of p3] by simp
    qed

    obtain pP' pIO where path M' (FSM.initial M') pP'
                    and path M' (target (FSM.initial M') pP') pIO
                    and p-io pP' = p-io pP
                    and p-io pIO = io
      using language-state-split[OF ‹((p-io pP) @ io) ∈ L M'›]
      by blast

    have target (initial M') pP' ∈ io-targets M' (p-io pP) (FSM.initial M')
      using ‹path M' (FSM.initial M') pP'›
      unfolding ‹p-io pP' = p-io pP›[symmetric]
      by auto

    let ?i = length pR' − 1
    have ?i < length pR'
      using ‹pR' ≠ []› by auto
```

have *?i* < *length* (*butlast io*)
using ‹*pR′* = *take* (*length pR′*) *pM*› ‹*pR′* ≠ []›
unfolding ‹*p-io* (*pM*@*pX*) = *butlast io*›[*symmetric*]
using *leI* **by** *fastforce*

have *t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*)) = *q′*
by (*metis* ‹*length pR′* − *1* < *length pR′*› ‹*length pR′* ≤ *length pM*› ‹*t-target* (*pM* ! (*length pR′* − *1*)) = *q′*›
dual-order.strict-trans1 nth-append)
then have (*t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*)), *PQ′*) ∈ *prs*
using ‹(*q′*,*PQ′*) ∈ *prs*› **by** *simp*
have *target* (*FSM.initial PQ′*) *pPQ* = *t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*))
using ‹*t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*)) = *q′*› ‹*target* (*FSM.initial PQ′*) *pPQ* = *q′*›
by *blast*

have *t-target* (*pIO* ! *?i*) ∉ *io-targets M′* (*p-io pPQ*) (*FSM.initial M′*)
using *minimal-sequence-to-failure-extending-preamble-no-repetitions-with-other-preambles*
[*OF* ‹*minimal-sequence-to-failure-extending-preamble-path M M′ prs pP io*› ‹*observable M*› ‹
path M (*target* (*initial M*) *pP*) (*pM*@*pX*)› ‹*p-io* (*pM*@*pX*) = *butlast io*›
‹*target* (*initial M′*) *pP′* ∈ *io-targets M′* (*p-io pP*) (*FSM.initial M′*)›
‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *pIO*› ‹*p-io pIO* = *io*› *t2*
‹*?i* < *length* (*butlast io*)› ‹(*t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*)), *PQ′*) ∈ *prs*›
‹*path PQ′* (*initial PQ′*) *pPQ*› ‹*target* (*FSM.initial PQ′*) *pPQ* = *t-target* ((*pM* @ *pX*) ! (*length pR′* − *1*))›]
by *blast*

moreover have *io-targets M′* (*p-io pPQ*) (*FSM.initial M′*) = {*target* (*initial M′*) *pPQ′*}
using ‹*path M′* (*initial M′*) *pPQ′*›
using ‹*io-targets M′* (*p-io pPQ*) (*FSM.initial M′*) = {*target* (*FSM.initial M′*) *pPQ′*}› ‹*p-io pPQ′* = *p-io pPQ*› **by** *auto*
moreover have *io-targets M′* (*p-io* (*pP*@*pR′*)) (*FSM.initial M′*) = {*t-target* (*pIO* ! *?i*)}
proof −
have (*take* (*length pR′*) *pIO*) ≠ []
using ‹*p-io pIO* = *io*› ‹*?i* < *length pR′*›
using ‹*io* = *p-io pM* @ *ioEx*› ‹*pR′* = *take* (*length pR′*) *pM*› **by** *auto*
moreover have *pIO* ! *?i* = *last* (*take* (*length pR′*) *pIO*)
using ‹*p-io pIO* = *io*› ‹*?i* < *length pR′*›
by (*metis* (*no-types*, *lifting*) ‹*io* = *p-io pM* @ *ioEx*› ‹*length pR′* ≤ *length pM*› ‹*pR′* = *take* (*length pR′*) *pM*›
butlast.simps(*1*) *last-conv-nth length-butlast length-map neq-iff nth-take take-le take-map*)

ultimately have *t-target* (*pIO* ! *?i*) = *target* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR′*) *pIO*)
unfolding *target.simps visited-states.simps*
by (*simp add*: *last-map*)
then have *t-target* (*pIO* ! *?i*) = *target* (*initial M′*) (*pP′* @ (*take* (*length pR′*) *pIO*))
by *auto*

have *path M′* (*target* (*FSM.initial M′*) *pP′*) (*take* (*length pR′*) *pIO*)
using ‹*path M′* (*target* (*FSM.initial M′*) *pP′*) *pIO*›
by (*simp add*: *path-prefix-take*)
then have *path M′* (*initial M′*) (*pP′* @ (*take* (*length pR′*) *pIO*))
using ‹*path M′* (*FSM.initial M′*) *pP′*› **by** *auto*
moreover have *p-io* (*pP′* @ (*take* (*length pR′*) *pIO*)) = (*p-io* (*pP@pR′*))
proof −
have *p-io* (*take* (*length pR′*) *pIO*) = *p-io pR′*
using ‹*p-io pIO* = *io*› ‹*pR′* = *take* (*length pR′*) *pM*› ‹*p-io* (*pM@pX*) = *butlast io*› ‹*length pR′* ≤ *length pM*›
by (*metis* (*no-types*, *lifting*) ‹*io* = *p-io pM* @ *ioEx*› *length-map take-le take-map*)
then show *?thesis*
using ‹*p-io pP′* = *p-io pP*› **by** *auto*
qed
ultimately have *io-targets M′* (*p-io* (*pP@pR′*)) (*FSM.initial M′*) = {*target* (*initial M′*) (*pP′* @ (*take* (*length pR′*) *pIO*))}
by (*metis* (*mono-tags*, *lifting*) *assms*(*4*) *observable-path-io-target*)

then show *?thesis*
unfolding ‹*t-target* (*pIO* ! *?i*) = *target* (*initial M′*) (*pP′* @ (*take* (*length pR′*) *pIO*))›
by *assumption*
qed

ultimately have *target* (*initial M′*) *pPQ′* ∉ *io-targets M′* (*p-io pR*) (*initial M′*)
unfolding ‹*pR* = *pP@pR′*› **by** *auto*
then show *False*
using ‹*target* (*initial M′*) *pPQ′* ∈ *io-targets M′* (*p-io pR*) (*initial M′*)›
by *blast*
qed

ultimately have *io-targets M′* (*p-io pPQ*) (*initial M′*) ∩ (⋃ (*image* (λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)) *?R*)) = {}
by *force*

then show (⋃ *pR* ∈ (*RP M q q′ pP pM prs M′*) . *io-targets M′* (*p-io pR*) (*initial M′*)) ≠ (⋃ *pR* ∈ (*R M q q′ pP pM*) . *io-targets M′* (*p-io pR*) (*initial M′*))
unfolding ‹*?RP* = *insert pPQ* (*R M q q′ pP pM*)›
using ‹*io-targets M′* (*p-io pPQ*) (*FSM.initial M′*) = {*target* (*FSM.initial M′*)

pPQ'}›
by *force*
qed

then obtain *f* **where** *f-def*: $\bigwedge q' . q' \in snd\ dM \Longrightarrow (RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ (f\ q')\ (R\ M\ q\ q'\ pP\ pM) \wedge (f\ q') \notin (R\ M\ q\ q'\ pP\ pM)$
proof −
define *f* **where** *f-def* : $f = (\lambda\ q' . SOME\ p . (RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ p\ (R\ M\ q\ q'\ pP\ pM) \wedge p \notin (R\ M\ q\ q'\ pP\ pM))$

have $\bigwedge q' . q' \in snd\ dM \Longrightarrow (RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ (f\ q')\ (R\ M\ q\ q'\ pP\ pM) \wedge (RP\ M\ q\ q'\ pP\ pM\ prs\ M') \neq (R\ M\ q\ q'\ pP\ pM)$
proof −
fix q' **assume** $q' \in snd\ dM$

have $(\bigcup pR \in RP\ M\ q\ q'\ pP\ pM\ prs\ M'.\ io\text{-}targets\ M'\ (p\text{-}io\ pR)\ (FSM.initial\ M')) \neq (\bigcup pR \in R\ M\ q\ q'\ pP\ pM.\ io\text{-}targets\ M'\ (p\text{-}io\ pR)\ (FSM.initial\ M'))$
using *snd-dM-prop*[*OF* ‹$q' \in snd\ dM$›]
by *assumption*
then have $(RP\ M\ q\ q'\ pP\ pM\ prs\ M') \neq (R\ M\ q\ q'\ pP\ pM)$
by *blast*
then obtain x **where** $(RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ x\ (R\ M\ q\ q'\ pP\ pM)$
using *RP-from-R*[*OF t2 pass1* ‹*completely-specified M'*› ‹*inputs M' = inputs M*›, *of prs* $\lambda\ q\ P\ io\ x\ y\ y' . q\ \lambda\ q\ P\ io\ x\ y\ y' . y\ q\ q'\ pP\ pM$]
by *force*
then have $x \notin (R\ M\ q\ q'\ pP\ pM)$
using ‹$(RP\ M\ q\ q'\ pP\ pM\ prs\ M') \neq (R\ M\ q\ q'\ pP\ pM)$›
by *auto*
then have $\exists\ p . (RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ p\ (R\ M\ q\ q'\ pP\ pM) \wedge p \notin (R\ M\ q\ q'\ pP\ pM)$
using ‹$(RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ x\ (R\ M\ q\ q'\ pP\ pM)$› **by** *blast*

show $(RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ (f\ q')\ (R\ M\ q\ q'\ pP\ pM) \wedge (RP\ M\ q\ q'\ pP\ pM\ prs\ M') \neq (R\ M\ q\ q'\ pP\ pM)$
using *someI-ex*[*OF* ‹$\exists\ p . (RP\ M\ q\ q'\ pP\ pM\ prs\ M') = insert\ p\ (R\ M\ q\ q'\ pP\ pM) \wedge p \notin (R\ M\ q\ q'\ pP\ pM)$›]
unfolding *f-def* **by** *auto*
qed

then show *?thesis* **using** *that* **by** *force*
qed

have $(\sum\ q' \in fst\ dM\ .\ card\ (\bigcup\ pR \in (RP\ M\ q\ q'\ pP\ pM\ prs\ M')\ .\ io\text{-}targets\ M'\ (p\text{-}io\ pR)\ (initial\ M'))) \geq Suc\ m$
proof −

have $\bigwedge\ nds\ .\ finite\ nds \Longrightarrow nds \subseteq fst\ dM \Longrightarrow (\sum\ q' \in nds\ .\ card\ (RP\ M\ q\ q'\ pP\ pM\ prs\ M')) \geq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in nds)\ pM) + card\ (nds \cap snd\ dM)$
proof −
fix *nds* **assume** *finite nds* **and** $nds \subseteq fst\ dM$
then show $(\sum\ q' \in nds\ .\ card\ (RP\ M\ q\ q'\ pP\ pM\ prs\ M')) \geq length\ (filter\ (\lambda t.\ t\text{-}target\ t \in nds)\ pM) + card\ (nds \cap snd\ dM)$
proof *induction*
case *empty*
then show *?case* **by** *auto*
next
case (*insert* q' *nds*)
then have *leq1*: $length\ (filter\ (\lambda t.\ t\text{-}target\ t \in nds)\ pM) + card\ (nds \cap snd\ dM) \leq (\sum q' \in nds.\ card\ (RP\ M\ q\ q'\ pP\ pM\ prs\ M'))$
by *blast*

have *p4*: $(card\ (R\ M\ q\ q'\ pP\ pM)) = length\ (filter\ (\lambda t.\ t\text{-}target\ t = q')\ pM)$

using ‹*path M q pM*› **proof** (*induction pM rule*: *rev-induct*)
case *Nil*
then show *?case* **unfolding** *R-def* **by** *auto*
next
case (*snoc t pM*)
then have *path M q pM* **and** $card\ (R\ M\ q\ q'\ pP\ pM) = length\ (filter\ (\lambda t.\ t\text{-}target\ t = q')\ pM)$
by *auto*

show *?case* **proof** (*cases target q* $(pM\ @\ [t]) = q'$)
case *True*
then have $(R\ M\ q\ q'\ pP\ (pM\ @\ [t])) = insert\ (pP\ @\ pM\ @\ [t])\ (R\ M\ q\ q'\ pP\ pM)$
unfolding *R-update*[*of M q q' pP pM t*] **by** *simp*
moreover have $(pP\ @\ pM\ @\ [t]) \notin (R\ M\ q\ q'\ pP\ pM)$
unfolding *R-def* **by** *auto*
ultimately have $card\ (R\ M\ q\ q'\ pP\ (pM\ @\ [t])) = Suc\ (card\ (R\ M\ q\ q'\ pP\ pM))$
using *finite-R*[*OF* ‹*path M q pM*›, *of q' pP*] **by** *simp*
then show *?thesis*
using *True* **unfolding** ‹$card\ (R\ M\ q\ q'\ pP\ pM) = length\ (filter\ (\lambda t.\ t\text{-}target\ t = q')\ pM)$› **by** *auto*
next
case *False*
then have $card\ (R\ M\ q\ q'\ pP\ (pM\ @\ [t])) = card\ (R\ M\ q\ q'\ pP\ pM)$
unfolding *R-update*[*of M q q' pP pM t*] **by** *simp*

then show *?thesis*
using *False* **unfolding** ‹*card* (*R M q q′ pP pM*) = *length* (*filter* (λ*t*. *t-target t* = *q′*) *pM*)› **by** *auto*
qed
qed

show *?case* **proof** (*cases q′* ∈ *snd dM*)
case *True*
then have *p0*: (*RP M q q′ pP pM prs M′*) = *insert* (*f q′*) (*R M q q′ pP pM*) **and** (*f q′*) ∉ (*R M q q′ pP pM*)
using *f-def* **by** *blast+*
then have *card* (*RP M q q′ pP pM prs M′*) = *Suc* (*card* (*R M q q′ pP pM*))
by (*simp add*: ‹*path M q pM*› *finite-R*)
then have *p1*: (∑ *q′* ∈ (*insert q′ nds*). *card* (*RP M q q′ pP pM prs M′*)) = (∑ *q′*∈*nds*. *card* (*RP M q q′ pP pM prs M′*)) + *Suc* (*card* (*R M q q′ pP pM*))
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))

have *p2*: *length* (*filter* (λ*t*. *t-target t* ∈ *insert q′ nds*) *pM*) = *length* (*filter* (λ*t*. *t-target t* ∈ *nds*) *pM*) + *length* (*filter* (λ*t*. *t-target t* = *q′*) *pM*)
using ‹*q′* ∉ *nds*› **by** (*induction pM*; *auto*)
have *p3*: *card* ((*insert q′ nds*) ∩ *snd dM*) = *Suc* (*card* (*nds* ∩ *snd dM*))
using *True* ‹*finite nds*› ‹*q′* ∉ *nds*› **by** *simp*

show *?thesis*
using *leq1*
unfolding *p1 p2 p3 p4* **by** *simp*
next
case *False*

have *card* (*RP M q q′ pP pM prs M′*) ≥ (*card* (*R M q q′ pP pM*))
proof (*cases* (*RP M q q′ pP pM prs M′*) = (*R M q q′ pP pM*))
case *True*
then show *?thesis* **using** *finite-R*[*OF* ‹*path M q pM*›, *of q′ pP*] **by** *auto*
next
case *False*
then obtain *pX* **where** (*RP M q q′ pP pM prs M′*) = *insert pX* (*R M q q′ pP pM*)
using *RP-from-R*[*OF t2 pass1* ‹*completely-specified M′*› ‹*inputs M′* = *inputs M*›, *of prs* λ *q P io x y y′* . *q* λ *q P io x y y′* . *y q q′ pP pM*]
by *force*
then show *?thesis* **using** *finite-R*[*OF* ‹*path M q pM*›, *of q′ pP*]
by (*simp add*: *card-insert-le*)
qed
then have *p1*: (∑ *q′* ∈ (*insert q′ nds*). *card* (*RP M q q′ pP pM prs M′*)) ≥ ((∑ *q′*∈*nds*. *card* (*RP M q q′ pP pM prs M′*)) + (*card* (*R M q q′ pP pM*)))
by (*simp add*: *insert.hyps*(*1*) *insert.hyps*(*2*))

have *p2*: *length* (*filter* (λ*t*. *t-target t* $\in$ *insert q′ nds*) *pM*) = *length* (*filter* (λ*t*. *t-target t* $\in$ *nds*) *pM*) + *length* (*filter* (λ*t*. *t-target t* = *q′*) *pM*)
using ‹*q′* $\notin$ *nds*› **by** (*induction pM*; *auto*)
have *p3*: *card* ((*insert q′ nds*) $\cap$ *snd dM*) = (*card* (*nds* $\cap$ *snd dM*))
using *False* ‹*finite nds*› ‹*q′* $\notin$ *nds*› **by** *simp*

have *length* (*filter* (λ*t*. *t-target t* $\in$ *nds*) *pM*) + *length* (*filter* (λ*t*. *t-target t* = *q′*) *pM*) + *card* (*nds* $\cap$ *snd dM*) $\leq$ ($\sum$ *q′*$\in$*nds*. *card* (*RP M q q′ pP pM prs M′*)) + *length* (*filter* (λ*t*. *t-target t* = *q′*) *pM*)
using *leq1 add-le-cancel-right* **by** *auto*

then show *?thesis*
using *p1*
unfolding *p2 p3 p4* **by** *simp*
qed
qed
qed

moreover have *finite* (*fst dM*)
using *t7*[*OF* ‹*dM* $\in$ *set repetition-sets*›] *fsm-states-finite*[*of M*]
using *rev-finite-subset* **by** *auto*
ultimately have ($\sum$ *q′* $\in$ *fst dM* . *card* (*RP M q q′ pP pM prs M′*)) $\geq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*) + *card* (*fst dM* $\cap$ *snd dM*)
by *blast*
have (*fst dM* $\cap$ *snd dM*) = (*snd dM*)
using *t8*[*OF* ‹*dM* $\in$ *set repetition-sets*›] **by** *blast*
have ($\sum$ *q′* $\in$ *fst dM* . *card* (*RP M q q′ pP pM prs M′*)) $\geq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*) + *card* (*snd dM*)
using ‹($\sum$ *q′* $\in$ *fst dM* . *card* (*RP M q q′ pP pM prs M′*)) $\geq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*) + *card* (*fst dM* $\cap$ *snd dM*)›
unfolding ‹(*fst dM* $\cap$ *snd dM*) = (*snd dM*)›
by *assumption*
moreover have ($\sum$ *q′*$\in$*fst dM*. *card* ($\bigcup$ *pR*$\in$*RP M q q′ pP pM prs M′*. *io-targets M′* (*p-io pR*) (*FSM.initial M′*))) = ($\sum$ *q′* $\in$ *fst dM* . *card* (*RP M q q′ pP pM prs M′*))
using *RP-card* ‹*FSM.initial P* = *FSM.initial M*› ‹*target* (*FSM.initial P*) *pP* = *q*› **by** *auto*
ultimately have ($\sum$ *q′*$\in$*fst dM*. *card* ($\bigcup$ *pR*$\in$*RP M q q′ pP pM prs M′*. *io-targets M′* (*p-io pR*) (*FSM.initial M′*))) $\geq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*) + *card* (*snd dM*)
by *linarith*
moreover have *Suc m* $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*) + *card* (*snd dM*)
using ‹*Suc* (*m* − *card* (*snd dM*)) $\leq$ *length* (*filter* (λ*t*. *t-target t* $\in$ *fst dM*) *pM*)›
by *linarith*
ultimately show *?thesis*
by *linarith*
qed

moreover have $(\sum\ q' \in fst\ dM\ .\ card\ (\bigcup\ pR \in (RP\ M\ q\ q'\ pP\ pM\ prs\ M')\ .\ io\text{-}targets\ M'\ (p\text{-}io\ pR)\ (initial\ M'))) \leq card\ (states\ M')$
proof −
have *finite* (*fst dM*)
by (*meson* ‹*dM* ∈ *set repetition-sets*› *fsm-states-finite rev-finite-subset t7*)

have $(\bigwedge x1.\ finite\ (RP\ M\ q\ x1\ pP\ pM\ prs\ M'))$
using *finite-RP*[*OF* ‹*path M q pM*› *t2 pass1* ‹*completely-specified M′*› ‹*inputs M′ = inputs M*›] **by** *force*

have $(\bigwedge y1.\ finite\ (io\text{-}targets\ M'\ (p\text{-}io\ y1)\ (FSM.initial\ M')))$
by (*meson io-targets-finite*)

have $(\bigwedge y1.\ io\text{-}targets\ M'\ (p\text{-}io\ y1)\ (FSM.initial\ M') \subseteq states\ M')$
by (*meson io-targets-states*)

show *?thesis*
using *distinct-union-union-card*
[*of fst dM* λ *q′* . (*RP M q q′ pP pM prs M′*) λ *pR* . *io-targets M′* (*p-io pR*) (*initial M′*)
, *OF* ‹*finite* (*fst dM*)›
no-shared-targets-for-distinct-states
no-shared-targets-for-identical-states
‹$(\bigwedge x1.\ finite\ (RP\ M\ q\ x1\ pP\ pM\ prs\ M'))$›
io-targets-finite
io-targets-states
fsm-states-finite[*of M′*]]
unfolding ‹(*target* (*FSM.initial M*) *pP*) = *q*› **by** *force*
qed

moreover have $card\ (states\ M') \leq m$
using ‹$size\ M' \leq m$› **by** *auto*

ultimately show *False*
by *linarith*
qed

41.6 Completeness of Sufficient Test Suites

This subsection combines the soundness and exhaustiveness properties of sufficient test suites to show completeness: for any System Under Test with at most m states a test suite sufficient for m passes if and only if the System Under Test is a reduction of the specification.

lemma *passes-test-suite-completeness* :

assumes *implies-completeness T M m*
and *observable M*
and *observable M′*
and *inputs M′ = inputs M*
and *inputs M ≠ {}*
and *completely-specified M*
and *completely-specified M′*
and *size M′ ≤ m*
shows *(L M′ ⊆ L M) ⟷ passes-test-suite M T M′*
using *passes-test-suite-exhaustiveness[OF - - assms(2−8)]*
passes-test-suite-soundness[OF - assms(2,3,4,6)]
assms(1)
test-suite.exhaust[of T]
by *metis*

41.7 Additional Test Suite Properties

fun *is-finite-test-suite* :: *('a,'b,'c,'d) test-suite ⇒ bool* **where**
is-finite-test-suite (Test-Suite prs tps rd-targets separators) =
((finite prs) ∧ (∀ q p . q ∈ fst ‘ prs ⟶ finite (rd-targets (q,p))) ∧ (∀ q q′ . finite (separators (q,q′))))

end

42 Representing Test Suites as Sets of Input-Output Sequences

This theory describes the representation of test suites as sets of input-output sequences and defines a pass relation for this representation.

theory *Test-Suite-IO*
imports *Test-Suite Maximal-Path-Trie*
begin

fun *test-suite-to-io* :: *('a,'b,'c) fsm ⇒ ('a,'b,'c,'d) test-suite ⇒ ('b × 'c) list set* **where**
test-suite-to-io M (Test-Suite prs tps rd-targets atcs) =
(⋃ (q,P) ∈ prs . L P)
∪ (⋃{(λ io′ . p-io p @ io′) ‘ (set (prefixes (p-io pt))) | p pt . ∃ q P . (q,P) ∈ prs ∧ path P (initial P) p ∧ target (initial P) p = q ∧ pt ∈ tps q})
∪ (⋃{(λ io-atc . p-io p @ p-io pt @ io-atc) ‘ (atc-to-io-set (from-FSM M (target q pt)) A) | p pt q A . ∃ P q′ t1 t2 . (q,P) ∈ prs ∧ path P (initial P) p ∧ target (initial P) p = q ∧ pt ∈ tps q ∧ q′ ∈ rd-targets (q,pt) ∧ (A,t1,t2) ∈ atcs (target q pt,q′) })

lemma *test-suite-to-io-language* :
assumes *implies-completeness T M m*

shows (*test-suite-to-io M T*) $\subseteq$ *L M*
proof
fix *io* **assume** *io* $\in$ *test-suite-to-io M T*

obtain *prs tps rd-targets atcs* **where** *T* = *Test-Suite prs tps rd-targets atcs*
by (*meson test-suite.exhaust*)

then obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets atcs*) *M m repetition-sets*
using *assms*(*1*) **unfolding** *implies-completeness-def*
by *blast*
then have *implies-completeness* (*Test-Suite prs tps rd-targets atcs*) *M m*
unfolding *implies-completeness-def*
by *blast*

have *t2*: $\bigwedge$ *q P*. (*q*, *P*) $\in$ *prs* $\Longrightarrow$ *is-preamble P M q*
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*]

by *blast*

have *t5*: $\bigwedge$*q*. *q* $\in$ *FSM.states M* $\Longrightarrow$ ($\exists$ *d*$\in$*set repetition-sets*. *q* $\in$ *fst d*)
using *implies-completeness-for-repetition-sets-simps*(*4*)[*OF repetition-sets-def*]

by *assumption*

have *t6*: $\bigwedge$ *q*. *q* $\in$ *fst* ‘ *prs* $\Longrightarrow$ *tps q* $\subseteq$ {*p1* . $\exists$ *p2 d* . (*p1*@*p2*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*} $\wedge$ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) $\subseteq$ *tps q*
using *implies-completeness-for-repetition-sets-simps*(*7*)[*OF repetition-sets-def*]

by *assumption*

have *t8*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
using *implies-completeness-for-repetition-sets-simps*(*5*,*6*)[*OF repetition-sets-def*]

by *blast*

from ‹*io* $\in$ *test-suite-to-io M T*› **consider**
(*a*) *io* $\in$ ($\bigcup$ (*q*,*P*) $\in$ *prs* . *L P*) |
(*b*) *io* $\in$ ($\bigcup${(λ *io*′ . *p-io p* @ *io*′) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . $\exists$ *q P* . (*q*,*P*) $\in$ *prs* $\wedge$ *path P* (*initial P*) *p* $\wedge$ *target* (*initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q*}) |
(*c*) *io* $\in$ ($\bigcup${(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . $\exists$ *P q*′ *t1 t2* . (*q*,*P*) $\in$ *prs* $\wedge$ *path P* (*initial P*) *p* $\wedge$ *target* (*initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q*′ $\in$ *rd-targets* (*q*,*pt*) $\wedge$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q*′) })
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*› *test-suite-to-io.simps*
by *blast*

then show *io* ∈ *L M* **proof** *cases*
case *a*
then obtain *q P* **where** (*q*, *P*) ∈ *prs* **and** *io* ∈ *L P*
by *blast*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›] **unfolding** *is-preamble-def* **by** *blast*

show *io* ∈ *L M*
using *submachine-language*[*OF* ‹*is-submachine P M*›] ‹*io* ∈ *L P*› **by** *blast*
next
case *b*
then obtain *p pt q P* **where** *io* ∈ (λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))
and (*q*,*P*) ∈ *prs*
and *path P* (*initial P*) *p*
and *target* (*initial P*) *p* = *q*
and *pt* ∈ *tps q*
by *blast*

then obtain *io′* **where** *io* = *p-io p* @ *io′* **and** *io′* ∈ (*set* (*prefixes* (*p-io pt*)))
by *blast*

then obtain *io″* **where** *p-io pt* = *io′* @ *io″* **and** *io* = *p-io p* @ *io′*
unfolding *prefixes-set* **using** ‹*io′* ∈ *set* (*prefixes* (*p-io pt*))› *prefixes-set-ob* **by** *blast*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*,*P*) ∈ *prs*›
by *force*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]
unfolding *is-preamble-def*
by *blast*
then have *initial P* = *initial M*
by *auto*

have *path M* (*initial M*) *p*
using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *p*›]
unfolding ‹*initial P* = *initial M*›
by *assumption*

have *target* (*initial M*) *p* = *q*
using ‹*target* (*initial P*) *p* = *q*›
unfolding ‹*initial P* = *initial M*›
by *assumption*

obtain *p2 d* **where** (*pt* @ *p2*, *d*) ∈ *m-traversal-paths-with-witness M q repeti-*

tion-sets m
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹*pt* ∈ *tps q*›
by *blast*
then have *path M q* (*pt* @ *p2*)
using *m-traversal-paths-with-witness-set*[*OF t5 t8 path-target-is-state*[*OF* ‹*path M* (*initial M*) *p*›], *of m*]
unfolding ‹*target* (*initial M*) *p* = *q*›
by *blast*
then have *path M* (*initial M*) (*p*@*pt*)
using ‹*path M* (*initial M*) *p*› ‹*target* (*initial M*) *p* = *q*›
by *auto*
then have *p-io p* @ *p-io pt* ∈ *L M*
by (*metis* (*mono-tags*, *lifting*) *language-intro map-append*)

then show *io* ∈ *L M*
unfolding ‹*io* = *p-io p* @ *io*′› ‹*p-io pt* = *io*′ @ *io*″› *append.assoc*[*symmetric*]
using *language-prefix*[*of p-io p* @ *io*′ *io*″ *M initial M*]
by *blast*
next
case *c*

then obtain *p pt q A P q*′ *t1 t2* **where** *io* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
and (*q*,*P*) ∈ *prs*
and *path P* (*initial P*) *p*
and *target* (*initial P*) *p* = *q*
and *pt* ∈ *tps q*
and *q*′ ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q*′)
by *blast*

obtain *ioA* **where** *io* = *p-io p* @ *p-io pt* @ *ioA*
and *ioA* ∈ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
using ‹*io* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)›
by *blast*
then have *ioA* ∈ *L* (*from-FSM M* (*target q pt*))
unfolding *atc-to-io-set.simps* **by** *blast*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*,*P*) ∈ *prs*› **by** *force*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›] **unfolding** *is-preamble-def* **by** *blast*
then have *initial P* = *initial M* **by** *auto*

have *path M* (*initial M*) *p*

using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *p*›]
unfolding ‹*initial P* = *initial M*›
by *assumption*
have *target* (*initial M*) *p* = *q*
using ‹*target* (*initial P*) *p* = *q*›
unfolding ‹*initial P* = *initial M*›
by *assumption*

obtain *p2 d* **where** (*pt* @ *p2*, *d*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹*pt* ∈ *tps q*› **by** *blast*

then have *path M q* (*pt* @ *p2*)
using *m-traversal-paths-with-witness-set*[*OF t5 t8 path-target-is-state*[*OF* ‹*path M* (*initial M*) *p*›], *of m*]
unfolding ‹*target* (*initial M*) *p* = *q*›
by *blast*
then have *path M* (*initial M*) (*p*@*pt*)
using ‹*path M* (*initial M*) *p*› ‹*target* (*initial M*) *p* = *q*›
by *auto*
moreover have (*target q pt*) = *target* (*initial M*) (*p*@*pt*)
using ‹*target* (*initial M*) *p* = *q*›
by *auto*
ultimately have (*target q pt*) ∈ *states M*
using *path-target-is-state*
by *metis*

have *ioA* ∈ *LS M* (*target q pt*)
using *from-FSM-language*[*OF* ‹(*target q pt*) ∈ *states M*›] ‹*ioA* ∈ *L* (*from-FSM M* (*target q pt*))›
by *blast*
then obtain *pA* **where** *path M* (*target q pt*) *pA* **and** *p-io pA* = *ioA*
by *auto*
then have *path M* (*initial M*) (*p* @ *pt* @ *pA*)
using ‹*path M* (*initial M*) (*p*@*pt*)› **unfolding** ‹(*target q pt*) = *target* (*initial M*) (*p*@*pt*)›
by *auto*
then have *p-io p* @ *p-io pt* @ *ioA* ∈ *L M*
unfolding ‹*p-io pA* = *ioA*›[*symmetric*]
using *language-intro* **by** *fastforce*
then show *io* ∈ *L M*
unfolding ‹*io* = *p-io p* @ *p-io pt* @ *ioA*›
by *assumption*
qed
qed

lemma *minimal-io-seq-to-failure* :
assumes ¬ (*L M′* ⊆ *L M*)

and *inputs* $M' = inputs\ M$
and *completely-specified* M
obtains $io\ x\ y\ y'$ **where** $io@[(x,y)] \in L\ M$ **and** $io@[(x,y')] \notin L\ M$ **and** $io@[(x,y')] \in L\ M'$
proof −
obtain ioF **where** $ioF \in L\ M'$ **and** $ioF \notin L\ M$
using *assms*(*1*) **by** *blast*

let $?prefs = \{ioF' \in set\ (prefixes\ ioF)\ .\ ioF' \in L\ M' \wedge ioF' \notin L\ M\}$
have *finite ?prefs*
using *prefixes-finite* **by** *auto*
moreover have $?prefs \neq \{\}$
unfolding *prefixes-set* **using** ‹$ioF \in L\ M'$› ‹$ioF \notin L\ M$› **by** *auto*
ultimately obtain ioF' **where** $ioF' \in ?prefs$ **and** $\bigwedge ioF''\ .\ ioF'' \in ?prefs \Longrightarrow length\ ioF' \leq length\ ioF''$
by (*meson leI min-length-elem*)

then have $ioF' \in L\ M'$ **and** $ioF' \notin L\ M$
by *auto*
then have $ioF' \neq []$
by *auto*
then have $ioF' = (butlast\ ioF')@[last\ ioF']$ **and** $length\ (butlast\ ioF') < length\ ioF'$
by *auto*
then have $butlast\ ioF' \notin ?prefs$
using ‹$\bigwedge ioF''\ .\ ioF'' \in ?prefs \Longrightarrow length\ ioF' \leq length\ ioF''$› *leD* **by** *blast*
moreover have $butlast\ ioF' \in L\ M'$
using ‹$ioF' \in L\ M'$› *language-prefix*[*of butlast* ioF' [*last* ioF'] M' *initial* M']
unfolding ‹$ioF' = (butlast\ ioF')@[last\ ioF']$›[*symmetric*] **by** *blast*
moreover have $butlast\ ioF' \in set\ (prefixes\ ioF)$
using ‹$ioF' = (butlast\ ioF')@[last\ ioF']$› ‹$ioF' \in ?prefs$› *prefixes-set*
proof −
have $\exists ps.\ (butlast\ ioF'\ @\ [last\ ioF'])\ @\ ps = ioF$
using ‹$ioF' = butlast\ ioF'\ @\ [last\ ioF']$› ‹$ioF' \in \{ioF' \in set\ (prefixes\ ioF).\ ioF' \in L\ M' \wedge ioF' \notin L\ M\}$›
unfolding *prefixes-set*
by *auto*
then show *?thesis*
using *prefixes-set* **by** *fastforce*
qed
ultimately have $butlast\ ioF' \in L\ M$
by *blast*

have $*$: $(butlast\ ioF')@[(fst\ (last\ ioF'),\ snd\ (last\ ioF'))] \in L\ M'$
using ‹$ioF' \in L\ M'$› ‹$ioF' = (butlast\ ioF')@[last\ ioF']$› **by** *auto*
have $**$: $(butlast\ ioF')@[(fst\ (last\ ioF'),\ snd\ (last\ ioF'))] \notin L\ M$
using ‹$ioF' \notin L\ M$› ‹$ioF' = (butlast\ ioF')@[last\ ioF']$› **by** *auto*

obtain p **where** *path M* (*initial M*) p **and** *p-io* p = *butlast ioF′*
using ‹*butlast ioF′* ∈ *L M*› **by** *auto*
moreover obtain t **where** t ∈ *transitions M*
and *t-source* t = *target* (*initial M*) p
and *t-input* t = *fst* (*last ioF′*)
proof −
have *fst* (*last ioF′*) ∈ *inputs M′*
using *language-io*(*1*)[*OF* ∗, *of fst* (*last ioF′*) *snd* (*last ioF′*)]
by *simp*
then have *fst* (*last ioF′*) ∈ *inputs M*
using *assms*(*2*) **by** *auto*
then show *?thesis*
using *that* ‹*completely-specified M*› *path-target-is-state*[*OF* ‹*path M* (*initial M*) *p*›]
unfolding *completely-specified.simps* **by** *blast*
qed
ultimately have ∗∗∗: (*butlast ioF′*)@[(*fst* (*last ioF′*), *t-output t*)] ∈ *L M*
proof −
have *p-io* (*p* @ [*t*]) ∈ *L M*
by (*metis* (*no-types*) ‹*path M* (*FSM.initial M*) *p*› ‹*t* ∈ *FSM.transitions M*› ‹*t-source t* = *target* (*FSM.initial M*) *p*›
language-intro path-append single-transition-path)
then show *?thesis*
by (*simp add*: ‹*p-io p* = *butlast ioF′*› ‹*t-input t* = *fst* (*last ioF′*)›)
qed

show *?thesis*
using *that*[*OF* ∗∗∗ ∗∗ ∗]
by *assumption*
qed

lemma *observable-minimal-path-to-failure* :
assumes ¬ (*L M′* ⊆ *L M*)
and *observable M*
and *observable M′*
and *inputs M′* = *inputs M*
and *completely-specified M*
and *completely-specified M′*
obtains p p' t t' **where** *path M* (*initial M*) (*p*@[*t*])
and *path M′* (*initial M′*) (*p′*@[*t′*])
and *p-io p′* = *p-io p*
and *t-input t′* = *t-input t*
and ¬(∃ *t″* . *t″* ∈ *transitions M* ∧ *t-source t″* = *target* (*initial M*) *p* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t′*)
proof −

obtain *io x y y′* **where** *io*@[(*x*,*y*)] ∈ *L M* **and** *io*@[(*x*,*y′*)] ∉ *L M* **and** *io*@[(*x*,*y′*)]

$\in$ *L M′*
using *minimal-io-seq-to-failure*[*OF assms*(*1*,*4*,*5*)] **by** *blast*

obtain *p t* **where** *path M* (*initial M*) (*p*@[*t*]) **and** *p-io p* = *io* **and** *t-input t* = *x* **and** *t-output t* = *y*
using *language-append-path-ob*[*OF* ‹*io*@[(*x*,*y*)] $\in$ *L M*›] **by** *blast*

moreover obtain *p′ t′* **where** *path M′* (*initial M′*) (*p′*@[*t′*]) **and** *p-io p′* = *io* **and** *t-input t′* = *x* **and** *t-output t′* = *y′*
using *language-append-path-ob*[*OF* ‹*io*@[(*x*,*y′*)] $\in$ *L M′*›] **by** *blast*

moreover have ¬(∃ *t″* . *t″* $\in$ *transitions M* ∧ *t-source t″* = *target* (*initial M*) *p* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t′*)
proof
assume ∃ *t″*. *t″* $\in$ *FSM.transitions M* ∧ *t-source t″* = *target* (*FSM.initial M*) *p* ∧ *t-input t″* = *t-input t* ∧ *t-output t″* = *t-output t′*
then obtain *t″* **where** *t″* $\in$ *FSM.transitions M* **and** *t-source t″* = *target* (*FSM.initial M*) *p* **and** *t-input t″* = *x* **and** *t-output t″* = *y′*
unfolding ‹*t-input t* = *x*› ‹*t-output t′* = *y′*› **by** *blast*

then have *path M* (*initial M*) (*p*@[*t″*])
using ‹*path M* (*initial M*) (*p*@[*t*])›
by (*meson path-append-elim path-append-transition*)
moreover have *p-io* (*p*@[*t″*]) = *io*@[(*x*,*y′*)]
using ‹*p-io p* = *io*› ‹*t-input t″* = *x*› ‹*t-output t″* = *y′*› **by** *auto*
ultimately have *io*@[(*x*,*y′*)] $\in$ *L M*
using *language-state-containment*
by (*metis* (*mono-tags*, *lifting*))
then show *False*
using ‹*io*@[(*x*,*y′*)] $\notin$ *L M*› **by** *blast*
qed

ultimately show *?thesis* **using** *that*[*of p t p′ t′*]
by *force*
qed

lemma *test-suite-to-io-pass* :
assumes *implies-completeness T M m*
and *observable M*
and *observable M′*
and *inputs M′* = *inputs M*
and *inputs M* ≠ {}
and *completely-specified M*
and *completely-specified M′*
shows *pass-io-set M′* (*test-suite-to-io M T*) = *passes-test-suite M T M′*
proof −
obtain *prs tps rd-targets atcs* **where** *T* = *Test-Suite prs tps rd-targets atcs*

by (*meson test-suite.exhaust*)
then obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets atcs*) *M m repetition-sets*
using *assms*(*1*) **unfolding** *implies-completeness-def* **by** *blast*
then have *implies-completeness* (*Test-Suite prs tps rd-targets atcs*) *M m*
unfolding *implies-completeness-def* **by** *blast*
then have *test-suite-language-prop*: *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*) $\subseteq$ *L M*
using *test-suite-to-io-language* **by** *blast*

have *t1*: (*initial M*, *initial-preamble M*) $\in$ *prs*
using *implies-completeness-for-repetition-sets-simps*(*1*)[*OF repetition-sets-def*]

by *assumption*
have *t2*: $\bigwedge$ *q P*. (*q*, *P*) $\in$ *prs* $\Longrightarrow$ *is-preamble P M q*
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*]

by *blast*
have *t3*: $\bigwedge$ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) $\in$ *atcs* (*q1*, *q2*) $\Longrightarrow$ (*A*, *d2*, *d1*) $\in$ *atcs* (*q2*, *q1*) $\wedge$ *is-separator M q1 q2 A d1 d2*
using *implies-completeness-for-repetition-sets-simps*(*3*)[*OF repetition-sets-def*]

by *assumption*

have *t5*: $\bigwedge$*q*. *q* $\in$ *FSM.states M* $\Longrightarrow$ ($\exists$ *d*$\in$*set repetition-sets*. *q* $\in$ *fst d*)
using *implies-completeness-for-repetition-sets-simps*(*4*)[*OF repetition-sets-def*]

by *assumption*

have *t6*: $\bigwedge$ *q*. *q* $\in$ *fst* ‘ *prs* $\Longrightarrow$ *tps q* $\subseteq$ {*p1* . $\exists$ *p2 d* . (*p1*@*p2*,*d*) $\in$ *m-traversal-paths-with-witness M q repetition-sets m*} $\wedge$ *fst* ‘ (*m-traversal-paths-with-witness M q repetition-sets m*) $\subseteq$ *tps q*
using *implies-completeness-for-repetition-sets-simps*(*7*)[*OF repetition-sets-def*]

by *assumption*

have *t7*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *fst d* $\subseteq$ *FSM.states M*
and *t8*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *snd d* $\subseteq$ *fst d*
and *t8'*: $\bigwedge$ *d*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *snd d* = *fst d* $\cap$ *fst* ‘ *prs*
and *t9*: $\bigwedge$ *d q1 q2*. *d* $\in$ *set repetition-sets* $\Longrightarrow$ *q1* $\in$ *fst d* $\Longrightarrow$ *q2* $\in$ *fst d* $\Longrightarrow$ *q1* $\neq$ *q2* $\Longrightarrow$ *atcs* (*q1*, *q2*) $\neq$ {}
using *implies-completeness-for-repetition-sets-simps*(*5*,*6*)[*OF repetition-sets-def*]

by *blast*+

have *pass-io-set M'* (*test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)) $\Longrightarrow$ *passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M'*

proof −
assume *pass-io-set M′* (*test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*))

then have *pass-io-prop*: $\bigwedge$ *io x y y′* . *io* @ [(*x*, *y*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*) ⟹ *io* @ [(*x*, *y′*)] ∈ *L M′* ⟹ *io* @ [(*x*, *y′*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *pass-io-set-def*
by *blast*

show *passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*
proof (*rule ccontr*)
assume ¬ *passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*

then consider (*a*) ¬ (∀ *q P io x y y′*. (*q*, *P*) ∈ *prs* ⟶ *io* @ [(*x*, *y*)] ∈ *L P* ⟶ *io* @ [(*x*, *y′*)] ∈ *L M′* ⟶ *io* @ [(*x*, *y′*)] ∈ *L P*) |
(*b*) ¬ ((∀ *q P pP ioT pT x y y′*.
(*q*, *P*) ∈ *prs* ⟶
path P (*FSM.initial P*) *pP* ⟶
target (*FSM.initial P*) *pP* = *q* ⟶
pT ∈ *tps q* ⟶
ioT @ [(*x*, *y*)] ∈ *set* (*prefixes* (*p-io pT*)) ⟶
p-io pP @ *ioT* @ [(*x*, *y′*)] ∈ *L M′* ⟶ (∃ *pT′*. *pT′* ∈ *tps q* ∧ *ioT* @ [(*x*, *y′*)] ∈ *set* (*prefixes* (*p-io pT′*))))) |
(*c*) ¬ ((∀ *q P pP pT*.
(*q*, *P*) ∈ *prs* ⟶
path P (*FSM.initial P*) *pP* ⟶
target (*FSM.initial P*) *pP* = *q* ⟶
pT ∈ *tps q* ⟶
p-io pP @ *p-io pT* ∈ *L M′* ⟶
(∀ *q′ A d1 d2 qT*.
q′ ∈ *rd-targets* (*q*, *pT*) ⟶
(*A*, *d1*, *d2*) ∈ *atcs* (*target q pT*, *q′*) ⟶ *qT* ∈ *io-targets M′* (*p-io pP* @ *p-io pT*) (*FSM.initial M′*) ⟶ *pass-separator-ATC M′ A qT d2*)))

unfolding *passes-test-suite.simps* **by** *blast*
then show *False* **proof** *cases*
case *a*
then obtain *q P io x y y′* **where** (*q*, *P*) ∈ *prs*
and *io* @ [(*x*, *y*)] ∈ *L P*
and *io* @ [(*x*, *y′*)] ∈ *L M′*
and *io* @ [(*x*, *y′*)] ∉ *L P*
by *blast*

have *is-preamble P M q*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›] **by** *assumption*

have *io* @ [(*x*, *y*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)

unfolding *test-suite-to-io.simps* **using** ‹$(q, P) \in prs$› ‹io @ $[(x, y)] \in L$ P›
by *fastforce*

have io @ $[(x, y')] \in$ *test-suite-to-io* M (*Test-Suite prs tps rd-targets atcs*)
using *pass-io-prop*[*OF* ‹io @ $[(x, y)] \in$ *test-suite-to-io* M (*Test-Suite prs tps rd-targets atcs*)› ‹io @ $[(x, y')] \in L\ M'$›]
by *assumption*

then have io @ $[(x, y')] \in L\ M$
using *test-suite-language-prop*
by *blast*

have io @ $[(x, y')] \in L\ P$
using *passes-test-suite-soundness-helper-1*[*OF* ‹*is-preamble* $P\ M\ q$› ‹*observable* M› ‹io @ $[(x, y)] \in L\ P$› ‹io @ $[(x, y')] \in L\ M$›]
by *assumption*
then show *False*
using ‹io @ $[(x, y')] \notin L\ P$›
by *blast*

next
case b
then obtain $q\ P\ pP\ ioT\ pT\ x\ y\ y'$ **where** $(q, P) \in prs$
and *path* P (*FSM.initial* P) pP
and *target* (*FSM.initial* P) $pP = q$
and $pT \in tps\ q$
and ioT @ $[(x, y)] \in$ *set* (*prefixes* (*p-io* pT))
and *p-io* pP @ ioT @ $[(x, y')] \in L\ M'$
and $\neg$ ($\exists\, pT'.\ pT' \in tps\ q \wedge ioT$ @ $[(x, y')] \in$ *set* (*prefixes* (*p-io* pT')))
by *blast*

have $\exists\, q\ P.\ (q, P) \in prs \wedge$ *path* P (*FSM.initial* P) $pP \wedge$ *target* (*FSM.initial* P) $pP = q \wedge pT \in tps\ q$
using ‹$(q, P) \in prs$› ‹*path* P (*FSM.initial* P) pP› ‹*target* (*FSM.initial* P) $pP = q$› ‹$pT \in tps\ q$› **by** *blast*
moreover have *p-io* pP @ ioT @ $[(x, y)] \in$ (@) (*p-io* pP) ‘ *set* (*prefixes* (*p-io* pT))
using ‹ioT @ $[(x, y)] \in$ *set* (*prefixes* (*p-io* pT))› **by** *auto*
ultimately have *p-io* pP @ ioT @ $[(x, y)] \in$ ($\bigcup$\{(@) (*p-io* p) ‘ *set* (*prefixes* (*p-io* pt)) |$p\ pt.\ \exists\, q\ P.\ (q, P) \in prs \wedge$ *path* P (*FSM.initial* P) $p \wedge$ *target* (*FSM.initial* P) $p = q \wedge pt \in tps\ q$\})
by *blast*
then have *p-io* pP @ ioT @ $[(x, y)] \in$ *test-suite-to-io* M (*Test-Suite prs tps rd-targets atcs*)
unfolding *test-suite-to-io.simps*
by *blast*
then have $*$: (*p-io* pP @ ioT) @ $[(x, y)] \in$ *test-suite-to-io* M (*Test-Suite*

prs tps rd-targets atcs)
by *auto*

have $**$: (*p-io pP* @ *ioT*) @ [(*x*, *y'*)] ∈ *L M'*
using ‹*p-io pP* @ *ioT* @ [(*x*, *y'*)] ∈ *L M'*› **by** *auto*

have (*p-io pP* @ *ioT*) @ [(*x*, *y'*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
using *pass-io-prop*[*OF* ∗ ∗∗] **by** *assumption*
then have (*p-io pP* @ *ioT*) @ [(*x*, *y'*)] ∈ *L M*
using *test-suite-language-prop* **by** *blast*

have *q* ∈ *states M*
using *is-preamble-is-state*[*OF t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]] **by** *assumption*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*, *P*) ∈ *prs*› **by** *force*

obtain *pT' d'* **where** (*pT* @ *pT'*, *d'*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹*pT* ∈ *tps q*› **by** *blast*

then have *path M q* (*pT* @ *pT'*)
and *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) (*pT* @ *pT'*))) *repetition-sets* = *Some d'*
and ⋀ *p' p''*. (*pT* @ *pT'*) = *p'* @ *p''* ⟹ *p''* ≠ [] ⟹ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p'*)) *repetition-sets* = *None*
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*]
by *blast*+

obtain *ioT'* **where** *p-io pT* = *ioT* @ [(*x*,*y*)] @ *ioT'*
using *prefixes-set-ob*[*OF* ‹*ioT* @ [(*x*, *y*)] ∈ *set* (*prefixes* (*p-io pT*))›]
unfolding *prefixes-set append.assoc*[*symmetric*]
by *blast*

let *?pt* = *take* (*length* (*ioT* @ [(*x*,*y*)])) *pT*
let *?p* = *butlast ?pt*
let *?t* = *last ?pt*

have *length ?pt* > *0*
using ‹*p-io pT* = *ioT* @ [(*x*,*y*)] @ *ioT'*›
unfolding *length-map*[*of* (λ *t* . (*t-input t*, *t-output t*)), *symmetric*]
by *auto*
then have *?pt* = *?p* @ [*?t*]
by *auto*
moreover have *path M q ?pt*
using ‹*path M q* (*pT* @ *pT'*)›
by (*meson path-prefix path-prefix-take*)

ultimately have *path M q* (*?p*@[*?t*])
  by *simp*

have *p-io ?p* = *ioT*
proof −
  have *p-io ?pt* = *take* (*length* (*ioT* @ [(*x*,*y*)])) (*p-io pT*)
    by (*simp add*: *take-map*)
  then have *p-io ?pt* = *ioT* @ [(*x*,*y*)]
    using ‹*p-io pT* = *ioT* @ [(*x*,*y*)] @ *ioT′*› **by** *auto*
  then show *?thesis*
    by (*simp add*: *map-butlast*)
qed

have *path M q ?p*
  using *path-append-transition-elim*[*OF* ‹*path M q* (*?p*@[*?t*])›] **by** *blast*

have *is-submachine P M*
  using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›] **unfolding** *is-preamble-def* **by** *blast*
then have *initial P* = *initial M* **by** *auto*

have *path M* (*initial M*) *pP*
  using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *pP*›]
  unfolding ‹*initial P* = *initial M*›
  by *assumption*
moreover have *target* (*initial M*) *pP* = *q*
  using ‹*target* (*initial P*) *pP* = *q*›
  unfolding ‹*initial P* = *initial M*›
  by *assumption*
ultimately have *path M* (*initial M*) (*pP*@*?p*)
  using ‹*path M q ?p*›
  by *auto*

have *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *?p*)) *repetition-sets* = *None*
proof −
  have ∗: (*pT* @ *pT′*) = *?p* @ ([*?t*] @ (*drop* (*length* (*ioT* @ [(*x*,*y*)])) *pT*) @ *pT′*)
    using ‹*?pt* = *?p* @ [*?t*]›
    by (*metis append-assoc append-take-drop-id*)
  have ∗∗: ([*?t*] @ (*drop* (*length* (*ioT* @ [(*x*,*y*)])) *pT*) @ *pT′*) ≠ []
    by *simp*
  show *?thesis*
    using ‹⋀ *p′ p″*. (*pT* @ *pT′*) = *p′* @ *p″* ⟹ *p″* ≠ [] ⟹ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p′*)) *repetition-sets* = *None*›[*OF* ∗ ∗∗]
    by *assumption*
qed

obtain p' t' **where** *path M* (*FSM.initial M*) (p' @ [t']) **and** *p-io* p' = *p-io* *pP* @ *ioT* **and** *t-input* t' = x **and** *t-output* t' = y'
using *language-append-path-ob*[*OF* ‹(*p-io pP* @ *ioT*) @ [(x, y')] ∈ *L M*›] **by** *blast*
then have *path M* (*FSM.initial M*) p' **and** *t-source* t' = *target* (*initial M*) p' **and** t' ∈ *transitions M*
by *auto*

have p' = *pP* @ *?p*
using *observable-path-unique*[*OF* ‹*observable M*› ‹*path M* (*FSM.initial M*) p'› ‹*path M* (*initial M*) (*pP*@*?p*)›]
‹*p-io ?p* = *ioT*›
unfolding ‹*p-io* p' = *p-io pP* @ *ioT*›
by *simp*
then have *t-source* t' = *target q ?p*
unfolding ‹*t-source* t' = *target* (*initial M*) p'› **using** ‹*target* (*initial M*) *pP* = *q*›
by *auto*

obtain *pTt′ dt′* **where** (*?p* @ [t'] @ *pTt′*, *dt′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *m-traversal-path-extension-exist-for-transition*[*OF* ‹*completely-specified M*› ‹*q* ∈ *states M*› ‹*FSM.inputs M* ≠ {}›
t5 t8 ‹*path M q ?p*› ‹*find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *?p*)) *repetition-sets* = *None*›
‹t' ∈ *transitions M*› ‹*t-source* t' = *target q ?p*›]
by *blast*

have *?p* @ [t'] @ *pTt′* ∈ *tps q*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹(*?p* @ [t'] @ *pTt′*, *dt′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*›
by *force*
moreover have *ioT* @ [(x, y')] ∈ *set* (*prefixes* (*p-io* (*?p* @ [t'] @ *pTt′*)))
proof −
have *p-io* (*?p* @ [t'] @ *pTt′*) = *ioT* @ [(x,y')] @ *p-io pTt′*
using ‹*p-io ?p* = *ioT*› **using** ‹*t-input* t' = x› ‹*t-output* t' = y'›
by *auto*
then show *?thesis*
unfolding *prefixes-set*
by *force*
qed

ultimately show *False*
using ‹¬ (∃ *pT′*. *pT′* ∈ *tps q* ∧ *ioT* @ [(x, y')] ∈ *set* (*prefixes* (*p-io pT′*)))›

by *blast*
next
case *c*

then obtain *q P pP pT q′ A d1 d2 qT* **where** $(q, P) \in prs$
and *path P* (*FSM.initial P*) *pP*
and *target* (*FSM.initial P*) *pP* = *q*
and $pT \in tps\ q$
and *p-io pP* @ *p-io pT* ∈ *L M′*
and $q' \in rd\text{-}targets\ (q, pT)$
and $(A, d1, d2) \in atcs\ (target\ q\ pT, q')$
and *qT* ∈ *io-targets M′* (*p-io pP* @ *p-io pT*) (*FSM.initial M′*)
and ¬*pass-separator-ATC M′ A qT d2*
by *blast*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]
unfolding *is-preamble-def*
by *blast*
then have *initial P* = *initial M* **by** *auto*

have *path M* (*initial M*) *pP*
using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *pP*›]
unfolding ‹*initial P* = *initial M*›
by *assumption*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*›
unfolding ‹*initial P* = *initial M*›
by *assumption*

have *q* ∈ *states M*
using *is-preamble-is-state*[*OF t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]]
by *assumption*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*, *P*) ∈ *prs*› **by** *force*

obtain *pT′ d′* **where** (*pT* @ *pT′*, *d′*) ∈ *m-traversal-paths-with-witness M q repetition-sets m*
using *t6*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] ‹*pT* ∈ *tps q*› **by** *blast*

then have *path M q* (*pT* @ *pT′*)
and *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) (*pT* @ *pT′*))) *repetition-sets* = *Some d′*
and ⋀ *p′ p″*. (*pT* @ *pT′*) = *p′* @ *p″* ⟹ *p″* ≠ [] ⟹ *find* (λ*d*. *Suc* (*m* − *card* (*snd d*)) ≤ *length* (*filter* (λ*t*. *t-target t* ∈ *fst d*) *p′*)) *repetition-sets* = *None*
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹*q* ∈ *states M*›, *of m*]

by *blast+*
then have *path M q pT*
by *auto*

have $qT \in$ *states* M'
using ‹$qT \in$ *io-targets* M' (*p-io pP* @ *p-io pT*) (*FSM.initial* M')› *io-targets-states subset-iff*
by *fastforce*

have *is-separator M* (*target q pT*) q' *A d1 d2*
using *t3*[*OF* ‹(*A*, *d1*, *d2*) $\in$ *atcs* (*target q pT*, q')›] **by** *blast*

have $\neg$ *pass-io-set* (*FSM.from-FSM* M' *qT*) (*atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*)
using ‹$\neg$*pass-separator-ATC* M' *A qT d2*›
pass-separator-pass-io-set-iff[*OF* ‹*is-separator M* (*target q pT*) q' *A d1 d2*› ‹*observable M*›
‹*observable* M'› *path-target-is-state*[*OF* ‹*path M q pT*›]
‹$qT \in$ *states* M'› ‹*inputs* M' = *inputs M*› ‹*completely-specified M*›]
by *simp*

have *pass-io-set* (*FSM.from-FSM* M' *qT*) (*atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*)
proof −
have $\bigwedge$ *io x y* y' . *io* @ [(*x*, *y*)] $\in$ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A* $\Longrightarrow$
io @ [(*x*, y')] $\in$ *L* (*FSM.from-FSM* M' *qT*) $\Longrightarrow$
io @ [(*x*, y')] $\in$ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*
proof −
fix *io x y* y' **assume** *io* @ [(*x*, *y*)] $\in$ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*
and *io* @ [(*x*, y')] $\in$ *L* (*FSM.from-FSM* M' *qT*)

define *tmp* **where** *tmp-def* : *tmp* = ($\bigcup$ {(λ*io-atc*. *p-io p* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A* |*p pt q A*. $\exists$ *P* q' *t1 t2*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ $q' \in$ *rd-targets* (*q*, *pt*) $\wedge$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, q')})
define *tmp2* **where** *tmp2-def* : *tmp2* = $\bigcup$ {(@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*)) |*p pt*. $\exists$ *q P*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q*}
have $\exists$ *P* q' *t1 t2*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *pP* $\wedge$ *target*

(*FSM.initial P*) *pP* = *q* ∧ *pT* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*, *pT*) ∧ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pT*, *q′*)
using ‹(*q*, *P*) ∈ *prs*› ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› ‹*pT* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*, *pT*)› ‹(*A*, *d1*, *d2*) ∈ *atcs* (*target q pT*, *q′*)› **by** *blast*
then have (λ*io-atc*. *p-io pP* @ *p-io pT* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A* ⊆ *tmp*
unfolding *tmp-def* **by** *blast*
then have (λ*io-atc*. *p-io pP* @ *p-io pT* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A* ⊆ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *test-suite-to-io.simps tmp-def*[*symmetric*] *tmp2-def*[*symmetric*] **by** *blast*
moreover have (*p-io pP* @ *p-io pT* @ (*io* @ [(*x*, *y*)])) ∈ (λ*io-atc*. *p-io pP* @ *p-io pT* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*
using ‹*io* @ [(*x*, *y*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*› **by** *auto*
ultimately have (*p-io pP* @ *p-io pT* @ (*io* @ [(*x*, *y*)])) ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
by *blast*
then have ∗: (*p-io pP* @ *p-io pT* @ *io*) @ [(*x*, *y*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
by *simp*

have *io* @ [(*x*, *y′*)] ∈ *LS M′ qT*
using ‹*io* @ [(*x*, *y′*)] ∈ *L* (*FSM.from-FSM M′ qT*)› ‹*qT* ∈ *states M′*›
by (*metis from-FSM-language*)
have ∗∗: (*p-io pP* @ *p-io pT* @ *io*) @ [(*x*, *y′*)] ∈ *L M′*
using *io-targets-language-append*[*OF* ‹*qT* ∈ *io-targets M′* (*p-io pP* @ *p-io pT*) (*FSM.initial M′*)› ‹*io* @ [(*x*, *y′*)] ∈ *LS M′ qT*›]
by *simp*

have (*p-io pP* @ *p-io pT*) @ (*io* @ [(*x*, *y′*)]) ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
using *pass-io-prop*[*OF* ∗ ∗∗] **by** *simp*
then have (*p-io pP* @ *p-io pT*) @ (*io* @ [(*x*, *y′*)]) ∈ *L M*
using *test-suite-language-prop* **by** *blast*

moreover have *target q pT* ∈ *io-targets M* (*p-io pP* @ *p-io pT*) (*initial M*)
proof −
have *target* (*initial M*) (*pP*@*pT*) = *target q pT*
unfolding ‹*target* (*initial M*) *pP* = *q*›[*symmetric*] **by** *auto*
moreover have *path M* (*initial M*) (*pP*@*pT*)
using ‹*path M* (*initial M*) *pP*› ‹*path M q pT*› **unfolding** ‹*target* (*initial M*) *pP* = *q*›[*symmetric*]
by *auto*
moreover have *p-io* (*pP*@*pT*) = (*p-io pP* @ *p-io pT*)
by *auto*

ultimately show *?thesis*
unfolding *io-targets.simps*
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq*)
qed

ultimately have *io* @ [(*x*, *y′*)] ∈ *LS M* (*target q pT*)
using *observable-io-targets-language*[*OF* - ‹*observable M*›, *of* (*p-io pP* @ *p-io pT*) (*io* @ [(*x*, *y′*)]) *initial M target q pT*]
by *blast*
then have *io* @ [(*x*, *y′*)] ∈ *L* (*FSM.from-FSM M* (*target q pT*))
unfolding *from-FSM-language*[*OF path-target-is-state*[*OF* ‹*path M q pT*›]]
by *assumption*

moreover have *io* @ [(*x*, *y′*)] ∈ *L A*
by (*metis Int-iff* ‹*io* @ [(*x*, *y′*)] ∈ *LS M* (*target q pT*)› ‹*io* @ [(*x*, *y*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*›
‹*is-separator M* (*target q pT*) *q′ A d1 d2*› *atc-to-io-set.simps is-separator-simps*(*9*))

ultimately show *io* @ [(*x*, *y′*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pT*)) *A*
unfolding *atc-to-io-set.simps* **by** *blast*
qed

then show *?thesis* **unfolding** *pass-io-set-def* **by** *blast*
qed

then show *False*
using *pass-separator-from-pass-io-set*[*OF* ‹*is-separator M* (*target q pT*) *q′ A d1 d2*› - ‹*observable M*›
‹*observable M′*› *path-target-is-state*[*OF* ‹*path M q pT*›]
‹*qT* ∈ *states M′*› ‹*inputs M′* = *inputs M*› ‹*completely-specified M*›]
‹¬*pass-separator-ATC M′ A qT d2*›
by *simp*
qed
qed
qed

moreover have *passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′* ⟹ *pass-io-set M′* (*test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*))
proof −
assume *passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*

have *pass1*: ⋀ *q P io x y y′* . (*q*,*P*) ∈ *prs* ⟹ *io*@[(*x*,*y*)] ∈ *L P* ⟹ *io*@[(*x*,*y′*)]

$\in L\ M' \Longrightarrow io@[(x,y')] \in L\ P$
using ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*›
unfolding *passes-test-suite.simps*
by *meson*

have *pass2*: $\bigwedge$ *q P pP ioT pT x y y′* . $(q,P) \in prs \Longrightarrow path\ P\ (initial\ P)\ pP \Longrightarrow target\ (initial\ P)\ pP = q \Longrightarrow pT \in tps\ q \Longrightarrow ioT@[(x,y)] \in set\ (prefixes\ (p\text{-}io\ pT)) \Longrightarrow (p\text{-}io\ pP)@ioT@[(x,y')] \in L\ M' \Longrightarrow (\exists\ pT'\ .\ pT' \in tps\ q \wedge ioT@[(x,y')] \in set\ (prefixes\ (p\text{-}io\ pT')))$
using ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*›
unfolding *passes-test-suite.simps* **by** *blast*

have *pass3*: $\bigwedge$ *q P pP pT q′ A d1 d2 qT* . $(q,P) \in prs \Longrightarrow path\ P\ (initial\ P)\ pP \Longrightarrow target\ (initial\ P)\ pP = q \Longrightarrow pT \in tps\ q \Longrightarrow (p\text{-}io\ pP)@(p\text{-}io\ pT) \in L\ M' \Longrightarrow q' \in rd\text{-}targets\ (q,pT) \Longrightarrow (A,d1,d2) \in atcs\ (target\ q\ pT,\ q') \Longrightarrow qT \in io\text{-}targets\ M'\ ((p\text{-}io\ pP)@(p\text{-}io\ pT))\ (initial\ M') \Longrightarrow pass\text{-}separator\text{-}ATC\ M'\ A\ qT\ d2$
using ‹*passes-test-suite M* (*Test-Suite prs tps rd-targets atcs*) *M′*›
unfolding *passes-test-suite.simps* **by** *blast*

show *pass-io-set M′* (*test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*))
proof (*rule ccontr*)
assume $\neg$ *pass-io-set M′* (*test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*))
then obtain *io x y y′* **where** $io\ @\ [(x,\ y)] \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
and $io\ @\ [(x,\ y')] \in L\ M'$
and $io\ @\ [(x,\ y')] \notin$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *pass-io-set-def* **by** *blast*

have *preamble-prop*: $\bigwedge$ *q P* . $(q,\ P) \in prs \Longrightarrow io\ @\ [(x,\ y)] \in L\ P \Longrightarrow False$
proof −
fix *q P* **assume** $(q,\ P) \in prs$ **and** $io\ @\ [(x,\ y)] \in L\ P$
have $io\ @\ [(x,\ y')] \in L\ P$ **using** *pass1*[*OF* ‹$(q,\ P) \in prs$› ‹$io\ @\ [(x,\ y)] \in L\ P$› ‹$io\ @\ [(x,\ y')] \in L\ M'$›]
by *assumption*
then have $io\ @\ [(x,\ y')] \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *test-suite-to-io.simps* **using** ‹$(q,\ P) \in prs$› **by** *blast*
then show *False* **using** ‹$io\ @\ [(x,\ y')] \notin$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)›
by *simp*
qed

have *traversal-path-prop* : $\bigwedge$ *pP pt q P* . *io* @ [(*x*, *y*)] ∈ (@) (*p-io pP*) ‘ *set* (*prefixes* (*p-io pt*)) ⟹ (*q*, *P*) ∈ *prs* ⟹ *path P* (*FSM.initial P*) *pP* ⟹ *target* (*FSM.initial P*) *pP* = *q* ⟹ *pt* ∈ *tps q* ⟹ *False*
proof −
fix *pP pt q P* **assume** *io* @ [(*x*, *y*)] ∈ (@) (*p-io pP*) ‘ *set* (*prefixes* (*p-io pt*))
and (*q*, *P*) ∈ *prs*
and *path P* (*FSM.initial P*) *pP*
and *target* (*FSM.initial P*) *pP* = *q*
and *pt* ∈ *tps q*

obtain *io′ io″* **where** *io* @ [(*x*, *y*)] = (*p-io pP*) @ *io′* **and** *io′*@*io″* = *p-io pt*
using ‹*io* @ [(*x*, *y*)] ∈ (@) (*p-io pP*) ‘ *set* (*prefixes* (*p-io pt*))›
unfolding *prefixes-set*
by *blast*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]
unfolding *is-preamble-def*
by *blast*
then have *initial P* = *initial M*
by *auto*

have *path M* (*initial M*) *pP*
using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *pP*›]
unfolding ‹*initial P* = *initial M*›
by *assumption*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*›
unfolding ‹*initial P* = *initial M*›
by *assumption*

have *q* ∈ *states M*
using *is-preamble-is-state*[*OF t2*[*OF* ‹(*q*, *P*) ∈ *prs*›]]
by *assumption*

have *q* ∈ *fst* ‘ *prs*
using ‹(*q*, *P*) ∈ *prs*› **by** *force*

show *False* **proof** (*cases io′ rule*: *rev-cases*)
case *Nil*
then have *p-io pP* = *io* @ [(*x*, *y*)]
using ‹*io* @ [(*x*, *y*)] = (*p-io pP*) @ *io′*›
by *auto*
show *?thesis*
using *preamble-prop*[*OF* ‹(*q*,*P*) ∈ *prs*› *language-state-containment*[*OF* ‹*path P* (*FSM.initial P*) *pP*› ‹*p-io pP* = *io* @ [(*x*, *y*)]›]]
by *assumption*

```
next
  case (snoc ioI xy)
  then have xy = (x,y) and io = (p-io pP) @ ioI
    using ‹io @ [(x, y)] = (p-io pP) @ io'› by auto
  then have p-io pP @ ioI @ [(x, y')] ∈ L M'
    using ‹io @ [(x, y')] ∈ L M'› by simp

  have ioI @ [(x, y)] ∈ set (prefixes (p-io pt))
    unfolding prefixes-set
    using ‹io' @ io'' = p-io pt› ‹xy = (x, y)› snoc
    by auto

  obtain pT' where pT' ∈ tps q and ioI @ [(x, y')] ∈ set (prefixes (p-io
pT'))
      using pass2[OF ‹(q, P) ∈ prs› ‹path P (FSM.initial P) pP› ‹target
(FSM.initial P) pP = q› ‹pt ∈ tps q›
                ‹ioI @ [(x, y)] ∈ set (prefixes (p-io pt))› ‹p-io pP @ ioI @
[(x, y')] ∈ L M'›] by blast

  have io @ [(x, y')] ∈ (@) (p-io pP) ` set (prefixes (p-io pT'))
    using ‹ioI @ [(x, y')] ∈ set (prefixes (p-io pT'))›
    unfolding ‹io = (p-io pP) @ ioI›
    by simp


  have io @ [(x, y')] ∈ (⋃ {(@) (p-io p) ` set (prefixes (p-io pt)) |p pt. ∃ q
P. (q, P) ∈ prs ∧ path P (FSM.initial P) p ∧ target (FSM.initial P) p = q ∧ pt
∈ tps q})
      using ‹(q, P) ∈ prs› ‹path P (FSM.initial P) pP› ‹target (FSM.initial
P) pP = q›
            ‹pT' ∈ tps q› ‹io @ [(x, y')] ∈ (@) (p-io pP) ` set (prefixes (p-io
pT'))›
    by blast
  then have io @ [(x, y')] ∈ test-suite-to-io M (Test-Suite prs tps rd-targets
atcs)
    unfolding test-suite-to-io.simps
    by blast

  then show False
      using ‹io @ [(x, y')] ∉ test-suite-to-io M (Test-Suite prs tps rd-targets
atcs)›
    by blast
qed
qed

consider (a) io @ [(x, y)] ∈ (⋃(q, P)∈prs. L P) |
        (b) io @ [(x, y)] ∈ (⋃ {(@) (p-io p) ` set (prefixes (p-io pt)) |p pt. ∃ q
P. (q, P) ∈ prs ∧ path P (FSM.initial P) p ∧ target (FSM.initial P) p = q ∧ pt
∈ tps q}) |
```

(c) io @ [(x, y)] ∈ (⋃ {(λio-atc. p-io p @ p-io pt @ io-atc) ‘ atc-to-io-set (FSM.from-FSM M (target q pt)) A |p pt q A.
∃ P q′ t1 t2.
(q, P) ∈ prs ∧
path P (FSM.initial P) p ∧ target (FSM.initial P) p = q ∧ pt ∈ tps q ∧ q′ ∈ rd-targets (q, pt) ∧ (A, t1, t2) ∈ atcs (target q pt, q′)})
using ‹io @ [(x, y)] ∈ test-suite-to-io M (Test-Suite prs tps rd-targets atcs)›
unfolding *test-suite-to-io.simps* **by** *blast*

then show *False* **proof** *cases*
case *a*
then show *?thesis* **using** *preamble-prop* **by** *blast*
next
case *b*
then show *?thesis* **using** *traversal-path-prop* **by** *blast*
next
case *c*

then obtain pP pt q A P q′ t1 t2 **where** io @ [(x, y)] ∈ (λio-atc. p-io pP @ p-io pt @ io-atc) ‘ atc-to-io-set (FSM.from-FSM M (target q pt)) A
and (q, P) ∈ prs
and path P (FSM.initial P) pP
and target (FSM.initial P) pP = q
and pt ∈ tps q
and q′ ∈ rd-targets (q, pt)
and (A, t1, t2) ∈ atcs (target q pt, q′)
by *blast*

obtain ioA **where** io @ [(x, y)] = p-io pP @ p-io pt @ ioA
using ‹io @ [(x, y)] ∈ (λio-atc. p-io pP @ p-io pt @ io-atc) ‘ atc-to-io-set (FSM.from-FSM M (target q pt)) A›
unfolding *prefixes-set*
by *blast*

show *False* **proof** (*cases ioA rule*: *rev-cases*)
case *Nil*
then have io @ [(x, y)] = p-io pP @ p-io pt
using ‹io @ [(x, y)] = p-io pP @ p-io pt @ ioA› **by** *simp*
then have io @ [(x, y)] ∈ (@) (p-io pP) ‘ set (prefixes (p-io pt))
unfolding *prefixes-set* **by** *blast*
show *?thesis*
using *traversal-path-prop*[*OF* ‹io @ [(x, y)] ∈ (@) (p-io pP) ‘ set (prefixes (p-io pt))› ‹(q, P) ∈ prs›
‹path P (FSM.initial P) pP› ‹target (FSM.initial P) pP = q› ‹pt ∈ tps q›]
by *assumption*
next
case (*snoc ioAI xy*)

then have *xy* = (*x*,*y*) **and** *io* = *p-io pP* @ *p-io pt* @ *ioAI*
using ‹*io* @ [(*x*, *y*)] = *p-io pP* @ *p-io pt* @ *ioA*› **by** *simp*+
then have *p-io pP* @ *p-io pt* @ *ioAI* @ [(*x*,*y*)] ∈ (λ*io-atc*. *p-io pP* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*
using ‹*io* @ [(*x*, *y*)] ∈ (λ*io-atc*. *p-io pP* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*›
by *auto*
then have *ioAI* @ [(*x*,*y*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*
by *auto*

have *p-io pP* @ *p-io pt* ∈ *L M′*
using ‹*io* @ [(*x*,*y′*)] ∈ *L M′*› *language-prefix*[*of p-io pP* @ *p-io pt ioAI* @ [(*x*, *y′*)] *M′ initial M′*]
unfolding ‹*io* = *p-io pP* @ *p-io pt* @ *ioAI*›
by *simp*
then obtain *pt′* **where** *path M′* (*initial M′*) *pt′* **and** *p-io pt′* = *p-io pP* @ *p-io pt*
by *auto*
then have *target* (*initial M′*) *pt′* ∈ *io-targets M′* (*p-io pP* @ *p-io pt*) (*FSM.initial M′*)
by *fastforce*

have *pass-separator-ATC M′ A* (*target* (*FSM.initial M′*) *pt′*) *t2*
using *pass3*[*OF* ‹(*q*, *P*) ∈ *prs*› ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› ‹*pt* ∈ *tps q*›
‹*p-io pP* @ *p-io pt* ∈ *L M′*› ‹*q′* ∈ *rd-targets* (*q*, *pt*)› ‹(*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)›
‹*target* (*initial M′*) *pt′* ∈ *io-targets M′* (*p-io pP* @ *p-io pt*) (*FSM.initial M′*)›]
by *assumption*

have *is-separator M* (*target q pt*) *q′ A t1 t2*
using *t3*[*OF* ‹(*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)›] **by** *blast*

have *is-submachine P M*
using *t2*[*OF* ‹(*q*, *P*) ∈ *prs*›] **unfolding** *is-preamble-def* **by** *blast*
then have *initial P* = *initial M* **by** *auto*

have *path M* (*initial M*) *pP*
using *submachine-path*[*OF* ‹*is-submachine P M*› ‹*path P* (*initial P*) *pP*›]

unfolding ‹*initial P* = *initial M*›
by *assumption*
have *target* (*initial M*) *pP* = *q*
using ‹*target* (*initial P*) *pP* = *q*›
unfolding ‹*initial P* = *initial M*›

by *assumption*

have $q \in$ *states M*
using *is-preamble-is-state*[*OF t2*[*OF* ‹$(q, P) \in$ *prs*›]]
by *assumption*

have $q \in$ *fst* ‘ *prs*
using ‹$(q, P) \in$ *prs*› **by** *force*

obtain pT' d' **where** $(pt$ @ $pT', d') \in$ *m-traversal-paths-with-witness M q repetition-sets m*
using *t6*[*OF* ‹$q \in$ *fst* ‘ *prs*›] ‹$pt \in$ *tps q*›
by *blast*

then have *path M q* $(pt$ @ $pT')$
and *find* $(\lambda d.$ *Suc* $(m -$ *card* (*snd d*)) $\leq$ *length* (*filter* $(\lambda t.$ *t-target t* $\in$ *fst d*) $(pt$ @ $pT'))$) *repetition-sets* = *Some* d'
and $\bigwedge p'\ p''.\ (pt$ @ $pT') = p'$ @ $p'' \Longrightarrow p'' \neq [] \Longrightarrow$ *find* $(\lambda d.$ *Suc* $(m -$ *card* (*snd d*)) $\leq$ *length* (*filter* $(\lambda t.$ *t-target t* $\in$ *fst d*) $p')$) *repetition-sets* = *None*
using *m-traversal-paths-with-witness-set*[*OF t5 t8* ‹$q \in$ *states M*›, *of m*]
by *blast+*
then have *path M q pt*
by *auto*

have *target* (*initial* M') $pt' \in$ *states* M'
using ‹*target* (*initial* M') $pt' \in$ *io-targets* M' (*p-io pP* @ *p-io pt*) (*FSM.initial* M')› *io-targets-states*
using *subset-iff*
by *fastforce*

have *pass-io-set* (*FSM.from-FSM* M' (*target* (*FSM.initial* M') pt')) (*atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*)
using *pass-io-set-from-pass-separator*[*OF* ‹*is-separator M* (*target q pt*) q' *A t1 t2*›
‹*pass-separator-ATC* M' *A* (*target* (*FSM.initial* M') pt') *t2*›
‹*observable M*› ‹*observable* M'› *path-target-is-state*[*OF* ‹*path M q pt*›]
‹*target* (*FSM.initial* M') $pt' \in$ *FSM.states* M'› ‹*inputs* M' = *inputs M*›]
by *assumption*
moreover note ‹*ioAI* @ $[(x,y)] \in$ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*›
moreover have *ioAI* @ $[(x, y')] \in L$ (*FSM.from-FSM* M' (*target* (*FSM.initial* M') pt'))
using ‹*io* @ $[(x,y')] \in L\ M'$› **unfolding** ‹*io* = *p-io pP* @ *p-io pt* @ *ioAI*›
by (*metis* (*no-types*, *lifting*) ‹*target* (*FSM.initial* M') $pt' \in$ *FSM.states*

M′›
‹*target* (*FSM.initial M′*) *pt′* ∈ *io-targets M′* (*p-io pP* @ *p-io pt*) (*FSM.initial M′*)›
append-assoc assms(*3*) *from-FSM-language observable-io-targets-language*)

ultimately have *ioAI* @ [(*x*,*y′*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*
unfolding *pass-io-set-def* **by** *blast*

define *tmp* **where** *tmp-def* : *tmp* = (⋃ {(λ*io-atc. p-io p* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A* |*p pt q A*. ∃ *P q′ t1 t2*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*, *pt*) ∧ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)})
define *tmp2* **where** *tmp2-def* : *tmp2* = ⋃ {(@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*)) |*p pt*. ∃ *q P*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q*}
have ∃ *P q′ t1 t2*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *pP* ∧ *target* (*FSM.initial P*) *pP* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*, *pt*) ∧ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)
using ‹(*q*, *P*) ∈ *prs*› ‹*path P* (*FSM.initial P*) *pP*› ‹*target* (*FSM.initial P*) *pP* = *q*› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*, *pt*)› ‹(*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)› **by** *blast*
then have (λ*io-atc. p-io pP* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A* ⊆ *tmp*
unfolding *tmp-def* **by** *blast*
then have (λ*io-atc. p-io pP* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A* ⊆ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *test-suite-to-io.simps tmp-def*[*symmetric*] *tmp2-def*[*symmetric*] **by** *blast*
moreover have (*p-io pP* @ *p-io pt* @ (*ioAI* @ [(*x*, *y′*)])) ∈ (λ*io-atc. p-io pP* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*
using ‹*ioAI* @ [(*x*, *y′*)] ∈ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*› **by** *auto*
ultimately have (*p-io pP* @ *p-io pt* @ (*ioAI* @ [(*x*, *y′*)])) ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
by *blast*
then have *io* @ [(*x*, *y′*)] ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding ‹*io* = *p-io pP* @ *p-io pt* @ *ioAI*› **by** *auto*
then show *False*
using ‹*io* @ [(*x*, *y′*)] ∉ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)›
by *blast*
qed
qed
qed
qed

ultimately show *?thesis*
unfolding ‹T = *Test-Suite prs tps rd-targets atcs*›
by *blast*
qed

lemma *test-suite-to-io-finite* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
shows *finite* (*test-suite-to-io M T*)
proof −
obtain *prs tps rd-targets atcs* **where** T = *Test-Suite prs tps rd-targets atcs*
by (*meson test-suite.exhaust*)
then obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets atcs*) *M m repetition-sets*
using *assms*(*1*)
unfolding *implies-completeness-def*
by *blast*
then have *implies-completeness* (*Test-Suite prs tps rd-targets atcs*) *M m*
unfolding *implies-completeness-def*
by *blast*
then have *test-suite-language-prop*: *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*) ⊆ *L M*
using *test-suite-to-io-language*
by *blast*

have *f1*: (*finite prs*)
and *f2*: ⋀ *q p* . *q* ∈ *fst* ‘ *prs* ⟹ *finite* (*rd-targets* (*q*,*p*))
and *f3*: ⋀ *q q′* . *finite* (*atcs* (*q*,*q′*))
using *assms*(*2*)
unfolding ‹T = *Test-Suite prs tps rd-targets atcs*› *is-finite-test-suite.simps*
by *blast*+

have *t1*: (*initial M*, *initial-preamble M*) ∈ *prs*
using *implies-completeness-for-repetition-sets-simps*(*1*)[*OF repetition-sets-def*]

by *assumption*
have *t2*: ⋀ *q P*. (*q*, *P*) ∈ *prs* ⟹ *is-preamble P M q*
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*]

by *blast*
have *t3*: ⋀ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) ∈ *atcs* (*q1*, *q2*) ⟹ (*A*, *d2*, *d1*) ∈ *atcs* (*q2*, *q1*) ∧ *is-separator M q1 q2 A d1 d2*
using *implies-completeness-for-repetition-sets-simps*(*3*)[*OF repetition-sets-def*]

by *assumption*

have *t5*: $\bigwedge q.\ q \in FSM.states\ M \Longrightarrow (\exists\, d \in set\ repetition\text{-}sets.\ q \in fst\ d)$
using *implies-completeness-for-repetition-sets-simps*(*4*)[*OF repetition-sets-def*]

by *assumption*

have *t6*: $\bigwedge q.\ q \in fst\ `\ prs \Longrightarrow tps\ q \subseteq \{p1\ .\ \exists\ p2\ d\ .\ (p1@p2,d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\} \wedge fst\ `\ (m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m) \subseteq tps\ q$
using *implies-completeness-for-repetition-sets-simps*(*7*)[*OF repetition-sets-def*]

by *assumption*

have *t7*: $\bigwedge\ d.\ d \in set\ repetition\text{-}sets \Longrightarrow fst\ d \subseteq FSM.states\ M$
and *t8*: $\bigwedge\ d.\ d \in set\ repetition\text{-}sets \Longrightarrow snd\ d \subseteq fst\ d$
and *t8'*: $\bigwedge\ d.\ d \in set\ repetition\text{-}sets \Longrightarrow snd\ d = fst\ d \cap fst\ `\ prs$
and *t9*: $\bigwedge\ d\ q1\ q2.\ d \in set\ repetition\text{-}sets \Longrightarrow q1 \in fst\ d \Longrightarrow q2 \in fst\ d \Longrightarrow q1 \neq q2 \Longrightarrow atcs\ (q1,\ q2) \neq \{\}$
using *implies-completeness-for-repetition-sets-simps*(*5*,*6*)[*OF repetition-sets-def*]

by *blast+*

have *f4*: $\bigwedge\ q\ .\ q \in fst\ `\ prs \Longrightarrow finite\ (tps\ q)$
proof −
fix *q* **assume** $q \in fst\ `\ prs$
then have $tps\ q \subseteq \{p1\ .\ \exists\ p2\ d\ .\ (p1@p2,d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\}$
using *t6* **by** *blast*
moreover have $\{p1\ .\ \exists\ p2\ d\ .\ (p1@p2,d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\} \subseteq (\bigcup\ p2 \in fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\ .\ set\ (prefixes\ p2))$
proof
fix *p1* **assume** $p1 \in \{p1\ .\ \exists\ p2\ d\ .\ (p1@p2,d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\}$
then obtain *p2 d* **where** $(p1@p2,d) \in m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m$ **by** *blast*
then have $p1@p2 \in fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m$ **by** *force*
moreover have $p1 \in set\ (prefixes\ (p1@p2))$ **unfolding** *prefixes-set* **by** *blast*
ultimately show $p1 \in (\bigcup\ p2 \in fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\ .\ set\ (prefixes\ p2))$ **by** *blast*
qed
ultimately have $tps\ q \subseteq (\bigcup\ p2 \in fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\ .\ set\ (prefixes\ p2))$
by *simp*
moreover have $finite\ (\bigcup\ p2 \in fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m\ .\ set\ (prefixes\ p2))$
proof −
have $finite\ (fst\ `\ m\text{-}traversal\text{-}paths\text{-}with\text{-}witness\ M\ q\ repetition\text{-}sets\ m)$

using *m-traversal-paths-with-witness-finite*[*of M q repetition-sets m*] **by** *auto*
moreover have $\bigwedge$ *p2* . *finite* (*set* (*prefixes p2*)) **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
ultimately show *finite* (*tps q*)
using *finite-subset* **by** *blast*
qed
then have *f4'* : $\bigwedge$ *q P* . (*q*,*P*) $\in$ *prs* $\Longrightarrow$ *finite* (*tps q*)
by *force*

define *T1* **where** *T1-def* : *T1* = ($\bigcup$(*q*, *P*)$\in$*prs*. *L P*)
define *T2* **where** *T2-def* : *T2* = $\bigcup$\{(@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*)) |*p pt*. $\exists$ *q P*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q*\}
define *T3* **where** *T3-def* : *T3* = $\bigcup$ \{(λ*io-atc*. *p-io p* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A* |*p pt q A*.
$\exists$ *P q' t1 t2*.
(*q*, *P*) $\in$ *prs* $\wedge$
path P (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q'* $\in$ *rd-targets* (*q*, *pt*) $\wedge$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*)\}

have *test-suite-to-io M T* = *T1* $\cup$ *T2* $\cup$ *T3*
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*› *test-suite-to-io.simps T1-def T2-def T3-def* **by** *simp*

moreover have *finite T1*
proof −
have $\bigwedge$ *q P* . (*q*, *P*)$\in$*prs* $\Longrightarrow$ *finite* (*L P*)
proof −
fix *q P* **assume** (*q*, *P*)$\in$*prs*
have *acyclic P*
using *t2*[*OF* ‹(*q*, *P*)$\in$*prs*›]
unfolding *is-preamble-def*
by *blast*
then show *finite* (*L P*)
using *acyclic-alt-def*
by *blast*
qed
then show *?thesis* **using** *f1* **unfolding** *T1-def*
by *auto*
qed

moreover have *finite T2*
proof −
have $*$: *T2* = ($\bigcup$ (*p*,*pt*) $\in$ \{(*p*,*pt*) | *p pt*. $\exists$ *q P*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q*\} . ((@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*))))
unfolding *T2-def*
by *auto*

have $\bigwedge$ *p pt* . *finite* ((@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*)))
by *auto*
moreover have *finite* {(*p*,*pt*) | *p pt*. ∃ *q P*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q*}
proof −
have {(*p*,*pt*) | *p pt*. ∃ *q P*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q*} ⊆ (⋃ (*q*,*P*) ∈ *prs* . {*p* . *path P* (*initial P*) *p*} × (*tps q*))
by *auto*
moreover have *finite* (⋃ (*q*,*P*) ∈ *prs* . {*p* . *path P* (*initial P*) *p*} × (*tps q*))
proof −
note ‹*finite prs*›
moreover have $\bigwedge$ *q P* . (*q*,*P*) ∈ *prs* ⟹ *finite* ({*p* . *path P* (*initial P*) *p*} × (*tps q*))
proof −
fix *q P* **assume** (*q*,*P*) ∈ *prs*

have *acyclic P* **using** *t2*[*OF* ‹(*q*, *P*)∈*prs*›]
unfolding *is-preamble-def*
by *blast*
then have *finite* {*p* . *path P* (*initial P*) *p*}
using *acyclic-paths-finite*[*of P initial P*]
unfolding *acyclic.simps*
by (*metis* (*no-types*, *lifting*) *Collect-cong*)
then show *finite* ({*p* . *path P* (*initial P*) *p*} × (*tps q*))
using *f4'*[*OF* ‹(*q*,*P*) ∈ *prs*›]
by *simp*
qed
ultimately show *?thesis*
by *force*
qed
ultimately show *?thesis*
by (*meson rev-finite-subset*)
qed
ultimately show *?thesis*
unfolding ∗ **by** *auto*
qed

moreover have *finite T3*
proof −
have *scheme*: $\bigwedge$ *f P* . (⋃ {*f a b c d* | *a b c d* . *P a b c d*}) = (⋃ (*a*,*b*,*c*,*d*) ∈ {(*a*,*b*,*c*,*d*) | *a b c d* . *P a b c d*} . *f a b c d*)
by *blast*

have ∗: *T3* = (⋃ (*p*,*pt*,*q*,*A*) ∈ {(*p*, *pt*, *q*, *A*) | *p pt q A* . ∃ *P q' t1 t2*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q'* ∈ *rd-targets* (*q*, *pt*) ∧ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q'*)}

. (λ*io-atc*. *p-io p* @ *p-io pt* @ *io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*)
unfolding *T3-def scheme* **by** *blast*

have {(*p*, *pt*, *q*, *A*) | *p pt q A* . ∃ *P q′ t1 t2*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*, *pt*) ∧ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*)}
⊆ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*, *pt*) . (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*}))
by *blast*
moreover have *finite* (⋃ (*q*,*P*) ∈ *prs* . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*, *pt*) . (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*}))
proof −
note ‹*finite prs*›
moreover have ⋀ *q P* . (*q*,*P*) ∈ *prs* ⟹ *finite* (⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*, *pt*) . (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*}))
proof −
fix *q P* **assume** (*q*,*P*) ∈ *prs*
then have *q* ∈ *fst* ‘ *prs* **by** *force*

have *finite* (*tps q*) **using** *f4′*[*OF* ‹(*q*,*P*) ∈ *prs*›] **by** *assumption*
moreover have ⋀ *pt* . *pt* ∈ *tps q* ⟹ *finite* (⋃ *q′* ∈ *rd-targets* (*q*, *pt*) . (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*}))
proof −
fix *pt* **assume** *pt* ∈ *tps q*

have *finite* (*rd-targets* (*q*,*pt*)) **using** *f2*[*OF* ‹*q* ∈ *fst* ‘ *prs*›] **by** *blast*
moreover have ⋀ *q′* . *q′* ∈ *rd-targets* (*q*, *pt*) ⟹ *finite* (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*})
proof −
fix *q′* **assume** *q′* ∈ *rd-targets* (*q*, *pt*)
have *finite* (*atcs* (*target q pt*, *q′*)) **using** *f3* **by** *blast*
moreover have *finite* {*p* . *path P* (*initial P*) *p*}
proof −
have *acyclic P* **using** *t2*[*OF* ‹(*q*, *P*)∈*prs*›] **unfolding** *is-preamble-def* **by** *blast*
then show *?thesis* **using** *acyclic-paths-finite*[*of P initial P*] **unfolding** *acyclic.simps* **by** (*metis* (*no-types*, *lifting*) *Collect-cong*)
qed
ultimately show *finite* (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*})
by *force*
qed
ultimately show *finite* (⋃ *q′* ∈ *rd-targets* (*q*, *pt*) . (⋃ (*A*, *t1*, *t2*) ∈ *atcs* (*target q pt*, *q′*) . {*p* . *path P* (*initial P*) *p*} × {*pt*} × {*q*} × {*A*}))
by *force*

qed
ultimately show *finite* ($\bigcup$ *pt* $\in$ *tps q* . $\bigcup$ *q'* $\in$ *rd-targets* (*q*, *pt*) . ($\bigcup$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*) . {*p* . *path P* (*initial P*) *p*} $\times$ {*pt*} $\times$ {*q*} $\times$ {*A*}))
by *force*
qed
ultimately show *?thesis* **by** *force*
qed
ultimately have *finite* {(*p*, *pt*, *q*, *A*) | *p pt q A* . $\exists$ *P q' t1 t2*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q'* $\in$ *rd-targets* (*q*, *pt*) $\wedge$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*)}
by (*meson rev-finite-subset*)

moreover have $\bigwedge$ *p pt q A* . (*p*,*pt*,*q*,*A*) $\in$ {(*p*, *pt*, *q*, *A*) | *p pt q A* . $\exists$ *P q' t1 t2*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q'* $\in$ *rd-targets* (*q*, *pt*) $\wedge$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*)}
$\Longrightarrow$ *finite* ((*λio-atc. p-io p @ p-io pt @ io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*)
proof −
fix *p pt q A* **assume** (*p*,*pt*,*q*,*A*) $\in$ {(*p*, *pt*, *q*, *A*) | *p pt q A* . $\exists$ *P q' t1 t2*. (*q*, *P*) $\in$ *prs* $\wedge$ *path P* (*FSM.initial P*) *p* $\wedge$ *target* (*FSM.initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q'* $\in$ *rd-targets* (*q*, *pt*) $\wedge$ (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*)}
then obtain *P q' t1 t2* **where** (*q*, *P*) $\in$ *prs* **and** *path P* (*FSM.initial P*) *p* **and** *target* (*FSM.initial P*) *p* = *q* **and** *pt* $\in$ *tps q* **and** *q'* $\in$ *rd-targets* (*q*, *pt*) **and** (*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*) **by** *blast*

have *is-separator M* (*target q pt*) *q' A t1 t2*
using *t3*[*OF* ‹(*A*, *t1*, *t2*) $\in$ *atcs* (*target q pt*, *q'*)›] **by** *blast*
then have *acyclic A*
using *is-separator-simps*(*2*) **by** *simp*
then have *finite* (*L A*)
unfolding *acyclic-alt-def* **by** *assumption*
then have *finite* (*atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*)
unfolding *atc-to-io-set.simps* **by** *blast*
then show *finite* ((*λio-atc. p-io p @ p-io pt @ io-atc*) ‘ *atc-to-io-set* (*FSM.from-FSM M* (*target q pt*)) *A*)
by *blast*
qed

ultimately show *?thesis* **unfolding** $*$ **by** *force*
qed

ultimately show *?thesis*
by *simp*
qed

42.1 Calculating the Sets of Sequences

abbreviation *L-acyclic M* $\equiv$ *LS-acyclic M* (*initial M*)

fun *test-suite-to-io′* :: (′a,′b,′c) *fsm* ⇒ (′a,′b,′c,′d) *test-suite* ⇒ (′b × ′c) *list set*
where
test-suite-to-io′ M (*Test-Suite prs tps rd-targets atcs*)
= (⋃ (*q*,*P*) ∈ *prs* .
L-acyclic P
∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) .
⋃ *pt* ∈ *tps q* .
((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))))
∪ (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) .
⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) .
(λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))

lemma *test-suite-to-io-code* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows *test-suite-to-io M T* = *test-suite-to-io′ M T*
proof −

obtain *prs tps rd-targets atcs* **where** *T* = *Test-Suite prs tps rd-targets atcs*
by (*meson test-suite.exhaust*)

then obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets atcs*) *M m repetition-sets*
using *assms*(*1*)
unfolding *implies-completeness-def*
by *blast*

have *t2*: ⋀ *q P*. (*q*, *P*) ∈ *prs* ⟹ *is-preamble P M q*
using *implies-completeness-for-repetition-sets-simps*(*2*)[*OF repetition-sets-def*]

by *blast*
have *t3*: ⋀ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) ∈ *atcs* (*q1*, *q2*) ⟹ (*A*, *d2*, *d1*) ∈ *atcs* (*q2*, *q1*) ∧ *is-separator M q1 q2 A d1 d2*
using *implies-completeness-for-repetition-sets-simps*(*3*)[*OF repetition-sets-def*]

by *assumption*

have *test-suite-to-io′-alt-def*: *test-suite-to-io′ M T*
= (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P*)
∪ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))))
∪ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc*

. *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
unfolding *test-suite-to-io′.simps* ‹*T* = *Test-Suite prs tps rd-targets atcs*›
by *fast*

have *test-suite-to-io-alt-def*: *test-suite-to-io M T* =
(⋃ (*q*,*P*) ∈ *prs* . *L P*)
∪ (⋃{(λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . ∃ *q P* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q*})
∪ (⋃{(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) })
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*› *test-suite-to-io.simps*
by *force*

have *preamble-language-prop*: ⋀ *q P* . (*q*,*P*) ∈ *prs* ⟹ *L-acyclic P* = *L P*
proof −
fix *q P* **assume** (*q*,*P*) ∈ *prs*
have *acyclic P* **using** *t2*[*OF* ‹(*q*, *P*)∈*prs*›] **unfolding** *is-preamble-def* **by** *blast*

then show *L-acyclic P* = *L P* **using** *LS-from-LS-acyclic* **by** *blast*
qed

have *preamble-path-prop*: ⋀ *q P ioP* . (*q*,*P*) ∈ *prs* ⟹ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) ⟷ (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*)
proof −
fix *q P ioP* **assume** (*q*,*P*) ∈ *prs*
have *is-preamble P M q* **using** *t2*[*OF* ‹(*q*, *P*)∈*prs*›] **by** *assumption*

have *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) ⟹ (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*)
proof −
assume *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
then have *ioP* ∈ *L P* **and** ∄ *x′*. *x′* ≠ [] ∧ *ioP* @ *x′* ∈ *L P*
unfolding *preamble-language-prop*[*OF* ‹(*q*,*P*) ∈ *prs*›] *remove-proper-prefixes-def* **by** *blast+*
show (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*)
using *preamble-maximal-io-paths-rev*[*OF* ‹*is-preamble P M q*› ‹*observable M*› ‹*ioP* ∈ *L P*› ‹∄ *x′*. *x′* ≠ [] ∧ *ioP* @ *x′* ∈ *L P*›] **by** *blast*
qed
moreover have (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*) ⟹ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
proof −
assume (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*)
then obtain *p* **where** *path P* (*initial P*) *p* **and** *target* (*initial P*) *p* = *q* **and** *p-io p* = *ioP*

by *blast*
then have $\nexists io'.\ io' \neq [] \wedge$ *p-io p* @ $io' \in L\ P$
using *preamble-maximal-io-paths*[*OF* ‹*is-preamble P M q*› ‹*observable M*›]
by *blast*
then show $ioP \in$ *remove-proper-prefixes* (*L-acyclic P*)
using *language-state-containment*[*OF* ‹*path P* (*initial P*) *p*› ‹*p-io p* = *ioP*›]
unfolding *preamble-language-prop*[*OF* ‹(*q*,*P*) ∈ *prs*›] *remove-proper-prefixes-def* ‹*p-io p* = *ioP*› **by** *blast*
qed
ultimately show $ioP \in$ *remove-proper-prefixes* (*L-acyclic P*) $\longleftrightarrow$ (∃ *p* . *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *p-io p* = *ioP*)
by *blast*
qed

have *eq1*: (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P*) = (⋃ (*q*,*P*) ∈ *prs* . *L P*)
using *preamble-language-prop* **by** *blast*

have *eq2*: (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))))) = (⋃{(λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . ∃ *q P* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q*})
proof
show (⋃(*q*, *P*)∈*prs*. ⋃*ioP*∈*remove-proper-prefixes* (*L-acyclic P*). ⋃*pt*∈*tps q*. (@) *ioP* ‘ *set* (*prefixes* (*p-io pt*))) ⊆ ⋃ {(@) (*p-io p*) ‘ *set* (*prefixes* (*p-io pt*)) |*p pt*. ∃*q P*. (*q*, *P*) ∈ *prs* ∧ *path P* (*FSM.initial P*) *p* ∧ *target* (*FSM.initial P*) *p* = *q* ∧ *pt* ∈ *tps q*}
proof
fix *io* **assume** *io* ∈ (⋃ (*q*,*P*) ∈ *prs* . (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))))))
then obtain *q P* **where** (*q*,*P*) ∈ *prs*
and *io* ∈ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))))
by *blast*
then obtain *ioP* **where** *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
and *io* ∈ (⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))))
by *blast*

obtain *p* **where** *path P* (*initial P*) *p* **and** *target* (*initial P*) *p* = *q* **and** *ioP* = *p-io p*
using *preamble-path-prop*[*OF* ‹(*q*,*P*) ∈ *prs*›, *of ioP*] ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› **by** *auto*

obtain *pt* **where** *pt* ∈ *tps q* **and** *io* ∈ ((λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))))
using ‹*io* ∈ (⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))))› **unfolding** ‹*ioP* = *p-io p*› **by** *blast*

show $io \in (\bigcup\{(\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt)))\ |\ p\ pt\ .\ \exists\ q\ P\ .\ (q,P) \in prs \wedge path\ P\ (initial\ P)\ p \wedge target\ (initial\ P)\ p = q \wedge pt \in tps\ q\})$
using ‹$io \in ((\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt))))$› ‹$(q,P) \in prs$› ‹$path\ P\ (initial\ P)\ p$› ‹$target\ (initial\ P)\ p = q$› ‹$pt \in tps\ q$› **by** *blast*
qed

show $\bigcup\ \{(@)\ (p\text{-}io\ p)\ `\ set\ (prefixes\ (p\text{-}io\ pt))\ |p\ pt.\ \exists q\ P.\ (q,\ P) \in prs \wedge path\ P\ (FSM.initial\ P)\ p \wedge target\ (FSM.initial\ P)\ p = q \wedge pt \in tps\ q\} \subseteq (\bigcup(q,\ P)\in prs.\ \bigcup ioP\in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P).\ \bigcup pt\in tps\ q.\ (@)\ ioP\ `\ set\ (prefixes\ (p\text{-}io\ pt)))$
proof
fix *io* **assume** $io \in (\bigcup\{(\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt)))\ |\ p\ pt\ .\ \exists\ q\ P\ .\ (q,P) \in prs \wedge path\ P\ (initial\ P)\ p \wedge target\ (initial\ P)\ p = q \wedge pt \in tps\ q\})$
then obtain $p\ pt\ q\ P$ **where** $io \in (\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt)))$
and $(q,P) \in prs$ **and** $path\ P\ (initial\ P)\ p$ **and** $target\ (initial\ P)\ p = q$ **and** $pt \in tps\ q$
by *blast*

then obtain ioP **where** $ioP \in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P)$
and $p\text{-}io\ p = ioP$
using *preamble-path-prop*[*OF* ‹$(q,P) \in prs$›, *of* $p\text{-}io\ p$] **by** *blast*

show $io \in (\bigcup\ (q,P) \in prs\ .\ (\bigcup\ ioP \in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P)\ .\ \bigcup\ pt \in tps\ q\ .\ ((\lambda\ io'\ .\ ioP\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt))))))$
using ‹$(q,P) \in prs$› ‹$ioP \in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P)$› ‹$pt \in tps\ q$› ‹$io \in (\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')\ `\ (set\ (prefixes\ (p\text{-}io\ pt)))$› **unfolding** ‹$p\text{-}io\ p = ioP$› **by** *blast*
qed
qed

have *eq3*: $(\bigcup\ (q,P) \in prs\ .\ \bigcup\ ioP \in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P)\ .\ \bigcup\ pt \in tps\ q\ .\ \bigcup\ q' \in rd\text{-}targets\ (q,pt)\ .\ \bigcup\ (A,t1,t2) \in atcs\ (target\ q\ pt,q')\ .\ (\lambda\ io\text{-}atc\ .\ ioP\ @\ p\text{-}io\ pt\ @\ io\text{-}atc)\ `\ (acyclic\text{-}language\text{-}intersection\ (from\text{-}FSM\ M\ (target\ q\ pt))\ A)) = (\bigcup\{(\lambda\ io\text{-}atc\ .\ p\text{-}io\ p\ @\ p\text{-}io\ pt\ @\ io\text{-}atc)\ `\ (atc\text{-}to\text{-}io\text{-}set\ (from\text{-}FSM\ M\ (target\ q\ pt))\ A)\ |\ p\ pt\ q\ A\ .\ \exists\ P\ q'\ t1\ t2\ .\ (q,P) \in prs \wedge path\ P\ (initial\ P)\ p \wedge target\ (initial\ P)\ p = q \wedge pt \in tps\ q \wedge q' \in rd\text{-}targets\ (q,pt) \wedge (A,t1,t2) \in atcs\ (target\ q\ pt,q')\ \})$
proof
show $(\bigcup\ (q,P) \in prs\ .\ \bigcup\ ioP \in remove\text{-}proper\text{-}prefixes\ (L\text{-}acyclic\ P)\ .\ \bigcup\ pt \in tps\ q\ .\ \bigcup\ q' \in rd\text{-}targets\ (q,pt)\ .\ \bigcup\ (A,t1,t2) \in atcs\ (target\ q\ pt,q')\ .\ (\lambda\ io\text{-}atc\ .\ ioP\ @\ p\text{-}io\ pt\ @\ io\text{-}atc)\ `\ (acyclic\text{-}language\text{-}intersection\ (from\text{-}FSM\ M\ (target\ q\ pt))\ A)) \subseteq (\bigcup\{(\lambda\ io\text{-}atc\ .\ p\text{-}io\ p\ @\ p\text{-}io\ pt\ @\ io\text{-}atc)\ `\ (atc\text{-}to\text{-}io\text{-}set\ (from\text{-}FSM\ M\ (target\ q\ pt))\ A)\ |\ p\ pt\ q\ A\ .\ \exists\ P\ q'\ t1\ t2\ .\ (q,P) \in prs \wedge path\ P\ (initial\ P)\ p \wedge target\ (initial\ P)\ p = q \wedge pt \in tps\ q \wedge q' \in rd\text{-}targets\ (q,pt) \wedge (A,t1,t2) \in atcs\ (target\ q\ pt,q')\ \})$
proof

fix *io* **assume** *io* ∈ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))

then obtain *q P ioP pt q′ A t1 t2* **where** (*q*,*P*) ∈ *prs*
and *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
and *pt* ∈ *tps q*
and *q′* ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)
and *io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)
by *blast*

obtain *p* **where** *path P* (*initial P*) *p* **and** *target* (*initial P*) *p* = *q* **and** *ioP* = *p-io p*
using *preamble-path-prop*[*OF* ‹(*q*,*P*) ∈ *prs*›, *of ioP*] ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› **by** *auto*

have *acyclic A*
using *t3*[*OF* ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)›] *is-separator-simps*(*2*) **by** *metis*
have (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*) = (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
unfolding *acyclic-language-intersection-completeness*[*OF* ‹*acyclic A*›] *atc-to-io-set.simps* **by** *simp*
have *io* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
using ‹*io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)› **unfolding** ‹(*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*) = (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)› ‹*ioP* = *p-io p*› **by** *simp*

then show *io* ∈ (⋃{(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) })
using ‹(*q*,*P*) ∈ *prs*› ‹*path P* (*initial P*) *p*› ‹*target* (*initial P*) *p* = *q*› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)› **by** *blast*
qed

show (⋃{(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) }) ⊆ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
proof
fix *io* **assume** *io* ∈ (⋃{(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set*

(*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) })

then obtain *p pt q A P q′ t1 t2* **where** *io* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
and (*q*,*P*) ∈ *prs*
and *path P* (*initial P*) *p*
and *target* (*initial P*) *p* = *q*
and *pt* ∈ *tps q*
and *q′* ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)
by *blast*

then obtain *ioP* **where** *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
and *p-io p* = *ioP*
using *preamble-path-prop*[*OF* ‹(*q*,*P*) ∈ *prs*›, *of p-io p*] **by** *blast*

have *acyclic A*
using *t3*[*OF* ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)›] *is-separator-simps*(*2*) **by** *metis*

have ∗: (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) = (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)
unfolding *acyclic-language-intersection-completeness*[*OF* ‹*acyclic A*›] *atc-to-io-set.simps* **by** *simp*

have *io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)
using ‹*io* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)› **unfolding** ∗ ‹*p-io p* = *ioP*› **by** *simp*

then show *io* ∈ (⋃ (*q*,*P*) ∈ *prs* . ⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
using ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)› **by** *force*
qed
qed

show *?thesis*
unfolding *test-suite-to-io′-alt-def test-suite-to-io-alt-def eq1 eq2 eq3* **by** *simp*
qed

42.2 Using Maximal Sequences Only

fun *test-suite-to-io-maximal* :: (′*a*::*linorder*,′*b*::*linorder*,′*c*) *fsm* ⇒ (′*a*,′*b*,′*c*,′*d*::*linorder*) *test-suite* ⇒ (′*b* × ′*c*) *list set* **where**
test-suite-to-io-maximal M (*Test-Suite prs tps rd-targets atcs*) =
remove-proper-prefixes (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes*

(*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))))

lemma *test-suite-to-io-maximal-code* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows {*io′* ∈ (*test-suite-to-io M T*) . ¬ (∃ *io″* . *io″* ≠ [] ∧ *io′*@*io″* ∈ (*test-suite-to-io M T*))} = *test-suite-to-io-maximal M T*
proof −

obtain *prs tps rd-targets atcs* **where** *T* = *Test-Suite prs tps rd-targets atcs*
by (*meson test-suite.exhaust*)

have *t-def*: *test-suite-to-io M T* = *test-suite-to-io′ M T*
using *test-suite-to-io-code*[*OF assms*] **by** *assumption*

have *t1-def* : *test-suite-to-io′ M T* = (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))) ∪ (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*› **by** *simp*

define *tmax* **where** *tmax-def*: *tmax* = (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))))
have *t2-def* : *test-suite-to-io-maximal M T* = *remove-proper-prefixes tmax*
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*› *tmax-def* **by** *simp*

have *tmax-sub*: *tmax* ⊆ (*test-suite-to-io M T*)
unfolding *tmax-def t-def t1-def*
proof
fix *io* **assume** *io* ∈ (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))))
then obtain *q P* **where** (*q*,*P*) ∈ *prs*
and *io* ∈ *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))

by *force*
then consider (*a*) *io* ∈ *L-acyclic P* |
(*b*) *io* ∈ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))
by *blast*
then show *io* ∈ (⋃ (*q*,*P*) ∈ *prs* . *L-acyclic P* ∪ (⋃ *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*) . ⋃ *pt* ∈ *tps q* . ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))) ∪ (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))
proof *cases*
case *a*
then show *?thesis* **using** ‹(*q*,*P*) ∈ *prs*› **by** *blast*
next
case *b*
then obtain *ioP pt* **where** *ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)
and *pt* ∈ *tps q*
and *io* ∈ *Set.insert* (*ioP* @ *p-io pt*) (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))
by *blast*
then consider (*b1*) *io* = (*ioP* @ *p-io pt*) |
(*b2*) *io* ∈ (⋃ *q′* ∈ *rd-targets* (*q*,*pt*) . ⋃ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))
by *blast*
then show *?thesis* **proof** *cases*
case *b1*
then have *io* ∈ ((λ *io′* . *ioP* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))) **unfolding** *prefixes-set* **by** *blast*
then show *?thesis* **using** ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› **by** *blast*
next
case *b2*
then obtain *q′ A t1 t2* **where** *q′* ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)
and *io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
by *blast*
then have *io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)
unfolding *remove-proper-prefixes-def* **by** *blast*
then show *?thesis* **using** ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)› **by** *force*
qed

qed
qed

have *tmax-max*: $\bigwedge$ *io* . *io* $\in$ *test-suite-to-io M T* $\Longrightarrow$ *io* $\notin$ *tmax* $\Longrightarrow$ $\exists$ *io''* . *io''* $\neq$ [] $\wedge$ *io@io''* $\in$ (*test-suite-to-io M T*)
proof −
fix *io* **assume** *io* $\in$ *test-suite-to-io M T* **and** *io* $\notin$ *tmax*

then have *io* $\in$ ($\bigcup$ (*q*,*P*) $\in$ *prs* . *L-acyclic P* $\cup$ ($\bigcup$ *ioP* $\in$ *remove-proper-prefixes* (*L-acyclic P*) . $\bigcup$ *pt* $\in$ *tps q* . ((λ *io'* . *ioP* @ *io'*) ‘ (*set* (*prefixes* (*p-io pt*)))) $\cup$ ($\bigcup$ *q'* $\in$ *rd-targets* (*q*,*pt*) . $\bigcup$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q'*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))
unfolding *t-def t1-def* **by** *blast*
then obtain *q P* **where** (*q*,*P*) $\in$ *prs*
and *io* $\in$ (*L-acyclic P* $\cup$ ($\bigcup$ *ioP* $\in$ *remove-proper-prefixes* (*L-acyclic P*) . $\bigcup$ *pt* $\in$ *tps q* . ((λ *io'* . *ioP* @ *io'*) ‘ (*set* (*prefixes* (*p-io pt*)))) $\cup$ ($\bigcup$ *q'* $\in$ *rd-targets* (*q*,*pt*) . $\bigcup$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q'*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))))
by *force*
then consider (*a*) *io* $\in$ *L-acyclic P* |
(*b*) *io* $\in$ ($\bigcup$ *ioP* $\in$ *remove-proper-prefixes* (*L-acyclic P*) . $\bigcup$ *pt* $\in$ *tps q* . ((λ *io'* . *ioP* @ *io'*) ‘ (*set* (*prefixes* (*p-io pt*)))) $\cup$ ($\bigcup$ *q'* $\in$ *rd-targets* (*q*,*pt*) . $\bigcup$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q'*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)))
by *blast*
then show $\exists$ *io''* . *io''* $\neq$ [] $\wedge$ *io@io''* $\in$ (*test-suite-to-io M T*) **proof** *cases*
case *a*
then have *io* $\in$ *tmax*
using ‹(*q*,*P*) $\in$ *prs*› **unfolding** *tmax-def* **by** *blast*
then show *?thesis*
using ‹*io* $\notin$ *tmax*› **by** *simp*
next
case *b*
then obtain *ioP pt* **where** *ioP* $\in$ *remove-proper-prefixes* (*L-acyclic P*)
and *pt* $\in$ *tps q*
and *io* $\in$ ((λ *io'* . *ioP* @ *io'*) ‘ (*set* (*prefixes* (*p-io pt*)))) $\cup$ ($\bigcup$ *q'* $\in$ *rd-targets* (*q*,*pt*) . $\bigcup$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q'*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
by *blast*
then consider (*b1*) *io* $\in$ (λ *io'* . *ioP* @ *io'*) ‘ (*set* (*prefixes* (*p-io pt*))) |
(*b2*) *io* $\in$ ($\bigcup$ *q'* $\in$ *rd-targets* (*q*,*pt*) . $\bigcup$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q'*) . (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
by *blast*
then show *?thesis* **proof** *cases*
case *b1*
then obtain *pt1 pt2* **where** *io* = *ioP* @ *pt1* **and** *p-io pt* = *pt1@pt2*
by (*metis* (*no-types*, *lifting*) *b1 image-iff prefixes-set-ob*)

have *ioP* @ (*p-io pt*) ∈ *tmax*
using ‹*io* ∉ *tmax*› ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› **unfolding** *tmax-def* **by** *force*
then have *io* ≠ *ioP* @ (*p-io pt*)
using ‹*io* ∉ *tmax*› **by** *blast*
then have *pt2* ≠ []
using ‹*io* = *ioP* @ *pt1*› **unfolding** ‹*p-io pt* = *pt1*@*pt2*› **by** *auto*

have *ioP* @ (*p-io pt*) ∈ *test-suite-to-io M T*
using ‹*ioP* @ (*p-io pt*) ∈ *tmax*› *tmax-sub* **by** *blast*
then show *?thesis*
unfolding ‹*io* = *ioP* @ *pt1*› *append.assoc* ‹*p-io pt* = *pt1*@*pt2*›
using ‹*pt2* ≠ []› **by** *blast*
next
case *b2*
then obtain *q′ A t1 t2* **where** *q′* ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)
and *io* ∈ (λ *io-atc* . *ioP* @ *p-io pt* @ *io-atc*) ‘ (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*)
by *blast*

then obtain *ioA* **where** *io* = *ioP* @ *p-io pt* @ *ioA*
and *ioA* ∈ *acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*
by *blast*

moreover have *ioA* ∉ (*remove-proper-prefixes* (*acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*))
proof
assume *ioA* ∈ *remove-proper-prefixes* (*acyclic-language-intersection* (*FSM.from-FSM M* (*target q pt*)) *A*)
then have *io* ∈ *tmax*
using ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)› **unfolding** *tmax-def* ‹*io* = *ioP* @ *p-io pt* @ *ioA*› **by** *force*
then show *False*
using ‹*io* ∉ *tmax*› **by** *simp*
qed
ultimately obtain *ioA2* **where** *ioA* @ *ioA2* ∈ *acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*
and *ioA2* ≠ []
unfolding *remove-proper-prefixes-def* **by** *blast*
then have *io*@*ioA2* ∈ *test-suite-to-io M T*
using ‹(*q*,*P*) ∈ *prs*› ‹*ioP* ∈ *remove-proper-prefixes* (*L-acyclic P*)› ‹*pt* ∈ *tps q*› ‹*q′* ∈ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)› ‹*ioA* @ *ioA2* ∈ *acyclic-language-intersection* (*from-FSM M* (*target q pt*)) *A*› **unfolding** *t-def t1-def* ‹*io* = *ioP* @ *p-io pt* @ *ioA*› **by** *force*

then show *?thesis*
using ‹*ioA2* ≠ []› **by** *blast*
qed
qed
qed

show *?thesis* **unfolding** *t2-def*
proof
show {*io′* ∈ *test-suite-to-io M T*. ∄ *io″*. *io″* ≠ [] ∧ *io′* @ *io″* ∈ *test-suite-to-io M T*} ⊆ *remove-proper-prefixes tmax*
proof
fix *io* **assume** *io* ∈ {*io′* ∈ *test-suite-to-io M T*. ∄ *io″*. *io″* ≠ [] ∧ *io′* @ *io″* ∈ *test-suite-to-io M T*}
then have *io* ∈ *test-suite-to-io M T* **and** ∄ *io″*. *io″* ≠ [] ∧ *io* @ *io″* ∈ *test-suite-to-io M T*
by *blast+*
show *io* ∈ *remove-proper-prefixes tmax*
using ‹∄ *io″*. *io″* ≠ [] ∧ *io* @ *io″* ∈ *test-suite-to-io M T*›
using *tmax-sub tmax-max*[*OF* ‹*io* ∈ *test-suite-to-io M T*›] **unfolding** *remove-proper-prefixes-def*
by *auto*
qed
show *remove-proper-prefixes tmax* ⊆ {*io′* ∈ *test-suite-to-io M T*. ∄ *io″*. *io″* ≠ [] ∧ *io′* @ *io″* ∈ *test-suite-to-io M T*}
proof
fix *io* **assume** *io* ∈ *remove-proper-prefixes tmax*
then have *io* ∈ *tmax* **and** ∄ *io″*. *io″* ≠ [] ∧ *io* @ *io″* ∈ *remove-proper-prefixes tmax*
unfolding *remove-proper-prefixes-def* **by** *blast+*
then have *io* ∈ *test-suite-to-io M T*
using *tmax-sub* **by** *blast*
moreover have ∄ *io″*. *io″* ≠ [] ∧ *io* @ *io″* ∈ *test-suite-to-io M T*
proof
assume ∃ *io″*. *io″* ≠ [] ∧ *io* @ *io″* ∈ *test-suite-to-io M T*
then obtain *io″* **where** *io″* ≠ [] **and** *io* @ *io″* ∈ *test-suite-to-io M T*
by *blast*
then obtain *io‴* **where** (*io* @ *io″*) @ *io‴* ∈ *test-suite-to-io M T*
and (∄ *zs*. *zs* ≠ [] ∧ (*io* @ *io″*) @ *io‴* @ *zs* ∈ *test-suite-to-io M T*)
using *finite-set-elem-maximal-extension-ex*[*OF* ‹*io* @ *io″* ∈ *test-suite-to-io M T*› *test-suite-to-io-finite*[*OF assms*(*1*,*2*)]] **by** *blast*

have *io* @ (*io″* @ *io‴*) ∈ *tmax*
using *tmax-max*[*OF* ‹(*io* @ *io″*) @ *io‴* ∈ *test-suite-to-io M T*›] ‹(∄ *zs*. *zs* ≠ [] ∧ (*io* @ *io″*) @ *io‴* @ *zs* ∈ *test-suite-to-io M T*)› **by** *force*
moreover have *io″*@*io‴* ≠ []
using ‹*io″* ≠ []› **by** *auto*

ultimately show *False*
using ‹$\nexists io''.\ io'' \neq [] \wedge io @ io'' \in$ *remove-proper-prefixes tmax*›
by (*metis* (*mono-tags*, *lifting*) ‹$io \in$ *remove-proper-prefixes tmax*› *mem-Collect-eq remove-proper-prefixes-def*)
qed

ultimately show $io \in \{io' \in$ *test-suite-to-io M T*$.\ \nexists io''.\ io'' \neq [] \wedge io' @ io'' \in$ *test-suite-to-io M T*$\}$
by *blast*
qed
qed
qed

lemma *test-suite-to-io-pass-maximal* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
shows *pass-io-set M′* (*test-suite-to-io M T*) = *pass-io-set-maximal M′* $\{io' \in$ (*test-suite-to-io M T*) . $\neg$ ($\exists$ *io″* . $io'' \neq [] \wedge io'@io'' \in$ (*test-suite-to-io M T*))$\}$
proof −

have *p1*: *finite* (*test-suite-to-io M T*)
using *test-suite-to-io-finite*[*OF assms*] **by** *assumption*

obtain *prs tps rd-targets atcs* **where** *T* = *Test-Suite prs tps rd-targets atcs*
by (*meson test-suite.exhaust*)
then obtain *repetition-sets* **where** *repetition-sets-def*: *implies-completeness-for-repetition-sets* (*Test-Suite prs tps rd-targets atcs*) *M m repetition-sets*
using *assms*(*1*) **unfolding** *implies-completeness-def* **by** *blast*
then have *implies-completeness* (*Test-Suite prs tps rd-targets atcs*) *M m*
unfolding *implies-completeness-def* **by** *blast*
then have *test-suite-language-prop*: *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*) $\subseteq$ *L M*
using *test-suite-to-io-language* **by** *blast*

have *p2*: $\bigwedge io'\ io''.\ io' @ io'' \in$ *test-suite-to-io M T* $\Longrightarrow io' \in$ *test-suite-to-io M T*
unfolding ‹*T* = *Test-Suite prs tps rd-targets atcs*›
proof −
fix *io′ io″* **assume** *io′* @ *io″* $\in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)

have *preamble-prop* : $\bigwedge$ *io‴ q P* . (*q*,*P*) $\in$ *prs* $\Longrightarrow io'@io''' \in L\ P \Longrightarrow io' \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
proof −
fix *io‴ q P* **assume** (*q*,*P*) $\in$ *prs* **and** $io'@io''' \in L\ P$

have $io' \in (\bigcup\ (q,P) \in prs\ .\ L\ P)$
using ‹$(q,P) \in prs$› *language-prefix*[*OF* ‹$io'@io''' \in L\ P$›] **by** *auto*
then show $io' \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
unfolding *test-suite-to-io.simps* **by** *blast*
qed

have *traversal-path-prop* : $\bigwedge io'''\ p\ pt\ q\ P\ .\ io'@io''' \in (\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')$ ‘ (*set* (*prefixes* (*p-io pt*))) $\Longrightarrow (q,P) \in prs \Longrightarrow$ *path P* (*initial P*) $p \Longrightarrow$ *target* (*initial P*) $p = q \Longrightarrow pt \in tps\ q \Longrightarrow io' \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
proof −
fix $io'''\ p\ pt\ q\ P$ **assume** $io'@io''' \in (\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')$ ‘ (*set* (*prefixes* (*p-io pt*)))
and $(q,P) \in prs$
and *path P* (*initial P*) p
and *target* (*initial P*) $p = q$
and $pt \in tps\ q$

obtain *ioP1* **where** $io'@io''' = p\text{-}io\ p\ @\ ioP1$ **and** $ioP1 \in$ (*set* (*prefixes* (*p-io pt*)))
using ‹$io'@io''' \in (\lambda\ io'\ .\ p\text{-}io\ p\ @\ io')$ ‘ (*set* (*prefixes* (*p-io pt*)))› **by** *auto*
then obtain *ioP2* **where** $ioP1\ @\ ioP2 = p\text{-}io\ pt$
unfolding *prefixes-set* **by** *blast*

show $io' \in$ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
proof (*cases* $length\ io' \leq length\ (p\text{-}io\ p)$)
case *True*
then have $io' = (take\ (length\ io')\ (p\text{-}io\ p))$
using ‹$io'@io''' = p\text{-}io\ p\ @\ ioP1$›
by (*metis* (*no-types*, *lifting*) *order-refl take-all take-le*)
then have $p\text{-}io\ p = io'@(drop\ (length\ io')\ (p\text{-}io\ p))$
by (*metis* (*no-types*, *lifting*) *append-take-drop-id*)
then have $io'@(drop\ (length\ io')\ (p\text{-}io\ p)) \in L\ P$
using *language-state-containment*[*OF* ‹*path P* (*initial P*) p›] **by** *blast*
then show *?thesis*
using *preamble-prop*[*OF* ‹$(q,P) \in prs$›] **by** *blast*
next
case *False*
then have $io' = p\text{-}io\ p\ @\ (take\ (length\ io' - length\ (p\text{-}io\ p))\ ioP1)$
using ‹$io'@io''' = p\text{-}io\ p\ @\ ioP1$›
by (*metis* (*no-types*, *lifting*) *le-cases take-all take-append take-le*)
moreover have $(take\ (length\ io' - length\ (p\text{-}io\ p))\ ioP1) \in$ (*set* (*prefixes* (*p-io pt*)))
proof −
have $ioP1 = (take\ (length\ io' - length\ (p\text{-}io\ p))\ ioP1)\ @\ (drop\ (length\ io' - length\ (p\text{-}io\ p))\ ioP1)$
by *auto*
then have $(take\ (length\ io' - length\ (p\text{-}io\ p))\ ioP1)\ @\ ((drop\ (length\ io'$

− *length* (*p-io p*)) *ioP1*) @ *ioP2*) = *p-io pt*
using ‹*ioP1* @ *ioP2* = *p-io pt*› **by** (*metis* (*mono-tags*, *lifting*) *append-assoc*)

then show *?thesis*
unfolding *prefixes-set* **by** *blast*
qed
ultimately have *io′* ∈ (λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*)))
by *blast*
then have *io′* ∈ (⋃{(λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . ∃ *q P* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q*})
using ‹(*q*,*P*) ∈ *prs*› ‹*path P* (*initial P*) *p*› ‹*target* (*initial P*) *p* = *q*› ‹*pt* ∈ *tps q*› **by** *blast*
then show *?thesis*
unfolding *test-suite-to-io.simps* **by** *blast*
qed
qed

from ‹*io′* @ *io″* ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)› **consider**
(*a*) *io′* @ *io″* ∈ (⋃ (*q*,*P*) ∈ *prs* . *L P*) |
(*b*) *io′* @ *io″* ∈ (⋃{(λ *io′* . *p-io p* @ *io′*) ‘ (*set* (*prefixes* (*p-io pt*))) | *p pt* . ∃ *q P* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q*}) |
(*c*) *io′* @ *io″* ∈ (⋃{(λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . ∃ *P q′ t1 t2* . (*q*,*P*) ∈ *prs* ∧ *path P* (*initial P*) *p* ∧ *target* (*initial P*) *p* = *q* ∧ *pt* ∈ *tps q* ∧ *q′* ∈ *rd-targets* (*q*,*pt*) ∧ (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*) })
unfolding *test-suite-to-io.simps*
by *blast*

then show *io′* ∈ *test-suite-to-io M* (*Test-Suite prs tps rd-targets atcs*)
proof *cases*
case *a*
then show *?thesis* **using** *preamble-prop* **by** *blast*
next
case *b*
then show *?thesis* **using** *traversal-path-prop* **by** *blast*
next
case *c*
then obtain *p pt q A P q′ t1 t2* **where** *io′* @ *io″* ∈ (λ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*)
and (*q*,*P*) ∈ *prs*
and *path P* (*initial P*) *p*
and *target* (*initial P*) *p* = *q*
and *pt* ∈ *tps q*
and *q′* ∈ *rd-targets* (*q*,*pt*)
and (*A*,*t1*,*t2*) ∈ *atcs* (*target q pt*,*q′*)
by *blast*

obtain *ioA* **where** *io′ @ io″ = p-io p @ p-io pt @ ioA*
and *ioA ∈ (atc-to-io-set (from-FSM M (target q pt)) A)*
using ‹*io′ @ io″ ∈ (λ io-atc . p-io p @ p-io pt @ io-atc) ‘ (atc-to-io-set (from-FSM M (target q pt)) A)*›
by *blast*

show *?thesis* **proof** (*cases length io′ ≤ length (p-io p @ p-io pt)*)
case *True*
then have *io′ @ (drop (length io′) (p-io p @ p-io pt)) = p-io p @ p-io pt*
using ‹*io′ @ io″ = p-io p @ p-io pt @ ioA*›
by (*simp add*: *append-eq-conv-conj*)
moreover have *p-io p @ p-io pt ∈ (λ io′ . p-io p @ io′) ‘ (set (prefixes (p-io pt)))*
unfolding *prefixes-set* **by** *blast*
ultimately have *io′@(drop (length io′) (p-io p @ p-io pt)) ∈ (λ io′ . p-io p @ io′) ‘ (set (prefixes (p-io pt)))*
by *simp*
then show *?thesis*
using *traversal-path-prop*[*OF* - ‹*(q,P) ∈ prs*› ‹*path P (initial P) p*› ‹*target (initial P) p = q*› ‹*pt ∈ tps q*›] **by** *blast*
next
case *False*
then have *io′ = (p-io p @ p-io pt) @ (take (length io′ − length (p-io p @ p-io pt)) ioA)*
proof −
have *io′ = take (length io′) (io′ @ io″)*
by *auto*
then show *?thesis*
using *False* ‹*io′ @ io″ = p-io p @ p-io pt @ ioA*› **by** *fastforce*
qed
moreover have (*take (length io′ − length (p-io p @ p-io pt)) ioA*) ∈ (*atc-to-io-set (from-FSM M (target q pt)) A*)
proof −
have (*take (length io′ − length (p-io p @ p-io pt)) ioA*)@(*drop (length io′ − length (p-io p @ p-io pt)) ioA*) ∈ *L (from-FSM M (target q pt)) ∩ L A*
using ‹*ioA ∈ (atc-to-io-set (from-FSM M (target q pt)) A)*› **by** *auto*
then have ∗: (*take (length io′ − length (p-io p @ p-io pt)) ioA*)@(*drop (length io′ − length (p-io p @ p-io pt)) ioA*) ∈ *L (from-FSM M (target q pt))*
and ∗∗: (*take (length io′ − length (p-io p @ p-io pt)) ioA*)@(*drop (length io′ − length (p-io p @ p-io pt)) ioA*) ∈ *L A*
by *blast+*
show *?thesis*
using *language-prefix*[*OF* ∗] *language-prefix*[*OF* ∗∗]
unfolding *atc-to-io-set.simps* **by** *blast*
qed
ultimately have *io′ ∈ (λ io-atc . p-io p @ p-io pt @ io-atc) ‘ (atc-to-io-set (from-FSM M (target q pt)) A)*

by *force*

then have $io' \in (\bigcup\{(\lambda$ *io-atc* . *p-io p* @ *p-io pt* @ *io-atc*) ‘ (*atc-to-io-set* (*from-FSM M* (*target q pt*)) *A*) | *p pt q A* . $\exists$ *P q′ t1 t2* . (*q*,*P*) $\in$ *prs* $\wedge$ *path P* (*initial P*) *p* $\wedge$ *target* (*initial P*) *p* = *q* $\wedge$ *pt* $\in$ *tps q* $\wedge$ *q′* $\in$ *rd-targets* (*q*,*pt*) $\wedge$ (*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q′*) }) **using** ‹(*q*,*P*) $\in$ *prs*› ‹*path P* (*initial P*) *p*› ‹*target* (*initial P*) *p* = *q*› ‹*pt* $\in$ *tps q*› ‹*q′* $\in$ *rd-targets* (*q*,*pt*)› ‹(*A*,*t1*,*t2*) $\in$ *atcs* (*target q pt*,*q′*)› **by** *blast*
then show *?thesis*
unfolding *test-suite-to-io.simps* **by** *blast*
qed
qed
qed

then show *?thesis*
using *pass-io-set-maximal-from-pass-io-set*[*OF p1*] **by** *blast*
qed

lemma *passes-test-suite-as-maximal-sequences-completeness* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
and *observable M′*
and *inputs M′* = *inputs M*
and *inputs M* $\neq$ {}
and *completely-specified M*
and *completely-specified M′*
and *size M′* $\leq$ *m*
shows (*L M′* $\subseteq$ *L M*) $\longleftrightarrow$ *pass-io-set-maximal M′* (*test-suite-to-io-maximal M T*)
unfolding *passes-test-suite-completeness*[*OF assms*(*1*,*3*−*9*)]
unfolding *test-suite-to-io-pass*[*OF assms*(*1*,*3*−*8*),*symmetric*]
unfolding *test-suite-to-io-pass-maximal*[*OF assms*(*1*,*2*)]
unfolding *test-suite-to-io-maximal-code*[*OF assms*(*1*,*2*,*3*)]
by *simp*

lemma *test-suite-to-io-maximal-finite* :
assumes *implies-completeness T M m*
and *is-finite-test-suite T*
and *observable M*
shows *finite* (*test-suite-to-io-maximal M T*)
using *test-suite-to-io-finite*[*OF assms*(*1*,*2*)]
unfolding *test-suite-to-io-maximal-code*[*OF assms*, *symmetric*]
by *simp*

end

43 Calculating Sufficient Test Suites

This theory describes algorithms to calculate test suites that satisfy the sufficiency criterion for a given specification FSM and upper bound m on the number of states in the System Under Test.

theory *Test-Suite-Calculation*
imports *Test-Suite-IO*
begin

43.1 Calculating Path Prefixes that are to be Extended With Adaptive Cest Cases

43.1.1 Calculating Tests along m-Traversal-Paths

fun *prefix-pair-tests* :: $'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ set \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ set$ **where**
$prefix\text{-}pair\text{-}tests\ q\ pds$
$= \bigcup\{\{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\} \mid p1\ p2\ .$
$\exists\ (p,(rd,dr)) \in pds\ .$
$(p1,p2) \in set\ (prefix\text{-}pairs\ p) \wedge$
$(target\ q\ p1) \in rd \wedge$
$(target\ q\ p2) \in rd \wedge$
$(target\ q\ p1) \neq (target\ q\ p2)\}$

lemma *prefix-pair-tests-code*[*code*] :
$prefix\text{-}pair\text{-}tests\ q\ pds = (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds))$
proof −
have $\bigwedge tp\ .\ tp \in prefix\text{-}pair\text{-}tests\ q\ pds \Longrightarrow tp \in (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds))$
proof −
fix *tp* **assume** $tp \in prefix\text{-}pair\text{-}tests\ q\ pds$
then obtain *tps* **where** $tp \in tps$
and $tps \in \{\{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\} \mid p1\ p2\ .\ \exists\ (p,(rd,dr)) \in pds\ .\ (p1,p2) \in set\ (prefix\text{-}pairs\ p) \wedge (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2)\}$
unfolding *prefix-pair-tests.simps*
by (*meson UnionE*)
then obtain *p1 p2* **where** $tps = \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\}$
and $\exists\ (p,(rd,dr)) \in pds\ .\ (p1,p2) \in set\ (prefix\text{-}pairs\ p) \wedge (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2)$
unfolding *mem-Collect-eq* **by** *blast*

then obtain *p rd dr* **where** $(p,(rd,dr)) \in pds$ **and** $(p1,p2) \in set\ (prefix\text{-}pairs\ p)$ **and** $(target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2)$
by *blast*

have *scheme* : $\bigwedge f\ x\ xs\ .\ x \in set\ xs \Longrightarrow f\ x \in set\ (map\ f\ xs)$
by *auto*

have $(p1,p2) \in set\ (filter\ (\lambda(p1,\ p2).\ target\ q\ p1 \in rd \wedge target\ q\ p2 \in rd \wedge target\ q\ p1 \neq target\ q\ p2)\ (prefix\text{-}pairs\ p))$
using ‹$(p1,p2) \in set\ (prefix\text{-}pairs\ p)$›
‹$(target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2)$›
by *auto*
have $\{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\} \in (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p))))$
using *scheme*[*OF* ‹$(p1,p2) \in set\ (filter\ (\lambda(p1,\ p2).\ target\ q\ p1 \in rd \wedge target\ q\ p2 \in rd \wedge target\ q\ p1 \neq target\ q\ p2)\ (prefix\text{-}pairs\ p))$›, *of* $(\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})$]
by *simp*

then show $tp \in (\bigcup (image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds))$
using ‹$tp \in tps$› ‹$(p,(rd,dr)) \in pds$›
unfolding ‹$tps = \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\}$›
by *blast*
qed
moreover have $\bigwedge tp\ .\ tp \in (\bigcup (image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds)) \Longrightarrow tp \in prefix\text{-}pair\text{-}tests\ q\ pds$
proof −
fix *tp* **assume** $tp \in (\bigcup (image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ pds))$
then obtain *prddr* **where** $prddr \in pds$
and $tp \in (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ prddr$
by *blast*
then obtain *p rd dr* **where** $prddr = (p,(rd,dr))$ **by** *auto*

then have $tp \in \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p))))$
using ‹$tp \in (\lambda\ (p,(rd,dr))\ .\ \bigcup\ (set\ (map\ (\lambda\ (p1,p2)\ .\ \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\})\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p)))))\ prddr$› **by** *auto*

then obtain *p1 p2* **where** $(p1,p2) \in set\ (filter\ (\lambda\ (p1,p2)\ .\ (target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2))\ (prefix\text{-}pairs\ p))$
and $tp \in \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\}$
by *auto*
then have $(target\ q\ p1) \in rd \wedge (target\ q\ p2) \in rd \wedge (target\ q\ p1) \neq (target\ q\ p2)$
and $(p1,p2) \in set\ (prefix\text{-}pairs\ p)$
by *auto*

then show $tp \in prefix\text{-}pair\text{-}tests\ q\ pds$
using ‹$prddr \in pds$› ‹$tp \in \{(q,p1,(target\ q\ p2)),\ (q,p2,(target\ q\ p1))\}$›
unfolding *prefix-pair-tests.simps* ‹$prddr = (p,(rd,dr))$›
by *blast*
qed
ultimately show *?thesis*
by *blast*
qed

43.1.2 Calculating Tests between Preambles

fun $preamble\text{-}prefix\text{-}tests'\ ::\ 'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ list \Rightarrow 'a\ list \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ list$ **where**
$preamble\text{-}prefix\text{-}tests'\ q\ pds\ drs =$
$concat\ (map\ (\lambda((p,(rd,dr)),q2,p1)\ .\ [(q,p1,q2),\ (q2,[],(target\ q\ p1))])$
$(filter\ (\lambda((p,(rd,dr)),q2,p1)\ .\ (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2)$
$(concat\ (map\ (\lambda((p,(rd,dr)),q2)\ .\ map\ (\lambda p1\ .\ ((p,(rd,dr)),q2,p1))\ (prefixes\ p))\ (List.product\ pds\ drs)))))$

definition $preamble\text{-}prefix\text{-}tests\ ::\ 'a \Rightarrow (('a,'b,'c)\ traversal\text{-}path \times ('a\ set \times 'a\ set))\ set \Rightarrow 'a\ set \Rightarrow ('a \times ('a,'b,'c)\ traversal\text{-}path \times 'a)\ set$ **where**
$preamble\text{-}prefix\text{-}tests\ q\ pds\ drs = \bigcup\{\{(q,p1,q2),\ (q2,[],(target\ q\ p1))\} \mid p1\ q2\ .\ \exists\ (p,(rd,dr)) \in pds\ .\ q2 \in drs \wedge (\exists\ p2\ .\ p = p1@p2) \wedge (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2\}$

lemma $preamble\text{-}prefix\text{-}tests\text{-}code[code]$:
$preamble\text{-}prefix\text{-}tests\ q\ pds\ drs = (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup(image\ (\lambda\ (p1,q2)\ .\ \{(q,p1,q2),\ (q2,[],(target\ q\ p1))\})\ (Set.filter\ (\lambda\ (p1,q2)\ .\ (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2)\ ((set\ (prefixes\ p)) \times drs))))\ pds))$
proof −
have $\bigwedge pp\ .\ pp \in preamble\text{-}prefix\text{-}tests\ q\ pds\ drs \Longrightarrow pp \in (\bigcup(image\ (\lambda\ (p,(rd,dr))\ .\ \bigcup(image\ (\lambda\ (p1,q2)\ .\ \{(q,p1,q2),\ (q2,[],(target\ q\ p1))\})\ (Set.filter\ (\lambda\ (p1,q2)\ .\ (target\ q\ p1) \in rd \wedge q2 \in rd \wedge (target\ q\ p1) \neq q2)\ ((set\ (prefixes\ p)) \times drs))))\ pds))$
proof −
fix *pp* **assume** $pp \in preamble\text{-}prefix\text{-}tests\ q\ pds\ drs$
then obtain *p1 q2* **where** $pp \in \{(q,p1,q2),\ (q2,[],(target\ q\ p1))\}$

and $\exists$ (p,(rd,dr)) $\in$ pds . q2 $\in$ drs $\wedge$ ($\exists$ p2 . p = p1@p2) $\wedge$ (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2
unfolding *preamble-prefix-tests-def* **by** *blast*
then obtain *p rd dr* **where** (p,(rd,dr)) $\in$ pds **and** q2 $\in$ drs **and** ($\exists$ p2 . p = p1@p2) **and** (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2
by *auto*

then have (p1,q2) $\in$ (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs))
unfolding *prefixes-set* **by** *force*
then show pp $\in$ ($\bigcup$(image (λ (p,(rd,dr)) . $\bigcup$(image (λ (p1,q2) . {(q,p1,q2), (q2,[],(target q p1))}) (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs)))) pds))
using ‹(p,(rd,dr)) $\in$ pds›
‹pp $\in$ {(q,p1,q2), (q2,[],(target q p1))}› **by** *blast*
qed
moreover have $\bigwedge$ pp . pp $\in$ ($\bigcup$(image (λ (p,(rd,dr)) . $\bigcup$(image (λ (p1,q2) . {(q,p1,q2), (q2,[],(target q p1))}) (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs)))) pds))
$\Longrightarrow$ pp $\in$ preamble-prefix-tests q pds drs
proof −
fix pp **assume** pp $\in$ ($\bigcup$(image (λ (p,(rd,dr)) . $\bigcup$(image (λ (p1,q2) . {(q,p1,q2), (q2,[],(target q p1))}) (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs)))) pds))
then obtain *prddr* **where** *prddr* $\in$ pds
and pp $\in$ (λ (p,(rd,dr)) . $\bigcup$(image (λ (p1,q2) . {(q,p1,q2), (q2,[],(target q p1))}) (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs)))) *prddr*
by *blast*

obtain *p rd dr* **where** *prddr* = (p,(rd,dr))
using *prod-cases3* **by** *blast*

obtain *p1 q2* **where** (p1,q2) $\in$ (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs))
and pp $\in$ {(q,p1,q2), (q2,[],(target q p1))}
using ‹pp $\in$ (λ (p,(rd,dr)) . $\bigcup$(image (λ (p1,q2) . {(q,p1,q2), (q2,[],(target q p1))}) (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs)))) *prddr*›
unfolding ‹*prddr* = (p,(rd,dr))›
by *blast*

have q2 $\in$ drs $\wedge$ ($\exists$ p2 . p = p1@p2) $\wedge$ (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2
using ‹(p1,q2) $\in$ (Set.filter (λ (p1,q2) . (target q p1) $\in$ rd $\wedge$ q2 $\in$ rd $\wedge$ (target q p1) $\neq$ q2) ((set (prefixes p)) $\times$ drs))›
unfolding *prefixes-set*
by *auto*
then have $\exists$(p, rd, dr)$\in$pds. q2 $\in$ drs $\wedge$ ($\exists$p2. p = p1 @ p2) $\wedge$ target q p1 $\in$

rd ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*
using ‹*prddr* ∈ *pds*› ‹*prddr* = (*p*,(*rd*,*dr*))›
by *blast*
then have ∗: {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))} ∈ {{(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)} |*p1 q2*.
∃(*p*, *rd*, *dr*)∈*pds*. *q2* ∈ *drs* ∧ (∃*p2*. *p* = *p1* @ *p2*) ∧ *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*} **by** *blast*

show *pp* ∈ *preamble-prefix-tests q pds drs*
using *UnionI*[*OF* ∗ ‹*pp* ∈ {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))}›]
unfolding *preamble-prefix-tests-def* **by** *assumption*
qed
ultimately show *?thesis* **by** *blast*
qed

43.1.3 Calculating Tests between m-Traversal-Paths Prefixes and Preambles

fun *preamble-pair-tests* :: *'a set set* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × (*'a*,*'b*,*'c*) *traversal-path* × *'a*) *set* **where**
preamble-pair-tests drss rds = (⋃ *drs* ∈ *drss* . (λ (*q1*,*q2*) . (*q1*,[],*q2*)) ‘ ((*drs* × *drs*) ∩ *rds*))

43.2 Calculating a Test Suite

definition *calculate-test-paths* ::
(*'a*,*'b*,*'c*) *fsm*
⇒ *nat*
⇒ *'a set*
⇒ (*'a* × *'a*) *set*
⇒ (*'a set* × *'a set*) *list*
⇒ ((*'a* ⇒ (*'a*,*'b*,*'c*) *traversal-path set*) × ((*'a* × (*'a*,*'b*,*'c*) *traversal-path*) ⇒ *'a set*))
where
calculate-test-paths M m d-reachable-states r-distinguishable-pairs repetition-sets =
(*let*
paths-with-witnesses
= (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q repetition-sets m*)) *d-reachable-states*);
get-paths
= *m2f* (*set-as-map paths-with-witnesses*);
PrefixPairTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *prefix-pair-tests q mrsps*;
PreamblePrefixTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *preamble-prefix-tests q mrsps d-reachable-states*;
PreamblePairTests

$= \textit{preamble-pair-tests}\ (\bigcup\ (q,pw) \in \textit{paths-with-witnesses}\ .\ ((\lambda\ (p,(rd,dr))\ .\ dr)\ `\ pw))\ \textit{r-distinguishable-pairs};$

$\textit{tests}$

$= \textit{PrefixPairTests} \cup \textit{PreamblePrefixTests} \cup \textit{PreamblePairTests};$

$\textit{tps}'$

$= \textit{m2f-by} \bigcup\ (\textit{set-as-map}\ (\textit{image}\ (\lambda\ (q,p)\ .\ (q,\ \textit{image fst}\ p))\ \textit{paths-with-witnesses}));$

$\textit{tps}''$

$= \textit{m2f}\ (\textit{set-as-map}\ (\textit{image}\ (\lambda\ (q,p,q')\ .\ (q,p))\ \textit{tests}));$

$\textit{tps}$

$= (\lambda\ q\ .\ \textit{tps}'\ q \cup \textit{tps}''\ q);$

$\textit{rd-targets}$

$= \textit{m2f}\ (\textit{set-as-map}\ (\textit{image}\ (\lambda\ (q,p,q')\ .\ ((q,p),q'))\ \textit{tests}))$

$\textit{in}$

$(\ \textit{tps},\ \textit{rd-targets}\ ))$

definition *combine-test-suite* ::

$('a,'b,'c)\ \textit{fsm}$

$\Rightarrow \textit{nat}$

$\Rightarrow ('a \times ('a,'b,'c)\ \textit{preamble})\ \textit{set}$

$\Rightarrow (('a \times 'a) \times (('d,'b,'c)\ \textit{separator} \times 'd \times 'd))\ \textit{set}$

$\Rightarrow ('a\ \textit{set} \times 'a\ \textit{set})\ \textit{list}$

$\Rightarrow ('a,'b,'c,'d)\ \textit{test-suite}$

where

$\textit{combine-test-suite}\ M\ m\ \textit{states-with-preambles}\ \textit{pairs-with-separators}\ \textit{repetition-sets} =$

$(\textit{let drs} = \textit{image fst states-with-preambles};$

$\textit{rds} = \textit{image fst pairs-with-separators};$

$\textit{tps-and-targets} = \textit{calculate-test-paths}\ M\ m\ \textit{drs rds repetition-sets};$

$\textit{atcs} = \textit{m2f}\ (\textit{set-as-map pairs-with-separators})$

$\textit{in}\ (\textit{Test-Suite states-with-preambles}\ (\textit{fst tps-and-targets})\ (\textit{snd tps-and-targets})\ \textit{atcs}))$

definition *calculate-test-suite-for-repetition-sets* ::

$('a{::}\textit{linorder},'b{::}\textit{linorder},'c)\ \textit{fsm} \Rightarrow \textit{nat} \Rightarrow ('a\ \textit{set} \times 'a\ \textit{set})\ \textit{list} \Rightarrow ('a,'b,'c,\ ('a \times 'a) + 'a)\ \textit{test-suite}$

where

$\textit{calculate-test-suite-for-repetition-sets}\ M\ m\ \textit{repetition-sets} =$

$(\textit{let}$

$\textit{states-with-preambles} = \textit{d-reachable-states-with-preambles}\ M;$

$\textit{pairs-with-separators} = \textit{image}\ (\lambda((q1,q2),A)\ .\ ((q1,q2),A,\textit{Inr}\ q1,\textit{Inr}\ q2))\ (\textit{r-distinguishable-state-pairs-with-separators}\ M)$

$\textit{in combine-test-suite}\ M\ m\ \textit{states-with-preambles pairs-with-separators repetition-sets})$

43.3 Sufficiency of the Calculated Test Suite

lemma *calculate-test-suite-for-repetition-sets-sufficient-and-finite* :
fixes M :: ($'a$::*linorder*,$'b$::*linorder*,$'c$) *fsm*
assumes *observable* M
and *completely-specified* M
and *inputs* $M \neq \{\}$
and $\bigwedge q.\ q \in FSM.states\ M \Longrightarrow \exists\, d{\in}set\ RepSets.\ q \in fst\ d$
and $\bigwedge d.\ d \in set\ RepSets \Longrightarrow fst\ d \subseteq states\ M \wedge (snd\ d = fst\ d \cap fst\ `$ *d-reachable-states-with-preambles* M)
and $\bigwedge\ q1\ q2\ d.\ d \in set\ RepSets \Longrightarrow q1{\in}fst\ d \Longrightarrow q2{\in}fst\ d \Longrightarrow q1 \neq q2 \Longrightarrow (q1,\ q2) \in fst\ `$ *r-distinguishable-state-pairs-with-separators* M
shows *implies-completeness* (*calculate-test-suite-for-repetition-sets* M m *RepSets*) M m
and *is-finite-test-suite* (*calculate-test-suite-for-repetition-sets* M m *RepSets*)
proof −
obtain *states-with-preambles tps rd-targets atcs* **where** *calculate-test-suite-for-repetition-sets* M m *RepSets*
= *Test-Suite states-with-preambles tps rd-targets atcs*
using *test-suite.exhaust* **by** *blast*

have $\bigwedge$ *a b c d* . *Test-Suite states-with-preambles tps rd-targets atcs* = *Test-Suite a b c d* $\Longrightarrow$ *tps* = *b*
by *blast*

have *states-with-preambles-def* : *states-with-preambles* = *d-reachable-states-with-preambles* M

and *tps-def* : *tps* = (λq. (*m2f-by* $\bigcup$ (*set-as-map* ((λ(q, p). (q, *fst* ` $p$)) ` (λq. (q, *m-traversal-paths-with-witness* M q *RepSets* m)) ` *fst* ` *d-reachable-states-with-preambles* M))) q
$\cup$ (*m2f* (*set-as-map* ((λ(q, p, q'). (q, p)) ` (($\bigcup q{\in}fst$ ` *d-reachable-states-with-preambles* M. $\bigcup$ (*prefix-pair-tests* q ` (*m2f* (*set-as-map* (($\lambda q$. ($q$, *m-traversal-paths-with-witness* $M$ $q$ *RepSets* $m$)) ` *fst* ` *d-reachable-states-with-preambles* M)) q)))
$\cup$ ($\bigcup q{\in}fst$ ` *d-reachable-states-with-preambles* $M$. $\bigcup mrsps{\in}$ *m2f* (*set-as-map* (($\lambda q$. ($q$, *m-traversal-paths-with-witness* $M$ $q$ *RepSets* $m$)) ` *fst* ` *d-reachable-states-with-preambles* $M$)) $q$ . *preamble-prefix-tests* $q$ *mrsps* (*fst* ` *d-reachable-states-with-preambles* M))
$\cup$ *preamble-pair-tests* ($\bigcup$(q, y)$\in$(λq. (q, *m-traversal-paths-with-witness* M q *RepSets* m)) ` *fst* ` *d-reachable-states-with-preambles* M. (λ(p, *rd*, *dr*). *dr*) ` $y$) (*fst* ` (λ(($q1$, $q2$), A). (($q1$, $q2$), A, *Inr* $q1$ :: $'a \times 'a + 'a$, *Inr* $q2$ :: $'a \times 'a + 'a$)) ` *r-distinguishable-state-pairs-with-separators* M))))) q)
and *rd-targets-def* : *rd-targets* = *m2f* (*set-as-map*
((λ(q, p, y). ((q, p), y)) ` (($\bigcup q{\in}fst$ ` *d-reachable-states-with-preambles* M. $\bigcup$ (*prefix-pair-tests* q ` (*m2f* (*set-as-map* (($\lambda q$. ($q$, *m-traversal-paths-with-witness* $M$ $q$ *RepSets* $m$)) ` *fst* ` *d-reachable-states-with-preambles* M)) q)))

$\cup$ ($\bigcup$ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*. $\bigcup$ *mrsps*∈ *m2f* (*set-as-map* ((λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*)) *q* . *preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*))
$\cup$ *preamble-pair-tests* ($\bigcup$ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))))
and *atcs-def* : *atcs* = *m2f* (*set-as-map* ((λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*))
using ‹*calculate-test-suite-for-repetition-sets M m RepSets* = *Test-Suite states-with-preambles tps rd-targets atcs*›[*symmetric*]
unfolding *calculate-test-suite-for-repetition-sets-def combine-test-suite-def Let-def calculate-test-paths-def fst-conv snd-conv* **by** *force*+

have *tps-alt-def*: $\bigwedge$ *q* . *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* ⟹
tps q = (*fst* ‘ *m-traversal-paths-with-witness M q RepSets m*) ∪
{*z*. (*q*, *z*)
∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘
((*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)) ∪
($\bigcup$ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
$\bigcup$ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests ($\bigcup$ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
and *rd-targets-alt-def*: $\bigwedge$ *q p* . *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* ⟹
rd-targets (*q*,*p*) = {*z*. ((*q*, *p*), *z*)
∈ (λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘
((*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)) ∪
($\bigcup$ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
$\bigcup$ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests ($\bigcup$ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
proof −
fix *q p* **assume** *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*

have *scheme0* : (*case set-as-map*

((λ(q, p). (q, fst ‘ p)) ‘
(λq. (q, m-traversal-paths-with-witness M q RepSets m)) ‘
fst ‘ d-reachable-states-with-preambles M)
q of
None ⇒ ⋃ {} | Some x ⇒ ⋃ x) = image fst (m-traversal-paths-with-witness M q RepSets m)
proof −
have ∗: ((λ(q, p). (q, fst ‘ p)) ‘
(λq. (q, m-traversal-paths-with-witness M q RepSets m)) ‘
fst ‘ d-reachable-states-with-preambles M)
= (λ q . (q , image fst (m-traversal-paths-with-witness M q RepSets m))) ‘ (fst ‘ d-reachable-states-with-preambles M)
by *force*
have ∗∗: ⋀ f q xs . (case set-as-map
((λq. (q, f q)) ‘ xs)
q of
None ⇒ ⋃ {} | Some xs ⇒ ⋃ xs) = (if q ∈ xs then ⋃ {f q} else ⋃ {})
unfolding *set-as-map-def* **by** *auto*

show *?thesis*
unfolding ∗ ∗∗
using ‹q ∈ fst ‘ d-reachable-states-with-preambles M›
by *auto*
qed

have *scheme1* : ⋀ f q xs . (case set-as-map
((λq. (q, f q)) ‘ xs)
q of
None ⇒ {} | Some xs ⇒ xs) = (if q ∈ xs then {f q} else {})
unfolding *set-as-map-def* **by** *auto*

have *scheme2*: (⋃q∈fst ‘ d-reachable-states-with-preambles M.
⋃ (prefix-pair-tests q ‘
(if q ∈ fst ‘ d-reachable-states-with-preambles M
then {m-traversal-paths-with-witness M q RepSets m} else {})))
= (⋃q∈fst ‘ d-reachable-states-with-preambles M. (⋃ (prefix-pair-tests q ‘ {m-traversal-paths-with-witness M q RepSets m})))
unfolding *set-as-map-def* **by** *auto*

have *scheme3*: (⋃q∈fst ‘ d-reachable-states-with-preambles M.
⋃mrsps∈if q ∈ fst ‘ d-reachable-states-with-preambles M
then {m-traversal-paths-with-witness M q RepSets m} else {}.
preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles M))
= (⋃q∈fst ‘ d-reachable-states-with-preambles M. (⋃mrsps∈{m-traversal-paths-with-witness

M q RepSets m} . *preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*)))
unfolding *set-as-map-def* **by** *auto*

have *scheme4* : (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)
= *image fst* (*r-distinguishable-state-pairs-with-separators M*)
by *force*

have ∗:*tps q* = (*fst* ‘ *m-traversal-paths-with-witness M q RepSets m*) ∪
{*z*. (*q*, *z*)
∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘
((⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*})) ∪
(⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests (⋃ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
unfolding *tps-def*
unfolding *scheme0 scheme1 scheme2 scheme3 scheme4*
unfolding *set-as-map-def*
by *auto*

have ∗∗: {*z*. (*q*, *z*)
∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘
((⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*})) ∪
(⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests (⋃ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
= {*z*. (*q*, *z*)
∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘
((*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)) ∪
(⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪

preamble-pair-tests (⋃ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
(**is** {*z*. (*q*, *z*) ∈ *?S1*} = {*z*. (*q*, *z*) ∈ *?S2*})
proof −
have ⋀ *z* . (*q*, *z*) ∈ *?S1* ⟹ (*q*, *z*) ∈ *?S2*
proof −
fix *z* **assume** (*q*, *z*) ∈ *?S1*
then consider (*q*, *z*) ∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘ (⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*}))
| (*q*,*z*) ∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘ (⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*))
| (*q*,*z*) ∈ (λ(*q*, *p*, *q*′). (*q*, *p*)) ‘ (*preamble-pair-tests* (⋃ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))
by *blast*
then show (*q*, *z*) ∈ *?S2* **proof** *cases*
case *1*
have *scheme*: ⋀ *f y xs* . *y* ∈ *image f xs* ⟹ ∃ *x* . *x* ∈ *xs* ∧ *f x* = *y* **by** *auto*

obtain *qzq* **where** *qzq* ∈ (⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*}))
and (λ(*q*, *p*, *q*′). (*q*, *p*)) *qzq* = (*q*,*z*)
using *scheme*[*OF 1*] **by** *blast*
then obtain *q*′ **where** *q*′∈*fst* ‘ *d-reachable-states-with-preambles M*
and *qzq* ∈ ⋃ (*prefix-pair-tests q*′ ‘ {*m-traversal-paths-with-witness M q*′ *RepSets m*})
by *blast*
then have *fst qzq* = *q*′
by *auto*
then have *q*′ = *q*
using ‹(λ(*q*, *p*, *q*′). (*q*, *p*)) *qzq* = (*q*,*z*)›
by (*simp add*: *prod.case-eq-if*)
then have *qzq* ∈ ⋃ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*})
using ‹*qzq* ∈ ⋃ (*prefix-pair-tests q*′ ‘ {*m-traversal-paths-with-witness M q*′ *RepSets m*})›
by *blast*
then have (λ(*q*, *p*, *q*′). (*q*, *p*)) *qzq* ∈ *?S2*
by *auto*
then show *?thesis*
unfolding ‹(λ(*q*, *p*, *q*′). (*q*, *p*)) *qzq* = (*q*,*z*)›

```
        by assumption
      next
        case 2
        then show ?thesis by blast
      next
        case 3
        then show ?thesis by blast
      qed
    qed
    moreover have ⋀ z . (q, z) ∈ ?S2 ⟹ (q, z) ∈ ?S1
      using ‹q ∈ fst ‘ d-reachable-states-with-preambles M› by blast
    ultimately show ?thesis
      by meson
  qed

  show tps q = (fst ‘ m-traversal-paths-with-witness M q RepSets m) ∪
              {z. (q, z)
                ∈ (λ(q, p, q′). (q, p)) ‘
                  ((prefix-pair-tests q (m-traversal-paths-with-witness M q RepSets
m)) ∪
                   (⋃ q∈fst ‘ d-reachable-states-with-preambles M.
                       ⋃ mrsps∈{m-traversal-paths-with-witness M q RepSets m}.
                    preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles
M)) ∪
                preamble-pair-tests (⋃ (q, y)∈(λq. (q, m-traversal-paths-with-witness
M q RepSets m)) ‘ fst ‘ d-reachable-states-with-preambles M. (λ(p, rd, dr). dr) ‘ y)
(fst ‘ (λ((q1, q2), A). ((q1, q2), A, Inr q1 :: ′a × ′a + ′a, Inr q2 :: ′a × ′a + ′a))
‘ r-distinguishable-state-pairs-with-separators M))}
    using ∗ unfolding ∗∗ by assumption

  have ∗∗∗: rd-targets (q,p) = {z. ((q, p), z)
                ∈ (λ(q, p, y). ((q, p), y)) ‘
                  ((⋃ q∈fst ‘ d-reachable-states-with-preambles M.
                       ⋃ (prefix-pair-tests q ‘ {m-traversal-paths-with-witness M q
RepSets m})) ∪
                   (⋃ q∈fst ‘ d-reachable-states-with-preambles M.
                       ⋃ mrsps∈{m-traversal-paths-with-witness M q RepSets m}.
                    preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles
M)) ∪
                preamble-pair-tests (⋃ (q, y)∈(λq. (q, m-traversal-paths-with-witness
M q RepSets m)) ‘ fst ‘ d-reachable-states-with-preambles M. (λ(p, rd, dr). dr) ‘ y)
(fst ‘ (λ((q1, q2), A). ((q1, q2), A, Inr q1 :: ′a × ′a + ′a, Inr q2 :: ′a × ′a + ′a))
‘ r-distinguishable-state-pairs-with-separators M))}
    unfolding rd-targets-def
    unfolding scheme1 scheme2 scheme3 scheme4
    unfolding set-as-map-def
    by auto
```

have ∗∗∗∗: {z. ((q, p), z)
∈ (λ(q, p, y). ((q, p), y)) ‘
((⋃ q∈fst ‘ d-reachable-states-with-preambles M.
⋃ (prefix-pair-tests q ‘ {m-traversal-paths-with-witness M q RepSets m})) ∪
(⋃ q∈fst ‘ d-reachable-states-with-preambles M.
⋃ mrsps∈{m-traversal-paths-with-witness M q RepSets m}.
preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles M)) ∪
preamble-pair-tests (⋃ (q, y)∈(λq. (q, m-traversal-paths-with-witness M q RepSets m)) ‘ fst ‘ d-reachable-states-with-preambles M. (λ(p, rd, dr). dr) ‘ y) (fst ‘ (λ((q1, q2), A). ((q1, q2), A, Inr q1 :: ′a × ′a + ′a, Inr q2 :: ′a × ′a + ′a)) ‘ r-distinguishable-state-pairs-with-separators M))}
= {z. ((q, p), z)
∈ (λ(q, p, y). ((q, p), y)) ‘
((prefix-pair-tests q (m-traversal-paths-with-witness M q RepSets m)) ∪
(⋃ q∈fst ‘ d-reachable-states-with-preambles M.
⋃ mrsps∈{m-traversal-paths-with-witness M q RepSets m}.
preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles M)) ∪
preamble-pair-tests (⋃ (q, y)∈(λq. (q, m-traversal-paths-with-witness M q RepSets m)) ‘ fst ‘ d-reachable-states-with-preambles M. (λ(p, rd, dr). dr) ‘ y) (fst ‘ (λ((q1, q2), A). ((q1, q2), A, Inr q1 :: ′a × ′a + ′a, Inr q2 :: ′a × ′a + ′a)) ‘ r-distinguishable-state-pairs-with-separators M))}
(**is** {z. ((q, p), z) ∈ ?S1} = {z. ((q, p), z) ∈ ?S2})
proof −
have ⋀ z . ((q, p), z) ∈ ?S1 ⟹ ((q, p), z) ∈ ?S2
proof −
fix z **assume** ((q, p), z) ∈ ?S1
then consider ((q, p), z) ∈ (λ(q, p, y). ((q, p), y)) ‘ (⋃ q∈fst ‘ d-reachable-states-with-preambles M.
⋃ (prefix-pair-tests q ‘ {m-traversal-paths-with-witness M q RepSets m}))
| ((q, p), z) ∈ (λ(q, p, y). ((q, p), y)) ‘ (⋃ q∈fst ‘ d-reachable-states-with-preambles M.
⋃ mrsps∈{m-traversal-paths-with-witness M q RepSets m}.
preamble-prefix-tests q mrsps (fst ‘ d-reachable-states-with-preambles M))
| ((q, p), z) ∈ (λ(q, p, y). ((q, p), y)) ‘ (preamble-pair-tests (⋃ (q, y)∈(λq. (q, m-traversal-paths-with-witness M q RepSets m)) ‘ fst ‘ d-reachable-states-with-preambles M. (λ(p, rd, dr). dr) ‘ y) (fst ‘ (λ((q1, q2), A). ((q1, q2), A, Inr q1 :: ′a × ′a + ′a, Inr q2 :: ′a × ′a + ′a)) ‘ r-distinguishable-state-pairs-with-separators M))
by *blast*
then show ((q, p), z) ∈ ?S2 **proof** *cases*
case *1*
have *scheme*: ⋀ f y xs . y ∈ image f xs ⟹ ∃ x . x ∈ xs ∧ f x = y **by** *auto*

obtain *qzq* **where** *qzq* ∈ (⋃ q∈fst ‘ d-reachable-states-with-preambles M.

$\bigcup$ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*}))
and (λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) *qzq* = ((*q*,*p*),*z*)
using *scheme*[*OF 1*] **by** *blast*
then obtain *q′* **where** *q′*∈*fst* ‘ *d-reachable-states-with-preambles M*
and *qzq* ∈ $\bigcup$ (*prefix-pair-tests q′* ‘ {*m-traversal-paths-with-witness M q′ RepSets m*})
by *blast*
then have *fst qzq* = *q′*
by *auto*
then have *q′* = *q*
using ‹(λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) *qzq* = ((*q*,*p*),*z*)›
by (*simp add*: *prod.case-eq-if*)
then have *qzq* ∈ $\bigcup$ (*prefix-pair-tests q* ‘ {*m-traversal-paths-with-witness M q RepSets m*})
using ‹*qzq* ∈ $\bigcup$ (*prefix-pair-tests q′* ‘ {*m-traversal-paths-with-witness M q′ RepSets m*})›
by *blast*
then have (λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) *qzq* ∈ *?S2*
by *auto*
then show *?thesis*
unfolding ‹(λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) *qzq* = ((*q*,*p*),*z*)›
by *assumption*
next
case *2*
then show *?thesis* **by** *blast*
next
case *3*
then show *?thesis* **by** *blast*
qed
qed
moreover have $\bigwedge$ *z* . ((*q*, *p*), *z*) ∈ *?S2* ⟹ ((*q*, *p*), *z*) ∈ *?S1*
using ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*› **by** *blast*
ultimately show *?thesis*
by *meson*
qed

show *rd-targets* (*q*,*p*) = {*z*. ((*q*, *p*), *z*)
∈ (λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘
((*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)) ∪
($\bigcup$ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
$\bigcup$ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests ($\bigcup$ (*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: ′*a* × ′*a* + ′*a*, *Inr q2* :: ′*a* × ′*a* + ′*a*)) ‘ *r-distinguishable-state-pairs-with-separators M*))}
using *** **unfolding** **** **by** *assumption*

qed

define *pps-alt* :: ('a × ('a,'b,'c) *traversal-path* × 'a) *set* **where** *pps-alt-def* : *pps-alt* = {(*q1*,[],*q2*) | *q1 q2* . ∃ *q p rd dr* . *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* ∧ (*p*,(*rd*,*dr*)) ∈ *m-traversal-paths-with-witness M q RepSets m* ∧ *q1* ∈ *dr* ∧ *q2* ∈ *dr* ∧ (*q1*,*q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*}

have *preamble-pair-tests-alt* :

preamble-pair-tests (⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1* :: 'a × 'a + 'a, *Inr q2* :: 'a × 'a + 'a)) ‘ *r-distinguishable-state-pairs-with-separator M*)

= *pps-alt*

(**is** *?PP1* = *?PP2*)

proof −

have ⋀ *x* . *x* ∈ *?PP1* ⟹ *x* ∈ *?PP2*

proof −

fix *x* **assume** *x* ∈ *?PP1*

then obtain *drs* **where** *drs* ∈ (⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*)

and *x* ∈ (λ(*q1*, *q2*). (*q1*, [], *q2*)) ‘ (*drs* × *drs* ∩ *fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)

unfolding *preamble-pair-tests.simps* **by** *force*

obtain *q y* **where** (*q*,*y*) ∈ (λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*

and *drs* ∈ (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*

using ‹*drs* ∈ (⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*. (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*)›

by *force*

have *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*

and *y* = *m-traversal-paths-with-witness M q RepSets m*

using ‹(*q*,*y*) ∈ (λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘ *fst* ‘ *d-reachable-states-with-preambles M*›

by *force*+

obtain *p rd* **where** (*p*,(*rd*,*drs*)) ∈ *m-traversal-paths-with-witness M q RepSets m*

using ‹*drs* ∈ (λ(*p*, *rd*, *dr*). *dr*) ‘ *y*› **unfolding** ‹*y* = *m-traversal-paths-with-witness M q RepSets m*›

by *force*

obtain *q1 q2* **where** (*q1*,*q2*) ∈ (*drs* × *drs* ∩ *fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)

and *x* = (*q1*, [], *q2*)

using ‹*x* ∈ (λ(*q1*, *q2*). (*q1*, [], *q2*)) ‘ (*drs* × *drs* ∩ *fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)›

by *force*

have $q1 \in drs \wedge q2 \in drs \wedge (q1,q2) \in fst$ ‘ *r-distinguishable-state-pairs-with-separators M*
using ‹$(q1,q2) \in (drs \times drs \cap fst$ ‘ $(\lambda((q1,\ q2),\ A).\ ((q1,\ q2),\ A,\ Inr\ q1,\ Inr\ q2))$ ‘ *r-distinguishable-state-pairs-with-separators M*)›
by *force*

then show $x \in ?PP2$
unfolding ‹$x = (q1,\ [],\ q2)$› *pps-alt-def*
using ‹$q \in fst$ ‘ *d-reachable-states-with-preambles M*› ‹$(p,(rd,drs)) \in$ *m-traversal-paths-with-witness M q RepSets m*›
by *blast*
qed

moreover have $\bigwedge x\ .\ x \in ?PP2 \Longrightarrow x \in ?PP1$
proof −
fix x **assume** $x \in ?PP2$
then obtain $q1\ q2$ **where** $x = (q1,[],q2)$ **unfolding** *pps-alt-def*
by *auto*
then obtain $q\ p\ rd\ dr$ **where** $q \in fst$ ‘ *d-reachable-states-with-preambles M*
and $(p,(rd,dr)) \in$ *m-traversal-paths-with-witness M q RepSets m*
and $q1 \in dr \wedge q2 \in dr \wedge (q1,q2) \in fst$ ‘ *r-distinguishable-state-pairs-with-separators M*
using ‹$x \in ?PP2$› **unfolding** *pps-alt-def* **by** *blast*

have $dr \in (\bigcup(q,\ y) \in (\lambda q.\ (q,$ *m-traversal-paths-with-witness M q RepSets m*$))$ ‘ fst ‘ *d-reachable-states-with-preambles M*. $(\lambda(p,\ rd,\ dr).\ dr)$ ‘ $y)$
using ‹$q \in fst$ ‘ *d-reachable-states-with-preambles M*› ‹$(p,(rd,dr)) \in$ *m-traversal-paths-with-witness M q RepSets m*› **by** *force*

moreover have $x \in (\lambda(q1,\ q2).\ (q1,\ [],\ q2))$ ‘ $(dr \times dr \cap fst$ ‘ $(\lambda((q1,\ q2),\ A).\ ((q1,\ q2),\ A,\ Inr\ q1,\ Inr\ q2))$ ‘ *r-distinguishable-state-pairs-with-separators M*)
unfolding ‹$x = (q1,[],q2)$› **using** ‹$q1 \in dr \wedge q2 \in dr \wedge (q1,q2) \in fst$ ‘ *r-distinguishable-state-pairs-with-separators M*› **by** *force*

ultimately show $x \in ?PP1$
unfolding *preamble-pair-tests.simps* **by** *force*
qed

ultimately show *?thesis* **by** *blast*
qed

have *p1*: (*initial M*,*initial-preamble M*) $\in$ *states-with-preambles*
using *fsm-initial*[*of M*]
unfolding *states-with-preambles-def d-reachable-states-with-preambles-def calculate-state-preamble-from-input-choices.simps* **by** *force*

have *p2a*: $\bigwedge$ *q P* . (*q*,*P*) $\in$ *states-with-preambles* $\Longrightarrow$ *is-preamble P M q*
using *assms*(*1*) *d-reachable-states-with-preambles-soundness*(*1*) *states-with-preambles-def* **by** *blast*

have *p2b*: $\bigwedge$ *q P* . (*q*,*P*) $\in$ *states-with-preambles* $\Longrightarrow$ (*tps q*) $\neq$ {}
proof −
fix *q P* **assume** (*q*,*P*) $\in$ *states-with-preambles*
then have *q* $\in$ (*image fst* (*d-reachable-states-with-preambles M*))
unfolding *states-with-preambles-def*
by (*simp add*: *rev-image-eqI*)

have *q* $\in$ *states M*
using ‹(*q*, *P*) $\in$ *states-with-preambles*› *assms*(*1*) *d-reachable-states-with-preambles-soundness*(*2*) *states-with-preambles-def* **by** *blast*

obtain *p′ d′* **where** (*p′*, *d′*) $\in$ *m-traversal-paths-with-witness M q RepSets m*
using *m-traversal-path-exist*[*OF assms*(*2*) ‹*q* $\in$ *states M*› *assms*(*3*) ‹$\bigwedge$*q*. *q* $\in$ *FSM.states M* $\Longrightarrow$ $\exists$ *d*$\in$*set RepSets*. *q* $\in$ *fst d*›] *assms*(*5*)
by *blast*
then have *p′* $\in$ *image fst* (*m-traversal-paths-with-witness M q RepSets m*)
using *image-iff* **by** *fastforce*

have (*q*, *image fst* (*m-traversal-paths-with-witness M q RepSets m*)) $\in$ (*image* (λ (*q*,*p*) . (*q*, *image fst p*)) (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q RepSets m*)) (*image fst* (*d-reachable-states-with-preambles M*))))
using ‹*q* $\in$ (*image fst* (*d-reachable-states-with-preambles M*))› **by** *force*
have (*image fst* (*m-traversal-paths-with-witness M q RepSets m*)) $\in$ (*m2f* (*set-as-map* (*image* (λ (*q*,*p*) . (*q*, *image fst p*)) (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q RepSets m*)) (*image fst* (*d-reachable-states-with-preambles M*)))))) *q*
using *set-as-map-containment*[*OF* ‹(*q*, *image fst* (*m-traversal-paths-with-witness M q RepSets m*)) $\in$ (*image* (λ (*q*,*p*) . (*q*, *image fst p*)) (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q RepSets m*)) (*image fst* (*d-reachable-states-with-preambles M*))))›]
by *assumption*
then have *p′* $\in$ ($\bigcup$ ((*m2f* (*set-as-map* (*image* (λ (*q*,*p*) . (*q*, *image fst p*)) (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q RepSets m*)) (*image fst* (*d-reachable-states-with-preambles M*)))))) *q*))
using ‹*p′* $\in$ *image fst* (*m-traversal-paths-with-witness M q RepSets m*)› **by** *blast*

then show (*tps q*) $\neq$ {}
unfolding *tps-def m2f-by-from-m2f* **by** *blast*
qed

have *p2*: ($\forall$ *q P*. (*q*, *P*) $\in$ *states-with-preambles* $\longrightarrow$ *is-preamble P M q* $\wedge$ *tps q* $\neq$ {})
using *p2a p2b* **by** *blast*

have $\bigwedge$ *q1 q2 A d1 d2* . ((*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*)) ⟹ ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separa*
M ∧ *d1* = *Inr q1* ∧ *d2* = *Inr q2*
proof −
fix *q1 q2 A d1 d2* **assume** ((*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*))
then have *atcs* (*q1*,*q2*) = {*z*. ((*q1*, *q2*), *z*) ∈ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*,
Inr q1, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*}
unfolding *atcs-def set-as-map-def* **by** *auto*
then show ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M* ∧ *d1*
= *Inr q1* ∧ *d2* = *Inr q2*
using ‹((*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*))› **by** *auto*
qed

have $\bigwedge$ *q1 q2 A d1 d2* . (*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*) ⟹ (*A*,*d2*,*d1*) ∈ *atcs* (*q2*,*q1*)
∧ *is-separator M q1 q2 A d1 d2*
proof −
fix *q1 q2 A d1 d2* **assume** (*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*)
then have ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M* **and**
d1 = *Inr q1* **and** *d2* = *Inr q2*
using ‹$\bigwedge$ *q1 q2 A d1 d2* . ((*A*,*d1*,*d2*) ∈ *atcs* (*q1*,*q2*)) ⟹ ((*q1*,*q2*),*A*) ∈
r-distinguishable-state-pairs-with-separators M ∧ *d1* = *Inr q1* ∧ *d2* = *Inr q2*›
by *blast+*
then have ((*q2*,*q1*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M*
unfolding *r-distinguishable-state-pairs-with-separators-def*
by *auto*
then have (*A*,*d2*,*d1*) ∈ *atcs* (*q2*,*q1*)
unfolding *atcs-def* ‹*d1* = *Inr q1*› ‹*d2* = *Inr q2*› *set-as-map-def* **by** *force*
moreover have *is-separator M q1 q2 A d1 d2*
using *r-distinguishable-state-pairs-with-separators-elem-is-separator*[*OF* ‹((*q1*,*q2*),*A*)
∈ *r-distinguishable-state-pairs-with-separators M*› *assms*(*1*,*2*)]
unfolding ‹*d1* = *Inr q1*› ‹*d2* = *Inr q2*›
by *assumption*
ultimately show (*A*,*d2*,*d1*) ∈ *atcs* (*q2*,*q1*) ∧ *is-separator M q1 q2 A d1 d2*
by *simp*
qed
then have *p3* : (∀ *q1 q2 A d1 d2*. (*A*, *d1*, *d2*) ∈ *atcs* (*q1*, *q2*) ⟶ (*A*, *d2*, *d1*)
∈ *atcs* (*q2*, *q1*) ∧ *is-separator M q1 q2 A d1 d2*)
by *blast*

have *p4*: $\bigwedge$ *q* . *q* ∈ *states M* ⟹ (∃ *d* ∈ *set RepSets*. *q* ∈ *fst d*)
by (*simp add*: *assms*(*4*))

have *p5*: $\bigwedge$ *d* . *d* ∈ *set RepSets* ⟹ ((*fst d* ⊆ *states M*) ∧ (*snd d* = *fst d* ∩ *fst*
‘ *states-with-preambles*) ∧ (∀ *q1 q2* . *q1* ∈ *fst d* ⟶ *q2* ∈ *fst d* ⟶ *q1* ≠ *q2* ⟶
atcs (*q1*,*q2*) ≠ {}))
proof −
fix *d* **assume** *d* ∈ *set RepSets*

then have $\bigwedge$ *q1 q2* . *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *atcs* (*q1*,*q2*)

$\neq \{\}$
proof −
fix *q1 q2* **assume** *q1* ∈ *fst d* **and** *q2* ∈ *fst d* **and** *q1* ≠ *q2*
then have (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
using *assms*(*6*)[*OF* ‹*d* ∈ *set RepSets*›] **by** *blast*
then obtain *A* **where** ((*q1*,*q2*),*A*) ∈ *r-distinguishable-state-pairs-with-separators M*
by *auto*
then have (*A*,*Inr q1*,*Inr q2*) ∈ *atcs* (*q1*,*q2*)
unfolding *atcs-def set-as-map-def*
by *force*
then show *atcs* (*q1*,*q2*) ≠ {}
by *blast*
qed
then show ((*fst d* ⊆ *states M*) ∧ (*snd d* = *fst d* ∩ *fst* ‘ *states-with-preambles*) ∧ (∀ *q1 q2* . *q1* ∈ *fst d* ⟶ *q2* ∈ *fst d* ⟶ *q1* ≠ *q2* ⟶ *atcs* (*q1*,*q2*) ≠ {}))
using *assms*(*5*)[*OF* ‹*d* ∈ *set RepSets*›] **unfolding** *states-with-preambles-def* **by** *blast*
qed

have *p6* : ⋀ *q* . *q* ∈ *image fst states-with-preambles* ⟹ *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q RepSets m*} ∧ *fst* ‘ (*m-traversal-paths-with-witness M q RepSets m*) ⊆ *tps q*
proof
fix *q* **assume** *q* ∈ *image fst states-with-preambles*
then have *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*
unfolding *states-with-preambles-def* **by** *assumption*
then have *q* ∈ *states M*
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *d-reachable-states-with-preambles-soundness*(*2*) *image-iff prod.collapse*)
show *fst* ‘ *m-traversal-paths-with-witness M q RepSets m* ⊆ *tps q*
unfolding *tps-alt-def*[*OF* ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *blast*

show *tps q* ⊆ {*p1* . ∃ *p2 d* . (*p1*@*p2*,*d*) ∈ *m-traversal-paths-with-witness M q RepSets m*}
proof
fix *p* **assume** *p* ∈ *tps q*

have ∗ : (⋀ *q* . *q* ∈ *states M* ⟹ (∃ *d* ∈ *set RepSets*. *q* ∈ *fst d*))
using *p4* **by** *blast*

have ∗∗: (⋀ *d* . *d* ∈ *set RepSets* ⟹ (*snd d* ⊆ *fst d*))
using *p5* **by** *simp*

from ‹*p* ∈ *tps q*› **consider**
(*a*) *p* ∈ *fst* ‘ *m-traversal-paths-with-witness M q RepSets m* |

(*b*) $(q, p) \in (\lambda(q, p, q').\ (q, p))$ ‘ (*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)) |
(*c*) $(q, p) \in (\lambda(q, p, q').\ (q, p))$ ‘ ($\bigcup$ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
$\bigcup$ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) |
(*d*) $(q, p) \in (\lambda(q, p, q').\ (q, p))$ ‘ *pps-alt*
unfolding *tps-alt-def*[*OF* ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›] *preamble-pair-tests-alt* **by** *blast*

then show *p* ∈ {*p1*. ∃ *p2 d*. (*p1* @ *p2*, *d*) ∈ *m-traversal-paths-with-witness M q RepSets m*}
proof *cases*
case *a*
then obtain *d* **where** (*p*,*d*) ∈ *m-traversal-paths-with-witness M q RepSets m*
by *auto*
then have ∃ *p2 d*. (*p* @ *p2*, *d*) ∈ *m-traversal-paths-with-witness M q RepSets m*
by (*metis append-eq-append-conv2*)
then show *?thesis*
by *blast*
next
case *b*

obtain *p1 p2* **where** (*q*,*p*) ∈ ((λ(*q*, *p*, *q*′). (*q*, *p*)) ‘{(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})
and ∃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q RepSets m*.
(*p1*, *p2*) ∈ *set* (*prefix-pairs p*) ∧ *target q p1* ∈ *rd* ∧ *target q p2* ∈ *rd* ∧ *target q p1* ≠ *target q p2*
using *b*
unfolding *prefix-pair-tests.simps* **by** *blast*

obtain *p*′ *d* **where** (*p*′, *d*) ∈ *m-traversal-paths-with-witness M q RepSets m*
and (*p1*, *p2*) ∈ *set* (*prefix-pairs p*′)
using ‹∃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q RepSets m*.
(*p1*, *p2*) ∈ *set* (*prefix-pairs p*) ∧ *target q p1* ∈ *rd* ∧ *target q p2* ∈ *rd* ∧ *target q p1* ≠ *target q p2*›
by *blast*

have ∃ *p*′′ . *p*′ = *p* @ *p*′′
using ‹(*p1*, *p2*) ∈ *set* (*prefix-pairs p*′)› **unfolding** *prefix-pairs-set-alt*
using ‹(*q*,*p*) ∈ ((λ(*q*, *p*, *q*′). (*q*, *p*)) ‘{(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})› **by** *auto*
then show *?thesis*
using ‹(*p*′, *d*) ∈ *m-traversal-paths-with-witness M q RepSets m*›
by *blast*
next

case *c*

obtain *q′* **where** *q′* ∈ *fst* ‘ *d-reachable-states-with-preambles M*
and (*q*,*p*) ∈ (λ(*q*, *p*, *q′*). (*q*, *p*)) ‘ (*preamble-prefix-tests q′* (*m-traversal-paths-with-witness M q′ RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*))
using *c* **by** *blast*

obtain *p1 q2* **where** (*q*,*p*) ∈ ((λ(*q*, *p*, *q′*). (*q*, *p*)) ‘{(*q′*, *p1*, *q2*), (*q2*, [], *target q′ p1*)})
and ∃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q′ RepSets m*.
q2 ∈ *fst* ‘ *d-reachable-states-with-preambles M* ∧ (∃ *p2*. *p* = *p1* @ *p2*) ∧ *target q′ p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q′ p1* ≠ *q2*
using ‹(*q*,*p*) ∈ (λ(*q*, *p*, *q′*). (*q*, *p*)) ‘ (*preamble-prefix-tests q′* (*m-traversal-paths-with-witness M q′ RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*))›
unfolding *preamble-prefix-tests-def*
by *blast*

obtain *p2 d* **where** (*p1*@*p2*, *d*)∈*m-traversal-paths-with-witness M q′ RepSets m*
and *q2* ∈ *fst* ‘ *d-reachable-states-with-preambles M*
using ‹∃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q′ RepSets m*.
q2 ∈ *fst* ‘ *d-reachable-states-with-preambles M* ∧ (∃ *p2*. *p* = *p1* @ *p2*) ∧ *target q′ p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q′ p1* ≠ *q2*›
by *blast*

consider (*a*) *q* = *q′* ∧ *p* = *p1* | (*b*) *q* = *q2* ∧ *p* = []
using ‹(*q*,*p*) ∈ ((λ(*q*, *p*, *q′*). (*q*, *p*)) ‘{(*q′*, *p1*, *q2*), (*q2*, [], *target q′ p1*)})› **by** *auto*
then show *?thesis* **proof** *cases*
case *a*
then show *?thesis*
using ‹(*p1* @ *p2*, *d*) ∈ *m-traversal-paths-with-witness M q′ RepSets m*› **by** *blast*

next
case *b*

then have *q* ∈ *states M* **and** *p* = []
using ‹*q2* ∈ *fst* ‘ *d-reachable-states-with-preambles M*› **unfolding** *d-reachable-states-with-preambles-def* **by** *auto*

have ∃ *p′ d′*. (*p′*, *d′*) ∈ *m-traversal-paths-with-witness M q RepSets m*
using *m-traversal-path-exist*[*OF assms*(*2*) ‹*q* ∈ *states M*› *assms*(*3*) ∗ ∗∗]
by *blast*
then show *?thesis*
unfolding ‹*p* = []›
by *simp*
qed

next
case d
then have $p = []$
unfolding *pps-alt-def* **by** *force*

have $q \in$ *states M*
using ‹$q \in$ *fst* ‘ *d-reachable-states-with-preambles M*› **unfolding** *d-reachable-states-with-preambles-def* **by** *auto*

have $\exists\, p'\ d'.\ (p',\ d') \in$ *m-traversal-paths-with-witness M q RepSets m*
using *m-traversal-path-exist*[*OF assms*(*2*) ‹$q \in$ *states M*› *assms*(*3*) $*$ $**$]
by *blast*
then show *?thesis*
unfolding ‹$p = []$›
by *simp*
qed
qed
qed

have *p7* : $\bigwedge$ *q p d* . $q \in$ *image fst states-with-preambles* $\Longrightarrow$ $(p,d) \in$ *m-traversal-paths-with-witness M q RepSets m* $\Longrightarrow$
(($\forall$ *p1 p2 p3* . *p=p1@p2@p3* $\longrightarrow$ *p2* $\neq$ [] $\longrightarrow$ *target q p1* $\in$ *fst d* $\longrightarrow$ *target q* (*p1@p2*) $\in$ *fst d* $\longrightarrow$ *target q p1* $\neq$ *target q* (*p1@p2*) $\longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ (*p1@p2*) $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*,(*p1@p2*)) $\wedge$ *target q* (*p1@p2*) $\in$ *rd-targets* (*q*,*p1*)))
$\wedge$ ($\forall$ *p1 p2 q'* . *p=p1@p2* $\longrightarrow$ *q'* $\in$ *image fst states-with-preambles* $\longrightarrow$ *target q p1* $\in$ *fst d* $\longrightarrow$ *q'* $\in$ *fst d* $\longrightarrow$ *target q p1* $\neq$ *q'* $\longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ [] $\in$ *tps q'* $\wedge$ *target q p1* $\in$ *rd-targets* (*q'*,[]) $\wedge$ *q'* $\in$ *rd-targets* (*q*,*p1*)))
$\wedge$ ($\forall$ *q1 q2* . *q1* $\neq$ *q2* $\longrightarrow$ *q1* $\in$ *snd d* $\longrightarrow$ *q2* $\in$ *snd d* $\longrightarrow$ ([] $\in$ *tps q1* $\wedge$ [] $\in$ *tps q2* $\wedge$ *q1* $\in$ *rd-targets* (*q2*,[]) $\wedge$ *q2* $\in$ *rd-targets* (*q1*,[]))))
proof −
fix *q p d* **assume** $q \in$ *image fst states-with-preambles* **and** $(p,d) \in$ *m-traversal-paths-with-witness M q RepSets m*
then have $(p,(fst\ d,\ snd\ d)) \in$ *m-traversal-paths-with-witness M q RepSets m* **by** *auto*

have $q \in$ *fst* ‘ *d-reachable-states-with-preambles M*
using ‹$q \in$ *image fst states-with-preambles*› **unfolding** *states-with-preambles-def* **by** *assumption*

have *p7c1* : $\bigwedge$ *p1 p2 p3* . *p=p1@p2@p3* $\Longrightarrow$ *p2* $\neq$ [] $\Longrightarrow$ *target q p1* $\in$ *fst d* $\Longrightarrow$ *target q* (*p1@p2*) $\in$ *fst d* $\Longrightarrow$ *target q p1* $\neq$ *target q* (*p1@p2*) $\Longrightarrow$ (*p1* $\in$ *tps q* $\wedge$ (*p1@p2*) $\in$ *tps q* $\wedge$ *target q p1* $\in$ *rd-targets* (*q*,(*p1@p2*)) $\wedge$ *target q* (*p1@p2*) $\in$ *rd-targets* (*q*,*p1*))
proof −
fix *p1 p2 p3* **assume** *p=p1@p2@p3* **and** *p2* $\neq$ [] **and** *target q p1* $\in$ *fst d* **and** *target q* (*p1@p2*) $\in$ *fst d* **and** *target q p1* $\neq$ *target q* (*p1@p2*)

have (*p1*,*p1@p2*) $\in$ *set* (*prefix-pairs p*)

using ‹p=$p1$@$p2$@$p3$› ‹$p2 \neq []$› **unfolding** *prefix-pairs-set*
by *simp*
then have ($p1$,$p1$@$p2$) $\in$ *set* (*filter* (λ($p1$, $p2$). *target q p1* $\in$ *fst d* $\wedge$ *target q p2* $\in$ *fst d* $\wedge$ *target q p1* $\neq$ *target q p2*) (*prefix-pairs p*))
using ‹*target q p1* $\in$ *fst d*› ‹*target q* ($p1$@$p2$) $\in$ *fst d*› ‹*target q p1* $\neq$ *target q* ($p1$@$p2$)›
by *auto*
have {(q, $p1$, *target q* ($p1$@$p2$)), (q, ($p1$@$p2$), *target q p1*)} $\in$ ((*set* (*map* (λ($p1$, $p2$). {(q, $p1$, *target q p2*), (q, $p2$, *target q p1*)})
(*filter* (λ($p1$, $p2$). *target q p1* $\in$ *fst d* $\wedge$ *target q p2* $\in$ *fst d* $\wedge$ *target q p1* $\neq$ *target q p2*) (*prefix-pairs p*)))))
using *map-set*[*OF* ‹($p1$,$p1$@$p2$) $\in$ *set* (*filter* (λ($p1$, $p2$). *target q p1* $\in$ *fst d* $\wedge$ *target q p2* $\in$ *fst d* $\wedge$ *target q p1* $\neq$ *target q p2*) (*prefix-pairs p*))›, *of* (λ($p1$, $p2$). {(q, $p1$, *target q p2*), (q, $p2$, *target q p1*)})]
by *force*
then have (q, $p1$, *target q* ($p1$@$p2$)) $\in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)
and (q, $p1$@$p2$, *target q p1*) $\in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)
unfolding *prefix-pair-tests-code*[*of q m-traversal-paths-with-witness M q RepSets m*]
using ‹(p,(*fst d*, *snd d*)) $\in$ *m-traversal-paths-with-witness M q RepSets m*›
by *blast+*

have $p1 \in$ *tps q*
proof −
have (q, $p1$) $\in$ ((λ(q, p, q'). (q, p)) ‘ (*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)))
using ‹(q, $p1$, *target q* ($p1$@$p2$)) $\in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)›
by (*simp add*: *rev-image-eqI*)
then show *?thesis*
unfolding *tps-alt-def*[*OF* ‹$q \in$ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *blast*
qed

moreover have ($p1$@$p2$) $\in$ *tps q*
proof −
have (q, $p1$@$p2$) $\in$ ((λ(q, p, q'). (q, p)) ‘ (*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)))
using ‹(q, $p1$@$p2$, *target q p1*) $\in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)›
by (*simp add*: *rev-image-eqI*)
then show *?thesis*
unfolding *tps-alt-def*[*OF* ‹$q \in$ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *blast*
qed

moreover have *target q p1* $\in$ *rd-targets* (q,($p1$@$p2$))

proof −
have $((q,\ p1@p2),\ target\ q\ p1) \in (\lambda(q,\ p,\ y).\ ((q,\ p),\ y))$ ‘ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)
using ‹$(q,\ p1@p2,\ target\ q\ p1) \in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)›
by (*simp add*: *rev-image-eqI*)
then show *?thesis*
unfolding *rd-targets-alt-def*[*OF* ‹$q \in$ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *blast*
qed

moreover have *target q* (*p1@p2*) $\in$ *rd-targets* (*q*,*p1*)
proof −
have $((q,\ p1),\ target\ q\ (p1@p2)) \in (\lambda(q,\ p,\ y).\ ((q,\ p),\ y))$ ‘ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)
using ‹$(q,\ p1,\ target\ q\ (p1@p2)) \in$ *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)›
by (*simp add*: *rev-image-eqI*)
then show *?thesis*
unfolding *rd-targets-alt-def*[*OF* ‹$q \in$ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *blast*
qed

ultimately show ($p1 \in tps\ q \wedge (p1@p2) \in tps\ q \wedge target\ q\ p1 \in$ *rd-targets* (q,($p1@p2$)) $\wedge$ *target q* (*p1@p2*) $\in$ *rd-targets* (*q*,*p1*))
by *blast*
qed

moreover have *p7c2*: $\bigwedge p1\ p2\ q'.\ p{=}p1@p2 \Longrightarrow q' \in$ *image fst states-with-preambles* $\Longrightarrow$ *target q p1* $\in$ *fst d* $\Longrightarrow q' \in$ *fst d* $\Longrightarrow$ *target q p1* $\neq q' \Longrightarrow (p1 \in tps\ q \wedge [] \in tps\ q' \wedge$ *target q p1* $\in$ *rd-targets* (q',[]) $\wedge\ q' \in$ *rd-targets* (q,$p1$))
proof −
fix *p1 p2 q′* **assume** *p=p1@p2* **and** $q' \in$ *image fst states-with-preambles* **and** *target q p1* $\in$ *fst d* **and** $q' \in$ *fst d* **and** *target q p1* $\neq q'$
then have $q' \in$ *fst* ‘ *d-reachable-states-with-preambles M*
unfolding *states-with-preambles-def* **by** *blast*

have *p1* $\in$ *set* (*prefixes p*)
using ‹*p=p1@p2*› **unfolding** *prefixes-set*
by *simp*
then have ($p1$,q') $\in$ *Set.filter* ($\lambda(p1,\ q2).$ *target q p1* $\in$ *fst d* $\wedge$ *q2* $\in$ *fst d* $\wedge$ *target q p1* $\neq$ *q2*) (*set* (*prefixes p*) $\times$ *fst* ‘ *d-reachable-states-with-preambles M*)
using ‹*target q p1* $\in$ *fst d*› ‹$q' \in$ *fst d*› ‹$q' \in$ *image fst states-with-preambles*› ‹*target q p1* $\neq q'$› **unfolding** *states-with-preambles-def*
by *force*
then have $\{(q,\ p1,\ q'),\ (q',\ [],\ target\ q\ p1)\} \subseteq$ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles*

M)
using *preamble-prefix-tests-code*[*of q m-traversal-paths-with-witness M q RepSets m* (*fst* ‘ *d-reachable-states-with-preambles M*)]
using ‹(*p*,(*fst d*, *snd d*)) ∈ *m-traversal-paths-with-witness M q RepSets m*›
by *blast*
then have (*q*, *p1*, *q′*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)
and (*q′*, [], *target q p1*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)
by *blast+*

have *p1* ∈ *tps q*
using ‹(*q*, *p1*, *q′*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)›
‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›
unfolding *tps-alt-def*[*OF* ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *force*

moreover have [] ∈ *tps q′*
using ‹(*q′*, [], *target q p1*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)›
‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›
unfolding *tps-alt-def*[*OF* ‹*q′* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *force*

moreover have *target q p1* ∈ *rd-targets* (*q′*,[])
using ‹(*q′*, [], *target q p1*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)›
‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›
unfolding *rd-targets-alt-def*[*OF* ‹*q′* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *force*

moreover have *q′* ∈ *rd-targets* (*q*,*p1*)
using ‹(*q*, *p1*, *q′*) ∈ *preamble-prefix-tests q* (*m-traversal-paths-with-witness M q RepSets m*) (*fst* ‘ *d-reachable-states-with-preambles M*)›
‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›
unfolding *rd-targets-alt-def*[*OF* ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
by *force*

ultimately show (*p1* ∈ *tps q* ∧ [] ∈ *tps q′* ∧ *target q p1* ∈ *rd-targets* (*q′*,[]) ∧ *q′* ∈ *rd-targets* (*q*,*p1*))
by *blast*
qed

moreover have *p7c3*: ⋀ *q1 q2* . *q1* ≠ *q2* ⟹ *q1* ∈ *snd d* ⟹ *q2* ∈ *snd d* ⟹ ([] ∈ *tps q1* ∧ [] ∈ *tps q2* ∧ *q1* ∈ *rd-targets* (*q2*,[]) ∧ *q2* ∈ *rd-targets* (*q1*,[]))

proof −
fix *q1 q2* **assume** *q1* ≠ *q2* **and** *q1* ∈ *snd d* **and** *q2* ∈ *snd d*

have (⋀*d*. *d* ∈ *set RepSets* ⟹ *snd d* ⊆ *fst d*)
using *p5* **by** *blast*
have *q* ∈ *states M*
by (*metis* (*no-types*, *lifting*) ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*› *assms*(*1*)
d-reachable-states-with-preambles-soundness(*2*) *image-iff prod.collapse*)

have *d* ∈ *set RepSets*
using *m-traversal-paths-with-witness-set*[*OF p4* ‹(⋀*d*. *d* ∈ *set RepSets* ⟹ *snd d* ⊆ *fst d*)› ‹*q* ∈ *states M*›, *of m*]
using ‹(*p*, *d*) ∈ *m-traversal-paths-with-witness M q RepSets m*› *find-set*
by *force*

have *fst d* ⊆ *states M*
and *snd d* = *fst d* ∩ *fst* ‘ *states-with-preambles*
and ⋀ *q1 q2*. *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *atcs* (*q1*, *q2*) ≠ {}
using *p5*[*OF* ‹*d* ∈ *set RepSets*›] **by** *blast*+

have *q1* ∈ *fst d*
and *q2* ∈ *fst d*
and *q1* ∈ *fst* ‘ *d-reachable-states-with-preambles M*
and *q2* ∈ *fst* ‘ *d-reachable-states-with-preambles M*
using ‹*q1* ∈ *snd d*› ‹*q2* ∈ *snd d*› **unfolding** ‹*snd d* = *fst d* ∩ *fst* ‘ *states-with-preambles*›
unfolding *states-with-preambles-def* **by** *blast*+

obtain *A t1 t2* **where** (*A*,*t1*,*t2*) ∈ *atcs* (*q1*, *q2*)
using ‹⋀ *q1 q2*. *q1* ∈ *fst d* ⟹ *q2* ∈ *fst d* ⟹ *q1* ≠ *q2* ⟹ *atcs* (*q1*, *q2*) ≠ {}›[*OF* ‹*q1* ∈ *fst d*› ‹*q2* ∈ *fst d*› ‹*q1* ≠ *q2*›]
by *auto*

then have (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
unfolding *atcs-def* **using** *set-as-map-elem* **by** *force*
then have (*q1*,[],*q2*) ∈ *pps-alt*
using ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*› ‹(*p*,(*fst d*, *snd d*)) ∈ *m-traversal-paths-with-witness M q RepSets m*›
unfolding *pps-alt-def*
by (*metis* (*mono-tags*, *lifting*) ‹*q1* ∈ *snd d*› ‹*q2* ∈ *snd d*› *mem-Collect-eq*)
then have [] ∈ *tps q1* **and** *q2* ∈ *rd-targets* (*q1*,[])
unfolding *tps-alt-def*[*OF* ‹*q1* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
rd-targets-alt-def[*OF* ‹*q1* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
preamble-pair-tests-alt
by *force*+

have (*A*,*t2*,*t1*) ∈ *atcs* (*q2*, *q1*)

using *p3* ‹(*A*,*t1*,*t2*) ∈ *atcs* (*q1*, *q2*)› **by** *blast*
then have (*q2*, *q1*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
unfolding *atcs-def* **using** *set-as-map-elem* **by** *force*
then have (*q2*,[],*q1*) ∈ *pps-alt*
using ‹*q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*› ‹(*p*,(*fst d*, *snd d*)) ∈ *m-traversal-paths-with-witness M q RepSets m*›
unfolding *pps-alt-def*
by (*metis* (*mono-tags*, *lifting*) ‹*q1* ∈ *snd d*› ‹*q2* ∈ *snd d*› *mem-Collect-eq*)
then have [] ∈ *tps q2* **and** *q1* ∈ *rd-targets* (*q2*,[])
unfolding *tps-alt-def*[*OF* ‹*q2* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
rd-targets-alt-def[*OF* ‹*q2* ∈ *fst* ‘ *d-reachable-states-with-preambles M*›]
preamble-pair-tests-alt
by *force*+

then show ([] ∈ *tps q1* ∧ [] ∈ *tps q2* ∧ *q1* ∈ *rd-targets* (*q2*,[]) ∧ *q2* ∈ *rd-targets* (*q1*,[]))
using ‹[] ∈ *tps q1*› ‹*q2* ∈ *rd-targets* (*q1*,[])›
by *simp*
qed

ultimately show (∀ *p1 p2 p3* . *p*=*p1*@*p2*@*p3* ⟶ *p2* ≠ [] ⟶ *target q p1* ∈ *fst d* ⟶ *target q* (*p1*@*p2*) ∈ *fst d* ⟶ *target q p1* ≠ *target q* (*p1*@*p2*) ⟶ (*p1* ∈ *tps q* ∧ (*p1*@*p2*) ∈ *tps q* ∧ *target q p1* ∈ *rd-targets* (*q*,(*p1*@*p2*)) ∧ *target q* (*p1*@*p2*) ∈ *rd-targets* (*q*,*p1*)))
∧ (∀ *p1 p2 q′* . *p*=*p1*@*p2* ⟶ *q′* ∈ *image fst states-with-preambles* ⟶ *target q p1* ∈ *fst d* ⟶ *q′* ∈ *fst d* ⟶ *target q p1* ≠ *q′* ⟶ (*p1* ∈ *tps q* ∧ [] ∈ *tps q′* ∧ *target q p1* ∈ *rd-targets* (*q′*,[]) ∧ *q′* ∈ *rd-targets* (*q*,*p1*)))
∧ (∀ *q1 q2* . *q1* ≠ *q2* ⟶ *q1* ∈ *snd d* ⟶ *q2* ∈ *snd d* ⟶ ([] ∈ *tps q1* ∧ [] ∈ *tps q2* ∧ *q1* ∈ *rd-targets* (*q2*,[]) ∧ *q2* ∈ *rd-targets* (*q1*,[])))
by *blast*
qed

have *implies-completeness-for-repetition-sets* (*Test-Suite states-with-preambles tps rd-targets atcs*) *M m RepSets*
unfolding *implies-completeness-for-repetition-sets.simps*
using *p1 p2 p3 p4 p5 p6 p7*
by *force*
then show *implies-completeness* (*calculate-test-suite-for-repetition-sets M m RepSets*) *M m*
unfolding ‹*calculate-test-suite-for-repetition-sets M m RepSets* = *Test-Suite states-with-preambles tps rd-targets atcs*›
implies-completeness-def
by *blast*

show *is-finite-test-suite* (*calculate-test-suite-for-repetition-sets M m RepSets*)
proof −
have *finite states-with-preambles*

unfolding *states-with-preambles-def d-reachable-states-with-preambles-def*
using *fsm-states-finite*[*of M*] **by** *simp*

moreover have $\bigwedge$ *q p. q* $\in$ *fst* ‘ *states-with-preambles* $\Longrightarrow$ *finite* (*rd-targets* (*q*, *p*))
proof −
fix *q p* **assume** *q* $\in$ *fst* ‘ *states-with-preambles*
then have *q* $\in$ *fst* ‘ *d-reachable-states-with-preambles M* **unfolding** *states-with-preambles-def* **by** *assumption*

have $*$: *finite* ((λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘ (*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*) ∪
(⋃*q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃*mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)) ∪
preamble-pair-tests
(⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘
fst ‘ *d-reachable-states-with-preambles M*.
(λ(*p*, *rd*, *dr*). *dr*) ‘ *y*)
(*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)))
proof −
have $*$: $\bigwedge$ *a b c f* . *finite* (*f* ‘ (*a* ∪ *b* ∪ *c*)) = (*finite* (*f*‘*a*) ∧ *finite* (*f*‘*b*) ∧ *finite* (*f*‘*c*))
by (*simp add*: *image-Un*)

have *finite* ((λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘ (*prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*)))
proof −
have *prefix-pair-tests q* (*m-traversal-paths-with-witness M q RepSets m*) ⊆
(⋃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q RepSets m* . ⋃ (*p1*, *p2*) ∈ *set* (*prefix-pairs p*) .{(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})
unfolding *prefix-pair-tests.simps* **by** *blast*
moreover have *finite* (⋃ (*p*, *rd*, *dr*)∈*m-traversal-paths-with-witness M q RepSets m* . ⋃ (*p1*, *p2*) ∈ *set* (*prefix-pairs p*) .{(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})
proof −
have *finite* (*m-traversal-paths-with-witness M q RepSets m*)
using *m-traversal-paths-with-witness-finite*[*of M q RepSets m*] **by** *assumption*
moreover have $\bigwedge$ *p rd dr* . *finite* (⋃ (*p1*, *p2*) ∈ *set* (*prefix-pairs p*) .{(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})
by *auto*
ultimately show *?thesis* **by** *force*
qed
ultimately show *?thesis* **using** *infinite-super* **by** *blast*
qed

moreover have *finite* ((λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘ (⋃ *q*∈*fst* ‘ *d-reachable-states-with-preambles M*.
⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}.
preamble-prefix-tests q mrsps (*fst* ‘ *d-reachable-states-with-preambles M*)))
proof −
have *finite* (*fst* ‘ *d-reachable-states-with-preambles M*) **using** ‹*finite states-with-preambles*› **unfolding** *states-with-preambles-def* **by** *auto*
moreover have ⋀ *q* . *q*∈*fst* ‘ *d-reachable-states-with-preambles M* ⟹ *finite* (⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}. *preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*))
proof −
fix *q* **assume** *q*∈*fst* ‘ *d-reachable-states-with-preambles M*

have *finite* {*m-traversal-paths-with-witness M q RepSets m*} **by** *simp*
moreover have ⋀ *mrsps* . *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*} ⟹ *finite* (*preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*))
proof −
fix *mrsps* **assume** *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}
then have ∗: *mrsps* = *m-traversal-paths-with-witness M q RepSets m* **by** *simp*

have *preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*)
⊆ (⋃ (*p*,*rd*,*dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . ⋃ *q2* ∈ (*fst* ‘ *d-reachable-states-with-preambles M*) . (⋃ *p1* ∈ *set* (*prefixes p*) . {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))}))
unfolding *preamble-prefix-tests-def* ∗ *prefixes-set* **by** *blast*
moreover have *finite* (⋃ (*p*,*rd*,*dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . ⋃ *q2* ∈ (*fst* ‘ *d-reachable-states-with-preambles M*) . (⋃ *p1* ∈ *set* (*prefixes p*) . {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))}))
proof −
have *finite* (*m-traversal-paths-with-witness M q RepSets m*)
using *m-traversal-paths-with-witness-finite* **by** *metis*
moreover have ⋀ *p rd dr* . (*p*,*rd*,*dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* ⟹ *finite* (⋃ *q2* ∈ (*fst* ‘ *d-reachable-states-with-preambles M*) . (⋃ *p1* ∈ *set* (*prefixes p*) . {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))}))
using ‹*finite* (*fst* ‘ *d-reachable-states-with-preambles M*)› **by** *blast*
ultimately show *?thesis* **by** *force*
qed
ultimately show *finite* (*preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*)) **using** *infinite-super* **by** *blast*
qed
ultimately show *finite* (⋃ *mrsps*∈{*m-traversal-paths-with-witness M q RepSets m*}. *preamble-prefix-tests q mrsps* (*fst* ‘ *d-reachable-states-with-preambles M*)) **by** *force*

qed
ultimately show *?thesis* **by** *blast*
qed

moreover have *finite* ((λ(*q*, *p*, *y*). ((*q*, *p*), *y*)) ‘ (*preamble-pair-tests*
(⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘
fst ‘ *d-reachable-states-with-preambles M*.
(λ(*p*, *rd*, *dr*). *dr*) ‘ *y*)
(*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)))
proof −

have *finite* (⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘
fst ‘ *d-reachable-states-with-preambles M*.
(λ(*p*, *rd*, *dr*). *dr*) ‘ *y*)
proof −

have ∗: (⋃(*q*, *y*)∈(λ*q*. (*q*, *m-traversal-paths-with-witness M q RepSets m*)) ‘
fst ‘ *d-reachable-states-with-preambles M*.
(λ(*p*, *rd*, *dr*). *dr*) ‘ *y*) =
(⋃ *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* . ⋃ (*p*, *rd*, *dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . {*dr*})
by *force*

have *finite* (⋃ *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* . ⋃ (*p*, *rd*, *dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . {*dr*})
proof −
have *finite* (*fst* ‘ *d-reachable-states-with-preambles M*)
using ‹*finite states-with-preambles*› **unfolding** *states-with-preambles-def* **by** *auto*

moreover have ⋀*q* . *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M* ⟹ *finite* (⋃ (*p*, *rd*, *dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . {*dr*})
proof −
fix *q* **assume** *q* ∈ *fst* ‘ *d-reachable-states-with-preambles M*

have *finite* (*m-traversal-paths-with-witness M q RepSets m*)
using *m-traversal-paths-with-witness-finite* **by** *metis*
moreover have ⋀ *p rd dr* . (*p*, *rd*, *dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* ⟹ *finite* {*dr*}
by *simp*
ultimately show *finite* (⋃ (*p*, *rd*, *dr*) ∈ *m-traversal-paths-with-witness M q RepSets m* . {*dr*})
by *force*
qed
ultimately show *?thesis* **by** *blast*
qed

then show *?thesis* **unfolding** $*$ **by** *assumption*
qed
moreover have *finite* (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*)
proof −
have (*fst* ‘ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*, *Inr q2*)) ‘ *r-distinguishable-state-pairs-with-separators M*) $\subseteq$ *states M* $\times$ *states M*
unfolding *r-distinguishable-state-pairs-with-separators-def* **by** *auto*
moreover have *finite* (*states M* $\times$ *states M*)
using *fsm-states-finite* **by** *auto*
ultimately show *?thesis* **using** *infinite-super* **by** *blast*
qed
ultimately show *?thesis*
unfolding *preamble-pair-tests.simps* **by** *blast*
qed

ultimately show *?thesis*
unfolding $*$ **by** *blast*

qed

show *finite* (*rd-targets* (*q*, *p*))
unfolding *rd-targets-alt-def*[*OF* ‹*q* $\in$ *fst* ‘ *d-reachable-states-with-preambles M*›]
using *finite-snd-helper*[*of - q p*, *OF* $*$] **by** *assumption*
qed

moreover have $\bigwedge$ *q q′*. *finite* (*atcs* (*q*, *q′*))
proof −
fix *q q′*
show *finite* (*atcs* (*q*,*q′*)) **proof** (*cases set-as-map* (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*:: ($'a \times 'a$) + $'a$, *Inr q2* :: ($'a \times 'a$) + $'a$)) ‘ *r-distinguishable-state-pairs-with-separators M*) (*q*, *q′*))
case *None*
then have *atcs* (*q*, *q′*) = {} **unfolding** *atcs-def* **by** *auto*
then show *?thesis* **by** *auto*
next
case (*Some a*)
then have *atcs* (*q*, *q′*) = *a* **unfolding** *atcs-def* **by** *auto*
then have $*$: *atcs* (*q*, *q′*) = {*z*. ((*q*, *q′*), *z*) $\in$ (λ((*q1*, *q2*), *A*). ((*q1*, *q2*), *A*, *Inr q1*:: ($'a \times 'a$) + $'a$, *Inr q2*:: ($'a \times 'a$) + $'a$)) ‘ *r-distinguishable-state-pairs-with-separators M*} **using** *Some* **unfolding** *set-as-map-def*
by (*metis* (*no-types*, *lifting*) *option.distinct*(*1*) *option.inject*)

have *finite* (*r-distinguishable-state-pairs-with-separators M*)
proof −
have *r-distinguishable-state-pairs-with-separators M* $\subseteq$ ($\bigcup$ *q1* $\in$ *states*

M . ⋃ *q2* ∈ *states M* . {((*q1*,*q2*), *the* (*state-separator-from-s-states M q1 q2*)), ((*q1*,*q2*), *the* (*state-separator-from-s-states M q2 q1*))})
proof
fix *x* **assume** *x* ∈ *r-distinguishable-state-pairs-with-separators M*
then obtain *q1 q2 Sep* **where** *x* = ((*q1*,*q2*),*Sep*)
and *q1* ∈ *states M*
and *q2* ∈ *states M*
and (*q1* < *q2* ∧ *state-separator-from-s-states M q1 q2* = *Some Sep*) ∨ (*q2* < *q1* ∧ *state-separator-from-s-states M q2 q1* = *Some Sep*)
unfolding *r-distinguishable-state-pairs-with-separators-def* **by** *blast*
then consider *state-separator-from-s-states M q1 q2* = *Some Sep* | *state-separator-from-s-states M q2 q1* = *Some Sep* **by** *blast*

then show *x* ∈ (⋃ *q1* ∈ *states M* . ⋃ *q2* ∈ *states M* . {((*q1*,*q2*), *the* (*state-separator-from-s-states M q1 q2*)), ((*q1*,*q2*), *the* (*state-separator-from-s-states M q2 q1*))})
using ‹*q1* ∈ *states M*› ‹*q2* ∈ *states M*› **unfolding** ‹*x* = ((*q1*,*q2*),*Sep*)› **by** (*cases*; *force*)
qed

moreover have *finite* (⋃ *q1* ∈ *states M* . ⋃ *q2* ∈ *states M* . {((*q1*,*q2*), *the* (*state-separator-from-s-states M q1 q2*)), ((*q1*,*q2*), *the* (*state-separator-from-s-states M q2 q1*))})
using *fsm-states-finite*[*of M*] **by** *force*

ultimately show *?thesis* **using** *infinite-super* **by** *blast*
qed
then show *?thesis* **unfolding** ∗ **by** (*simp add*: *finite-snd-helper*)
qed
qed

ultimately show *?thesis*
unfolding ‹*calculate-test-suite-for-repetition-sets M m RepSets* = *Test-Suite states-with-preambles tps rd-targets atcs*›
is-finite-test-suite.simps
by *blast*
qed
qed

43.4 Two Complete Example Implementations

43.4.1 Naive Repetition Set Strategy

definition *calculate-test-suite-naive* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm* ⇒ *nat* ⇒ (*'a*,*'b*,*'c*, (*'a* × *'a*) + *'a*) *test-suite* **where**
calculate-test-suite-naive M m = *calculate-test-suite-for-repetition-sets M m* (*maximal-repetition-sets-from-sep M*)

definition *calculate-test-suite-naive-as-io-sequences* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*)

fsm ⇒ *nat* ⇒ ('*b* × '*c*) *list set* **where**
calculate-test-suite-naive-as-io-sequences M m = *test-suite-to-io-maximal M* (*calculate-test-suite-naive M m*)

lemma *calculate-test-suite-naive-completeness* :
fixes *M* :: ('*a*::*linorder*,'*b*::*linorder*,'*c*) *fsm*
assumes *observable M*
and *observable M'*
and *inputs M'* = *inputs M*
and *inputs M* ≠ {}
and *completely-specified M*
and *completely-specified M'*
and *size M'* ≤ *m*
shows (*L M'* ⊆ *L M*) ⟷ *passes-test-suite M* (*calculate-test-suite-naive M m*) *M'*
and (*L M'* ⊆ *L M*) ⟷ *pass-io-set-maximal M'* (*calculate-test-suite-naive-as-io-sequences M m*)
proof −

have ⋀*q*. *q* ∈ *FSM.states M* ⟹ ∃ *d*∈*set* (*maximal-repetition-sets-from-separators-list-naive M*). *q* ∈ *fst d*
unfolding *maximal-repetition-sets-from-separators-list-naive-def Let-def*
by (*metis* (*mono-tags*, *lifting*) *list.set-map maximal-pairwise-r-distinguishable-state-sets-from-separators-code maximal-repetition-sets-from-separators-code maximal-repetition-sets-from-separators-cover*)
moreover have ⋀*d*. *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-naive M*) ⟹ *fst d* ⊆ *states M* ∧ (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
and ⋀ *d q1 q2*. *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-naive M*) ⟹ *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
proof
fix *d* **assume** *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-naive M*)
then have *d* ∈ *maximal-repetition-sets-from-separators M*
by (*simp add*: *maximal-repetition-sets-from-separators-code-alt*)

then show *fst d* ⊆ *states M* **and** (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
and ⋀ *q1 q2* . *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
unfolding *maximal-repetition-sets-from-separators-def*
maximal-pairwise-r-distinguishable-state-sets-from-separators-def
pairwise-r-distinguishable-state-sets-from-separators-def
by *force*+
qed
ultimately have *implies-completeness* (*calculate-test-suite-naive M m*) *M m*
and *is-finite-test-suite* (*calculate-test-suite-naive M m*)
using *calculate-test-suite-for-repetition-sets-sufficient-and-finite*[*OF* ‹*observable*

M› ‹*completely-specified M*› ‹*inputs M* ≠ {}›]
unfolding *calculate-test-suite-naive-def* **by** *force*+

then show (*L M′* ⊆ *L M*) ⟷ *passes-test-suite M* (*calculate-test-suite-naive M m*) *M′*
and (*L M′* ⊆ *L M*) ⟷ *pass-io-set-maximal M′* (*calculate-test-suite-naive-as-io-sequences M m*)
using *passes-test-suite-completeness*[*OF - assms*]
passes-test-suite-as-maximal-sequences-completeness[*OF - - assms*]
unfolding *calculate-test-suite-naive-as-io-sequences-def*
by *blast*+
qed

definition *calculate-test-suite-naive-as-io-sequences-with-assumption-check* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm* ⇒ *nat* ⇒ *String.literal* + ((*′b* × *′c*) *list set*) **where**
calculate-test-suite-naive-as-io-sequences-with-assumption-check M m =
(*if inputs M* ≠ {}
then if observable M
then if completely-specified M
then (*Inr* (*test-suite-to-io-maximal M* (*calculate-test-suite-naive M m*)))
else (*Inl* (*STR ′′specification is not completely specified′′*))
else (*Inl* (*STR ′′specification is not observable′′*))
else (*Inl* (*STR ′′specification has no inputs′′*)))

lemma *calculate-test-suite-naive-as-io-sequences-with-assumption-check-completeness*
:
fixes *M* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm*
assumes *observable M′*
and *inputs M′* = *inputs M*
and *completely-specified M′*
and *size M′* ≤ *m*
and *calculate-test-suite-naive-as-io-sequences-with-assumption-check M m* = *Inr ts*
shows (*L M′* ⊆ *L M*) ⟷ *pass-io-set-maximal M′ ts*
proof −

have *inputs M* ≠ {}
and *observable M*
and *completely-specified M*
using ‹*calculate-test-suite-naive-as-io-sequences-with-assumption-check M m* = *Inr ts*›
unfolding *calculate-test-suite-naive-as-io-sequences-with-assumption-check-def*
by (*meson Inl-Inr-False*)+
then have *ts* = (*test-suite-to-io-maximal M* (*calculate-test-suite-naive M m*))
using ‹*calculate-test-suite-naive-as-io-sequences-with-assumption-check M m* = *Inr ts*›
unfolding *calculate-test-suite-naive-as-io-sequences-with-assumption-check-def*
by (*metis sum.inject*(*2*))

then show *?thesis*
using *calculate-test-suite-naive-completeness*(*2*)[*OF* ‹*observable M*› *assms*(*1*,*2*) ‹*inputs M* ≠ {}›
‹*completely-specified M*› *assms*(*3*,*4*)]
unfolding *calculate-test-suite-naive-as-io-sequences-def*
by *simp*
qed

43.4.2 Greedy Repetition Set Strategy

definition *calculate-test-suite-greedy* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm* ⇒ *nat* ⇒ (*'a*,*'b*,*'c*, (*'a* × *'a*) + *'a*) *test-suite* **where**
calculate-test-suite-greedy M m = *calculate-test-suite-for-repetition-sets M m* (*maximal-repetition-sets-from-se*
M)

definition *calculate-test-suite-greedy-as-io-sequences* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm* ⇒ *nat* ⇒ (*'b* × *'c*) *list set* **where**
calculate-test-suite-greedy-as-io-sequences M m = *test-suite-to-io-maximal M* (*calculate-test-suite-greedy M m*)

lemma *calculate-test-suite-greedy-completeness* :
fixes *M* :: (*'a*::*linorder*,*'b*::*linorder*,*'c*) *fsm*
assumes *observable M*
and *observable M'*
and *inputs M'* = *inputs M*
and *inputs M* ≠ {}
and *completely-specified M*
and *completely-specified M'*
and *size M'* ≤ *m*
shows (*L M'* ⊆ *L M*) ⟷ *passes-test-suite M* (*calculate-test-suite-greedy M m*) *M'*
and (*L M'* ⊆ *L M*) ⟷ *pass-io-set-maximal M'* (*calculate-test-suite-greedy-as-io-sequences M m*)
proof −

have ⋀*q*. *q* ∈ *FSM.states M* ⟹ ∃ *d*∈*set* (*maximal-repetition-sets-from-separators-list-greedy M*). *q* ∈ *fst d*
unfolding *maximal-repetition-sets-from-separators-list-greedy-def Let-def*
using *greedy-pairwise-r-distinguishable-state-sets-from-separators-cover*[*of - M*]
by *simp*

moreover have ⋀*d*. *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-greedy M*) ⟹ *fst d* ⊆ *states M* ∧ (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
and ⋀ *d q1 q2*. *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-greedy M*) ⟹ *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
proof
fix *d* **assume** *d* ∈ *set* (*maximal-repetition-sets-from-separators-list-greedy M*)

then have *fst d* ∈ *set* (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*)
and (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
unfolding *maximal-repetition-sets-from-separators-list-greedy-def Let-def* **by** *force+*

then have *fst d* ∈ *pairwise-r-distinguishable-state-sets-from-separators M*
using *greedy-pairwise-r-distinguishable-state-sets-from-separators-soundness* **by** *blast*
then show *fst d* ⊆ *states M* **and** (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
and ⋀ *q1 q2* . *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
using ‹(*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)›
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def*
by *force+*
qed
ultimately have *implies-completeness* (*calculate-test-suite-greedy M m*) *M m*
and *is-finite-test-suite* (*calculate-test-suite-greedy M m*)
using *calculate-test-suite-for-repetition-sets-sufficient-and-finite*[*OF* ‹*observable M*› ‹*completely-specified M*› ‹*inputs M* ≠ {}›]
unfolding *calculate-test-suite-greedy-def* **by** *force+*

then show (*L M′* ⊆ *L M*) ⟷ *passes-test-suite M* (*calculate-test-suite-greedy M m*) *M′*
and (*L M′* ⊆ *L M*) ⟷ *pass-io-set-maximal M′* (*calculate-test-suite-greedy-as-io-sequences M m*)
using *passes-test-suite-completeness*[*OF - assms*]
passes-test-suite-as-maximal-sequences-completeness[*OF - - assms*]
unfolding *calculate-test-suite-greedy-as-io-sequences-def*
by *blast+*
qed

definition *calculate-test-suite-greedy-as-io-sequences-with-assumption-check* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm* ⇒ *nat* ⇒ *String.literal* + ((*′b* × *′c*) *list set*) **where**
calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m =
(*if inputs M* ≠ {}
then if observable M
then if completely-specified M
then (*Inr* (*test-suite-to-io-maximal M* (*calculate-test-suite-greedy M m*)))
else (*Inl* (*STR ′′specification is not completely specified′′*))
else (*Inl* (*STR ′′specification is not observable′′*))
else (*Inl* (*STR ′′specification has no inputs′′*)))

lemma *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-completeness*
:
fixes *M* :: (*′a*::*linorder*,*′b*::*linorder*,*′c*) *fsm*
assumes *observable M′*

and *inputs M′ = inputs M*
and *completely-specified M′*
and *size M′ ≤ m*
and *calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m = Inr ts*
shows *(L M′ ⊆ L M) ⟷ pass-io-set-maximal M′ ts*
proof −

have *inputs M ≠ {}*
and *observable M*
and *completely-specified M*
using ‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m = Inr ts*›
unfolding *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-def*

by (*meson Inl-Inr-False*)+
then have *ts = (test-suite-to-io-maximal M (calculate-test-suite-greedy M m))*
using ‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check M m = Inr ts*›
unfolding *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-def*
by (*metis sum.inject*(*2*))
then show *?thesis*
using *calculate-test-suite-greedy-completeness*(*2*)[*OF* ‹*observable M*› *assms*(*1*,*2*) ‹*inputs M ≠ {}*›
‹*completely-specified M*› *assms*(*3*,*4*)]
unfolding *calculate-test-suite-greedy-as-io-sequences-def*
by *simp*
qed

end

44 Refined Test Suite Calculation

This theory refines some of the algorithms defined in *Test-Suite-Calculation* using containers from the Containers framework.

theory *Test-Suite-Calculation-Refined*
imports *Test-Suite-Calculation*
../Util-Refined
Deriving.Compare
Containers.Containers
begin

44.1 New Instances

44.1.1 Order on FSMs

instantiation *fsm* :: (*ord*,*ord*,*ord*) *ord*
begin

```
fun less-eq-fsm :: ('a,'b,'c) fsm ⇒ ('a,'b,'c) fsm ⇒ bool where
  less-eq-fsm M1 M2 =
    (if initial M1 < initial M2
      then True
      else ((initial M1 = initial M2) ∧ (if set-less-aux (states M1) (states M2)
        then True
        else ((states M1 = states M2) ∧ (if set-less-aux (inputs M1) (inputs M2)
          then True
           else ((inputs M1 = inputs M2) ∧ (if set-less-aux (outputs M1) (outputs
M2)
            then True
                else ((outputs M1 = outputs M2) ∧ (set-less-aux (transitions M1)
(transitions M2) ∨ (transitions M1) = (transitions M2)))))))))

fun less-fsm :: ('a,'b,'c) fsm ⇒ ('a,'b,'c) fsm ⇒ bool where
  less-fsm a b = (a ≤ b ∧ a ≠ b)

instance by (intro-classes)
end


instantiation fsm :: (linorder,linorder,linorder) linorder
begin

lemma less-le-not-le-FSM :
  fixes x :: ('a,'b,'c) fsm
  and   y :: ('a,'b,'c) fsm
shows (x < y) = (x ≤ y ∧ ¬ y ≤ x)
proof
  show x < y ⟹ x ≤ y ∧ ¬ y ≤ x

  proof −
    assume x < y
    then show x ≤ y ∧ ¬ y ≤ x
    proof (cases FSM.initial x < FSM.initial y)
      case True
      then show ?thesis unfolding less-fsm.simps less-eq-fsm.simps by auto
    next
      case False
      then have ∗: FSM.initial x = FSM.initial y
        using ‹x < y› unfolding less-fsm.simps less-eq-fsm.simps by auto

      show ?thesis proof (cases set-less-aux (FSM.states x) (FSM.states y))
        case True
        then show ?thesis
          unfolding less-fsm.simps less-eq-fsm.simps
          using ∗ set-less-aux-antisym by fastforce
      next
```

```
case False
then have **: FSM.states x = FSM.states y
  using ‹x < y› * unfolding less-fsm.simps less-eq-fsm.simps by auto

show ?thesis proof (cases set-less-aux (FSM.inputs x) (FSM.inputs y))
  case True
  then show ?thesis
    unfolding less-fsm.simps less-eq-fsm.simps
    using * ** set-less-aux-antisym by fastforce
next
  case False
  then have ***: FSM.inputs x = FSM.inputs y
    using ‹x < y› * **
    unfolding less-fsm.simps less-eq-fsm.simps
    by (simp add: set-less-def)

 show ?thesis proof (cases set-less-aux (FSM.outputs x) (FSM.outputs y))
    case True
    then show ?thesis
      unfolding less-fsm.simps less-eq-fsm.simps
      using * ** *** set-less-aux-antisym
      by fastforce
  next
    case False
    then have ****: FSM.outputs x = FSM.outputs y
      using ‹x < y› * ** ***
      unfolding less-fsm.simps less-eq-fsm.simps
      by (simp add: set-less-def)


    have x ≠ y using ‹x < y› by auto
    then have FSM.transitions x ≠ FSM.transitions y
      using * ** *** **** apply transfer
      by (metis fsm-impl.exhaust-sel)
    then have *****: set-less-aux (FSM.transitions x) (FSM.transitions y)
      using ‹x < y› * ** *** ****
      unfolding less-fsm.simps less-eq-fsm.simps
      by (simp add: set-less-aux-def)

     then have ¬(set-less-aux (FSM.transitions y) (FSM.transitions x) ∨
transitions y = transitions x)
      using ‹FSM.transitions x ≠ FSM.transitions y› fsm-transitions-finite
set-less-aux-antisym
      by auto
    then have ¬ y ≤ x
      using * ** *** ****
      unfolding less-fsm.simps less-eq-fsm.simps
      by (simp add: set-less-def)
    then show ?thesis using ‹x < y›
```

```
            using less-fsm.elims(2)
            by blast
          qed
        qed
      qed
    qed
  qed

  show x ≤ y ∧ ¬ y ≤ x ⟹ x < y
    using less-fsm.elims(3)
    by blast
qed


lemma order-refl-FSM :
  fixes x :: ('a,'b,'c) fsm
  shows x ≤ x
  by auto

lemma order-trans-FSM :
  fixes x :: ('a,'b,'c) fsm
  fixes y :: ('a,'b,'c) fsm
  fixes z :: ('a,'b,'c) fsm
  shows x ≤ y ⟹ y ≤ z ⟹ x ≤ z
  unfolding less-eq-fsm.simps
  using less-trans[of initial x initial y initial z]
        set-less-aux-trans[of states x states y states z]
        set-less-aux-trans[of inputs x inputs y inputs z]
        set-less-aux-trans[of outputs x outputs y outputs z]
        set-less-aux-trans[of transitions x transitions y transitions z]
  by metis

lemma antisym-FSM :
  fixes x :: ('a,'b,'c) fsm
  fixes y :: ('a,'b,'c) fsm
shows x ≤ y ⟹ y ≤ x ⟹ x = y
  unfolding less-eq-fsm.simps
  using equal-fsm-def[of x y]
  unfolding equal-class.equal
  by (metis order.asym set-less-aux-antisym)

lemma linear-FSM :
  fixes x :: ('a,'b,'c) fsm
  fixes y :: ('a,'b,'c) fsm
shows x ≤ y ∨ y ≤ x
  unfolding less-eq-fsm.simps
 by (metis fsm-inputs-finite fsm-states-finite fsm-outputs-finite fsm-transitions-finite
neq-iff set-less-aux-finite-total)
```

instance
using *less-le-not-le-FSM order-refl-FSM order-trans-FSM antisym-FSM linear-FSM*

by (*intro-classes*; *metis*+)
end

instantiation *fsm* :: (*linorder*,*linorder*,*linorder*) *compare*
begin
fun *compare-fsm* :: ('*a*, '*b*, '*c*) *fsm* ⇒ ('*a*, '*b*, '*c*) *fsm* ⇒ *order* **where**
compare-fsm x y = *comparator-of x y*

instance
using *comparator-of compare-fsm.elims*
by (*intro-classes*; *simp add*: *comparator-def*)
end

44.1.2 Derived Instances

derive (*eq*) *ceq fsm*

derive (*dlist*) *set-impl fsm*
derive (*assoclist*) *mapping-impl fsm*

derive (*no*) *cenum fsm*
derive (*no*) *ccompare fsm*

44.1.3 Finiteness and Cardinality Instantiations for FSMs

lemma *finiteness-fsm-UNIV* : *finite* (*UNIV* :: ('*a*,'*b*,'*c*) *fsm set*) =
(*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∧ *finite* (*UNIV* :: '*c set*))
proof

define *f* :: '*a* ⇒ ('*a*) *fset* **where** *f-def*: *f* = (λ *q* . {| *q* |})
have *inj f*
proof
fix *x y* **assume** *x* ∈ (*UNIV* :: '*a set*) **and** *y* ∈ *UNIV* **and** *f x* = *f y*
then show *x* = *y* **unfolding** *f-def* **by** (*transfer*; *auto*)
qed

show *finite* (*UNIV* :: ('*a*,'*b*,'*c*) *fsm set*) ⟹ (*finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*) ∧ *finite* (*UNIV* :: '*c set*))
proof (*rule ccontr*)

obtain *q* **where** *q* ∈ (*UNIV* :: '*a set*) **by** *auto*
obtain *x* **where** *x* ∈ (*UNIV* :: '*b set*) **by** *auto*

obtain y **where** $y \in (UNIV :: 'c\ set)$ **by** *auto*

assume *finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$ **and** $\neg$ (*finite* $(UNIV :: 'a\ set) \wedge$ *finite* $(UNIV :: 'b\ set) \wedge$ *finite* $(UNIV :: 'c\ set)$)

then consider (a) $\neg$ *finite* $(UNIV :: 'a\ set)$ | (b) $\neg$ *finite* $(UNIV :: 'b\ set)$ | (c) $\neg$ *finite* $(UNIV :: 'c\ set)$
by *blast*
then show *False* **proof** *cases*
case a
define $f :: 'a \Rightarrow ('a,'b,'c)\ fsm$ **where** $f = (\lambda\ q\ .\ fsm\text{-}from\text{-}list\ q\ [])$
have *inj f*
unfolding *inj-def f-def* **by** (*transfer*; *auto*)
then have $\neg$ *finite* $(f\ `\ UNIV)$
using ‹*inj f*› *finite-imageD a* **by** *auto*
then have $\neg$ *finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$
by (*meson infinite-iff-countable-subset top-greatest*)
then show *?thesis*
using ‹*finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$› **by** *blast*
next
case b
define $f :: 'b \Rightarrow ('a,'b,'c)\ fsm$ **where** $f = (\lambda\ x\ .\ fsm\text{-}from\text{-}list\ q\ [(q,x,y,q)])$
have *inj f*
unfolding *inj-def f-def* **by** (*transfer*; *auto*)
then have $\neg$ *finite* $(f\ `\ UNIV)$
using ‹*inj f*› *finite-imageD b* **by** *auto*
then have $\neg$ *finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$
by (*meson infinite-iff-countable-subset top-greatest*)
then show *?thesis*
using ‹*finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$› **by** *blast*
next
case c
define $f :: 'c \Rightarrow ('a,'b,'c)\ fsm$ **where** $f = (\lambda\ y\ .\ fsm\text{-}from\text{-}list\ q\ [(q,x,y,q)])$
have *inj f*
unfolding *inj-def f-def* **by** (*transfer*; *auto*)
then have $\neg$ *finite* $(f\ `\ UNIV)$
using ‹*inj f*› *finite-imageD c* **by** *auto*
then have $\neg$ *finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$
by (*meson infinite-iff-countable-subset top-greatest*)
then show *?thesis*
using ‹*finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$› **by** *blast*
qed
qed

show (*finite* $(UNIV :: 'a\ set) \wedge$ *finite* $(UNIV :: 'b\ set) \wedge$ *finite* $(UNIV :: 'c\ set)$) $\Longrightarrow$ *finite* $(UNIV :: ('a,'b,'c)\ fsm\ set)$
proof −
define $f :: ('a,'b,'c)\ fsm \Rightarrow ('a \times 'a\ set \times 'b\ set \times 'c\ set \times ('a \times 'b \times 'c \times 'a)\ set)$ **where**

f = (λ *m* . (*initial m*, *states m*, *inputs m*, *outputs m*, *transitions m*))
assume (*finite* (*UNIV* :: *'a set*) ∧ *finite* (*UNIV* :: *'b set*) ∧ *finite* (*UNIV* :: *'c set*))
then have *finite* (*UNIV* :: (*'a* × *'a set* × *'b set* × *'c set* × (*'a* × *'b* × *'c* × *'a*) *set*) *set*)
by (*simp add*: *Finite-Set.finite-set finite-prod*)
moreover have *f* ‘ (*UNIV* :: (*'a*,*'b*,*'c*) *fsm set*) ⊆ (*UNIV* :: (*'a* × *'a set* × *'b set* × *'c set* × (*'a* × *'b* × *'c* × *'a*) *set*) *set*)
by *auto*
moreover have *inj f*
unfolding *inj-def f-def* **apply** *transfer*
by (*simp add*: *fsm-impl.expand*)
ultimately show *?thesis* **by** (*metis inj-on-finite*)
qed
qed

instantiation *fsm* :: (*finite-UNIV*,*finite-UNIV*,*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*((*'a*,*'b*,*'c*) *fsm*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*) ∧
of-phantom (*finite-UNIV* :: *'b finite-UNIV*)
∧
of-phantom (*finite-UNIV* :: *'c finite-UNIV*))

instance by(*intro-classes*)(*simp add*: *finite-UNIV-fsm-def finite-UNIV finiteness-fsm-UNIV*)
end

instantiation *fsm* :: (*card-UNIV*,*card-UNIV*,*card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*((*'a*,*'b*,*'c*) *fsm*)
(*if CARD*(*'a*) = *0* ∨ *CARD*(*'b*) = *0* ∨ *CARD*(*'c*) = *0*
then 0
else card ((λ(*q*::*'a*, *Q*, *X*::*'b set*, *Y*::*'c set*, *T*). *FSM.create-fsm-from-sets q Q X Y T*) ‘ *UNIV*))
instance apply *intro-classes*
proof (*cases CARD*(*'a*) = *0* ∨ *CARD*(*'b*) = *0* ∨ *CARD*(*'c*) = *0*)
case *True*
then have ¬ (*finite* (*UNIV* :: *'a set*) ∧ *finite* (*UNIV* :: *'b set*) ∧ *finite* (*UNIV* :: *'c set*))
by *force*
then have *infinite* (*UNIV* :: (*'a*, *'b*, *'c*) *fsm set*)
using *finiteness-fsm-UNIV* **by** *blast*
then have *card* (*UNIV* :: (*'a*, *'b*, *'c*) *fsm set*) = *0*

by *auto*
then show *card-UNIV-class.card-UNIV* = *Phantom*(('*a*, '*b*, '*c*) *fsm*) *CARD*(('*a*, '*b*, '*c*) *fsm*)
using *True*
by (*simp add*: *card-UNIV-fsm-def*)
next
case *False*
then have *finite* (*UNIV* :: '*a set*) **and** *finite* (*UNIV* :: '*b set*) **and** *finite* (*UNIV* :: '*c set*)
by *force+*
then have *surj* (λ(*q*::'*a*, *Q*, *X*::'*b set*, *Y*::'*c set*, *T*). *FSM.create-fsm-from-sets q Q X Y T*)
using *create-fsm-from-sets-surj* **by** *blast*
then show *card-UNIV-class.card-UNIV* = *Phantom*(('*a*, '*b*, '*c*) *fsm*) *CARD*(('*a*, '*b*, '*c*) *fsm*)
using *False*
by (*simp add*: *card-UNIV-fsm-def*)
qed
end

instantiation *fsm* :: (*type*,*type*,*type*) *cproper-interval* **begin**
definition *cproper-interval-fsm* :: (('*a*,'*b*,'*c*) *fsm*) *proper-interval* **where**
cproper-interval-fsm m1 m2 = *undefined*
instance by(*intro-classes*)(*simp add*: *ID-None ccompare-fsm-def*)
end

44.2 Updated Code Equations

44.2.1 New Code Equations for *remove-proper-prefixes*

lemma *remove-proper-prefixes-refined*[*code*] :
fixes *t* :: ('*a* :: *ccompare*) *list set-rbt*
shows *remove-proper-prefixes* (*RBT-set t*) = (*case ID CCOMPARE*(('*a list*)) *of*
Some - ⇒ (*if* (*is-empty t*) *then* {} *else set* (*paths* (*from-list* (*RBT-Set2.keys t*))))
|
None ⇒ *Code.abort* (*STR* ''*remove-proper-prefixes RBT-set*: *ccompare* = *None*'') (λ-. *remove-proper-prefixes* (*RBT-set t*)))
(**is** *?v1* = *?v2*)
proof (*cases ID CCOMPARE*(('*a list*)))
case *None*
then show *?thesis* **by** *simp*
next
case (*Some a*)
then have ∗:*ID ccompare* ≠ (*None* :: ('*a*::*ccompare list* ⇒ '*a*::*ccompare list* ⇒ *order*) *option*) **by** *auto*

show *?thesis* **proof** (*cases is-empty t*)
case *True*
then show *?thesis* **unfolding** *Some remove-proper-prefixes-def* **by** *auto*

next
case *False*
then have *?v2* = *set* (*paths* (*from-list* (*RBT-Set2.keys t*))) **using** *Some* **by** *auto*
moreover have *?v1* = *set* (*paths* (*from-list* (*RBT-Set2.keys t*)))
using *False* **unfolding** *RBT-set-conv-keys*[*OF* ∗, *of t*] *remove-proper-prefixes-code-trie*

by (*cases RBT-Set2.keys t*; *auto*)
ultimately show *?thesis* **by** *simp*
qed
qed

44.2.2 Special Handling for *set-as-map* on *image*

Avoid creating an intermediate set for (*image f xs*) when evaluating (*set-as-map* (*image f xs*)).

definition *set-as-map-image* :: ('*a1* × '*a2*) *set* ⇒ (('*a1* × '*a2*) ⇒ ('*b1* × '*b2*)) ⇒ ('*b1* ⇒ '*b2 set option*) **where**
set-as-map-image xs f = (*set-as-map* (*image f xs*))

definition *dual-set-as-map-image* :: ('*a1* × '*a2*) *set* ⇒ (('*a1* × '*a2*) ⇒ ('*b1* × '*b2*)) ⇒ (('*a1* × '*a2*) ⇒ ('*c1* × '*c2*)) ⇒ (('*b1* ⇒ '*b2 set option*) × ('*c1* ⇒ '*c2 set option*)) **where**
dual-set-as-map-image xs f1 f2 = (*set-as-map* (*image f1 xs*), *set-as-map* (*image f2 xs*))

lemma *set-as-map-image-code*[*code*] :
fixes *t* :: ('*a1* ::*ccompare* × '*a2* :: *ccompare*) *set-rbt*
and *f1* :: ('*a1* × '*a2*) ⇒ ('*b1* :: *ccompare* × '*b2* ::*ccompare*)
shows *set-as-map-image* (*RBT-set t*) *f1* = (*case ID CCOMPARE*(('*a1* × '*a2*)) *of*
Some - ⇒ *Mapping.lookup*
(*RBT-Set2.fold* (λ *kv m1* .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
t
Mapping.empty) |
None ⇒ *Code.abort* (*STR* ''*set-as-map-image RBT-set*: *ccompare* = *None*'')
(λ-. *set-as-map-image* (*RBT-set t*) *f1*))
proof (*cases ID CCOMPARE*(('*a1* × '*a2*)))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

let *?f'* = λ *t* . (*RBT-Set2.fold* (λ *kv m1* .

(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
t
Mapping.empty)

let *?f* = λ *xs* . (*fold* (λ *kv m1* . *case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*))
xs Mapping.empty)
have ⋀ *xs* :: (*'a1* × *'a2*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*)
proof −
fix *xs* :: (*'a1* × *'a2*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *f1 xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

then have ∗: (*?f* (*xs*@[*xz*])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k'* . *if k'* = *k then Some v else Mapping.lookup m k'*)
by (*metis lookup-update'*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x'* . *if x'* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x'*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*)

by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) **and** {*z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} = {*z*}
using ‹*f1* *xz* = (*x*,*z*)› **by** *fastforce*+
then have *m2*: (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} *else None*)
= (λ *x′* . *if x′* = *x then Some* {*z*} *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x′*)
using ‹*f1* *xz* = (*x*,*z*)› **by** *fastforce*

show *?thesis* **using** *m1 m2 snoc*
using ‹*f1 xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** ∗ **by** *auto*
have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x′*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have (∃ *z′* . (*x*,*z′*) ∈ *f1* ‘ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have (∃ *z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) **by** *fastforce*

have {*z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} = *Some zs*
using ‹(λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} = *zs* **by** *auto*
then show *?thesis*
using ‹*f1 xz* = (*x*, *z*)› **by** *auto*
qed

have ⋀ *a* . (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x′*) *a*
proof −

fix *a* **show** (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x′*) *a*
using ‹{*z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} = *Set.insert z zs*› ‹(∃ *z′* . (*x*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*]))› ‹*f1 xz* = (*x*, *z*)›
by (*cases a* = *x*; *auto*)
qed

then have *m2*: (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z′* . (*x′*,*z′*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} *else None*)
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x′*)
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹*f1 xz* = (*x*, *z*)› **by** *presburger*
qed
qed
qed

then have *Mapping.lookup* (*?f′ t*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set* (*RBT-Set2.keys t*)) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set* (*RBT-Set2.keys t*)} *else None*)
unfolding *fold-conv-fold-keys* **by** *metis*
moreover have *set* (*RBT-Set2.keys t*) = (*RBT-set t*)
using *Some* **by** (*simp add*: *RBT-set-conv-keys*)
ultimately have *Mapping.lookup* (*?f′ t*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ (*RBT-set t*)) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ (*RBT-set t*)} *else None*)
by *force*
then show *?thesis*
using *Some* **unfolding** *set-as-map-image-def set-as-map-def* **by** *simp*
qed

lemma *dual-set-as-map-image-code*[*code*] :
fixes *t* :: (*′a1* ::*ccompare* × *′a2* :: *ccompare*) *set-rbt*
and *f1* :: (*′a1* × *′a2*) ⇒ (*′b1* :: *ccompare* × *′b2* ::*ccompare*)
and *f2* :: (*′a1* × *′a2*) ⇒ (*′c1* :: *ccompare* × *′c2* ::*ccompare*)
shows *dual-set-as-map-image* (*RBT-set t*) *f1 f2* = (*case ID CCOMPARE*((*′a1* × *′a2*)) *of*
Some - ⇒ *let mm* = (*RBT-Set2.fold* (λ *kv* (*m1*,*m2*) .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)
, *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m2* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m2* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*)

m2)))
t
(*Mapping.empty*,*Mapping.empty*))
in (*Mapping.lookup* (*fst mm*), *Mapping.lookup* (*snd mm*)) |
None ⇒ *Code.abort* (*STR ''dual-set-as-map-image RBT-set*: *ccompare* = *None''*)
(λ-. (*dual-set-as-map-image* (*RBT-set t*) *f1 f2*)))
proof (*cases ID CCOMPARE*(('*a1* × '*a2*)))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

let *?f1* = λ *xs* . (*fold* (λ *kv m* . *case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)) *xs Mapping.empty*)
let *?f2* = λ *xs* . (*fold* (λ *kv m* . *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)) *xs Mapping.empty*)

let *?f12* = λ *xs* . *fold* (λ *kv* (*m1*,*m2*) .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)
, *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m2* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m2* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m2*)))
xs
(*Mapping.empty*,*Mapping.empty*)

let *?f1'* = λ *t* . (*RBT-Set2.fold* (λ *kv m* . *case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)) *t Mapping.empty*)
let *?f2'* = λ *t* . (*RBT-Set2.fold* (λ *kv m* . *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m*)) *t Mapping.empty*)

let *?f12'* = λ *t* . *RBT-Set2.fold* (λ *kv* (*m1*,*m2*) .
(*case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)
, *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m2* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m2* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m2*)))
t
(*Mapping.empty*,*Mapping.empty*)

have ⋀*xs* . *?f12 xs* = (*?f1 xs*, *?f2 xs*)
unfolding *fold-dual*[*symmetric*] **by** *simp*

then have *?f12* (*RBT-Set2.keys t*) = (*?f1* (*RBT-Set2.keys t*), *?f2* (*RBT-Set2.keys t*))
by *simp*
then have *?f12' t* = (*?f1' t*, *?f2' t*)
unfolding *fold-conv-fold-keys* **by** *metis*

have *Mapping.lookup* (*fst* (*?f12' t*)) = *set-as-map* (*f1* ‘ (*RBT-set t*))
unfolding ‹*?f12' t* = (*?f1' t*, *?f2' t*)› *fst-conv set-as-map-image-def*[*symmetric*]
using *set-as-map-image-code*[*of t f1*] *Some* **by** *simp*
moreover have *Mapping.lookup* (*snd* (*?f12' t*)) = *set-as-map* (*f2* ‘ (*RBT-set t*))

unfolding ‹*?f12' t* = (*?f1' t*, *?f2' t*)› *snd-conv set-as-map-image-def*[*symmetric*]
using *set-as-map-image-code*[*of t f2*] *Some* **by** *simp*
ultimately show *?thesis*
unfolding *dual-set-as-map-image-def Let-def* **using** *Some* **by** *simp*
qed

44.2.3 New Code Equations for *h*

lemma *h-refined*[*code*] : *h M* (*q*,*x*)
= (*let m* = *set-as-map-image* (*transitions M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in (*case m* (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {}))
apply *transfer*
unfolding *h-code set-as-map-image-def* **by** *simp*

44.2.4 New Code Equations for *canonical-separator'*

lemma *canonical-separator'-refined*[*code*] :
fixes *M* :: ('*a*,'*b*,'*c*) *fsm-impl*
shows
FSM-Impl.canonical-separator' M P q1 q2 = (*if FSM-Impl.fsm-impl.initial P* = (*q1*,*q2*)
then
(*let f'* = *set-as-map-image* (*FSM-Impl.fsm-impl.transitions M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*));
f = (λ*qx* . (*case f' qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions' = *shifted-transitions* (*FSM-Impl.fsm-impl.transitions P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*FSM-Impl.fsm-impl.states P*) (*FSM-Impl.fsm-impl.inputs P*);
ts = *shifted-transitions'* ∪ *distinguishing-transitions-lr*
in FSMI
(*Inl* (*q1*,*q2*))
((*image Inl* (*FSM-Impl.fsm-impl.states P*)) ∪ {*Inr q1*, *Inr q2*})
(*FSM-Impl.fsm-impl.inputs M* ∪ *FSM-Impl.fsm-impl.inputs P*)
(*FSM-Impl.fsm-impl.outputs M* ∪ *FSM-Impl.fsm-impl.outputs P*)
(*ts*))
else FSMI
(*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {})
unfolding *set-as-map-image-def* **by** *simp*

44.2.5 New Code Equations for *calculate-test-paths*

lemma *calculate-test-paths-refined*[*code*] :
calculate-test-paths M m d-reachable-states r-distinguishable-pairs repetition-sets =
(*let*
paths-with-witnesses
= (*image* (λ *q* . (*q*,*m-traversal-paths-with-witness M q repetition-sets m*)) *d-reachable-states*);
get-paths
= *m2f* (*set-as-map paths-with-witnesses*);
PrefixPairTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *prefix-pair-tests q mrsps*;
PreamblePrefixTests
= ⋃ *q* ∈ *d-reachable-states* . ⋃ *mrsps* ∈ *get-paths q* . *preamble-prefix-tests q mrsps d-reachable-states*;
PreamblePairTests
= *preamble-pair-tests* (⋃ (*q*,*pw*) ∈ *paths-with-witnesses* . ((λ (*p*,(*rd*,*dr*)) . *dr*) ‘ *pw*)) *r-distinguishable-pairs*;
tests
= *PrefixPairTests* ∪ *PreamblePrefixTests* ∪ *PreamblePairTests*;
tps′
= *m2f-by* ⋃ (*set-as-map-image paths-with-witnesses* (λ (*q*,*p*) . (*q*, *image fst p*)));
dual-maps
= *dual-set-as-map-image tests* (λ (*q*,*p*,*q′*) . (*q*,*p*)) (λ (*q*,*p*,*q′*) . ((*q*,*p*),*q′*));
tps′′
= *m2f* (*fst dual-maps*);
tps
= (λ *q* . *tps′ q* ∪ *tps′′ q*);
rd-targets
= *m2f* (*snd dual-maps*)
in (*tps*, *rd-targets*))

unfolding *calculate-test-paths-def Let-def dual-set-as-map-image-def fst-conv snd-conv set-as-map-image-def*
by *simp*

44.2.6 New Code Equations for *prefix-pair-tests*

fun *target′* :: *′state* ⇒ (*′state*, *′input*, *′output*) *path* ⇒ *′state* **where**
target′ q [] = *q* |
target′ q p = *t-target* (*last p*)

lemma *target-refined*[*code*] :
target q p = *target′ q p*
proof (*cases p rule*: *rev-cases*)
case *Nil*
then show *?thesis* **by** *auto*

next
case (*snoc p′ t*)
then have $p \neq []$ **by** *auto*
then show *?thesis* **unfolding** *snoc target.simps visited-states.simps*
by (*metis* (*no-types*, *lifting*) *last-ConsR last-map list.map-disc-iff target′.elims*)

qed

lemma *prefix-pair-tests-refined*[*code*] :
fixes *t* :: (('*a* ::*ccompare*,'*b*::*ccompare*,'*c*::*ccompare*) *traversal-path* × ('*a set* × '*a set*)) *set-rbt*
shows *prefix-pair-tests q* (*RBT-set t*) = (*case ID CCOMPARE*((('*a*,'*b*,'*c*) *traversal-path* × ('*a set* × '*a set*))) *of*
Some - ⇒ *set*
(*concat* (*map* (λ (*p*,(*rd*,*dr*)) .
(*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))])
(*filter* (λ (*p1*,*p2*) . (*target q p1*) ≠ (*target q p2*) ∧ (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd*) (*prefix-pairs p*)))))
(*RBT-Set2.keys t*))) |
None ⇒ *Code.abort* (*STR* ''*prefix-pair-tests RBT-set*: *ccompare* = *None*'')
(λ-. (*prefix-pair-tests q* (*RBT-set t*))))
(**is** *prefix-pair-tests q* (*RBT-set t*) = *?C*)
proof (*cases ID CCOMPARE*((('*a* ::*ccompare*,'*b*::*ccompare*,'*c*::*ccompare*) *traversal-path* × ('*a set* × '*a set*))))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

have ∗: *?C* = (⋃(*image* (λ (*p*,(*rd*,*dr*)) . ⋃ (*set* (*map* (λ (*p1*,*p2*) . {(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))}) (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*))))) (*set* (*RBT-Set2.keys t*))))
proof −
let *?S1* = *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]) (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*))))) (*RBT-Set2.keys t*)))
let *?S2* = (⋃(*image* (λ (*p*,(*rd*,*dr*)) . ⋃ (*set* (*map* (λ (*p1*,*p2*) . {(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))}) (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*))))) (*set* (*RBT-Set2.keys t*))))

have ∗: *?C* = *?S1*
proof −
have ∗: ⋀ *rd p* . (*filter* (λ (*p1*,*p2*) . (*target q p1*) ≠ (*target q p2*) ∧ (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd*) (*prefix-pairs p*)) = (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*))
by *meson*
have *?C* = *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]) (*filter* (λ (*p1*,*p2*) . (*target q p1*)

$\neq$ (*target q p2*) $\wedge$ (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd*) (*prefix-pairs p*))))) (*RBT-Set2.keys t*)))
using *Some* **by** *auto*
then show *?thesis*
unfolding $*$ **by** *presburger*
qed

have *union-filter-helper*: $\bigwedge$ *xs f x1 x2 y* . *y* $\in$ *f* (*x1*,*x2*) $\Longrightarrow$ (*x1*,*x2*) $\in$ *set xs* $\Longrightarrow$ *y* $\in \bigcup$ (*set* (*map f xs*))
by *auto*
have *concat-set-helper* : $\bigwedge$ *xss xs x* . *x* $\in$ *set xs* $\Longrightarrow$ *xs* $\in$ *set xss* $\Longrightarrow$ *x* $\in$ *set* (*concat xss*)
by *auto*

have $\bigwedge$ *x* . *x* $\in$ *?S1* $\Longrightarrow$ *x* $\in$ *?S2*
proof −
fix *x* **assume** *x* $\in$ *?S1*
then obtain *p rd dr p1 p2* **where** (*p*,(*rd*,*dr*)) $\in$ *set* (*RBT-Set2.keys t*)
and (*p1*,*p2*) $\in$ *set* ((*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*)))
and *x* $\in$ *set* [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]
by *auto*
then have *x* $\in$ {(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))}
by *auto*
then have *x* $\in \bigcup$ (*set* (*map* (λ (*p1*,*p2*) . {(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))}) (*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*))))
using *union-filter-helper*[*OF* - ‹(*p1*,*p2*) $\in$ *set* ((*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*)))›, *of x* (λ(*p1*, *p2*). {(*q*, *p1*, *target q p2*), (*q*, *p2*, *target q p1*)})] **by** *simp*
then show *x* $\in$ *?S2*
using ‹(*p*,(*rd*,*dr*)) $\in$ *set* (*RBT-Set2.keys t*)› **by** *blast*
qed

moreover have $\bigwedge$ *x* . *x* $\in$ *?S2* $\Longrightarrow$ *x* $\in$ *?S1*
proof −
fix *x* **assume** *x* $\in$ *?S2*
then obtain *p rd dr p1 p2* **where** (*p*,(*rd*,*dr*)) $\in$ *set* (*RBT-Set2.keys t*)
and (*p1*,*p2*) $\in$ *set* ((*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*)))
and *x* $\in$ {(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))}
by *auto*

then have $*$: *x* $\in$ *set* [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))] **by** *auto*
have $**$: [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))] $\in$ *set* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]) (*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*)))
using ‹(*p1*,*p2*) $\in$ *set* ((*filter* (λ (*p1*,*p2*) . (*target q p1*) $\in$ *rd* $\wedge$ (*target q p2*) $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ (*target q p2*)) (*prefix-pairs p*)))› **by** *force*

have ∗∗∗: (*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]) (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*)))) ∈ *set* ((*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*p2*) . [(*q*,*p1*,(*target q p2*)), (*q*,*p2*,(*target q p1*))]) (*filter* (λ (*p1*,*p2*) . (*target q p1*) ∈ *rd* ∧ (*target q p2*) ∈ *rd* ∧ (*target q p1*) ≠ (*target q p2*)) (*prefix-pairs p*))))) (*RBT-Set2.keys t*)))
using ‹(*p*,(*rd*,*dr*)) ∈ *set* (*RBT-Set2.keys t*)› **by** *force*

show *x* ∈ *?S1*
using *concat-set-helper*[*OF concat-set-helper*[*OF* ∗ ∗∗] ∗∗∗] **by** *assumption*
qed

ultimately show *?thesis* **unfolding** ∗ **by** *blast*
qed

show *?thesis*
unfolding ∗ **unfolding** *prefix-pair-tests-code*
using *Some* **by** (*simp add*: *RBT-set-conv-keys*)
qed

44.2.7 New Code Equations for *preamble-prefix-tests*

lemma *preamble-prefix-tests-refined*[*code*] :
fixes *t1* :: (('*a* ::*ccompare*,'*b*::*ccompare*,'*c*::*ccompare*) *traversal-path* × ('*a set* × '*a set*)) *set-rbt*
and *t2* :: '*a set-rbt*
shows *preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*) = (*case ID CCOMPARE*((('*a*,'*b*,'*c*) *traversal-path* × ('*a set* × '*a set*))) *of*
Some - ⇒ (*case ID CCOMPARE*('*a*) *of*
Some - ⇒ *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) .
(*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))])
(*filter* (λ (*p1*,*q2*) . (*target q p1*) ≠ *q2* ∧ (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd*)
(*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))))
(*RBT-Set2.keys t1*))) |
None ⇒ *Code.abort* (*STR* ''*prefix-pair-tests RBT-set*: *ccompare* = *None*'') (λ-. (*preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*)))) |
None ⇒ *Code.abort* (*STR* ''*prefix-pair-tests RBT-set*: *ccompare* = *None*'') (λ-. (*preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*))))
(**is** *preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*) = *?C*)
proof (*cases ID CCOMPARE*((('*a*,'*b*,'*c*) *traversal-path* × ('*a set* × '*a set*))))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)
then have *k1*: (*RBT-set t1*) = *set* (*RBT-Set2.keys t1*)
by (*simp add*: *RBT-set-conv-keys*)

show *?thesis* **proof** (*cases ID CCOMPARE*('*a*))

case *None*
then show *?thesis* **using** *Some* **by** *auto*
next
case (*Some b*)
then have *k2*: (*RBT-set t2*) = *set* (*RBT-Set2.keys t2*)
by (*simp add*: *RBT-set-conv-keys*)

have *preamble-prefix-tests q* (*RBT-set t1*) (*RBT-set t2*) = (⋃(*p*, *rd*, *dr*)∈ *set* (*RBT-Set2.keys t1*). ⋃(*p1*, *q2*)∈*Set.filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*set* (*prefixes p*) × (*set* (*RBT-Set2.keys t2*))). {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)})
unfolding *preamble-prefix-tests-code k1 k2* **by** *simp*

moreover have *?C* = (⋃(*p*, *rd*, *dr*)∈ *set* (*RBT-Set2.keys t1*). ⋃(*p1*, *q2*)∈*Set.filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*set* (*prefixes p*) × (*set* (*RBT-Set2.keys t2*))). {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)})
proof −
let *?S1* = *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))) (*RBT-Set2.keys t1*)))
let *?S2* = (⋃(*p*, *rd*, *dr*)∈ *set* (*RBT-Set2.keys t1*). ⋃(*p1*, *q2*)∈*Set.filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*set* (*prefixes p*) × (*set* (*RBT-Set2.keys t2*))). {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)})

have ∗: *?C* = *?S1*
proof −
have ∗: ⋀ *rd p* . (*filter* (λ (*p1*,*q2*) . (*target q p1*) ≠ *q2* ∧ (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))) = (*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))
by *meson*
have *?C* = *set* (*concat* (*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) ≠ *q2* ∧ (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))) (*RBT-Set2.keys t1*)))
using *Some* ‹*ID ccompare* = *Some a*› **by** *auto*
then show *?thesis*
unfolding ∗ **by** *presburger*
qed

have *union-filter-helper*: ⋀ *xs f x1 x2 y* . *y* ∈ *f* (*x1*,*x2*) ⟹ (*x1*,*x2*) ∈ *set xs* ⟹ *y* ∈ ⋃ (*set* (*map f xs*))
by *auto*
have *concat-set-helper* : ⋀ *xss xs x* . *x* ∈ *set xs* ⟹ *xs* ∈ *set xss* ⟹ *x* ∈ *set* (*concat xss*)
by *auto*

have $\bigwedge x . x \in ?S1 \Longrightarrow x \in ?S2$
proof −
fix x **assume** $x \in ?S1$

obtain *prddr* **where** *prddr* ∈ *set* (*RBT-Set2.keys t1*)
and $x \in$ *set* ((λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))) *prddr*)
using *concat-map-elem*[*OF* ‹$x \in ?S1$›] **by** *blast*

moreover obtain *p rd dr* **where** *prddr* = (*p*,(*rd*,*dr*))
using *prod-cases3* **by** *blast*

ultimately have (*p*,(*rd*,*dr*)) ∈ *set* (*RBT-Set2.keys t1*)
and $x \in$ *set* ((*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))))
by *auto*
then obtain *p1 q2* **where** (*p1*,*q2*) ∈ *set* ((*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))
and $x \in$ *set* [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]
by *auto*

then have $x \in$ {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))}
by *auto*
then have $x \in \bigcup$ (*set* (*map* (λ(*p1*, *q2*). {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)}) (*filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))
using *union-filter-helper*[*OF* - ‹(*p1*,*q2*) ∈ *set* ((*filter* (λ (*p1*,*q2*) . (*target q p1*) ∈ *rd* ∧ *q2* ∈ *rd* ∧ (*target q p1*) ≠ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))›, *of x* (λ (*p1*,*q2*) . {(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))})] **by** *simp*
then have $x \in$ (⋃(*p1*, *q2*)∈*Set.filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*set* (*prefixes p*) × (*set* (*RBT-Set2.keys t2*))). {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)})
by *auto*
then show $x \in ?S2$
using ‹(*p*,(*rd*,*dr*)) ∈ *set* (*RBT-Set2.keys t1*)› **by** *blast*
qed

moreover have $\bigwedge x . x \in ?S2 \Longrightarrow x \in ?S1$
proof −
fix x **assume** $x \in ?S2$
then obtain *p rd dr p1 q2* **where** (*p*, *rd*, *dr*)∈ *set* (*RBT-Set2.keys t1*)
and (*p1*, *q2*)∈*Set.filter* (λ(*p1*, *q2*). *target q p1* ∈ *rd* ∧ *q2* ∈ *rd* ∧ *target q p1* ≠ *q2*) (*set* (*prefixes p*) × (*set* (*RBT-Set2.keys t2*)))
and $x \in$ {(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)}
by *blast*

then have $*$:$x \in$ *set* [(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)]
by *auto*

have (*p1*,*q2*) $\in$ *set* (*filter* (λ(*p1*, *q2*). *target q p1* $\in$ *rd* $\wedge$ *q2* $\in$ *rd* $\wedge$ *target q p1* $\neq$ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))
using ‹(*p1*, *q2*)$\in$*Set.filter* (λ(*p1*, *q2*). *target q p1* $\in$ *rd* $\wedge$ *q2* $\in$ *rd* $\wedge$ *target q p1* $\neq$ *q2*) (*set* (*prefixes p*) $\times$ (*set* (*RBT-Set2.keys t2*)))›
by *auto*
then have $**$:[(*q*, *p1*, *q2*), (*q2*, [], *target q p1*)] $\in$ *set* ((*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) $\in$ *rd* $\wedge$ *q2* $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))
by *force*

have $***$: (*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) $\in$ *rd* $\wedge$ *q2* $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*))))) $\in$ *set* (*map* (λ (*p*,(*rd*,*dr*)) . (*concat* (*map* (λ (*p1*,*q2*) . [(*q*,*p1*,*q2*), (*q2*,[],(*target q p1*))]) (*filter* (λ (*p1*,*q2*) . (*target q p1*) $\in$ *rd* $\wedge$ *q2* $\in$ *rd* $\wedge$ (*target q p1*) $\neq$ *q2*) (*List.product* (*prefixes p*) (*RBT-Set2.keys t2*)))))) (*RBT-Set2.keys t1*))
using ‹(*p*, *rd*, *dr*)$\in$ *set* (*RBT-Set2.keys t1*)› **by** *force*

show $x \in$ *?S1*
using *concat-set-helper*[*OF concat-set-helper*[*OF* $*$ $**$] $***$] **by** *assumption*
qed

ultimately show *?thesis* **unfolding** $*$ **by** *blast*
qed

ultimately show *?thesis* **by** *simp*
qed
qed

end

45 Data Refinement on FSM Representations

This section introduces a refinement of the type of finite state machines for code generation, maintaining mappings to access the transition relation to avoid repeated computations.

theory *FSM-Code-Datatype*
imports *FSM HOL−Library.Mapping Containers.Containers*
begin

45.1 Mappings and Function *h*

fun *list-as-mapping* :: ($'a \times 'c$) *list* $\Rightarrow$ ($'a$,$'c$ *set*) *mapping* **where**

list-as-mapping xs = (*foldr* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*insert z zs*) *m*)
xs
Mapping.empty)

lemma *list-as-mapping-lookup*:
fixes *xs* :: ('*a* × '*c*) *list*
shows (*Mapping.lookup* (*list-as-mapping xs*)) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ (*set xs*)) *then Some* {*z* . (*x*,*z*) ∈ (*set xs*)} *else None*)
proof −
let *?P* = λ*m* :: ('*a*,'*c set*) *mapping* . (*Mapping.lookup m*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ (*set xs*)) *then Some* {*z* . (*x*,*z*) ∈ (*set xs*)} *else None*)

have *?P* (*list-as-mapping xs*)
proof (*induction xs*)
case *Nil*
then show *?case*
using *Mapping.lookup-empty* **by** *fastforce*
next
case (*Cons xz xs*)
then obtain *x z* **where** *xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

have ∗: (*list-as-mapping* ((*x*,*z*)#*xs*)) = (*case Mapping.lookup* (*list-as-mapping xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*list-as-mapping xs*) |
Some zs ⇒ *Mapping.update x* (*insert z zs*) (*list-as-mapping xs*))
unfolding *list-as-mapping.simps*
by *auto*

show *?case* **proof** (*cases Mapping.lookup* (*list-as-mapping xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*list-as-mapping* ((*x*,*z*)#*xs*)) = (*Mapping.lookup* (*Mapping.update x* {*z*} (*list-as-mapping xs*)))
using ∗ **by** *auto*
then have *m1*: *Mapping.lookup* (*list-as-mapping* ((*x*,*z*)#*xs*)) = (λ *x*′ . *if x*′ = *x then Some* {*z*} *else Mapping.lookup* (*list-as-mapping xs*) *x*′)
by (*metis* (*lifting*) *lookup-update*′)

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *None*
using *None Cons* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z* . (*x*,*z*) ∈ *set* ((*x*,*z*)#*xs*)) **and** {*z*′ . (*x*,*z*′) ∈ *set* ((*x*,*z*)#*xs*)} = {*z*}

by *auto*
then have *m2*: (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* $((x,z)\#xs)$)
then Some $\{z'$. $(x',z') \in$ *set* $((x,z)\#xs)\}$
else None)
= (λ x' . *if* $x' = x$
then Some $\{z\}$ *else* (λ x . *if* ($\exists$ z . $(x,z) \in$ *set xs*)
then Some $\{z$. $(x,z) \in$ *set xs*$\}$
else None) x')
by *force*

show *?thesis* **using** *m1 m2 Cons*
using ‹$xz = (x, z)$› **by** *presburger*
next
case (*Some zs*)
then have $**$: *Mapping.lookup* (*list-as-mapping* $((x,z)\#xs)$) = (*Mapping.lookup* (*Mapping.update x* (*insert z zs*) (*list-as-mapping xs*)))
using $*$ **by** *auto*
then have *m1*: *Mapping.lookup* (*list-as-mapping* $((x,z)\#xs)$) = (λ x' . *if* x' = x *then Some* (*insert z zs*) *else Mapping.lookup* (*list-as-mapping xs*) x')
by (*metis* (*lifting*) *lookup-update'*)

have (λ x . *if* ($\exists$ z . $(x,z) \in$ *set xs*) *then Some* $\{z$. $(x,z) \in$ *set xs*$\}$ *else None*) x = *Some zs*
using *Some Cons* **by** *auto*
then have ($\exists$ z . $(x,z) \in$ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have ($\exists$ z . $(x,z) \in$ *set* $((x,z)\#xs)$) **by** *simp*

have $\{z'$. $(x,z') \in$ *set* $((x,z)\#xs)\}$ = *insert z zs*
proof −
have *Some* $\{z$. $(x,z) \in$ *set xs*$\}$ = *Some zs*
using ‹(λ x . *if* ($\exists$ z . $(x,z) \in$ *set xs*) *then Some* $\{z$. $(x,z) \in$ *set xs*$\}$ *else None*) x
= *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have $\{z$. $(x,z) \in$ *set xs*$\}$ = *zs* **by** *auto*
then show *?thesis* **by** *auto*
qed

have $\bigwedge$ a . (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* $((x,z)\#xs)$)
then Some $\{z'$. $(x',z') \in$ *set* $((x,z)\#xs)\}$ *else None*) a
= (λ x' . *if* $x' = x$
then Some (*insert z zs*)
else (λ x . *if* ($\exists$ z . $(x,z) \in$ *set xs*)
then Some $\{z$. $(x,z) \in$ *set xs*$\}$ *else None*) x') a
proof −
fix a **show** (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* $((x,z)\#xs)$)
then Some $\{z'$. $(x',z') \in$ *set* $((x,z)\#xs)\}$ *else None*) a
= (λ x' . *if* $x' = x$

then Some (*insert z zs*)
else (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*)
then Some {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x*′) *a*
using ‹{*z*′ . (*x*,*z*′) $\in$ *set* ((*x*,*z*)#*xs*)} = *insert z zs*› ‹($\exists$ *z* . (*x*,*z*) $\in$ *set* ((*x*,*z*)#*xs*))›
by (*cases a* = *x*; *auto*)
qed

then have *m2*: (λ *x*′ . *if* ($\exists$ *z*′ . (*x*′,*z*′) $\in$ *set* ((*x*,*z*)#*xs*))
then Some {*z*′ . (*x*′,*z*′) $\in$ *set* ((*x*,*z*)#*xs*)} *else None*)
= (λ *x*′ . *if x*′ = *x*
then Some (*insert z zs*)
else (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*)
then Some {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x*′)
by *auto*

show *?thesis* **using** *m1 m2 Cons*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
qed
qed
then show *?thesis* **.**
qed

lemma *list-as-mapping-lookup-transitions* :
(*case* (*Mapping.lookup* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′)) *ts*)) (*q*,*x*))
of Some ts $\Rightarrow$ *ts* | *None* $\Rightarrow$ {}) = { (*y*,*q*′) . (*q*,*x*,*y*,*q*′) $\in$ *set ts*}
(**is** *?S1* = *?S2*)
proof (*cases* $\exists$*z*. ((*q*, *x*), *z*) $\in$ *set* (*map* (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*, *q*′)) *ts*))
case *True*
then have *?S1* = {*z*. ((*q*, *x*), *z*) $\in$ *set* (*map* (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*, *q*′)) *ts*)}
unfolding *list-as-mapping-lookup* **by** *auto*
also have ... = *?S2*
by (*induction ts*; *auto*)
finally show *?thesis* **.**
next
case *False*
then have *?S1* = {}
unfolding *list-as-mapping-lookup* **by** *auto*
also have ... = *?S2*
using *False* **by** (*induction ts*; *auto*)
finally show *?thesis* **.**
qed

lemma *list-as-mapping-Nil* :
list-as-mapping [] = *Mapping.empty*
by *auto*

```
definition set-as-mapping :: ('a × 'c) set ⇒ ('a,'c set) mapping where
  set-as-mapping s = (THE m . Mapping.lookup m = (set-as-map s))

lemma set-as-mapping-ob :
  obtains m where set-as-mapping s = m and Mapping.lookup m = set-as-map
s
proof −
  obtain m where *:Mapping.lookup m = set-as-map s
    using Mapping.lookup.abs-eq by auto
  moreover have (THE x. Mapping.lookup x = set-as-map s) = m
    using the-equality[of λm . Mapping.lookup m = set-as-map s, OF *]
    unfolding *[symmetric]
    by (simp add: mapping-eqI)
  ultimately show ?thesis
    using that[of m] unfolding set-as-mapping-def by blast
qed

lemma set-as-mapping-refined [code]:
  fixes t :: ('a :: ccompare × 'c :: ccompare) set-rbt
  and   xs:: ('b :: ceq × 'd :: ceq) set-dlist
  shows set-as-mapping (DList-set xs) = (case ID CEQ(('b × 'd)) of
          Some - ⇒ (DList-Set.fold (λ (x,z) m . case Mapping.lookup m (x) of
                      None ⇒ Mapping.update (x) {z} m |
                      Some zs ⇒ Mapping.update (x) (Set.insert z zs) m)
                    xs
                    Mapping.empty) |
          None   ⇒ Code.abort (STR ''set-as-map RBT-set: ccompare = None'')
                        (λ-. set-as-mapping (DList-set xs)))
    (is set-as-mapping (DList-set xs) = ?C2 (DList-set xs))
    and set-as-mapping (RBT-set t) = (case ID CCOMPARE(('a × 'c)) of
          Some - ⇒ (RBT-Set2.fold (λ (x,z) m . case Mapping.lookup m (x) of
                      None ⇒ Mapping.update (x) {z} m |
                      Some zs ⇒ Mapping.update (x) (Set.insert z zs) m)
                    t
                    Mapping.empty) |
          None   ⇒ Code.abort (STR ''set-as-map RBT-set: ccompare = None'')
                        (λ-. set-as-mapping (RBT-set t)))
    (is set-as-mapping (RBT-set t) = ?C1 (RBT-set t))
proof −
  show set-as-mapping (RBT-set t) = ?C1 (RBT-set t)
  proof (cases ID CCOMPARE(('a × 'c)))
    case None
    then show ?thesis by auto
  next
    case (Some a)

    let ?f' = (λ t' . (RBT-Set2.fold (λ (x,z) m . case Mapping.lookup m x of
                                  None ⇒ Mapping.update x {z} m |
```

Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
t′
Mapping.empty))

let *?f* = λ *xs* . (*fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
xs Mapping.empty)
have ⋀ *xs* :: (′*a* × ′*c*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof −
fix *xs* :: (′*a* × ′*c*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

have ∗: (*?f* (*xs*@[(*x*,*z*)])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x′*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *set xs*)

by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) **and** {*z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} = {*z*}
by *fastforce*+
then have *m2*: (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x′* . *if x′* = *x then Some* {*z*} *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*)
by *force*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** ∗ **by** *auto*
have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x′*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have (∃ *z′* . (*x*,*z′*) ∈ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have (∃ *z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) **by** *fastforce*

have {*z′* . (*x*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) ∈ *set xs*} = *Some zs*
using ‹(λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) ∈ *set xs*} = *zs* **by** *auto*
then show *?thesis* **by** *auto*
qed

have ⋀ *a* . (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])} *else None*) *a*
= (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*) *x′*) *a*
proof −
fix *a* **show** (λ *x′* . *if* (∃ *z′* . (*x′*,*z′*) ∈ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z′* .

$(x',z') \in set\ (xs@[(x,z)])\}$ *else None*) *a*
$= (\lambda\ x'\ .\ if\ x' = x\ then\ Some\ (Set.insert\ z\ zs)\ else\ (\lambda\ x\ .\ if\ (\exists\ z\ .\ (x,z) \in set\ xs)\ then\ Some\ \{z\ .\ (x,z) \in set\ xs\}\ else\ None)\ x')\ a$
using ‹$\{z'\ .\ (x,z') \in set\ (xs@[(x,z)])\} = Set.insert\ z\ zs$› ‹$(\exists\ z'\ .\ (x,z') \in set\ (xs@[(x,z)]))$›
by (*cases* $a = x$; *auto*)
qed

then have *m2*: $(\lambda\ x'\ .\ if\ (\exists\ z'\ .\ (x',z') \in set\ (xs@[(x,z)]))\ then\ Some\ \{z'\ .\ (x',z') \in set\ (xs@[(x,z)])\}\ else\ None)$
$= (\lambda\ x'\ .\ if\ x' = x\ then\ Some\ (Set.insert\ z\ zs)\ else\ (\lambda\ x\ .\ if\ (\exists\ z\ .\ (x,z) \in set\ xs)\ then\ Some\ \{z\ .\ (x,z) \in set\ xs\}\ else\ None)\ x')$
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹$xz = (x,\ z)$› **by** *presburger*
qed
qed
qed
then have $Mapping.lookup\ (?f'\ t) = (\lambda\ x\ .\ if\ (\exists\ z\ .\ (x,z) \in set\ (RBT\text{-}Set2.keys\ t))\ then\ Some\ \{z\ .\ (x,z) \in set\ (RBT\text{-}Set2.keys\ t)\}\ else\ None)$
unfolding *fold-conv-fold-keys* **by** *metis*
moreover have $set\ (RBT\text{-}Set2.keys\ t) = (RBT\text{-}set\ t)$
using *Some* **by** (*simp add*: *RBT-set-conv-keys*)
ultimately have $Mapping.lookup\ (?f'\ t) = (\lambda\ x\ .\ if\ (\exists\ z\ .\ (x,z) \in (RBT\text{-}set\ t))\ then\ Some\ \{z\ .\ (x,z) \in (RBT\text{-}set\ t)\}\ else\ None)$
by *force*
then have $Mapping.lookup\ (?f'\ t) = set\text{-}as\text{-}map\ (RBT\text{-}set\ t)$
unfolding *set-as-map-def* **by** *blast*
then have $*$:$Mapping.lookup\ (?C1\ (RBT\text{-}set\ t)) = set\text{-}as\text{-}map\ (RBT\text{-}set\ t)$
unfolding *Some* **by** *force*

have $\bigwedge\ t'\ .\ Mapping.lookup\ (?C1\ (RBT\text{-}set\ t)) = Mapping.lookup\ (?C1\ t') \Longrightarrow (?C1\ (RBT\text{-}set\ t)) = (?C1\ t')$
by (*simp add*: *Some*)
then have $**$: $(\bigwedge x.\ Mapping.lookup\ x = set\text{-}as\text{-}map\ (RBT\text{-}set\ t) \Longrightarrow x = (?C1\ (RBT\text{-}set\ t)))$
by (*simp add*: $*$ *mapping-eqI*)

show *?thesis*
using *the-equality*[*of* $\lambda\ m\ .\ Mapping.lookup\ m = (set\text{-}as\text{-}map\ (RBT\text{-}set\ t))$, *OF* $*$ $**$]
unfolding *set-as-mapping-def* **by** *blast*
qed

show $set\text{-}as\text{-}mapping\ (DList\text{-}set\ xs) = ?C2\ (DList\text{-}set\ xs)$
proof (*cases ID CEQ*(('$b \times$ 'd)))

case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

let *?f′* = (λ *t′* . (*DList-Set.fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
t′
Mapping.empty))

let *?f* = λ *xs* . (*fold* (λ (*x*,*z*) *m* . *case Mapping.lookup m x of*
None ⇒ *Mapping.update x* {*z*} *m* |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) *m*)
xs Mapping.empty)
have ∗: ⋀ *xs* :: (*′b* × *′d*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof −
fix *xs* :: (*′b* × *′d*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

have ∗: (*?f* (*xs*@[(*x*,*z*)])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x′* . *if x′* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x′*)

unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have $\neg$($\exists$ *z* . (*x*,*z*) $\in$ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) **and** {*z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} = {*z*}
by *fastforce+*
then have *m2*: (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} *else None*)
= (λ *x'* . *if x'* = *x then Some* {*z*} *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x'*)
by *force*

show *?thesis* **using** *m1 m2 snoc*
using ‹*xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have $**$: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** $*$ **by** *auto*
have *scheme*: $\bigwedge$ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k'* . *if k'* = *k then Some v else Mapping.lookup m k'*)
by (*metis lookup-update'*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[(*x*,*z*)])) = (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x'*)
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])) **by** *fastforce*

have {*z'* . (*x*,*z'*) $\in$ *set* (*xs*@[(*x*,*z*)])} = *Set.insert z zs*
proof $-$
have *Some* {*z* . (*x*,*z*) $\in$ *set xs*} = *Some zs*
using ‹(λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) $\in$ *set xs*} = *zs* **by** *auto*
then show *?thesis* **by** *auto*
qed

have $\bigwedge a$. (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* (xs@[(x,z)])) *then Some* {z' . $(x',z') \in$ *set* (xs@[(x,z)])} *else None*) a
= (λ x' . *if* $x' = x$ *then Some* (*Set.insert* z zs) *else* (λ x . *if* ($\exists$ z . $(x,z) \in$ *set* xs) *then Some* {z . $(x,z) \in$ *set* xs} *else None*) x') a
proof −
fix a **show** (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* (xs@[(x,z)])) *then Some* {z' . $(x',z') \in$ *set* (xs@[(x,z)])} *else None*) a
= (λ x' . *if* $x' = x$ *then Some* (*Set.insert* z zs) *else* (λ x . *if* ($\exists$ z . $(x,z) \in$ *set* xs) *then Some* {z . $(x,z) \in$ *set* xs} *else None*) x') a
using ‹{z' . $(x,z') \in$ *set* (xs@[(x,z)])} = *Set.insert* z zs› ‹($\exists$ z' . $(x,z') \in$ *set* (xs@[(x,z)]))›
by (*cases* $a = x$; *auto*)
qed

then have *m2*: (λ x' . *if* ($\exists$ z' . $(x',z') \in$ *set* (xs@[(x,z)])) *then Some* {z' . $(x',z') \in$ *set* (xs@[(x,z)])} *else None*)
= (λ x' . *if* $x' = x$ *then Some* (*Set.insert* z zs) *else* (λ x . *if* ($\exists$ z . $(x,z) \in$ *set* xs) *then Some* {z . $(x,z) \in$ *set* xs} *else None*) x')
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹$xz = (x, z)$› **by** *presburger*
qed
qed
qed

have *ID CEQ*($'b \times 'd$) $\neq$ *None*
using *Some* **by** *auto*
then have $**$: $\bigwedge$ x . $x \in$ *set* (*list-of-dlist xs*) = ($x \in$ (*DList-set xs*))
by (*simp add*: *Collect-member DList-set-def*)
have *Mapping.lookup* (*?f′ xs*) = (λ x . *if* ($\exists$ z . $(x,z) \in$ (*DList-set xs*)) *then Some* {z . $(x,z) \in$ (*DList-set xs*)} *else None*)
using $*$[*of* (*list-of-dlist xs*)]
unfolding *DList-Set.fold.rep-eq* $**$ **by** *assumption*
then have *Mapping.lookup* (*?f′ xs*) = *set-as-map* (*DList-set xs*)
unfolding *set-as-map-def* **by** *blast*
then have $*$:*Mapping.lookup* (*?C2* (*DList-set xs*)) = *set-as-map* (*DList-set xs*)
unfolding *Some* **by** *force*

have $\bigwedge$ t' . *Mapping.lookup* (*?C2* (*DList-set xs*)) = *Mapping.lookup* (*?C2* t') $\Longrightarrow$ (*?C2* (*DList-set xs*)) = (*?C2* t')
by (*simp add*: *Some*)
then have $**$: ($\bigwedge x$. *Mapping.lookup* x = *set-as-map* (*DList-set xs*) $\Longrightarrow$ x = (*?C2* (*DList-set xs*)))
by (*simp add*: $*$ *mapping-eqI*)

```
    show ?thesis
      using the-equality[of λ m . Mapping.lookup m = (set-as-map (DList-set xs)),
OF * **]
      unfolding set-as-mapping-def by blast
  qed
qed


fun h-obs-impl-from-h :: (('state × 'input), ('output × 'state) set) mapping ⇒
('state × 'input, ('output, 'state) mapping) mapping where
  h-obs-impl-from-h h' = Mapping.map-values
                         (λ - yqs . let m' = set-as-mapping yqs;
                                      m'' = Mapping.filter (λ y qs . card qs = 1) m';
                                        m''' = Mapping.map-values (λ - qs . the-elem
qs) m''
                                    in m''')
                         h'

fun h-obs-impl :: (('state × 'input), ('output × 'state) set) mapping ⇒ 'state ⇒
'input ⇒ 'output ⇒ 'state option where
  h-obs-impl h' q x y  = (let
       tgts = snd ‘ Set.filter (λ(y',q') . y' = y) (case (Mapping.lookup h' (q,x)) of
Some ts ⇒ ts | None ⇒ {})
    in if card tgts = 1
      then Some (the-elem tgts)
      else None)

abbreviation(input) h-obs-lookup ≡ (λ h' q x y . (case Mapping.lookup h' (q,x)
of Some m ⇒ Mapping.lookup m y | None ⇒ None))

lemma h-obs-impl-from-h-invar : h-obs-impl h' q x y = h-obs-lookup (h-obs-impl-from-h
h') q x y
  (is ?A q x y = ?B q x y)
proof (cases Mapping.lookup h' (q,x))
  case None

  then have Mapping.lookup (h-obs-impl-from-h h') (q,x) = None
    unfolding h-obs-impl-from-h.simps Mapping.lookup-map-values
    by auto
  then have ?B q x y = None
    by auto
  moreover have ?A q x y = None
    unfolding h-obs-impl.simps Let-def None
    by (simp add:)
  ultimately show ?thesis
    by presburger
next
  case (Some yqs)
```

define m' **where** $m' = set\text{-}as\text{-}mapping\ yqs$
define m'' **where** $m'' = Mapping.filter\ (\lambda\ y\ qs\ .\ card\ qs = 1)\ m'$
define m''' **where** $m''' = Mapping.map\text{-}values\ (\lambda\ \text{-}\ qs\ .\ the\text{-}elem\ qs)\ m''$

have $Mapping.lookup\ (h\text{-}obs\text{-}impl\text{-}from\text{-}h\ h')\ (q,x) = Some\ m'''$
unfolding $m'''\text{-}def\ m''\text{-}def\ m'\text{-}def\ h\text{-}obs\text{-}impl\text{-}from\text{-}h.simps\ Let\text{-}def$
unfolding $Mapping.lookup\text{-}map\text{-}values\ Some$
by $auto$

have $Mapping.lookup\ m' = set\text{-}as\text{-}map\ yqs$
using $set\text{-}as\text{-}mapping\text{-}ob\ m'\text{-}def$
by $auto$

have $*$:$(snd\ `\ Set.filter\ (\lambda(y',\ q').\ y' = y)\ (case\ Some\ yqs\ of\ None \Rightarrow \{\} \mid Some\ ts \Rightarrow ts)) = \{z.\ (y,\ z) \in yqs\}$
by $force$

have $\bigwedge\ qs\ .\ Mapping.lookup\ m''\ y = Some\ qs \longleftrightarrow qs = \{z.\ (y,\ z) \in yqs\} \wedge card\ \{z.\ (y,\ z) \in yqs\} = 1$
unfolding $m''\text{-}def\ Mapping.lookup\text{-}filter$
unfolding ‹$Mapping.lookup\ m' = set\text{-}as\text{-}map\ yqs$› $set\text{-}as\text{-}map\text{-}def$
by $auto$
then have $**$:$\bigwedge\ q'\ .\ Mapping.lookup\ m'''\ y = Some\ q' \longleftrightarrow card\ \{z.\ (y,\ z) \in yqs\} = 1 \wedge q' = the\text{-}elem\ \{z.\ (y,\ z) \in yqs\}$
unfolding $m'''\text{-}def\ lookup\text{-}map\text{-}values$ **by** $auto$
then show $?thesis$
unfolding $h\text{-}obs\text{-}impl.simps\ Let\text{-}def$
unfolding ‹$Mapping.lookup\ (h\text{-}obs\text{-}impl\text{-}from\text{-}h\ h')\ (q,x) = Some\ m'''$›
using $*\ Some$ **by** $force$
qed

definition $set\text{-}as\text{-}mapping\text{-}image :: ('a1 \times 'a2)\ set \Rightarrow (('a1 \times 'a2) \Rightarrow ('b1 \times 'b2)) \Rightarrow ('b1,\ 'b2\ set)\ mapping$ **where**
$set\text{-}as\text{-}mapping\text{-}image\ s\ f = (THE\ m\ .\ Mapping.lookup\ m = set\text{-}as\text{-}map\ (image\ f\ s))$

lemma $set\text{-}as\text{-}mapping\text{-}image\text{-}ob$:
obtains m **where** $set\text{-}as\text{-}mapping\text{-}image\ s\ f = m$ **and** $Mapping.lookup\ m = set\text{-}as\text{-}map\ (image\ f\ s)$
proof −
obtain m **where** $*$:$Mapping.lookup\ m = set\text{-}as\text{-}map\ (image\ f\ s)$
using $Mapping.lookup.abs\text{-}eq$ **by** $auto$
moreover have $(THE\ x.\ Mapping.lookup\ x = set\text{-}as\text{-}map\ (image\ f\ s)) = m$

```
    using the-equality[of λm . Mapping.lookup m = set-as-map (image f s), OF *]
    unfolding *[symmetric]
    by (simp add: mapping-eqI)
  ultimately show ?thesis
    using that[of m] unfolding set-as-mapping-image-def by blast
qed

lemma set-as-mapping-image-code [code]:
  fixes t :: ('a1 ::ccompare × 'a2 :: ccompare) set-rbt
  and   f1 :: ('a1 × 'a2) ⇒ ('b1 :: ccompare × 'b2 ::ccompare)
  and   xs :: ('c1 :: ceq × 'c2 :: ceq) set-dlist
  and   f2 :: ('c1 × 'c2) ⇒ ('d1 × 'd2)
  shows set-as-mapping-image (DList-set xs) f2 = (case ID CEQ(('c1 × 'c2)) of
          Some - ⇒ (DList-Set.fold (λ kv m1 .
                    ( case f2 kv of (x,z) ⇒ (case Mapping.lookup m1 (x) of None
⇒ Mapping.update (x) {z} m1 | Some zs ⇒ Mapping.update (x) (Set.insert z zs)
m1)))
                  xs
                  Mapping.empty) |
          None   ⇒ Code.abort (STR ''set-as-map-image DList-set: ccompare =
None'')
                        (λ-. set-as-mapping-image (DList-set xs) f2))
  (is set-as-mapping-image (DList-set xs) f2 = ?C2 (DList-set xs))
    and set-as-mapping-image (RBT-set t) f1 = (case ID CCOMPARE(('a1 ×
'a2)) of
          Some - ⇒ (RBT-Set2.fold (λ kv m1 .
                    ( case f1 kv of (x,z) ⇒ (case Mapping.lookup m1 (x) of None
⇒ Mapping.update (x) {z} m1 | Some zs ⇒ Mapping.update (x) (Set.insert z zs)
m1)))
                  t
                  Mapping.empty) |
          None   ⇒ Code.abort (STR ''set-as-map-image RBT-set: ccompare =
None'')
                        (λ-. set-as-mapping-image (RBT-set t) f1))
  (is set-as-mapping-image (RBT-set t) f1 = ?C1 (RBT-set t))
proof −
  show set-as-mapping-image (RBT-set t) f1 = ?C1 (RBT-set t)

  proof (cases ID CCOMPARE(('a1 × 'a2)))
    case None
    then show ?thesis by auto
  next
    case (Some a)

    let ?f' = λ t . (RBT-Set2.fold (λ kv m1 .
                    ( case f1 kv of (x,z) ⇒ (case Mapping.lookup m1 (x) of None
⇒ Mapping.update (x) {z} m1 | Some zs ⇒ Mapping.update (x) (Set.insert z zs)
m1)))
                  t
```

Mapping.empty)

let *?f* = λ *xs* . (*fold* (λ *kv m1* . *case f1 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*))
xs Mapping.empty)
have ⋀ *xs* :: (*'a1* × *'a2*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*)
proof −
fix *xs* :: (*'a1* × *'a2*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *f1 xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

then have ∗: (*?f* (*xs*@[*xz*])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k'* . *if k'* = *k then Some v else Mapping.lookup m k'*)
by (*metis lookup-update'*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x'* . *if x'* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x'*)
unfolding ∗∗
unfolding *scheme* **by** *force*

have (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f1* ‘ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have ¬(∃ *z* . (*x*,*z*) ∈ *f1* ‘ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have (∃ *z'* . (*x*,*z'*) ∈ *f1* ‘ *set* (*xs*@[*xz*])) **and** {*z'* . (*x*,*z'*) ∈ *f1* ‘ *set* (*xs*@[*xz*])} = {*z*}
using ‹*f1 xz* = (*x*,*z*)› **by** *fastforce*+

then have *m2*: (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])} *else None*)
= (λ *x'* . *if x'* = *x then Some* {*z*} *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} *else None*) *x'*)
using ‹*f1 xz* = (*x*,*z*)› **by** *fastforce*

show *?thesis* **using** *m1 m2 snoc*
using ‹*f1 xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have $**$: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** $*$ **by** *auto*
have *scheme*: $\bigwedge$ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k'* . *if k'* = *k then Some v else Mapping.lookup m k'*)
by (*metis lookup-update'*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x'*)
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *f1* ‘ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have ($\exists$ *z'* . (*x*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])) **by** *fastforce*

have {*z'* . (*x*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} = *Some zs*
using ‹(λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} = *zs* **by** *auto*
then show *?thesis*
using ‹*f1 xz* = (*x*, *z*)› **by** *auto*
qed

have $\bigwedge$ *a* . (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} *else None*) *x'*) *a*
proof −
fix *a* **show** (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f1* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f1* ‘ *set xs*} *else None*) *x'*) *a*

using ‹{z′ . (x,z′) ∈ f1 ‘ set (xs@[xz])} = Set.insert z zs› ‹(∃ z′ . (x,z′) ∈ f1 ‘ set (xs@[xz]))› ‹f1 xz = (x, z)›
by (*cases* a = x; *auto*)
qed

then have *m2*: (λ x′ . *if* (∃ z′ . (x′,z′) ∈ f1 ‘ set (xs@[xz])) *then Some* {z′ . (x′,z′) ∈ f1 ‘ set (xs@[xz])} *else None*)
= (λ x′ . *if* x′ = x *then Some* (*Set.insert* z zs) *else* (λ x . *if* (∃ z . (x,z) ∈ f1 ‘ set xs) *then Some* {z . (x,z) ∈ f1 ‘ set xs} *else None*) x′)
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹f1 xz = (x, z)› **by** *presburger*
qed
qed
qed

then have *Mapping.lookup* (*?f′ t*) = (λ x . *if* (∃ z . (x,z) ∈ f1 ‘ set (*RBT-Set2.keys* t)) *then Some* {z . (x,z) ∈ f1 ‘ set (*RBT-Set2.keys* t)} *else None*)
unfolding *fold-conv-fold-keys* **by** *metis*
moreover have *set* (*RBT-Set2.keys* t) = (*RBT-set* t)
using *Some* **by** (*simp add*: *RBT-set-conv-keys*)
ultimately have *Mapping.lookup* (*?f′ t*) = (λ x . *if* (∃ z . (x,z) ∈ f1 ‘ (*RBT-set* t)) *then Some* {z . (x,z) ∈ f1 ‘ (*RBT-set* t)} *else None*)
by *force*

then have *Mapping.lookup* (*?f′ t*) = *set-as-map* (*image f1* (*RBT-set* t))
unfolding *set-as-map-def* **by** *blast*
then have ∗:*Mapping.lookup* (*?C1* (*RBT-set* t)) = *set-as-map* (*image f1* (*RBT-set* t))
unfolding *Some* **by** *force*

have ⋀ t′ . *Mapping.lookup* (*?C1* (*RBT-set* t)) = *Mapping.lookup* (*?C1* t′) ⟹ (*?C1* (*RBT-set* t)) = (*?C1* t′)
by (*simp add*: *Some*)
then have ∗∗: (⋀x. *Mapping.lookup* x = *set-as-map* (*image f1* (*RBT-set* t)) ⟹ x = (*?C1* (*RBT-set* t)))
by (*simp add*: ∗ *mapping-eqI*)

show *?thesis*
using *the-equality*[*of* λ m . *Mapping.lookup* m = (*set-as-map* (*image f1* (*RBT-set* t))), *OF* ∗ ∗∗]
unfolding *set-as-mapping-image-def* **by** *blast*
qed

show *set-as-mapping-image* (*DList-set xs*) *f2* = *?C2* (*DList-set xs*)

proof (*cases ID CEQ*(('*c1* × '*c2*)))
case *None*
then show *?thesis* **by** *auto*
next
case (*Some a*)

let *?f'* = λ *t* . (*DList-Set.fold* (λ *kv m1* .
(*case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*)))
t
Mapping.empty)

let *?f* = λ *xs* . (*fold* (λ *kv m1* . *case f2 kv of* (*x*,*z*) ⇒ (*case Mapping.lookup m1* (*x*) *of None* ⇒ *Mapping.update* (*x*) {*z*} *m1* | *Some zs* ⇒ *Mapping.update* (*x*) (*Set.insert z zs*) *m1*))
xs Mapping.empty)
have ∗:⋀ *xs* :: ('*c1* × '*c2*) *list* . *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f2* ‘ *set xs*} *else None*)
proof −
fix *xs* :: ('*c1* × '*c2*) *list*
show *Mapping.lookup* (*?f xs*) = (λ *x* . *if* (∃ *z* . (*x*,*z*) ∈ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) ∈ *f2* ‘ *set xs*} *else None*)
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case*
by (*simp add*: *Mapping.empty.abs-eq Mapping.lookup.abs-eq*)
next
case (*snoc xz xs*)
then obtain *x z* **where** *f2 xz* = (*x*,*z*)
by (*metis* (*mono-tags*, *opaque-lifting*) *surj-pair*)

then have ∗: (*?f* (*xs*@[*xz*])) = (*case Mapping.lookup* (*?f xs*) *x of*
None ⇒ *Mapping.update x* {*z*} (*?f xs*) |
Some zs ⇒ *Mapping.update x* (*Set.insert z zs*) (*?f xs*))
by *auto*

then show *?case* **proof** (*cases Mapping.lookup* (*?f xs*) *x*)
case *None*
then have ∗∗: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* {*z*} (*?f xs*)) **using** ∗ **by** *auto*

have *scheme*: ⋀ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k'* . *if k'* = *k then Some v else Mapping.lookup m k'*)
by (*metis lookup-update'*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x'* . *if x'* = *x then Some* {*z*} *else Mapping.lookup* (*?f xs*) *x'*)

unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x* = *None*
using *None snoc* **by** *auto*
then have $\neg$($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*)
by (*metis* (*mono-tags*, *lifting*) *option.distinct*(*1*))
then have ($\exists$ *z′* . (*x*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) **and** {*z′* . (*x*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} = {*z*}
using ‹*f2* *xz* = (*x*,*z*)› **by** *fastforce*+
then have *m2*: (λ *x′* . *if* ($\exists$ *z′* . (*x′*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) *then Some* {*z′* . (*x′*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} *else None*)
= (λ *x′* . *if x′* = *x then Some* {*z*} *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x′*)
using ‹*f2* *xz* = (*x*,*z*)› **by** *fastforce*

show *?thesis* **using** *m1 m2 snoc*
using ‹*f2 xz* = (*x*, *z*)› **by** *presburger*
next
case (*Some zs*)
then have $**$: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = *Mapping.lookup* (*Mapping.update x* (*Set.insert z zs*) (*?f xs*)) **using** $*$ **by** *auto*
have *scheme*: $\bigwedge$ *m k v* . *Mapping.lookup* (*Mapping.update k v m*) = (λ*k′* . *if k′* = *k then Some v else Mapping.lookup m k′*)
by (*metis lookup-update′*)

have *m1*: *Mapping.lookup* (*?f* (*xs*@[*xz*])) = (λ *x′* . *if x′* = *x then Some* (*Set.insert z zs*) *else Mapping.lookup* (*?f xs*) *x′*)
unfolding $**$
unfolding *scheme* **by** *force*

have (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x* = *Some zs*
using *Some snoc* **by** *auto*
then have ($\exists$ *z′* . (*x*,*z′*) $\in$ *f2* ‘ *set xs*)
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have ($\exists$ *z′* . (*x*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) **by** *fastforce*

have {*z′* . (*x*,*z′*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} = *Set.insert z zs*
proof −
have *Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} = *Some zs*
using ‹(λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x* = *Some zs*›
unfolding *case-prod-conv* **using** *option.distinct*(*2*) **by** *metis*
then have {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} = *zs* **by** *auto*
then show *?thesis*
using ‹*f2 xz* = (*x*, *z*)› **by** *auto*

qed

have $\bigwedge$ *a* . (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x'*) *a*
proof −
fix *a* **show** (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} *else None*) *a*
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x'*) *a*
using ‹{*z'* . (*x*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} = *Set.insert z zs*› ‹($\exists$ *z'* . (*x*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*]))› ‹*f2 xz* = (*x*, *z*)›
by (*cases a* = *x*; *auto*)
qed

then have *m2*: (λ *x'* . *if* ($\exists$ *z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])) *then Some* {*z'* . (*x'*,*z'*) $\in$ *f2* ‘ *set* (*xs*@[*xz*])} *else None*)
= (λ *x'* . *if x'* = *x then Some* (*Set.insert z zs*) *else* (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ *set xs*} *else None*) *x'*)
by *auto*

show *?thesis* **using** *m1 m2 snoc*
using ‹*f2 xz* = (*x*, *z*)› **by** *presburger*
qed
qed
qed

have *ID CEQ*('*c1* × '*c2*) $\neq$ *None*
using *Some* **by** *auto*
then have $**$: $\bigwedge$ *x* . *x* $\in$ *f2* ‘ *set* (*list-of-dlist xs*) = (*x* $\in$ *f2* ‘ (*DList-set xs*))
by (*simp add*: *Collect-member DList-set-def*)

have *Mapping.lookup* (*?f' xs*) = (λ *x* . *if* ($\exists$ *z* . (*x*,*z*) $\in$ *f2* ‘ (*DList-set xs*)) *then Some* {*z* . (*x*,*z*) $\in$ *f2* ‘ (*DList-set xs*)} *else None*)
using $*$[*of* (*list-of-dlist xs*)]
unfolding *DList-Set.fold.rep-eq* $**$ **.**
then have *Mapping.lookup* (*?f' xs*) = *set-as-map* (*image f2* (*DList-set xs*))
unfolding *set-as-map-def* **by** *blast*
then have $*$:*Mapping.lookup* (*?C2* (*DList-set xs*)) = *set-as-map* (*image f2* (*DList-set xs*))
unfolding *Some* **by** *force*

have $\bigwedge$ *t'* . *Mapping.lookup* (*?C2* (*DList-set xs*)) = *Mapping.lookup* (*?C2 t'*) $\Longrightarrow$ (*?C2* (*DList-set xs*)) = (*?C2 t'*)
by (*simp add*: *Some*)
then have $**$: ($\bigwedge$*x*. *Mapping.lookup x* = *set-as-map* (*image f2* (*DList-set xs*))

$\Longrightarrow x = (?C2\ (DList\text{-}set\ xs)))$
by (*simp add*: $*$ *mapping-eqI*)
then show *?thesis*
using $*$
using *set-as-mapping-image-ob* **by** *blast*
qed
qed

45.2 Impl Datatype

The following type extends *fsm-impl* with fields for *h* and *h-obs*.

datatype ('*state*, '*input*, '*output*) *fsm-with-precomputations-impl* =
FSMWPI (*initial-wpi* : '*state*)
(*states-wpi* : '*state set*)
(*inputs-wpi* : '*input set*)
(*outputs-wpi* : '*output set*)
(*transitions-wpi* : ('*state* × '*input* × '*output* × '*state*) *set*)
(*h-wpi* : (('*state* × '*input*), ('*output* × '*state*) *set*) *mapping*)
(*h-obs-wpi*: ('*state* × '*input*, ('*output*, '*state*) *mapping*) *mapping*)

fun *fsm-with-precomputations-impl-from-list* :: '*a* ⇒ ('*a* × '*b* × '*c* × '*a*) *list* ⇒ ('*a*, '*b*, '*c*) *fsm-with-precomputations-impl* **where**
fsm-with-precomputations-impl-from-list q [] = *FSMWPI q* {*q*} {} {} {} *Mapping.empty Mapping.empty* |
fsm-with-precomputations-impl-from-list q (*t*#*ts*) = (*let ts'* = *set* (*t*#*ts*)
in FSMWPI (*t-source t*)
((*image t-source ts'*) ∪ (*image t-target ts'*))
(*image t-input ts'*)
(*image t-output ts'*)
(*ts'*)
(*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)))
(*h-obs-impl-from-h* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)))))

fun *fsm-with-precomputations-impl-from-list'* :: '*a* ⇒ ('*a* × '*b* × '*c* × '*a*) *list* ⇒ ('*a*, '*b*, '*c*) *fsm-with-precomputations-impl* **where**
fsm-with-precomputations-impl-from-list' q [] = *FSMWPI q* {*q*} {} {} {} *Mapping.empty Mapping.empty* |
fsm-with-precomputations-impl-from-list' q (*t*#*ts*) = (*let tsr* = (*remdups* (*t*#*ts*));
h' = (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) *tsr*))
in FSMWPI (*t-source t*)
(*set* (*remdups* ((*map t-source tsr*) @ (*map t-target tsr*))))
(*set* (*remdups* (*map t-input tsr*)))
(*set* (*remdups* (*map t-output tsr*)))
(*set tsr*)
h'

(*h-obs-impl-from-h h'*))

lemma *fsm-impl-from-list-code*[*code*] :
fsm-with-precomputations-impl-from-list q ts = *fsm-with-precomputations-impl-from-list'* *q ts*
proof (*cases ts*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*Cons t ts*)
have ∗∗: *set* (*t*#*ts*) = *set* (*remdups* (*t*#*ts*))
by *auto*
have ∗: *set* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)) = *set* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*remdups* (*t*#*ts*)))
by (*metis remdups-map-remdups set-remdups*)
have *Mapping.lookup* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)))
= *Mapping.lookup* (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*remdups* (*t*#*ts*))))
unfolding *list-as-mapping-lookup* ∗ **by** *simp*
then have ∗∗∗: *list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*t*#*ts*)) = *list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*remdups* (*t*#*ts*)))
by (*simp add*: *mapping-eqI*)

have ∗∗∗∗: (*set* (*map t-source* (*remdups* (*t* # *ts*)) @ *map t-target* (*remdups* (*t* # *ts*)))) = (*t-source* ‘ *set* (*t* # *ts*) ∪ *t-target* ‘ *set* (*t* # *ts*))
by *auto*

have ∗∗∗∗∗: ⋀ *f xs* . *set* (*map f* (*remdups xs*)) = *f* ‘ *set xs*
by *auto*

show *?thesis*
unfolding *Cons fsm-with-precomputations-impl-from-list'.simps fsm-with-precomputations-impl-from-list.sim Let-def*
unfolding ∗∗ ∗∗∗
unfolding *set-remdups* ∗∗∗∗ ∗∗∗∗∗
unfolding *remdups-map-remdups*
by *presburger*
qed

45.3 Refined Datatype

Well-formedness now also encompasses the new fields for *h* and *h-obs*.

fun *well-formed-fsm-with-precomputations* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations-impl* ⇒ *bool* **where**
well-formed-fsm-with-precomputations M = (*initial-wpi M* ∈ *states-wpi M*
∧ *finite* (*states-wpi M*)
∧ *finite* (*inputs-wpi M*)

$\wedge$ *finite* (*outputs-wpi M*)
$\wedge$ *finite* (*transitions-wpi M*)
$\wedge$ ($\forall$ *t* $\in$ *transitions-wpi M* . *t-source t* $\in$ *states-wpi M* $\wedge$
t-input t $\in$ *inputs-wpi M* $\wedge$
t-target t $\in$ *states-wpi M* $\wedge$
t-output t $\in$ *outputs-wpi M*)
$\wedge$ ($\forall$ *q x* . (*case* (*Mapping.lookup* (*h-wpi M*) (*q*,*x*)) *of Some ts* $\Rightarrow$ *ts* | *None* $\Rightarrow$ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) $\in$ *transitions-wpi M* })
$\wedge$ ($\forall$ *q x y* . *h-obs-impl* (*h-wpi M*) *q x y* = *h-obs-lookup* (*h-obs-wpi M*) *q x y*))

lemma *well-formed-h-set-as-mapping* :
assumes *h-wpi M* = *set-as-mapping-image* (*transitions-wpi M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
shows (*case* (*Mapping.lookup* (*h-wpi M*) (*q*,*x*)) *of Some ts* $\Rightarrow$ *ts* | *None* $\Rightarrow$ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) $\in$ *transitions-wpi M* }
(**is** *?A q x* = *?B q x*)
proof −
have $*$:*Mapping.lookup* (*h-wpi M*) = (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*)) (*transitions-wpi M*)))
unfolding *assms* **using** *set-as-mapping-image-ob*
by *auto*
have $**$: (*case Mapping.lookup* (*h-wpi M*) (*q*, *x*) *of None* $\Rightarrow$ {} | *Some ts* $\Rightarrow$ *ts*) = {*a*. *case a of* (*y*, *q'*) $\Rightarrow$ (*q*, *x*, *y*, *q'*) $\in$ (*transitions-wpi M*)}
unfolding $*$
unfolding *set-as-map-def* **by** *force*

show *?thesis*
unfolding $**$ **by** *force*
qed

lemma *well-formed-h-obs-impl-from-h* :
assumes *h-obs-wpi M* = *h-obs-impl-from-h* (*h-wpi M*)
shows *h-obs-impl* (*h-wpi M*) *q x y* = (*h-obs-lookup* (*h-obs-wpi M*) *q x y*)
unfolding *assms h-obs-impl-from-h-invar* **by** *presburger*

typedef (*'state*, *'input*, *'output*) *fsm-with-precomputations* =
{ *M* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations-impl* . *well-formed-fsm-with-precomputations M*}
morphisms *fsm-with-precomputations-impl-of-fsm-with-precomputations Abs-fsm-with-precomputations*
proof −
obtain *q* :: *'state* **where** *True* **by** *blast*
define *M* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations-impl* **where**
M: *M* = *FSMWPI q* {*q*} {} {} {} *Mapping.empty Mapping.empty*

have ($\bigwedge$ *q x* . (*case* (*Mapping.lookup* (*h-wpi M*) (*q*,*x*)) *of Some ts* $\Rightarrow$ *ts* | *None* $\Rightarrow$ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) $\in$ *transitions-wpi M* })
proof −
fix *q x*

have { (y,q') . $(q,x,y,q') \in$ *transitions-wpi M* } = {}
unfolding *M* **by** *auto*
moreover have (*case* (*Mapping.lookup* (*h-wpi M*) (q,x)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = {}
unfolding *M* **by** (*metis fsm-with-precomputations-impl.sel*(*6*) *lookup-default-def lookup-default-empty*)
ultimately show (*case* (*Mapping.lookup* (*h-wpi M*) (q,x)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (y,q') . $(q,x,y,q') \in$ *transitions-wpi M* }
by *blast*
qed
moreover have (∀ *q x y* . *h-obs-impl* (*h-wpi M*) *q x y* = (*h-obs-lookup* (*h-obs-wpi M*) *q x y*))
unfolding *h-obs-impl.simps Let-def*
unfolding *calculation M*
by (*simp add*: *Mapping.empty-def Mapping.lookup.abs-eq*)
ultimately have *well-formed-fsm-with-precomputations M*
unfolding *M* **by** *auto*
then show *?thesis*
by *blast*
qed

setup-lifting *type-definition-fsm-with-precomputations*

lift-definition *initial-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'state* **is** *FSM-Code-Datatype.initial-wpi* **done**
lift-definition *states-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'state set* **is** *FSM-Code-Datatype.states-wpi* **done**
lift-definition *inputs-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'input set* **is** *FSM-Code-Datatype.inputs-wpi* **done**
lift-definition *outputs-wp* :: (*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ *'output set* **is** *FSM-Code-Datatype.outputs-wpi* **done**
lift-definition *transitions-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ (*'state* × *'input* × *'output* × *'state*) *set*
is *FSM-Code-Datatype.transitions-wpi* **done**
lift-definition *h-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ ((*'state* × *'input*), (*'output* × *'state*) *set*) *mapping*
is *FSM-Code-Datatype.h-wpi* **done**
lift-definition *h-obs-wp* ::
(*'state*, *'input*, *'output*) *fsm-with-precomputations* ⇒ ((*'state* × *'input*), (*'output*, *'state*) *mapping*) *mapping*
is *FSM-Code-Datatype.h-obs-wpi* **done**

lemma *fsm-with-precomputations-initial*: *initial-wp M* ∈ *states-wp M*
by (*transfer*; *auto*)
lemma *fsm-with-precomputations-states-finite*: *finite* (*states-wp M*)

by (*transfer*; *auto*)
lemma *fsm-with-precomputations-inputs-finite*: *finite* (*inputs-wp M*)
by (*transfer*; *auto*)
lemma *fsm-with-precomputations-outputs-finite*: *finite* (*outputs-wp M*)
by (*transfer*; *auto*)
lemma *fsm-with-precomputations-transitions-finite*: *finite* (*transitions-wp M*)
by (*transfer*; *auto*)
lemma *fsm-with-precomputations-transition-props*: $t \in$ *transitions-wp M* $\Longrightarrow$ *t-source t* $\in$ *states-wp M* $\wedge$
t-input t $\in$ *inputs-wp M* $\wedge$
t-target t $\in$ *states-wp M* $\wedge$
t-output t $\in$ *outputs-wp M*
by (*transfer*; *auto*)
lemma *fsm-with-precomputations-h-prop*: (*case* (*Mapping.lookup* (*h-wp M*) (*q*,*x*)) *of Some ts* $\Rightarrow$ *ts* | *None* $\Rightarrow$ {}) = { (*y*,*q'*) . (*q*,*x*,*y*,*q'*) $\in$ *transitions-wp M* }
by (*transfer*; *auto*)

lemma *fsm-with-precomputations-h-obs-prop*: (*h-obs-lookup* (*h-obs-wp M*) *q x y*) = *h-obs-impl* (*h-wp M*) *q x y*
proof −
define *M'* **where** *M'* = *fsm-with-precomputations-impl-of-fsm-with-precomputations M*
then have *well-formed-fsm-with-precomputations M'*
by (*transfer*;*blast*)
then have ∗:*h-obs-impl* (*fsm-with-precomputations-impl.h-wpi M'*) *q x y* = (*h-obs-lookup* (*h-obs-wpi M'*) *q x y*)
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*

have ∗∗: (*h-obs-lookup* (*h-obs-wpi M'*) *q x y*) = *h-obs-impl* (*fsm-with-precomputations-impl.h-wpi M'*) *q x y*
unfolding ∗ **by** *auto*
have ∗∗∗: *h-wp M* = (*fsm-with-precomputations-impl.h-wpi M'*)
unfolding *M'-def* **apply** *transfer* **by** *presburger*
have ∗∗∗∗: *h-obs-wp M* = (*fsm-with-precomputations-impl.h-obs-wpi M'*)
unfolding *M'-def* **apply** *transfer* **by** *presburger*

show *?thesis*
using ∗∗ ∗∗∗ ∗∗∗∗ **by** *presburger*
qed

lemma *map-values-empty* : *Mapping.map-values f Mapping.empty* = *Mapping.empty*
by (*metis Mapping.keys-empty empty-iff keys-map-values mapping-eqI'*)

lift-definition *fsm-with-precomputations-from-list* :: *'a* $\Rightarrow$ (*'a* $\times$ *'b* $\times$ *'c* $\times$ *'a*) *list* $\Rightarrow$ (*'a*, *'b*, *'c*) *fsm-with-precomputations*
is *fsm-with-precomputations-impl-from-list*
proof −
fix *q* :: *'a*

fix *ts* :: $('a \times 'b \times 'c \times 'a)$ *list*

define *M* **where** *M* = *fsm-with-precomputations-impl-from-list q ts*

have *base-props*: (*initial-wpi M* ∈ *states-wpi M*
∧ *finite* (*states-wpi M*)
∧ *finite* (*inputs-wpi M*)
∧ *finite* (*outputs-wpi M*)
∧ *finite* (*transitions-wpi M*))
proof (*cases ts*)
case *Nil*
show *?thesis*
unfolding *M-def Nil fsm-with-precomputations-impl-from-list.simps* **by** *auto*
next
case (*Cons t ts'*)
show *?thesis*
unfolding *M-def Cons fsm-with-precomputations-impl-from-list.simps Let-def* **by** *force*
qed

have *transition-prop*: (∀ *t* ∈ *transitions-wpi M* . *t-source t* ∈ *states-wpi M* ∧
t-input t ∈ *inputs-wpi M* ∧
t-target t ∈ *states-wpi M* ∧
t-output t ∈ *outputs-wpi M*)
proof (*cases ts*)
case *Nil*
show *?thesis*
unfolding *M-def Nil fsm-with-precomputations-impl-from-list.simps* **by** *auto*
next
case (*Cons t ts'*)
show *?thesis*
unfolding *M-def Cons fsm-with-precomputations-impl-from-list.simps Let-def* **by** *force*
qed

have *h-prop*:⋀ *qa x* .
(*case Mapping.lookup* (*h-wpi M*) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) =
{*a*. *case a of* (*y*, *q'*) ⇒ (*qa*, *x*, *y*, *q'*) ∈ *transitions-wpi M*}
(**is** ⋀ *qa x* . *?P qa x*)
proof −
fix *qa x*
show *?P qa x* **unfolding** *M-def*
proof (*induction ts*)
case *Nil*
have (*case Mapping.lookup* (*h-wpi* (*fsm-with-precomputations-impl-from-list q* []) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) = {}
by *simp*

moreover have *transitions-wpi* (*fsm-with-precomputations-impl-from-list q* []) = {}
by *auto*
ultimately show *?case*
by *blast*
next
case (*Cons t ts*)
have ∗: (*h-wpi* (*fsm-with-precomputations-impl-from-list q* (*t*#*ts*))) = (*list-as-mapping* (*map* (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′)) (*t*#*ts*)))
unfolding *fsm-with-precomputations-impl-from-list.simps Let-def* **by** *simp*
show *?case* **proof** (*cases* ∃ *z*. ((*qa*, *x*), *z*) ∈ *set* (*map* (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*, *q*′)) (*t* # *ts*)))
case *True*
then have (*case Mapping.lookup* (*h-wpi* (*fsm-with-precomputations-impl-from-list q* (*t*#*ts*))) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) = {*z*. ((*qa*, *x*), *z*) ∈ *set* (*map* (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*, *q*′)) (*t* # *ts*))}
unfolding ∗ *list-as-mapping-lookup* **by** *auto*
also have . . . = {*a*. *case a of* (*y*, *q*′) ⇒ (*qa*, *x*, *y*, *q*′) ∈ *transitions-wpi* (*fsm-with-precomputations-impl-from-list q* (*t*#*ts*))}
unfolding *fsm-with-precomputations-impl-from-list.simps Let-def*
by (*induction ts*; *cases t*; *auto*)
finally show *?thesis* **.**
next
case *False*
then have (*case Mapping.lookup* (*h-wpi* (*fsm-with-precomputations-impl-from-list q* (*t*#*ts*))) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) = {}
unfolding ∗ *list-as-mapping-lookup* **by** *auto*
also have . . . = {*a*. *case a of* (*y*, *q*′) ⇒ (*qa*, *x*, *y*, *q*′) ∈ *transitions-wpi* (*fsm-with-precomputations-impl-from-list q* (*t*#*ts*))}
using *False* **unfolding** *fsm-with-precomputations-impl-from-list.simps Let-def*
by (*induction ts*; *cases t*; *auto*)
finally show *?thesis* **.**
qed
qed
qed

have *h-obs-prop*: (∀ *q x y* . *h-obs-impl* (*h-wpi M*) *q x y* = (*h-obs-lookup* (*h-obs-wpi M*) *q x y*))
proof −
have ∗∗∗:*h-obs-wpi M* = (*h-obs-impl-from-h* (*h-wpi M*))
proof (*cases ts*)
case *Nil*
then have ∗:*h-wpi M* = *Mapping.empty* **and** ∗∗:*h-obs-wpi M* = *Mapping.empty*
unfolding *M-def* **by** *auto*
show *?thesis*
unfolding ∗ ∗∗ *h-obs-impl-from-h.simps map-values-empty* **by** *simp*
next

case (*Cons t ts'*)
show *?thesis*
unfolding *Cons M-def fsm-with-precomputations-impl-from-list.simps Let-def*
by *simp*
qed
then show *?thesis*
unfolding *h-obs-impl-from-h-invar*
by *simp*
qed

show *well-formed-fsm-with-precomputations* (*fsm-with-precomputations-impl-from-list q ts*)
using *base-props transition-prop h-prop h-obs-prop*
unfolding *well-formed-fsm-with-precomputations.simps M-def*[*symmetric*]
by *blast*
qed

lemma *fsm-with-precomputations-from-list-Nil-simps* :
initial-wp (*fsm-with-precomputations-from-list q* []) = *q*
states-wp (*fsm-with-precomputations-from-list q* []) = {*q*}
inputs-wp (*fsm-with-precomputations-from-list q* []) = {}
outputs-wp (*fsm-with-precomputations-from-list q* []) = {}
transitions-wp (*fsm-with-precomputations-from-list q* []) = {}
by (*transfer*; *auto*)+

lemma *fsm-with-precomputations-from-list-Cons-simps* :
initial-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*t-source t*)
states-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = ((*image t-source* (*set* (*t*#*ts*))) ∪ (*image t-target* (*set* (*t*#*ts*))))
inputs-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*image t-input* (*set* (*t*#*ts*)))
outputs-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*image t-output* (*set* (*t*#*ts*)))
transitions-wp (*fsm-with-precomputations-from-list q* (*t*#*ts*)) = (*set* (*t*#*ts*))
by (*transfer*; *auto*)+

definition *Fsm-with-precomputations* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations* **where**
Fsm-with-precomputations M = *Abs-fsm-with-precomputations* (*if well-formed-fsm-with-precomputations M then M else FSMWPI undefined* {*undefined*} {} {} {} *Mapping.empty Mapping.empty*)

lemma *fsm-with-precomputations-code-abstype* [*code abstype*] :
Fsm-with-precomputations (*fsm-with-precomputations-impl-of-fsm-with-precomputations M*) = *M*
proof −
have *well-formed-fsm-with-precomputations* (*fsm-with-precomputations-impl-of-fsm-with-precomputations M*)
using *fsm-with-precomputations-impl-of-fsm-with-precomputations*[*of M*] **by**

blast
then show *?thesis*
unfolding *Fsm-with-precomputations-def*
using *fsm-with-precomputations-impl-of-fsm-with-precomputations-inverse*[*of M*]
by *presburger*
qed

lemma *fsm-with-precomputations-impl-of-fsm-with-precomputations-code* [*code*] :
fsm-with-precomputations-impl-of-fsm-with-precomputations (*fsm-with-precomputations-from-list q ts*) = *fsm-with-precomputations-impl-from-list q ts*
by (*fact fsm-with-precomputations-from-list.rep-eq*)

definition *FSMWP* :: ('*state*, '*input*, '*output*) *fsm-with-precomputations* ⇒ ('*state*, '*input*, '*output*) *fsm-impl* **where**
FSMWP M = *FSMI* (*initial-wp M*)
(*states-wp M*)
(*inputs-wp M*)
(*outputs-wp M*)
(*transitions-wp M*)

code-datatype *FSMWP*

45.4 Lifting

lemma *fsm-impl-from-list*[*code*] :
fsm-impl-from-list q ts = *FSMWP* (*fsm-with-precomputations-from-list q ts*)
proof (*induction ts*)
case *Nil*
show *?case* **unfolding** *fsm-impl-from-list.simps FSMWP-def fsm-with-precomputations-from-list-Nil-simps*
by *simp*
next
case (*Cons t ts*)
show *?case* **unfolding** *fsm-impl-from-list.simps FSMWP-def fsm-with-precomputations-from-list-Cons-simps Let-def* **by** *simp*
qed

lemma *fsm-impl-FSMWP-initial*[*code*,*simp*] : *fsm-impl.initial* (*FSMWP M*) = *initial-wp M*
by (*simp add*: *FSMWP-def*)
lemma *fsm-impl-FSMWP-states*[*code*,*simp*] : *fsm-impl.states* (*FSMWP M*) = *states-wp M*
by (*simp add*: *FSMWP-def*)
lemma *fsm-impl-FSMWP-inputs*[*code*,*simp*] : *fsm-impl.inputs* (*FSMWP M*) = *inputs-wp M*
by (*simp add*: *FSMWP-def*)
lemma *fsm-impl-FSMWP-outputs*[*code*,*simp*] : *fsm-impl.outputs* (*FSMWP M*) = *outputs-wp M*

by (*simp add*: *FSMWP-def*)
lemma *fsm-impl-FSMWP-transitions*[*code*,*simp*] : *fsm-impl.transitions* (*FSMWP M*) = *transitions-wp M*
by (*simp add*: *FSMWP-def*)

lemma *well-formed-FSMWP*: *well-formed-fsm* (*FSMWP M*)
proof −
have ∗: *well-formed-fsm-with-precomputations* (*fsm-with-precomputations-impl-of-fsm-with-precomputations M*)
using *fsm-with-precomputations-impl-of-fsm-with-precomputations* **by** *blast*
then have (*initial-wp M* ∈ *states-wp M*
∧ *finite* (*states-wp M*)
∧ *finite* (*inputs-wp M*)
∧ *finite* (*outputs-wp M*)
∧ *finite* (*transitions-wp M*)
∧ (∀ *t* ∈ *transitions-wp M* . *t-source t* ∈ *states-wp M* ∧
t-input t ∈ *inputs-wp M* ∧
t-target t ∈ *states-wp M* ∧
t-output t ∈ *outputs-wp M*))
unfolding *well-formed-fsm-with-precomputations.simps*
by (*simp add*: *FSM-Code-Datatype.initial-wp.rep-eq FSM-Code-Datatype.inputs-wp.rep-eq FSM-Code-Datatype.outputs-wp.rep-eq FSM-Code-Datatype.states-wp.rep-eq FSM-Code-Datatype.transitions-w*
then show *?thesis*
unfolding *FSMWP-def* **by** *simp*
qed

lemma *h-with-precomputations-code* [*code*] : *FSM-Impl.h* ((*FSMWP M*)) = (λ (*q*,*x*) . *case Mapping.lookup* (*h-wp M*) (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {})
proof −
have ∗: ⋀ *q x* . (*case* (*Mapping.lookup* (*h-wp M*) (*q*,*x*)) *of Some ts* ⇒ *ts* | *None* ⇒ {}) = { (*y*,*q′*) . (*q*,*x*,*y*,*q′*) ∈ *transitions-wp M* }
by (*transfer*; *auto*)

have ∗∗: *fsm-impl.transitions* ((*FSMWP M*)) = *transitions-wp M*
by (*simp add*: *FSMWP-def*)

have ⋀ *q x* . *FSM-Impl.h* ((*FSMWP M*)) (*q*,*x*) = (λ (*q*,*x*) . *case Mapping.lookup* (*h-wp M*) (*q*,*x*) *of Some yqs* ⇒ *yqs* | *None* ⇒ {}) (*q*,*x*)
unfolding ∗ *FSM-Impl.h.simps case-prod-unfold fst-conv snd-conv* ∗∗ **by** *blast*
then show *?thesis*
by *blast*
qed

lemma *h-obs-with-precomputations-code* [*code*] : *FSM-Impl.h-obs* ((*FSMWP M*)) *q x y* = (*h-obs-lookup* (*h-obs-wp M*) *q x y*)
unfolding *fsm-with-precomputations-h-obs-prop*
unfolding *FSM-Impl.h-obs.simps*
unfolding *h-obs-impl.simps*

```
  unfolding Let-def
  unfolding FSM-Impl.h.simps[of FSMWP M q x]
  unfolding fsm-with-precomputations-h-prop[of M q x]
  by auto


fun filter-states-impl :: ('a,'b,'c) fsm-with-precomputations-impl ⇒ ('a ⇒ bool) ⇒
('a,'b,'c) fsm-with-precomputations-impl where
  filter-states-impl M P = (if P (initial-wpi M)
                        then (let
                               h' = Mapping.filter (λ (q,x) yqs . P q) (h-wpi M);
                              h'' = Mapping.map-values (λ - yqs . Set.filter (λ (y,q')
. P q') yqs) h'
                             in
                               FSMWPI (initial-wpi M)
                               (Set.filter P (states-wpi M))
                               (inputs-wpi M)
                               (outputs-wpi M)
                                    (Set.filter (λ t . P (t-source t) ∧ P (t-target t))
(transitions-wpi M))
                               h''
                               (h-obs-impl-from-h h''))
                        else M)

lift-definition filter-states :: ('a,'b,'c) fsm-with-precomputations ⇒ ('a ⇒ bool) ⇒
('a,'b,'c) fsm-with-precomputations
  is filter-states-impl
proof −
  fix M :: ('a,'b,'c) fsm-with-precomputations-impl
  fix P :: ('a ⇒ bool)

  let ?M = (filter-states-impl M P)

 show well-formed-fsm-with-precomputations M ⟹ well-formed-fsm-with-precomputations
?M
  proof −
    assume assm: well-formed-fsm-with-precomputations M
    show well-formed-fsm-with-precomputations ?M
    proof (cases P (initial-wpi M))
      case False
      then have ?M = M by auto
      then show ?thesis using assm by presburger
    next
      case True

      have initial-wpi ?M = initial-wpi M
        unfolding filter-states-impl.simps Let-def by auto
      have states-wpi ?M = Set.filter P (states-wpi M)
```

using *True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*
have *inputs-wpi ?M = inputs-wpi M*
unfolding *filter-states-impl.simps Let-def* **by** *auto*
have *outputs-wpi ?M = outputs-wpi M*
unfolding *filter-states-impl.simps Let-def* **by** *auto*
have *transitions-wpi ?M = (Set.filter (λ t . P (t-source t) ∧ P (t-target t)) (transitions-wpi M))*
using *True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*

define *h′* **where** *h′ = Mapping.filter (λ (q,x) yqs . P q) (h-wpi M)*
define *h″* **where** *h″ = Mapping.map-values (λ - yqs . Set.filter (λ (y,q′) . P q′) yqs) h′*

have *h-wpi ?M = h″*
unfolding *h″-def h′-def* **using** *True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*
then have *h-obs-wpi ?M = h-obs-impl-from-h (h-wpi ?M)*
using *True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*

have *base-props*: *(initial-wpi ?M ∈ states-wpi ?M*
∧ finite (states-wpi ?M)
∧ finite (inputs-wpi ?M)
∧ finite (outputs-wpi ?M)
∧ finite (transitions-wpi ?M))
using *assm True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*

have *transition-prop*: *(∀ t ∈ transitions-wpi ?M . t-source t ∈ states-wpi ?M ∧*
t-input t ∈ inputs-wpi ?M ∧
t-target t ∈ states-wpi ?M ∧
t-output t ∈ outputs-wpi ?M)
using *assm True* **unfolding** *filter-states-impl.simps Let-def* **by** *auto*

have *h-prop:⋀ qa x .*
(case Mapping.lookup (h-wpi ?M) (qa, x) of None ⇒ {} | Some ts ⇒ ts) =
{a. case a of (y, q′) ⇒ (qa, x, y, q′) ∈ transitions-wpi ?M}
(is *⋀ qa x . ?A qa x = ?B qa x)*
proof −
fix *q x*
show *?A q x = ?B q x*
proof *(cases P q)*
case *False*
then have *Mapping.lookup h′ (q,x) = None*
unfolding *h′-def*
unfolding *Mapping.lookup-filter case-prod-conv*
by *(metis (mono-tags) not-None-eq option.simps(4) option.simps(5))*

```
        then have ?A q x = {}
          unfolding ‹h-wpi ?M = h''› h''-def
          unfolding Mapping.lookup-map-values
          by simp
        moreover have ?B q x = {}
           unfolding ‹transitions-wpi ?M = (Set.filter (λ t . P (t-source t) ∧ P
(t-target t)) (transitions-wpi M))›
          using False by auto
        ultimately show ?thesis by blast
      next
        case True
        then have Mapping.lookup h' (q,x) = Mapping.lookup (h-wpi M) (q,x)
          unfolding h'-def
          unfolding Mapping.lookup-filter case-prod-conv
          by (cases Mapping.lookup (h-wpi M) (q, x); auto)
       have ?A q x = Set.filter (λ (y,q') . P q') (case Mapping.lookup (h-wpi M)
(q, x) of None ⇒ {} | Some ts ⇒ ts)
          unfolding ‹h-wpi ?M = h''› h''-def
          unfolding Mapping.lookup-map-values
         unfolding ‹Mapping.lookup h' (q,x) = Mapping.lookup (h-wpi M) (q,x)›
          by (cases Mapping.lookup (h-wpi M) (q, x); auto)
        also have ... = ?B q x
        proof −
          have *:(case Mapping.lookup (h-wpi M) (q, x) of None ⇒ {} | Some ts
⇒ ts) = {a. case a of (y, q') ⇒ (q, x, y, q') ∈ transitions-wpi M}
            using assm by auto
          show ?thesis
            unfolding *
            unfolding ‹transitions-wpi ?M = (Set.filter (λ t . P (t-source t) ∧ P
(t-target t)) (transitions-wpi M))›
            using True
            by auto
        qed
        finally show ?thesis .
      qed
    qed

    show ?thesis
      using base-props transition-prop h-prop well-formed-h-obs-impl-from-h[OF
‹h-obs-wpi ?M = h-obs-impl-from-h (h-wpi ?M)›]
      unfolding well-formed-fsm-with-precomputations.simps by blast
  qed
 qed
qed

lemma filter-states-simps:
  initial-wp (filter-states M P) = initial-wp M
  states-wp (filter-states M P) = (if P (initial-wp M) then Set.filter P (states-wp
M) else states-wp M)
```

inputs-wp (*filter-states M P*) = *inputs-wp M*
outputs-wp (*filter-states M P*) = *outputs-wp M*
transitions-wp (*filter-states M P*) = (*if P* (*initial-wp M*) *then* (*Set.filter* (λ *t* . *P* (*t-source t*) ∧ *P* (*t-target t*)) (*transitions-wp M*)) *else transitions-wp M*)
by (*transfer*; *simp add*: *Let-def*)+

lemma *filter-states-with-precomputations-code* [*code*] : *FSM-Impl.filter-states* ((*FSMWP M*)) *P* = *FSMWP* (*filter-states M P*)
unfolding *FSM-Impl.filter-states.simps Let-def*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
using *filter-states-simps*[*of M P*]
by (*simp add*: *FSMWP-def*)

fun *create-unconnected-fsm-from-fsets-impl* :: *'a* ⇒ *'a fset* ⇒ *'b fset* ⇒ *'c fset* ⇒ (*'a*,*'b*,*'c*) *fsm-with-precomputations-impl* **where**
create-unconnected-fsm-from-fsets-impl q ns ins outs = *FSMWPI q* (*insert q* (*fset ns*)) (*fset ins*) (*fset outs*) {} *Mapping.empty Mapping.empty*

lift-definition *create-unconnected-fsm-from-fsets* :: *'a* ⇒ *'a fset* ⇒ *'b fset* ⇒ *'c fset* ⇒ (*'a*,*'b*,*'c*) *fsm-with-precomputations*
is *create-unconnected-fsm-from-fsets-impl*
proof −
fix *q* :: *'a*
fix *ns*
fix *ins* :: *'b fset*
fix *outs* :: *'c fset*

let *?M* = (*create-unconnected-fsm-from-fsets-impl q ns ins outs*)

show *well-formed-fsm-with-precomputations* (*create-unconnected-fsm-from-fsets-impl q ns ins outs*)
proof −

have *base-props*: (*initial-wpi ?M* ∈ *states-wpi ?M*
∧ *finite* (*states-wpi ?M*)
∧ *finite* (*inputs-wpi ?M*)
∧ *finite* (*outputs-wpi ?M*)
∧ *finite* (*transitions-wpi ?M*))
by *auto*

have *transition-prop*: (∀ *t* ∈ *transitions-wpi ?M* . *t-source t* ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧
t-target t ∈ *states-wpi ?M* ∧

t-output t ∈ *outputs-wpi ?M*)
by *auto*

have ∗: (*h-wpi ?M*) = *Mapping.empty*
by *auto*
have ∗∗: *transitions-wpi ?M* = {}
by *auto*
have ∗∗∗: (*h-obs-wpi ?M*) = *Mapping.empty*
by *auto*

have *h-prop*:⋀ *qa x* .
(*case Mapping.lookup* (*h-wpi ?M*) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) =
{*a*. *case a of* (*y*, *q′*) ⇒ (*qa*, *x*, *y*, *q′*) ∈ *transitions-wpi ?M*}
unfolding ∗ ∗∗ *Mapping.lookup-empty* **by** *auto*

have *h-obs-prop*: ⋀ *q x y* . *h-obs-impl* (*h-wpi ?M*) *q x y* = *h-obs-lookup* (*h-obs-wpi ?M*) *q x y*
unfolding *h-obs-impl.simps Let-def*
unfolding ∗ ∗∗∗ *Mapping.lookup-empty*
by (*simp add*:)

show *?thesis*
using *base-props transition-prop h-prop h-obs-prop*
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed
qed

lemma *fsm-with-precomputations-impl-of-code* [*code*] :
fsm-with-precomputations-impl-of-fsm-with-precomputations (*create-unconnected-fsm-from-fsets q ns ins outs*) = *create-unconnected-fsm-from-fsets-impl q ns ins outs*
by (*fact create-unconnected-fsm-from-fsets.rep-eq*)

lemma *create-unconnected-fsm-from-fsets-simps*:
initial-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = *q*
states-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = (*insert q* (*fset ns*))
inputs-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset ins*
outputs-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = *fset outs*
transitions-wp (*create-unconnected-fsm-from-fsets q ns ins outs*) = {}
by (*transfer*; *simp add*: *Let-def*)+

lemma *create-unconnected-fsm-with-precomputations-code* [*code*] : *FSM-Impl.create-unconnected-fsm-from-fset q ns ins outs* = *FSMWP* (*create-unconnected-fsm-from-fsets q ns ins outs*)
unfolding *FSM-Impl.create-unconnected-fsm-from-fsets.simps*
unfolding *FSMWP-def*
unfolding *create-unconnected-fsm-from-fsets-simps*
by *presburger*

fun *add-transitions-impl* :: (*′a*,*′b*,*′c*) *fsm-with-precomputations-impl* ⇒ (*′a* × *′b* ×

$'c \times 'a)$ *set* $\Rightarrow$ $('a,'b,'c)$ *fsm-with-precomputations-impl* **where**
add-transitions-impl M ts = (*if* ($\forall$ *t* $\in$ *ts* . *t-source t* $\in$ *states-wpi M* $\wedge$ *t-input t* $\in$ *inputs-wpi M* $\wedge$ *t-output t* $\in$ *outputs-wpi M* $\wedge$ *t-target t* $\in$ *states-wpi M*)
then (*let ts'* = ((*transitions-wpi M*) $\cup$ *ts*);
h' = *set-as-mapping-image ts'* (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))
in FSMWPI
(*initial-wpi M*)
(*states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)
ts'
h'
(*h-obs-impl-from-h h'*))
else M)

lift-definition *add-transitions* :: $('a,'b,'c)$ *fsm-with-precomputations* $\Rightarrow$ $('a \times 'b \times 'c \times 'a)$ *set* $\Rightarrow$ $('a,'b,'c)$ *fsm-with-precomputations*
is *add-transitions-impl*
proof −
fix *M* :: $('a,'b,'c)$ *fsm-with-precomputations-impl*
fix *ts*

let *?M* = (*add-transitions-impl M ts*)

show *well-formed-fsm-with-precomputations M* $\Longrightarrow$ *well-formed-fsm-with-precomputations ?M*
proof −
assume *assm*: *well-formed-fsm-with-precomputations M*

show *well-formed-fsm-with-precomputations ?M*
proof (*cases* ($\forall$ *t* $\in$ *ts* . *t-source t* $\in$ *states-wpi M* $\wedge$ *t-input t* $\in$ *inputs-wpi M* $\wedge$ *t-output t* $\in$ *outputs-wpi M* $\wedge$ *t-target t* $\in$ *states-wpi M*))
case *False*
then have *?M* = *M* **by** *auto*
then show *?thesis* **using** *assm* **by** *presburger*
next
case *True*
then have *ts* $\subseteq$ *states-wpi M* $\times$ *inputs-wpi M* $\times$ *outputs-wpi M* $\times$ *states-wpi M*
by *fastforce*
moreover have *finite* (*states-wpi M* $\times$ *inputs-wpi M* $\times$ *outputs-wpi M* $\times$ *states-wpi M*)
using *assm* **unfolding** *well-formed-fsm-with-precomputations.simps* **by** *blast*
ultimately have *finite ts*
using *rev-finite-subset* **by** *auto*

have *initial-wpi ?M* = *initial-wpi M*

unfolding *add-transitions-impl.simps Let-def* **by** *auto*
have *states-wpi ?M = states-wpi M*
unfolding *add-transitions-impl.simps Let-def* **by** *auto*
have *inputs-wpi ?M = inputs-wpi M*
unfolding *add-transitions-impl.simps Let-def* **by** *auto*
have *outputs-wpi ?M = outputs-wpi M*
unfolding *add-transitions-impl.simps Let-def* **by** *auto*
have *transitions-wpi ?M* = (*transitions-wpi M*) ∪ *ts*
using *True* **unfolding** *add-transitions-impl.simps Let-def* **by** *auto*

define *ts′* **where** *ts′* = ((*transitions-wpi M*) ∪ *ts*)
define *h′* **where** *h′* = *set-as-mapping-image ts′* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))

have *h-wpi ?M = set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
unfolding *h′-def ts′-def* **using** *True* **unfolding** *add-transitions-impl.simps Let-def* **by** *auto*
have *h-obs-wpi ?M = h-obs-impl-from-h* (*h-wpi ?M*)
unfolding *h′-def ts′-def* **using** *True* **unfolding** *add-transitions-impl.simps Let-def* **by** *auto*

have *base-props*: (*initial-wpi ?M* ∈ *states-wpi ?M*
∧ *finite* (*states-wpi ?M*)
∧ *finite* (*inputs-wpi ?M*)
∧ *finite* (*outputs-wpi ?M*)
∧ *finite* (*transitions-wpi ?M*))
using *assm True* ‹*finite ts*› **unfolding** *add-transitions-impl.simps Let-def* **by** *auto*

have *transition-prop*: (∀ *t* ∈ *transitions-wpi ?M* . *t-source t* ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧
t-target t ∈ *states-wpi ?M* ∧
t-output t ∈ *outputs-wpi ?M*)
using *assm True* **unfolding** *add-transitions-impl.simps Let-def* **by** *auto*

have *h-prop*:⋀ *qa x* .
(*case Mapping.lookup* (*h-wpi ?M*) (*qa*, *x*) *of None* ⇒ {} | *Some ts* ⇒ *ts*) =
{*a*. *case a of* (*y*, *q′*) ⇒ (*qa*, *x*, *y*, *q′*) ∈ *transitions-wpi ?M*}
(**is** ⋀ *qa x* . *?A qa x* = *?B qa x*)
proof −
fix *q x*
show *?A q x* = *?B q x*
proof −

have ∗:*Mapping.lookup* (*h-wpi ?M*) = (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) .

((q,x),y,q′)) (transitions-wpi ?M)))
unfolding ‹h-wpi ?M = set-as-mapping-image (transitions-wpi ?M) (λ(q,x,y,q′) . ((q,x),y,q′))› **using** set-as-mapping-image-ob
by auto

have ∗∗: ⋀ z . ((q, x), z) ∈ (λ(q, x, y, q′). ((q, x), y, q′)) ‘ (transitions-wpi ?M) = ((q,x,z) ∈ (transitions-wpi ?M))
by force

show ?thesis
unfolding ∗ set-as-map-def ∗∗ **by** force
qed
qed

show ?thesis
using base-props transition-prop h-prop well-formed-h-obs-impl-from-h[OF ‹h-obs-wpi ?M = h-obs-impl-from-h (h-wpi ?M)›]
unfolding well-formed-fsm-with-precomputations.simps **by** blast
qed
qed
qed

lemma add-transitions-simps:
initial-wp (add-transitions M ts) = initial-wp M
states-wp (add-transitions M ts) = states-wp M
inputs-wp (add-transitions M ts) = inputs-wp M
outputs-wp (add-transitions M ts) = outputs-wp M
transitions-wp (add-transitions M ts) = (if (∀ t ∈ ts . t-source t ∈ states-wp M ∧ t-input t ∈ inputs-wp M ∧ t-output t ∈ outputs-wp M ∧ t-target t ∈ states-wp M)
then transitions-wp M ∪ ts else transitions-wp M)
by (transfer; simp add: Let-def)+

lemma add-transitions-with-precomputations-code [code] : FSM-Impl.add-transitions ((FSMWP M)) ts = FSMWP (add-transitions M ts)
unfolding FSM-Impl.add-transitions.simps
unfolding fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions
unfolding FSMWP-def
unfolding add-transitions-simps
by presburger

fun rename-states-impl :: (′a,′b,′c) fsm-with-precomputations-impl ⇒ (′a ⇒ ′d) ⇒ (′d,′b,′c) fsm-with-precomputations-impl **where**
rename-states-impl M f = (let ts = ((λt . (f (t-source t), t-input t, t-output t, f (t-target t))) ‘ transitions-wpi M);
h′ = set-as-mapping-image ts (λ(q,x,y,q′) . ((q,x),y,q′))

in
FSMWPI (*f* (*initial-wpi M*))
(*f* ‘ *states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)
ts
h′
(*h-obs-impl-from-h h′*))

lift-definition *rename-states* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations* ⇒(′*a* ⇒ ′*d*) ⇒ (′*d*,′*b*,′*c*) *fsm-with-precomputations*
is *rename-states-impl*
proof −
fix *M* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl*
fix *f* :: (′*a* ⇒ ′*d*)

let *?M* = (*rename-states-impl M f*)

show *well-formed-fsm-with-precomputations M* ⟹ *well-formed-fsm-with-precomputations ?M*
proof −
assume *assm*: *well-formed-fsm-with-precomputations M*

show *well-formed-fsm-with-precomputations ?M*
proof −

have *initial-wpi ?M* = *f* (*initial-wpi M*)
unfolding *rename-states-impl.simps Let-def* **by** *auto*
have *states-wpi ?M* = *f* ‘ *states-wpi M*
unfolding *rename-states-impl.simps Let-def* **by** *auto*
have *inputs-wpi ?M* = *inputs-wpi M*
unfolding *rename-states-impl.simps Let-def* **by** *auto*
have *outputs-wpi ?M* = *outputs-wpi M*
unfolding *rename-states-impl.simps Let-def* **by** *auto*
have *transitions-wpi ?M* = ((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions-wpi M*)
unfolding *rename-states-impl.simps Let-def* **by** *auto*

define *ts* **where** *ts* = ((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions-wpi M*)
define *h′* **where** *h′* = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))

have *transitions-wpi ?M* = *ts*
unfolding *ts-def rename-states-impl.simps Let-def* **by** *auto*
then have *h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
unfolding *h′-def* **unfolding** *rename-states-impl.simps Let-def* **by** *auto*
then have *h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)
unfolding *rename-states-impl.simps Let-def* **by** *auto*

have *base-props*: (*initial-wpi ?M* ∈ *states-wpi ?M*
∧ *finite* (*states-wpi ?M*)
∧ *finite* (*inputs-wpi ?M*)
∧ *finite* (*outputs-wpi ?M*)
∧ *finite* (*transitions-wpi ?M*))
using *assm* **unfolding** *rename-states-impl.simps Let-def* **by** *auto*

have *transition-prop*: (∀ *t* ∈ *transitions-wpi ?M* . *t-source t* ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧
t-target t ∈ *states-wpi ?M* ∧
t-output t ∈ *outputs-wpi ?M*)
using *assm* **unfolding** *rename-states-impl.simps Let-def* **by** *auto*

show *?thesis*
using *base-props transition-prop*
well-formed-h-set-as-mapping[*OF* ‹*h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′))›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed
qed
qed

lemma *rename-states-simps*:
initial-wp (*rename-states M f*) = *f* (*initial-wp M*)
states-wp (*rename-states M f*) = *f* ‘ *states-wp M*
inputs-wp (*rename-states M f*) = *inputs-wp M*
outputs-wp (*rename-states M f*) = *outputs-wp M*
transitions-wp (*rename-states M f*) = ((λ*t* . (*f* (*t-source t*), *t-input t*, *t-output t*, *f* (*t-target t*))) ‘ *transitions-wp M*)
by (*transfer*; *simp add*: *Let-def*)+

lemma *rename-states-with-precomputations-code*[*code*] : *FSM-Impl.rename-states* ((*FSMWP M*)) *f* = *FSMWP* (*rename-states M f*)
unfolding *FSM-Impl.rename-states.simps*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *FSMWP-def*
unfolding *rename-states-simps*
by *presburger*

fun *filter-transitions-impl* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ ((′*a* × ′*b*

$\times$ $'c$ $\times$ $'a$) $\Rightarrow$ *bool*) $\Rightarrow$ ($'a$,$'b$,$'c$) *fsm-with-precomputations-impl* **where**
filter-transitions-impl M P = (*let ts* = (*Set.filter P* (*transitions-wpi M*));
h' = (*set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q'*) .
((*q*,*x*),*y*,*q'*)))
in FSMWPI (*initial-wpi M*)
(*states-wpi M*)
(*inputs-wpi M*)
(*outputs-wpi M*)
ts
h'
(*h-obs-impl-from-h h'*))

lift-definition *filter-transitions* :: ($'a$,$'b$,$'c$) *fsm-with-precomputations* $\Rightarrow$ (($'a$ $\times$ $'b$
$\times$ $'c$ $\times$ $'a$) $\Rightarrow$ *bool*) $\Rightarrow$ ($'a$,$'b$,$'c$) *fsm-with-precomputations*
is *filter-transitions-impl*
proof −
fix *M* :: ($'a$,$'b$,$'c$) *fsm-with-precomputations-impl*
fix *P* :: (($'a$ $\times$ $'b$ $\times$ $'c$ $\times$ $'a$) $\Rightarrow$ *bool*)

let *?M* = *filter-transitions-impl M P*

show *well-formed-fsm-with-precomputations M* $\Longrightarrow$ *well-formed-fsm-with-precomputations*
?M
proof −
assume *assm*: *well-formed-fsm-with-precomputations M*

show *well-formed-fsm-with-precomputations ?M*
proof −

have *initial-wpi ?M* = *initial-wpi M*
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
have *states-wpi ?M* = *states-wpi M*
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
have *inputs-wpi ?M* = *inputs-wpi M*
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
have *outputs-wpi ?M* = *outputs-wpi M*
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
have *transitions-wpi ?M* = (*Set.filter P* (*transitions-wpi M*))
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*

have *h-wpi ?M* = (*set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q'*) .
((*q*,*x*),*y*,*q'*)))
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
then have *h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*

have *base-props*: (*initial-wpi ?M* $\in$ *states-wpi ?M*
$\wedge$ *finite* (*states-wpi ?M*)
$\wedge$ *finite* (*inputs-wpi ?M*)

$\wedge$ *finite* (*outputs-wpi ?M*)
$\wedge$ *finite* (*transitions-wpi ?M*))
using *assm* **unfolding** *filter-transitions-impl.simps Let-def* **by** *auto*

have *transition-prop*: ($\forall$ *t* $\in$ *transitions-wpi ?M* . *t-source t* $\in$ *states-wpi ?M* $\wedge$
t-input t $\in$ *inputs-wpi ?M* $\wedge$
t-target t $\in$ *states-wpi ?M* $\wedge$
t-output t $\in$ *outputs-wpi ?M*)
using *assm* **unfolding** *filter-transitions-impl.simps Let-def* **by** *auto*

show *?thesis*
using *base-props transition-prop*
well-formed-h-set-as-mapping[*OF* ‹*h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q'*) . ((*q*,*x*),*y*,*q'*))›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed
qed
qed

lemma *filter-transitions-simps*:
initial-wp (*filter-transitions M P*) = *initial-wp M*
states-wp (*filter-transitions M P*) = *states-wp M*
inputs-wp (*filter-transitions M P*) = *inputs-wp M*
outputs-wp (*filter-transitions M P*) = *outputs-wp M*
transitions-wp (*filter-transitions M P*) = *Set.filter P* (*transitions-wp M*)
by (*transfer*; *simp add*: *Let-def*)+

lemma *filter-transitions-with-precomputations-code* [*code*] : *FSM-Impl.filter-transitions* ((*FSMWP M*)) *P* = *FSMWP* (*filter-transitions M P*)
unfolding *FSM-Impl.filter-transitions.simps*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *FSMWP-def*
unfolding *filter-transitions-simps*
by *presburger*

fun *initial-singleton-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* $\Rightarrow$ ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
initial-singleton-impl M = *FSMWPI* (*initial-wpi M*)
{*initial-wpi M*}
(*inputs-wpi M*)
(*outputs-wpi M*)
{}

Mapping.empty
Mapping.empty

lemma *set-as-mapping-empty* :
set-as-mapping-image {} *f* = *Mapping.empty*
proof −
obtain *m* **where** *set-as-mapping-image* {} *f* = *m* **and** *Mapping.lookup m* = *set-as-map* (*f* ‘ {})
using *set-as-mapping-image-ob* **by** *blast*
then have ⋀ *k* . *Mapping.lookup m k* = *None*
unfolding *set-as-map-def*
by *simp*
then show *?thesis*
unfolding ‹*set-as-mapping-image* {} *f* = *m*›
by (*simp add*: *mapping-eqI*)
qed

lemma *h-obs-from-impl-h* : *h-obs-impl-from-h Mapping.empty* = *Mapping.empty*
unfolding *h-obs-impl-from-h.simps*
by (*simp add*: *map-values-empty*)

lift-definition *initial-singleton* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations* ⇒ (′*a*,′*b*,′*c*) *fsm-with-precomputations*
is *initial-singleton-impl*
proof −
fix *M* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl*

let *?M* = *initial-singleton-impl M*

show *well-formed-fsm-with-precomputations M* ⟹ *well-formed-fsm-with-precomputations ?M*
proof −
assume *assm*: *well-formed-fsm-with-precomputations M*

show *well-formed-fsm-with-precomputations ?M*
proof −

have *transitions-wpi ?M* = {}
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*

have *h-wpi ?M* = *Mapping.empty* **and** *h-obs-wpi ?M* = *Mapping.empty*
unfolding *filter-transitions-impl.simps Let-def* **by** *auto*
have *h-wpi ?M* = (*set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′)))
unfolding ‹*h-wpi ?M* = *Mapping.empty*› ‹*transitions-wpi ?M* = {}›
unfolding *set-as-mapping-empty* **by** *presburger*
have *h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)
unfolding ‹*h-wpi ?M* = *Mapping.empty*› ‹*h-obs-wpi ?M* = *Mapping.empty*›
unfolding *h-obs-from-impl-h* **by** *simp*

have *base-props*: (*initial-wpi ?M* ∈ *states-wpi ?M*
∧ *finite* (*states-wpi ?M*)
∧ *finite* (*inputs-wpi ?M*)
∧ *finite* (*outputs-wpi ?M*)
∧ *finite* (*transitions-wpi ?M*))
using *assm* **unfolding** *filter-transitions-impl.simps Let-def* **by** *auto*

have *transition-prop*: (∀ *t* ∈ *transitions-wpi ?M* . *t-source t* ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧
t-target t ∈ *states-wpi ?M* ∧
t-output t ∈ *outputs-wpi ?M*)
using *assm* **unfolding** *filter-transitions-impl.simps Let-def* **by** *auto*

show *?thesis*
using *base-props transition-prop*
well-formed-h-set-as-mapping[*OF* ‹*h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′))›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed
qed
qed

lemma *initial-singleton-simps*:
initial-wp (*initial-singleton M*) = *initial-wp M*
states-wp (*initial-singleton M*) = {*initial-wp M*}
inputs-wp (*initial-singleton M*) = *inputs-wp M*
outputs-wp (*initial-singleton M*) = *outputs-wp M*
transitions-wp (*initial-singleton M*) = {}
by (*transfer*; *simp add*: *Let-def*)+

lemma *initial-singleton-with-precomputations-code*[*code*] : *FSM-Impl.initial-singleton* ((*FSMWP M*)) = *FSMWP* (*initial-singleton M*)
unfolding *FSM-Impl.initial-singleton.simps*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *FSMWP-def*
unfolding *initial-singleton-simps*
by *presburger*

fun *canonical-separator′-impl* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ ((′*a* × ′*a*),′*b*,′*c*) *fsm-with-precomputations-impl* ⇒ ′*a* ⇒ ′*a* ⇒ ((′*a* × ′*a*) + ′*a*,′*b*,′*c*) *fsm-with-precomputations-impl* **where**

canonical-separator′-impl M P q1 q2 = (*if initial-wpi P* = (*q1*,*q2*)
then
(*let f′* = *set-as-map* (*image* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*)) (*transitions-wpi M*));
f = (λ*qx* . (*case f′ qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions′ = *shifted-transitions* (*transitions-wpi P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states-wpi P*) (*inputs-wpi P*);
ts = *shifted-transitions′* ∪ *distinguishing-transitions-lr*;
h′ = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
in

FSMWPI (*Inl* (*q1*,*q2*))
((*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*})
(*inputs-wpi M* ∪ *inputs-wpi P*)
(*outputs-wpi M* ∪ *outputs-wpi P*)
ts
h′
(*h-obs-impl-from-h h′*))
else FSMWPI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {} *Mapping.empty Mapping.empty*)

lemma *canonical-separator′-impl-refined*[*code*]:
canonical-separator′-impl M P q1 q2 = (*if initial-wpi P* = (*q1*,*q2*)
then
(*let f′* = *set-as-mapping-image* (*transitions-wpi M*) (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*));
f = (λ*qx* . (*case Mapping.lookup f′ qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}));
shifted-transitions′ = *shifted-transitions* (*transitions-wpi P*);
distinguishing-transitions-lr = *distinguishing-transitions f q1 q2* (*states-wpi P*) (*inputs-wpi P*);
ts = *shifted-transitions′* ∪ *distinguishing-transitions-lr*;
h′ = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
in

FSMWPI (*Inl* (*q1*,*q2*))
((*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*})
(*inputs-wpi M* ∪ *inputs-wpi P*)
(*outputs-wpi M* ∪ *outputs-wpi P*)
ts
h′
(*h-obs-impl-from-h h′*))
else FSMWPI (*Inl* (*q1*,*q2*)) {*Inl* (*q1*,*q2*)} {} {} {} *Mapping.empty Mapping.empty*)
unfolding *canonical-separator′-impl.simps*
using *set-as-mapping-image-ob*[*of* (*transitions-wpi M*) (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*))]
by *fastforce*

lift-definition *canonical-separator′* :: (*′a*,*′b*,*′c*) *fsm-with-precomputations* ⇒ ((*′a* × *′a*),*′b*,*′c*) *fsm-with-precomputations* ⇒ *′a* ⇒ *′a* ⇒ ((*′a* × *′a*) + *′a*,*′b*,*′c*) *fsm-with-precomputations*
is *canonical-separator′-impl*
proof −

fix *M* :: (′*a*,′*b*,′*c*) *fsm-with-precomputations-impl*
fix *P q1 q2*
show *well-formed-fsm-with-precomputations M* ⟹ *well-formed-fsm-with-precomputations P* ⟹ *well-formed-fsm-with-precomputations* (*canonical-separator′-impl M P q1 q2*)
proof −
assume *a1*: *well-formed-fsm-with-precomputations M*
assume *a2*: *well-formed-fsm-with-precomputations P*

let *?M* = *canonical-separator′-impl M P q1 q2*

show *well-formed-fsm-with-precomputations ?M*
proof (*cases initial-wpi P* = (*q1*,*q2*))
case *False*

have *h-wpi ?M* = *Mapping.empty* **and** *h-obs-wpi ?M* = *Mapping.empty* **and** *transitions-wpi ?M* = {}
using *False* **unfolding** *canonical-separator′-impl.simps Let-def* **by** *auto*
have *h-wpi ?M* = (*set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′)))
unfolding ‹*h-wpi ?M* = *Mapping.empty*› ‹*transitions-wpi ?M* = {}›
unfolding *set-as-mapping-empty* **by** *presburger*
have *h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)
unfolding ‹*h-wpi ?M* = *Mapping.empty*› ‹*h-obs-wpi ?M* = *Mapping.empty*›
unfolding *h-obs-from-impl-h* **by** *simp*

have *base-props*: (*initial-wpi ?M* ∈ *states-wpi ?M*
∧ *finite* (*states-wpi ?M*)
∧ *finite* (*inputs-wpi ?M*)
∧ *finite* (*outputs-wpi ?M*)
∧ *finite* (*transitions-wpi ?M*))
using *a1 False* **unfolding** *canonical-separator′-impl.simps Let-def* **by** *auto*

have *transition-prop*: (∀ *t* ∈ *transitions-wpi ?M* . *t-source t* ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧
t-target t ∈ *states-wpi ?M* ∧
t-output t ∈ *outputs-wpi ?M*)
using *a1 False* **unfolding** *canonical-separator′-impl.simps Let-def* **by** *auto*

show *?thesis*
using *base-props transition-prop*
well-formed-h-set-as-mapping[*OF* ‹*h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′))›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*

next
case *True*

let *?f* = (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*)) (*transitions-wpi M*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {}))

have ⋀ *qx* . (λ*qx* . (*case* (*set-as-map* (*image* (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*)) (*transitions-wpi M*))) *qx of Some yqs* ⇒ *yqs* | *None* ⇒ {})) *qx* = (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*)) ‘ *transitions-wpi M*}) *qx*
by (*metis* (*mono-tags*, *lifting*) *Collect-cong Collect-mem-eq set-as-map-containment set-as-map-elem*)
moreover have ⋀ *qx* . (λ *qx* . {*z*. (*qx*, *z*) ∈ (λ(*q*, *x*, *y*, *q*′). ((*q*, *x*), *y*)) ‘ *transitions-wpi M*}) *qx* = (λ *qx* . {*y* | *y* . ∃ *q*′ . (*fst qx*, *snd qx*, *y*, *q*′) ∈ *transitions-wpi M*}) *qx*
by *force*
ultimately have ∗: *?f* = (λ *qx* . {*y* | *y* . ∃ *q*′ . (*fst qx*, *snd qx*, *y*, *q*′) ∈ *transitions-wpi M*})
by *blast*

let *?shifted-transitions*′ = *shifted-transitions* (*transitions-wpi P*)
let *?distinguishing-transitions-lr* = *distinguishing-transitions ?f q1 q2* (*states-wpi P*) (*inputs-wpi P*)
let *?ts* = *?shifted-transitions*′ ∪ *?distinguishing-transitions-lr*

have *states-wpi ?M* = (*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*}
and *transitions-wpi ?M* = *?ts*
unfolding *canonical-separator*′*-impl.simps Let-def True* **by** *simp+*

have *p2*: *finite* (*states-wpi ?M*)
unfolding ‹*states-wpi ?M* = (*image Inl* (*states-wpi P*)) ∪ {*Inr q1*, *Inr q2*}› **using** *a2* **by** *auto*

have *initial-wpi ?M* = *Inl* (*q1*,*q2*) **by** *auto*
then have *p1*: *initial-wpi ?M* ∈ *states-wpi ?M*
using *a1 a2* **unfolding** *canonical-separator*′*.simps True* **by** *auto*
have *p3*: *finite* (*inputs-wpi ?M*)
using *a1 a2* **by** *auto*
have *p4*: *finite* (*outputs-wpi ?M*)
using *a1 a2* **by** *auto*

have *finite* (*states-wpi P* × *inputs-wpi P*)
using *a2* **by** *auto*
moreover have ∗∗: ⋀ *x q1* . *finite* ({*y* |*y*. ∃ *q*′. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q*′) ∈ *transitions-wpi M*})
proof −
fix *x q1*
have {*y* |*y*. ∃ *q*′. (*fst* (*q1*, *x*), *snd* (*q1*, *x*), *y*, *q*′) ∈ *transitions-wpi M*} = {*t-output t* | *t* . *t* ∈ *transitions-wpi M* ∧ *t-source t* = *q1* ∧ *t-input t* = *x*}
by *auto*

then have $\{y \mid y.\ \exists\, q'.\ (fst\ (q1,\ x),\ snd\ (q1,\ x),\ y,\ q') \in transitions\text{-}wpi\ M\}$ $\subseteq$ *image t-output* (*transitions-wpi M*)
unfolding *fst-conv snd-conv* **by** *blast*
moreover have *finite* (*image t-output* (*transitions-wpi M*))
using *a1* **by** *auto*
ultimately show *finite* ($\{y \mid y.\ \exists\, q'.\ (fst\ (q1,\ x),\ snd\ (q1,\ x),\ y,\ q') \in transitions\text{-}wpi\ M\}$)
by (*simp add*: *finite-subset*)
qed
ultimately have *finite ?distinguishing-transitions-lr*
unfolding $*$ *distinguishing-transitions-def* **by** *force*
moreover have *finite ?shifted-transitions′*
unfolding *shifted-transitions-def* **using** *a2* **by** *auto*
ultimately have *finite ?ts* **by** *blast*
then have *p5*: *finite* (*transitions-wpi ?M*)
by *simp*

have *inputs-wpi ?M* = *inputs-wpi M* $\cup$ *inputs-wpi P*
using *True* **by** *auto*
have *outputs-wpi ?M* = *outputs-wpi M* $\cup$ *outputs-wpi P*
using *True* **by** *auto*

have $\bigwedge$ *t* . *t* $\in$ *?shifted-transitions′* $\Longrightarrow$ *t-source t* $\in$ *states-wpi ?M* $\wedge$ *t-target t* $\in$ *states-wpi ?M*
unfolding ‹*states-wpi ?M* = (*image Inl* (*states-wpi P*)) $\cup$ {*Inr q1*, *Inr q2*}› *shifted-transitions-def*
using *a2* **by** *auto*
moreover have $\bigwedge$ *t* . *t* $\in$ *?distinguishing-transitions-lr* $\Longrightarrow$ *t-source t* $\in$ *states-wpi ?M* $\wedge$ *t-target t* $\in$ *states-wpi ?M*
unfolding ‹*states-wpi ?M* = (*image Inl* (*states-wpi P*)) $\cup$ {*Inr q1*, *Inr q2*}› *distinguishing-transitions-def* $*$ **by** *force*
ultimately have $\bigwedge$ *t* . *t* $\in$ *?ts* $\Longrightarrow$ *t-source t* $\in$ *states-wpi ?M* $\wedge$ *t-target t* $\in$ *states-wpi ?M*
by *blast*
moreover have $\bigwedge$ *t* . *t* $\in$ *?shifted-transitions′* $\Longrightarrow$ *t-input t* $\in$ *inputs-wpi ?M* $\wedge$ *t-output t* $\in$ *outputs-wpi ?M*
proof −
have $\bigwedge$ *t* . *t* $\in$ *?shifted-transitions′* $\Longrightarrow$ *t-input t* $\in$ *inputs-wpi P* $\wedge$ *t-output t* $\in$ *outputs-wpi P*
unfolding *shifted-transitions-def* **using** *a2* **by** *auto*
then show $\bigwedge$ *t* . *t* $\in$ *?shifted-transitions′* $\Longrightarrow$ *t-input t* $\in$ *inputs-wpi ?M* $\wedge$ *t-output t* $\in$ *outputs-wpi ?M*
unfolding ‹*inputs-wpi ?M* = *inputs-wpi M* $\cup$ *inputs-wpi P*›
‹*outputs-wpi ?M* = *outputs-wpi M* $\cup$ *outputs-wpi P*› **by** *blast*
qed
moreover have $\bigwedge$ *t* . *t* $\in$ *?distinguishing-transitions-lr* $\Longrightarrow$ *t-input t* $\in$ *inputs-wpi ?M* $\wedge$ *t-output t* $\in$ *outputs-wpi ?M*
unfolding $*$ *distinguishing-transitions-def* **using** *a1 a2 True* **by** *auto*
ultimately have *p6*: ($\forall$ *t*$\in$*transitions-wpi ?M*.

t-source t ∈ *states-wpi ?M* ∧
t-input t ∈ *inputs-wpi ?M* ∧ *t-target t* ∈ *states-wpi ?M* ∧
t-output t ∈ *outputs-wpi ?M*)
unfolding ‹*transitions-wpi ?M* = *?ts*› **by** *blast*

have *h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′))
and *h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)
using *True* **unfolding** *canonical-separator′-impl.simps Let-def* **by** *auto*

show *well-formed-fsm-with-precomputations ?M*
using *p1 p2 p3 p4 p5 p6*
using *well-formed-h-set-as-mapping*[*OF* ‹*h-wpi ?M* = *set-as-mapping-image* (*transitions-wpi ?M*) (λ(*q*,*x*,*y*,*q*′) . ((*q*,*x*),*y*,*q*′))›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?M* = *h-obs-impl-from-h* (*h-wpi ?M*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed
qed
qed

lemma *canonical-separator′-simps* :
initial-wp (*canonical-separator′ M P q1 q2*) = *Inl* (*q1*,*q2*)
states-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then* (*image Inl* (*states-wp P*)) ∪ {*Inr q1*, *Inr q2*} *else* {*Inl* (*q1*,*q2*)})
inputs-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then inputs-wp M* ∪ *inputs-wp P else* {})
outputs-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then outputs-wp M* ∪ *outputs-wp P else* {})
transitions-wp (*canonical-separator′ M P q1 q2*) = (*if initial-wp P* = (*q1*,*q2*) *then shifted-transitions* (*transitions-wp P*) ∪ *distinguishing-transitions* (λ (*q*,*x*) . {*y* . ∃ *q*′ . (*q*,*x*,*y*,*q*′) ∈ *transitions-wp M*}) *q1 q2* (*states-wp P*) (*inputs-wp P*) *else* {})
unfolding *h-out-impl-helper* **by** (*transfer*; *simp add*: *Let-def*)+

lemma *canonical-separator-with-precomputations-code* [*code*] : *FSM-Impl.canonical-separator′* ((*FSMWP M*)) ((*FSMWP P*)) *q1 q2* = *FSMWP* (*canonical-separator′ M P q1 q2*)
proof −

have ∗:⋀ *M1 M2* . (*M1* = *M2*) = (*fsm-impl.initial M1* = *fsm-impl.initial M2*
∧ *fsm-impl.states M1* = *fsm-impl.states M2*
∧ *fsm-impl.inputs M1* = *fsm-impl.inputs M2*
∧ *fsm-impl.outputs M1* = *fsm-impl.outputs M2*
∧ *fsm-impl.transitions M1* = *fsm-impl.transitions M2*)
by (*meson fsm-impl.expand*)

show *?thesis*
unfolding ∗
unfolding *FSM-Impl.canonical-separator′-simps*

unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *canonical-separator′-simps*
by *blast*
qed

fun *product-impl* :: ('a,'b,'c) *fsm-with-precomputations-impl* ⇒ ('d,'b,'c) *fsm-with-precomputations-impl* ⇒ ('a × 'd,'b,'c) *fsm-with-precomputations-impl* **where**
product-impl A B = (*let ts* = (*image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) (*Set.filter* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . *x* = *x′* ∧ *y* = *y′*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wpi B*)) (*transitions-wpi A*)))));
h′ = *set-as-mapping-image ts* (λ(*q*,*x*,*y*,*q′*) . ((*q*,*x*),*y*,*q′*))
in
FSMWPI ((*initial-wpi A*, *initial-wpi B*))
((*states-wpi A*) × (*states-wpi B*))
(*inputs-wpi A* ∪ *inputs-wpi B*)
(*outputs-wpi A* ∪ *outputs-wpi B*)
ts
h′
(*h-obs-impl-from-h h′*))

lift-definition *product* :: ('a,'b,'c) *fsm-with-precomputations* ⇒ ('d,'b,'c) *fsm-with-precomputations* ⇒ ('a × 'd,'b,'c) *fsm-with-precomputations* **is** *product-impl*
proof −
fix *A* :: ('a,'b,'c) *fsm-with-precomputations-impl*
fix *B* :: ('d,'b,'c) *fsm-with-precomputations-impl*
assume *a1*: *well-formed-fsm-with-precomputations A* **and** *a2*: *well-formed-fsm-with-precomputations B*

let *?P* = *product-impl A B*

have (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wpi B*)) (*transitions-wpi A*))) = {(*tA*,*tB*) | *tA tB* . *tA* ∈ *transitions-wpi A* ∧ *tB* ∈ *transitions-wpi B*}
by *auto*
then have (*Set.filter* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . *x* = *x′* ∧ *y* = *y′*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wpi B*)) (*transitions-wpi A*)))) = {((*qA*,*x*,*y*,*qA′*),(*qB*,*x*,*y*,*qB′*)) | *qA qB x y qA′ qB′* . (*qA*,*x*,*y*,*qA′*) ∈ *transitions-wpi A* ∧ (*qB*,*x*,*y*,*qB′*) ∈ *transitions-wpi B*}
by *auto*
then have *image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) (*Set.filter* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . *x* = *x′* ∧ *y* = *y′*) (⋃(*image* (λ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wpi B*)) (*transitions-wpi A*))))
= *image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) {((*qA*,*x*,*y*,*qA′*),(*qB*,*x*,*y*,*qB′*)) | *qA qB x y qA′ qB′* . (*qA*,*x*,*y*,*qA′*) ∈ *transitions-wpi A* ∧ (*qB*,*x*,*y*,*qB′*) ∈ *transitions-wpi B*}
by *auto*
then have *transitions-wpi ?P* = *image* (λ((*qA*,*x*,*y*,*qA′*), (*qB*,*x′*,*y′*,*qB′*)) . ((*qA*,*qB*),*x*,*y*,(*qA′*,*qB′*))) {((*qA*,*x*,*y*,*qA′*),(*qB*,*x*,*y*,*qB′*)) | *qA qB x y qA′ qB′* . (*qA*,*x*,*y*,*qA′*) ∈ *transitions-wpi*

$A \wedge (qB,x,y,qB') \in$ *transitions-wpi* $B\}$
by *auto*
also have ... = $\{((qA,qB),x,y,(qA',qB')) \mid qA\ qB\ x\ y\ qA'\ qB' . (qA,x,y,qA') \in$ *transitions-wpi* $A \wedge (qB,x,y,qB') \in$ *transitions-wpi* $B\}$
by *force*
finally have *transitions-wpi ?P* = $\{((qA,qB),x,y,(qA',qB')) \mid qA\ qB\ x\ y\ qA'\ qB'$ $. (qA,x,y,qA') \in$ *transitions-wpi* $A \wedge (qB,x,y,qB') \in$ *transitions-wpi* $B\}$ **.**

have *h-wpi ?P* = *set-as-mapping-image* (*transitions-wpi ?P*) ($\lambda(q,x,y,q') . ((q,x),y,q')$)
and *h-obs-wpi ?P* = *h-obs-impl-from-h* (*h-wpi ?P*)
unfolding *canonical-separator'-impl.simps Let-def* **by** *auto*

have *initial-wpi ?P* $\in$ *states-wpi ?P*
using *a1 a2* **by** *auto*
moreover have *finite* (*states-wpi ?P*)
using *a1 a2* **by** *auto*
moreover have *finite* (*inputs-wpi ?P*)
using *a1 a2* **by** *auto*
moreover have *finite* (*outputs-wpi ?P*)
using *a1 a2* **by** *auto*
moreover have *finite* (*transitions-wpi ?P*)
using *a1 a2* **unfolding** *product-code-naive* **by** *auto*
moreover have ($\forall$ *t*$\in$*transitions-wpi ?P*.
t-source t $\in$ *states-wpi ?P* $\wedge$
t-input t $\in$ *inputs-wpi ?P* $\wedge$ *t-target t* $\in$ *states-wpi ?P* $\wedge$
t-output t $\in$ *outputs-wpi ?P*)
using *a1 a2* **unfolding** *well-formed-fsm-with-precomputations.simps*
unfolding ‹*transitions-wpi ?P* = $\{((qA,qB),x,y,(qA',qB')) \mid qA\ qB\ x\ y\ qA'\ qB'$ $. (qA,x,y,qA') \in$ *transitions-wpi* $A \wedge (qB,x,y,qB') \in$ *transitions-wpi* $B\}$›
by *fastforce*
ultimately show *well-formed-fsm-with-precomputations ?P*
using *well-formed-h-set-as-mapping*[*OF* ‹*h-wpi ?P* = *set-as-mapping-image* (*transitions-wpi ?P*) ($\lambda(q,x,y,q') . ((q,x),y,q')$)›]
well-formed-h-obs-impl-from-h[*OF* ‹*h-obs-wpi ?P* = *h-obs-impl-from-h* (*h-wpi ?P*)›]
unfolding *well-formed-fsm-with-precomputations.simps* **by** *blast*
qed

lemma *product-simps*:
initial-wp (*product A B*) = (*initial-wp A*, *initial-wp B*)
states-wp (*product A B*) = (*states-wp A*) $\times$ (*states-wp B*)
inputs-wp (*product A B*) = *inputs-wp A* $\cup$ *inputs-wp B*
outputs-wp (*product A B*) = *outputs-wp A* $\cup$ *outputs-wp B*
transitions-wp (*product A B*) = (*image* ($\lambda((qA,x,y,qA'), (qB,x',y',qB')) . ((qA,qB),x,y,(qA',qB'))$) (*Set.filter* ($\lambda((qA,x,y,qA'), (qB,x',y',qB'))\ .\ x = x' \wedge y = y'$) ($\bigcup$(*image* ($\lambda$ *tA* . *image* (λ *tB* . (*tA*,*tB*)) (*transitions-wp B*)) (*transitions-wp A*)))))
by (*transfer*; *simp*)+

lemma *product-with-precomputations-code* [*code*] : *FSM-Impl.product* ((*FSMWP*

A)) ((*FSMWP B*)) = *FSMWP* (*product A B*)
unfolding *FSM-Impl.product-code-naive*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *FSMWP-def*
unfolding *product-simps*
by *presburger*

fun *from-FSMI-impl* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl* **where**
from-FSMI-impl M q = (*if q* ∈ *states-wpi M then FSMWPI q* (*states-wpi M*) (*inputs-wpi M*) (*outputs-wpi M*) (*transitions-wpi M*) (*h-wpi M*) (*h-obs-wpi M*) *else M*)

lift-definition *from-FSMI* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations* ⇒ '*a* ⇒ ('*a*,'*b*,'*c*) *fsm-with-precomputations* **is** *from-FSMI-impl*
proof −
fix *M* :: ('*a*,'*b*,'*c*) *fsm-with-precomputations-impl*
fix *q*
assume *well-formed-fsm-with-precomputations M*
then show *well-formed-fsm-with-precomputations* (*from-FSMI-impl M q*)
by (*cases q* ∈ *states-wpi M*; *auto*)
qed

lemma *from-FSMI-simps*:
initial-wp (*from-FSMI M q*) = (*if q* ∈ *states-wp M then q else initial-wp M*)
states-wp (*from-FSMI M q*) = *states-wp M*
inputs-wp (*from-FSMI M q*) = *inputs-wp M*
outputs-wp (*from-FSMI M q*) = *outputs-wp M*
transitions-wp (*from-FSMI M q*) = *transitions-wp M*
by (*transfer*; *simp add*: *Let-def*)+

lemma *from-FSMI-with-precomputations-code* [*code*] : *FSM-Impl.from-FSMI* ((*FSMWP M*)) *q* = *FSMWP* (*from-FSMI M q*)
unfolding *FSM-Impl.from-FSMI.simps*
unfolding *fsm-impl-FSMWP-initial fsm-impl-FSMWP-states fsm-impl-FSMWP-inputs fsm-impl-FSMWP-outputs fsm-impl-FSMWP-transitions*
unfolding *FSMWP-def*
unfolding *from-FSMI-simps*
by *presburger*

end

46 Code Export

This theory exports various functions developed in this library.

theory *Test-Suite-Generator-Code-Export*

imports *EquivalenceTesting/H-Method-Implementations*
EquivalenceTesting/HSI-Method-Implementations
EquivalenceTesting/W-Method-Implementations
EquivalenceTesting/Wp-Method-Implementations
EquivalenceTesting/SPY-Method-Implementations
EquivalenceTesting/SPYH-Method-Implementations
EquivalenceTesting/Partial-S-Method-Implementations
AdaptiveStateCounting/Test-Suite-Calculation-Refined
Prime-Transformation
Prefix-Tree-Refined
EquivalenceTesting/Test-Suite-Representations-Refined
HOL−Library.List-Lexorder
HOL−Library.Code-Target-Nat
HOL−Library.Code-Target-Int
Native-Word.Uint64
FSM-Code-Datatype
begin

46.1 Reduction Testing

definition *generate-reduction-test-suite-naive* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *String.literal* + (*uint64*×*uint64*) *list list* **where**
generate-reduction-test-suite-naive M m = (*case* (*calculate-test-suite-naive-as-io-sequences-with-assumption-cl*
M (*nat-of-integer m*)) *of*
Inl err ⇒ *Inl err* |
Inr ts ⇒ *Inr* (*sorted-list-of-set ts*))

definition *generate-reduction-test-suite-greedy* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *String.literal* + (*uint64*×*uint64*) *list list* **where**
generate-reduction-test-suite-greedy M m = (*case* (*calculate-test-suite-greedy-as-io-sequences-with-assumption-*
M (*nat-of-integer m*)) *of*
Inl err ⇒ *Inl err* |
Inr ts ⇒ *Inr* (*sorted-list-of-set ts*))

46.1.1 Fault Detection Capabilities of the Test Harness

The test harness for reduction testing (see https://bitbucket.org/Robert-Sachtleben/an-approach-for-the-verification-and-synthesis-of-complete) applies a test suite to a system under test (SUT) by repeatedly applying each IO-sequence (test case) in the test suite input by input to the SUT until either the test case has been fully applied or the first output is observed that does not correspond to the outputs in the IO-sequence and then checks whether the observed IO-sequence (consisting of a prefix of the test case possibly followed by an IO-pair consisting of the next input in the test case and an output that is not the next output in the test case) is prefix of some test case in the test suite. If such a prefix exists, then the application passes, else it fails and the overall application is aborted, reporting a failure.

The following lemma shows that the SUT (whose behaviour corresponds

to an FSM M') conforms to the specification (here FSM M) if and only if the above application procedure does not fail. As the following lemma uses quantification over all possible responses of the SUT to each test case, a further testability hypothesis is required to transfer this result to the actual test application process, which by necessity can only perform a finite number of applications: we assume that some value k exists such that by applying each test case k times, all responses of the SUT to it can be observed.

lemma *reduction-test-harness-soundness* :
fixes M :: (*uint64*,*uint64*,*uint64*) *fsm*
assumes *observable* M'
and *FSM.inputs* M' = *FSM.inputs* M
and *completely-specified* M'
and *size* $M' \leq$ *nat-of-integer* m
and *generate-reduction-test-suite-greedy* M m = *Inr ts*
shows (L M' $\subseteq$ L M) $\longleftrightarrow$ (*list-all* (λ *io* . $\neg$ ($\exists$ *ioPre x y y' ioSuf* . *io* = *ioPre*@[(x,y)]@*ioSuf* $\wedge$ *ioPre*@[(x,y')] $\in$ L M' $\wedge$ $\neg$($\exists$ *ioSuf'* . *ioPre*@[(x,y')]@*ioSuf'* $\in$ *list.set ts*))) *ts*)
proof −

obtain *tss* **where** *calculate-test-suite-greedy-as-io-sequences-with-assumption-check* M (*nat-of-integer* m) = *Inr tss*
using *assms*(*5*) **unfolding** *generate-reduction-test-suite-greedy-def*
by (*metis Inr-Inl-False old.sum.exhaust old.sum.simps*(*5*))

have *FSM.inputs* M $\neq$ {}
and *observable* M
and *completely-specified* M
using ‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check* M (*nat-of-integer* m) = *Inr tss*›
unfolding *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-def*

by (*meson Inl-Inr-False*)+
then have *tss* = (*test-suite-to-io-maximal* M (*calculate-test-suite-greedy* M (*nat-of-integer* m)))
using ‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check* M (*nat-of-integer* m) = *Inr tss*›
unfolding *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-def*
by (*metis sum.inject*(*2*))

have $\bigwedge q.$ $q \in$ *FSM.states* $M \Longrightarrow \exists$ $d \in$ *list.set* (*maximal-repetition-sets-from-separators-list-greedy* M). $q \in$ *fst* d
unfolding *maximal-repetition-sets-from-separators-list-greedy-def Let-def*
using *greedy-pairwise-r-distinguishable-state-sets-from-separators-cover*[*of* - M]
by *simp*
moreover have $\bigwedge d.$ $d \in$ *list.set* (*maximal-repetition-sets-from-separators-list-greedy* M) $\Longrightarrow$ *fst* $d \subseteq$ *FSM.states* M $\wedge$ (*snd* d = *fst* d $\cap$ *fst* ‘ *d-reachable-states-with-preambles* M)

and $\bigwedge$ *d q1 q2*. *d* ∈ *list.set* (*maximal-repetition-sets-from-separators-list-greedy M*) ⟹ *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
proof
fix *d* **assume** *d* ∈ *list.set* (*maximal-repetition-sets-from-separators-list-greedy M*)
then have *fst d* ∈ *list.set* (*greedy-pairwise-r-distinguishable-state-sets-from-separators M*)
and (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
unfolding *maximal-repetition-sets-from-separators-list-greedy-def Let-def* **by** *force+*

then have *fst d* ∈ *pairwise-r-distinguishable-state-sets-from-separators M*
using *greedy-pairwise-r-distinguishable-state-sets-from-separators-soundness* **by** *blast*
then show *fst d* ⊆ *FSM.states M* **and** (*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)
and $\bigwedge$ *q1 q2* . *q1*∈*fst d* ⟹ *q2*∈*fst d* ⟹ *q1* ≠ *q2* ⟹ (*q1*, *q2*) ∈ *fst* ‘ *r-distinguishable-state-pairs-with-separators M*
using ‹(*snd d* = *fst d* ∩ *fst* ‘ *d-reachable-states-with-preambles M*)›
unfolding *pairwise-r-distinguishable-state-sets-from-separators-def*
by *force+*
qed
ultimately have *implies-completeness* (*calculate-test-suite-greedy M* (*nat-of-integer m*)) *M* (*nat-of-integer m*)
and *is-finite-test-suite* (*calculate-test-suite-greedy M* (*nat-of-integer m*))
using *calculate-test-suite-for-repetition-sets-sufficient-and-finite*[*OF* ‹*observable M*› ‹*completely-specified M*› ‹*FSM.inputs M* ≠ {}›]
unfolding *calculate-test-suite-greedy-def*
by *simp+*

then have *finite tss*
using *test-suite-to-io-maximal-finite*[*OF* - - ‹*observable M*›]
unfolding ‹*tss* = (*test-suite-to-io-maximal M* (*calculate-test-suite-greedy M* (*nat-of-integer m*)))›
by *blast*

have *list.set ts* = *test-suite-to-io-maximal M* (*calculate-test-suite-greedy M* (*nat-of-integer m*))
and *ts* = *sorted-list-of-set tss*
using *sorted-list-of-set*(*1*)[*OF* ‹*finite tss*›]
using *assms*(*5*)
unfolding ‹*tss* = (*test-suite-to-io-maximal M* (*calculate-test-suite-greedy M* (*nat-of-integer m*)))›
‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check M* (*nat-of-integer m*) = *Inr tss*›
generate-reduction-test-suite-greedy-def
by *simp+*

then have $(L\ M' \subseteq L\ M) = pass\text{-}io\text{-}set\text{-}maximal\ M'\ (list.set\ ts)$
using *calculate-test-suite-greedy-as-io-sequences-with-assumption-check-completeness*[*OF assms*(*1*,*2*,*3*,*4*)]
‹*calculate-test-suite-greedy-as-io-sequences-with-assumption-check M* (*nat-of-integer m*) = *Inr tss*›
‹*tss* = *test-suite-to-io-maximal M* (*calculate-test-suite-greedy M* (*nat-of-integer m*))›
by *simp*

moreover have *pass-io-set-maximal M′* (*list.set ts*)
= (*list-all* (λ *io* . ¬ (∃ *ioPre x y y′ ioSuf* . *io* = *ioPre*@[(*x*,*y*)]@*ioSuf* ∧ *ioPre*@[(*x*,*y′*)] ∈ *L M′* ∧ ¬(∃ *ioSuf′* . *ioPre*@[(*x*,*y′*)]@*ioSuf′* ∈ *list.set ts*))) *ts*)
proof −
have ⋀ *P* . *list-all P* (*sorted-list-of-set tss*) = (∀ *x* ∈ *tss* . *P x*)
by (*simp add*: ‹*finite tss*› *list-all-iff*)
then have *scheme*: ⋀ *P* . *list-all P ts* = (∀ *x* ∈ (*list.set ts*) . *P x*)
unfolding ‹*ts* = *sorted-list-of-set tss*› *sorted-list-of-set*(*1*)[*OF* ‹*finite tss*›]
by *simp*

show *?thesis*
using *scheme*[*of* (λ *io* . ¬ (∃ *ioPre x y y′ ioSuf* . *io* = *ioPre*@[(*x*,*y*)]@*ioSuf* ∧ *ioPre*@[(*x*,*y′*)] ∈ *L M′* ∧ ¬(∃ *ioSuf′* . *ioPre*@[(*x*,*y′*)]@*ioSuf′* ∈ *list.set ts*)))]
unfolding *pass-io-set-maximal-def*
by *fastforce*
qed

ultimately show *?thesis*
by *simp*
qed

46.2 Equivalence Testing

46.2.1 Test Strategy Application and Transformation

fun *apply-method-to-prime* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*,*uint64*,*uint64*) *fsm* ⇒ *nat* ⇒ (*uint64*×*uint64*) *prefix-tree*) ⇒ (*uint64*×*uint64*) *prefix-tree* **where**
apply-method-to-prime M additionalStates isAlreadyPrime f = (*let*
M′ = (*if isAlreadyPrime then M else to-prime-uint64 M*);
m = *size-r M′* + (*nat-of-integer additionalStates*)
in f M′ m)

lemma *apply-method-to-prime-completeness* :
fixes *M2* :: (′*a*,*uint64*,*uint64*) *fsm*
assumes ⋀ *M1 m* (*M2* :: (′*a*,*uint64*,*uint64*) *fsm*) .
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹

size-r M1 $\leq$ *m* $\Longrightarrow$
size M2 $\leq$ *m* $\Longrightarrow$
FSM.inputs M2 = *FSM.inputs M1* $\Longrightarrow$
FSM.outputs M2 = *FSM.outputs M1* $\Longrightarrow$
(*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*f M1 m*)) = (*L M2* $\cap$ *set* (*f M1 m*)))
and *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ((*L M1* $\cap$ *set* (*apply-method-to-prime M1 additionalStates isAlreadyPrime f*)) = (*L M2* $\cap$ *set* (*apply-method-to-prime M1 additionalStates isAlreadyPrime f*)))
proof −

define *M′* **where** *M′* = (*if isAlreadyPrime then M1 else to-prime-uint64 M1*)
have *observable M′* **and** *minimal M′* **and** *L M1* = *L M′* **and** *FSM.inputs M′* = *FSM.inputs M1* **and** *FSM.outputs M′* = *FSM.outputs M1*
unfolding *M′-def* **using** *to-prime-uint64-props*[*OF assms*(*8*)] *assms*(*7*)
by (*metis* (*full-types*))+
then have *FSM.inputs M2* = *FSM.inputs M′* **and** *FSM.outputs M2* = *FSM.outputs M′*
using *assms*(*5*,*6*) **by** *auto*

have *size-r M′* = *size-r* (*to-prime M1*)
by (*metis* (*no-types*) ‹*L M1* = *L M′*› ‹*minimal M′*› ‹*observable M′*› *minimal-equivalence-size-r to-prime-props*(*1*) *to-prime-props*(*2*) *to-prime-props*(*3*))
then have *size-r M′* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
by *simp*

show *?thesis*
using *assms*(*1*)[*OF* ‹*observable M′*› *assms*(*2*) ‹*minimal M′*› *assms*(*3*) ‹*size-r M′* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)› *assms*(*4*) ‹*FSM.inputs M2* = *FSM.inputs M′*› ‹*FSM.outputs M2* = *FSM.outputs M′*›]
unfolding *apply-method-to-prime.simps Let-def* ‹*size-r M′* = *size-r* (*to-prime M1*)›[*symmetric*] *M′-def* ‹*L M1* = *L M′*› **.**
qed

fun *apply-to-prime-and-return-io-lists* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *nat* $\Rightarrow$ (*uint64*$\times$*uint64*) *prefix-tree*) $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime f = (*let M′* = (*if isAlreadyPrime then M else to-prime-uint64 M*) *in*
sorted-list-of-maximal-sequences-in-tree (*test-suite-from-io-tree M′* (*FSM.initial*

M′) (*apply-method-to-prime M additionalStates isAlreadyPrime f*)))

lemma *apply-to-prime-and-return-io-lists-completeness* :
fixes *M2* :: (*′a*,*uint64*,*uint64*) *fsm*
assumes ⋀ *M1 m* (*M2* :: (*′a*,*uint64*,*uint64*) *fsm*) .
observable M1 ⟹
observable M2 ⟹
minimal M1 ⟹
minimal M2 ⟹
size-r M1 ≤ *m* ⟹
size M2 ≤ *m* ⟹
FSM.inputs M2 = *FSM.inputs M1* ⟹
FSM.outputs M2 = *FSM.outputs M1* ⟹
((*L M1* = *L M2*) ⟷ ((*L M1* ∩ *set* (*f M1 m*)) = (*L M2* ∩ *set* (*f M1 m*))))
∧ *finite-tree* (*f M1 m*)
and *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*apply-to-prime-and-return-io-lists M1 additionalStates isAlreadyPrime f*)
proof −

define *M′* **where** *M′* = (*if isAlreadyPrime then M1 else to-prime-uint64 M1*)
have *observable M′* **and** *minimal M′* **and** *L M1* = *L M′* **and** *FSM.inputs M′* = *FSM.inputs M1* **and** *FSM.outputs M′* = *FSM.outputs M1*
unfolding *M′-def* **using** *to-prime-uint64-props*[*OF assms*(*8*)] *assms*(*7*)
by (*metis* (*full-types*))+
then have *FSM.inputs M2* = *FSM.inputs M′* **and** *FSM.outputs M2* = *FSM.outputs M′*
using *assms*(*5*,*6*) **by** *auto*

have *L M′* = *L* (*to-prime M1*)
using *to-prime-props*(*1*) *M′-def*
using ‹*L M1* = *L M′*› **by** *blast*

have *size-r M′* = *size-r* (*to-prime M1*)
using *minimal-equivalence-size-r*[*OF* ‹*minimal M′*› - ‹*observable M′*› - ‹*L M′* = *L* (*to-prime M1*)›]
using *assms*(*7*) *to-prime-props*(*2*,*3*)
unfolding *M′-def*
by *blast*
then have *size-r M′* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)

by *simp*

have $*$:$(L\ M1 = L\ M2) \longleftrightarrow ((L\ M1 \cap set\ (f\ M'\ (size\text{-}r\ (to\text{-}prime\ M1) + nat\text{-}of\text{-}integer\ additionalStates))) = (L\ M2 \cap set\ (f\ M'\ (size\text{-}r\ (to\text{-}prime\ M1) + nat\text{-}of\text{-}integer\ additionalStates))))$
and $**$:$finite\text{-}tree\ (f\ M'\ (size\text{-}r\ (to\text{-}prime\ M1) + nat\text{-}of\text{-}integer\ additionalStates))$
using $assms(1)[OF\ ‹observable\ M'›\ assms(2)\ ‹minimal\ M'›\ assms(3)\ ‹size\text{-}r\ M' \leq size\text{-}r\ (to\text{-}prime\ M1) + (nat\text{-}of\text{-}integer\ additionalStates)›\ assms(4)\ ‹FSM.inputs\ M2 = FSM.inputs\ M'›\ ‹FSM.outputs\ M2 = FSM.outputs\ M'›]$
unfolding $‹L\ M1 = L\ M'›$ **by** *blast+*

show *?thesis*
unfolding $*$
using $passes\text{-}test\text{-}cases\text{-}from\text{-}io\text{-}tree[OF\ ‹observable\ M'›\ assms(2)\ fsm\text{-}initial[of\ M']\ fsm\text{-}initial[of\ M2]\ **\ ]$
unfolding $‹size\text{-}r\ M' = size\text{-}r\ (to\text{-}prime\ M1)›[symmetric]$
unfolding *apply-to-prime-and-return-io-lists.simps apply-method-to-prime.simps Let-def* $‹L\ M1 = L\ M'›$
unfolding *M'-def* **by** *blast*
qed

fun $apply\text{-}to\text{-}prime\text{-}and\text{-}return\text{-}input\text{-}lists :: (uint64,uint64,uint64)\ fsm \Rightarrow integer \Rightarrow bool \Rightarrow ((uint64,uint64,uint64)\ fsm \Rightarrow nat \Rightarrow (uint64 \times uint64)\ prefix\text{-}tree) \Rightarrow uint64\ list\ list$ **where**
$apply\text{-}to\text{-}prime\text{-}and\text{-}return\text{-}input\text{-}lists\ M\ additionalStates\ isAlreadyPrime\ f = test\text{-}suite\text{-}to\text{-}input\text{-}sequences\ (apply\text{-}method\text{-}to\text{-}prime\ M\ additionalStates\ isAlreadyPrime\ f)$

lemma *apply-to-prime-and-return-input-lists-completeness* :
fixes $M2 :: ('a,uint64,uint64)\ fsm$
assumes $\bigwedge M1\ m\ (M2 :: ('a,uint64,uint64)\ fsm)$.
$observable\ M1 \Longrightarrow$
$observable\ M2 \Longrightarrow$
$minimal\ M1 \Longrightarrow$
$minimal\ M2 \Longrightarrow$
$size\text{-}r\ M1 \leq m \Longrightarrow$
$size\ M2 \leq m \Longrightarrow$
$FSM.inputs\ M2 = FSM.inputs\ M1 \Longrightarrow$
$FSM.outputs\ M2 = FSM.outputs\ M1 \Longrightarrow$
$((L\ M1 = L\ M2) \longleftrightarrow ((L\ M1 \cap set\ (f\ M1\ m)) = (L\ M2 \cap set\ (f\ M1\ m))))$
$\wedge\ finite\text{-}tree\ (f\ M1\ m)$
and $observable\ M2$
and $minimal\ M2$
and $size\ M2 \leq size\text{-}r\ (to\text{-}prime\ M1) + (nat\text{-}of\text{-}integer\ additionalStates)$
and $FSM.inputs\ M2 = FSM.inputs\ M1$
and $FSM.outputs\ M2 = FSM.outputs\ M1$
and $isAlreadyPrime \Longrightarrow observable\ M1 \wedge minimal\ M1 \wedge reachable\text{-}states\ M1 = states\ M1$

and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*apply-to-prime-and-return-input-lists M1 additionalStates isAlreadyPrime f*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
proof −
define *M'* **where** *M'* = (*if isAlreadyPrime then M1 else to-prime-uint64 M1*)
have *observable M'* **and** *minimal M'* **and** *L M1* = *L M'* **and** *FSM.inputs M'* = *FSM.inputs M1* **and** *FSM.outputs M'* = *FSM.outputs M1*
unfolding *M'-def* **using** *to-prime-uint64-props*[*OF assms*(*8*)] *assms*(*7*)
by (*metis* (*full-types*))+
then have *FSM.inputs M2* = *FSM.inputs M'* **and** *FSM.outputs M2* = *FSM.outputs M'*
using *assms*(*5*,*6*) **by** *auto*

have *L M'* = *L* (*to-prime M1*)
using *to-prime-props*(*1*) *M'-def* ‹*L M1* = *L M'*› **by** *metis*

have *size-r M'* = *size-r* (*to-prime M1*)
using *minimal-equivalence-size-r*[*OF* ‹*minimal M'*› - ‹*observable M'*› - ‹*L M'* = *L* (*to-prime M1*)›]
using *assms*(*7*) *to-prime-props*(*2*,*3*)
unfolding *M'-def*
by *blast*
then have *size-r M'* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
by *simp*

have ∗:(*L M1* = *L M2*) = ((*L M1* ∩ *set* (*f M'* (*size-r* (*to-prime M1*) + *nat-of-integer additionalStates*))) = (*L M2* ∩ *set* (*f M'* (*size-r* (*to-prime M1*) + *nat-of-integer additionalStates*))))
and ∗∗:*finite-tree* (*f M'* (*size-r* (*to-prime M1*) + *nat-of-integer additionalStates*))
using *assms*(*1*)[*OF* ‹*observable M'*› *assms*(*2*) ‹*minimal M'*› *assms*(*3*) ‹*size-r M'* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)› *assms*(*4*) ‹*FSM.inputs M2* = *FSM.inputs M'*› ‹*FSM.outputs M2* = *FSM.outputs M'*›]
unfolding ‹*L M1* = *L M'*› **by** *blast*+

show *?thesis*
using *test-suite-to-input-sequences-pass-alt-def*[*OF* ∗∗ ∗]
unfolding ‹*size-r M'* = *size-r* (*to-prime M1*)›[*symmetric*]
unfolding *apply-to-prime-and-return-input-lists.simps apply-method-to-prime.simps Let-def M'-def* **.**
qed

46.2.2 W-Method

definition *w-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-h-framework*

lemma *w-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*w-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-h-framework-completeness-and-finiteness*
unfolding *w-method-via-h-framework-ts-def*
by *metis*

definition *w-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
w-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-h-framework*

lemma *w-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*w-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*w-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-h-framework-completeness-and-finiteness*
unfolding *w-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *w-method-via-h-framework-2-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
w-method-via-h-framework-2-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-h-framework-2*

lemma *w-method-via-h-framework-2-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)

and *FSM.inputs M2 = FSM.inputs M1*
and *FSM.outputs M2 = FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-h-framework-2-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*w-method-via-h-framework-2* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-h-framework-2-completeness-and-finiteness*
unfolding *w-method-via-h-framework-2-ts-def*
by *metis*

definition *w-method-via-h-framework-2-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
w-method-via-h-framework-2-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-h-framework-2*

lemma *w-method-via-h-framework-2-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*w-method-via-h-framework-2-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*w-method-via-h-framework-2* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-h-framework-2-completeness-and-finiteness*
unfolding *w-method-via-h-framework-2-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *w-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64* $\times$ *uint64*) $\times$ *bool*) *list list* **where**
w-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-spy-framework*

lemma *w-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*

and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*w-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-spy-framework-completeness-and-finiteness*
unfolding *w-method-via-spy-framework-ts-def*
by *metis*

definition *w-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime w-method-via-spy-framework*

lemma *w-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*w-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*w-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-spy-framework-completeness-and-finiteness*
unfolding *w-method-via-spy-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *w-method-via-pair-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
w-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime w-method-via-pair-framework*

lemma *w-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*w-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*w-method-via-pair-framework*

and *isAlreadyPrime=isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-pair-framework-completeness-and-finiteness*
unfolding *w-method-via-pair-framework-ts-def*
by *metis*

definition *w-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
w-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-list M additionalStates isAlreadyPrime w-method-via-pair-framework*

lemma *w-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*w-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*w-method-via-pair-framework*
and *isAlreadyPrime=isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *w-method-via-pair-framework-completeness-and-finiteness*
unfolding *w-method-via-pair-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.3 Wp-Method

definition *wp-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
wp-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime wp-method-via-h-framework*

lemma *wp-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*wp-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*wp-method-via-h-framework*
and *isAlreadyPrime=isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *wp-method-via-h-framework-completeness-and-finiteness*

unfolding *wp-method-via-h-framework-ts-def*
by *metis*

definition *wp-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
wp-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime wp-method-via-h-framework*

lemma *wp-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*wp-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*wp-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *wp-method-via-h-framework-completeness-and-finiteness*
unfolding *wp-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *wp-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
wp-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime wp-method-via-spy-framework*

lemma *wp-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*wp-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*wp-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *wp-method-via-spy-framework-completeness-and-finiteness*
unfolding *wp-method-via-spy-framework-ts-def*
by *metis*

definition *wp-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒

integer ⇒ *bool* ⇒ *uint64 list list* **where**
wp-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime wp-method-via-spy-framework*

lemma *wp-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*wp-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*wp-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *wp-method-via-spy-framework-completeness-and-finiteness*
unfolding *wp-method-via-spy-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.4 HSI-Method

definition *hsi-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
hsi-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-h-framework*

lemma *hsi-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*hsi-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-h-framework-completeness-and-finiteness*
unfolding *hsi-method-via-h-framework-ts-def*
by *metis*

definition *hsi-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
hsi-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists*

M additionalStates isAlreadyPrime hsi-method-via-h-framework

lemma *hsi-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*hsi-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*hsi-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-h-framework-completeness-and-finiteness*
unfolding *hsi-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *hsi-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
hsi-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-spy-framework*

lemma *hsi-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*hsi-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-spy-framework-completeness-and-finiteness*
unfolding *hsi-method-via-spy-framework-ts-def*
by *metis*

definition *hsi-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
hsi-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime hsi-method-via-spy-framework*

lemma *hsi-method-via-spy-framework-input-completeness* :
assumes *observable M2*

and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*hsi-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*hsi-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-spy-framework-completeness-and-finiteness*
unfolding *hsi-method-via-spy-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *hsi-method-via-pair-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
hsi-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime hsi-method-via-pair-framework*

lemma *hsi-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*hsi-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*hsi-method-via-pair-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-pair-framework-completeness-and-finiteness*
unfolding *hsi-method-via-pair-framework-ts-def*
by *metis*

definition *hsi-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
hsi-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime hsi-method-via-pair-framework*

lemma *hsi-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*

and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*hsi-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*hsi-method-via-pair-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *hsi-method-via-pair-framework-completeness-and-finiteness*
unfolding *hsi-method-via-pair-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.5 H-Method

definition *h-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
h-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-h-framework M m c b*)

lemma *h-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-h-framework-ts M1 additionalStates isAlreadyPrime c b*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-h-framework-completeness-and-finiteness*
unfolding *h-method-via-h-framework-ts-def*
by *metis*

definition *h-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
h-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-h-framework M m c b*)

lemma *h-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1*

= *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*h-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-h-framework-completeness-and-finiteness*
unfolding *h-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *h-method-via-pair-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
h-method-via-pair-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime h-method-via-pair-framework*

lemma *h-method-via-pair-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*h-method-via-pair-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-ts-def*
by *metis*

definition *h-method-via-pair-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-pair-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime h-method-via-pair-framework*

lemma *h-method-via-pair-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*

shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*h-method-via-pair-framework-input M1 additionalStates isAlreadyPrime*). $\forall$ *xs'*$\in$*list.set* (*prefixes xs*). {*io* $\in$ *L M1*. *map fst io* = *xs'*} = {*io* $\in$ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*h-method-via-pair-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *h-method-via-pair-framework-2-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ ((*uint64*$\times$*uint64*)$\times$*bool*) *list list* **where**
h-method-via-pair-framework-2-ts M additionalStates isAlreadyPrime c = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-2 M m c*)

lemma *h-method-via-pair-framework-2-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-2-ts M1 additionalStates isAlreadyPrime c*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-pair-framework-2 M m c*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-2-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-2-ts-def*
by *metis*

definition *h-method-via-pair-framework-2-input* :: (*uint64*,*uint64*,*uint64*) *fsm* $\Rightarrow$ *integer* $\Rightarrow$ *bool* $\Rightarrow$ *bool* $\Rightarrow$ *uint64 list list* **where**
h-method-via-pair-framework-2-input M additionalStates isAlreadyPrime c = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-2 M m c*)

lemma *h-method-via-pair-framework-2-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* $\leq$ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* $\Longrightarrow$ *observable M1* $\wedge$ *minimal M1* $\wedge$ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) $\longleftrightarrow$ ($\forall$ *xs*$\in$*list.set* (*h-method-via-pair-framework-2-input*

M1 additionalStates isAlreadyPrime c). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-pair-framework-2 M m c*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-2-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-2-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *h-method-via-pair-framework-3-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
h-method-via-pair-framework-3-ts M additionalStates isAlreadyPrime c1 c2 = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-3 M m c1 c2*)

lemma *h-method-via-pair-framework-3-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*h-method-via-pair-framework-3-ts M1 additionalStates isAlreadyPrime c1 c2*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-pair-framework-3 M m c1 c2*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-3-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-3-ts-def*
by *metis*

definition *h-method-via-pair-framework-3-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
h-method-via-pair-framework-3-input M additionalStates isAlreadyPrime c1 c2 = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *h-method-via-pair-framework-3 M m c1 c2*)

lemma *h-method-via-pair-framework-3-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*h-method-via-pair-framework-3-input*

M1 additionalStates isAlreadyPrime c1 c2). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *h-method-via-pair-framework-3 M m c1 c2*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *h-method-via-pair-framework-3-completeness-and-finiteness*
unfolding *h-method-via-pair-framework-3-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.6 SPY-Method

definition *spy-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spy-method-via-h-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime spy-method-via-h-framework*

lemma *spy-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spy-method-via-h-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*spy-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spy-method-via-h-framework-completeness-and-finiteness*
unfolding *spy-method-via-h-framework-ts-def*
by *metis*

definition *spy-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
spy-method-via-h-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime spy-method-via-h-framework*

lemma *spy-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spy-method-via-h-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst*

io = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*spy-method-via-h-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spy-method-via-h-framework-completeness-and-finiteness*
unfolding *spy-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *spy-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spy-method-via-spy-framework-ts M additionalStates isAlreadyPrime = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime spy-method-via-spy-framework*

lemma *spy-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spy-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=*spy-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spy-method-via-spy-framework-completeness-and-finiteness*
unfolding *spy-method-via-spy-framework-ts-def*
by *metis*

definition *spy-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *uint64 list list* **where**
spy-method-via-spy-framework-input M additionalStates isAlreadyPrime = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime spy-method-via-spy-framework*

lemma *spy-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spy-method-via-spy-framework-input M1 additionalStates isAlreadyPrime*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=*spy-method-via-spy-framework* **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spy-method-via-spy-framework-completeness-and-finiteness*

unfolding *spy-method-via-spy-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.7 SPYH-Method

definition *spyh-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spyh-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-h-framework M m c b*)

lemma *spyh-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spyh-method-via-h-framework-ts M1 additionalStates isAlreadyPrime c b*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *spyh-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spyh-method-via-h-framework-completeness-and-finiteness*
unfolding *spyh-method-via-h-framework-ts-def*
by *metis*

definition *spyh-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
spyh-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-h-framework M m c b*)

lemma *spyh-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spyh-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *spyh-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]

using *spyh-method-via-h-framework-completeness-and-finiteness*
unfolding *spyh-method-via-h-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

definition *spyh-method-via-spy-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
spyh-method-via-spy-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-spy-framework M m c b*)

lemma *spyh-method-via-spy-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*spyh-method-via-spy-framework-ts M1 additionalStates isAlreadyPrime c b*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *spyh-method-via-spy-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spyh-method-via-spy-framework-completeness-and-finiteness*
unfolding *spyh-method-via-spy-framework-ts-def*
by *metis*

definition *spyh-method-via-spy-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
spyh-method-via-spy-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *spyh-method-via-spy-framework M m c b*)

lemma *spyh-method-via-spy-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*spyh-method-via-spy-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *spyh-method-via-spy-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *spyh-method-via-spy-framework-completeness-and-finiteness*

unfolding *spyh-method-via-spy-framework-input-def*[*symmetric*]
by (*metis* (*no-types*, *lifting*))

46.2.8 Partial S-Method

definition *partial-s-method-via-h-framework-ts* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ ((*uint64*×*uint64*)×*bool*) *list list* **where**
partial-s-method-via-h-framework-ts M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-io-lists M additionalStates isAlreadyPrime* (λ *M m* . *partial-s-method-via-h-framework M m c b*)

lemma *partial-s-method-via-h-framework-ts-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ *list-all* (*passes-test-case M2* (*FSM.initial M2*)) (*partial-s-method-via-h-framework-M1 additionalStates isAlreadyPrime c b*)
using *apply-to-prime-and-return-io-lists-completeness*[**where** *f*=(λ *M m* . *partial-s-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF - assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
using *partial-s-method-via-h-framework-completeness-and-finiteness*
unfolding *partial-s-method-via-h-framework-ts-def*
by *metis*

definition *partial-s-method-via-h-framework-input* :: (*uint64*,*uint64*,*uint64*) *fsm* ⇒ *integer* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *uint64 list list* **where**
partial-s-method-via-h-framework-input M additionalStates isAlreadyPrime c b = *apply-to-prime-and-return-input-lists M additionalStates isAlreadyPrime* (λ *M m* . *partial-s-method-via-h-framework M m c b*)

lemma *partial-s-method-via-h-framework-input-completeness* :
assumes *observable M2*
and *minimal M2*
and *size M2* ≤ *size-r* (*to-prime M1*) + (*nat-of-integer additionalStates*)
and *FSM.inputs M2* = *FSM.inputs M1*
and *FSM.outputs M2* = *FSM.outputs M1*
and *isAlreadyPrime* ⟹ *observable M1* ∧ *minimal M1* ∧ *reachable-states M1* = *states M1*
and *size* (*to-prime M1*) < *2^64*
shows (*L M1* = *L M2*) ⟷ (∀ *xs*∈*list.set* (*partial-s-method-via-h-framework-input M1 additionalStates isAlreadyPrime c b*). ∀ *xs'*∈*list.set* (*prefixes xs*). {*io* ∈ *L M1*. *map fst io* = *xs'*} = {*io* ∈ *L M2*. *map fst io* = *xs'*})
using *apply-to-prime-and-return-input-lists-completeness*[**where** *f*=(λ *M m* . *partial-s-method-via-h-framework M m c b*) **and** *isAlreadyPrime*=*isAlreadyPrime*, *OF*

- *assms*(*1*,*2*,*3*,*4*,*5*,*6*,*7*)]
  using *partial-s-method-via-h-framework-completeness-and-finiteness*
  unfolding *partial-s-method-via-h-framework-input-def*[*symmetric*]
  by (*metis* (*no-types*, *lifting*))

46.3 New Instances

lemma *finiteness-fset-UNIV* : *finite* (*UNIV* :: *'a fset set*) = *finite* (*UNIV* :: *'a set*)
proof

  define *f* :: *'a* ⇒ (*'a*) *fset* **where** *f-def*: *f* = (λ *q* . {| *q* |})
  have *inj f*
  proof
    fix *x y* **assume** *x* ∈ (*UNIV* :: *'a set*) **and** *y* ∈ *UNIV* **and** *f x* = *f y*
    then show *x* = *y* **unfolding** *f-def* **by** (*transfer*; *auto*)
  qed

  show *finite* (*UNIV* :: *'a fset set*) ⟹ *finite* (*UNIV* :: *'a set*)
  proof (*rule ccontr*)
    assume *finite* (*UNIV* :: *'a fset set*) **and** ¬ *finite* (*UNIV* :: *'a set*)

    then have ¬ *finite* (*f* ‘ *UNIV*)
      using ‹*inj f*›
      using *finite-imageD* **by** *blast*
    then have ¬ *finite* (*UNIV* :: *'a fset set*)
      by (*meson infinite-iff-countable-subset top-greatest*)
    then show *False*
      using ‹*finite* (*UNIV* :: *'a fset set*)› **by** *auto*
  qed

  show *finite* (*UNIV* :: *'a set*) ⟹ *finite* (*UNIV* :: *'a fset set*)
  proof −
    assume *finite* (*UNIV* :: *'a set*)
    then have *finite* (*UNIV* :: *'a set set*)
      by (*simp add*: *Finite-Set.finite-set*)
    moreover have *fset* ‘ (*UNIV* :: *'a fset set*) ⊆ (*UNIV* :: *'a set set*)
      by *auto*
    moreover have *inj fset*
      by (*meson fset-inject injI*)
    ultimately show *?thesis* **by** (*metis inj-on-finite*)
  qed
qed

instantiation *fset* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a fset*) (*of-phantom* (*finite-UNIV* :: *'a finite-UNIV*))
instance by(*intro-classes*)(*simp add*: *finite-UNIV-fset-def finite-UNIV finiteness-fset-UNIV*)
end

```
derive (eq) ceq fset
derive (no) cenum fset
derive (no) ccompare fset
derive (dlist) set-impl fset

instantiation fset :: (type) cproper-interval begin
definition cproper-interval-fset :: (('a) fset) proper-interval
  where cproper-interval-fset - - = undefined
instance by(intro-classes)(simp add: ID-None ccompare-fset-def)
end

lemma card-fPow: card (Pow (fset A)) = 2 ^ card (fset A)
  using card-Pow[of fset A]
  by simp



lemma finite-sets-finite-univ :
  assumes finite (UNIV :: 'a set)
  shows finite (xs :: 'a set)
  by (metis Diff-UNIV Diff-infinite-finite assms finite-Diff)


lemma card-UNIV-fset: CARD('a fset) = (if CARD('a) = 0 then 0 else 2 ^
CARD('a))
  apply (simp add: card-eq-0-iff)
proof

  have inj fset
    by (meson fset-inject injI)
  have card (UNIV :: 'a fset set) = card (fset ` (UNIV :: 'a fset set))
    by (simp add: ‹inj fset› card-image)


  show finite (UNIV :: 'a set) ⟶ CARD('a fset) = 2 ^ CARD('a)
  proof
    assume finite (UNIV :: 'a set)
    then have CARD('a set) = 2 ^ CARD('a)
      by (metis Pow-UNIV card-Pow)

    have finite (UNIV :: 'a set set)
      using ‹finite (UNIV :: 'a set)›
      by (simp add: Finite-Set.finite-set)

    have finite (UNIV :: 'a fset set)
      using ‹finite (UNIV :: 'a set)› finiteness-fset-UNIV by auto


    have ⋀ xs :: 'a set . finite xs
```

```
      using finite-sets-finite-univ[OF ‹finite (UNIV :: 'a set)›] .
    then have (UNIV :: 'a set set) = fset ` (UNIV :: 'a fset set)
      by (metis UNIV-I UNIV-eq-I fset-to-fset image-iff)

    have CARD('a fset) ≤ CARD('a set)
      unfolding ‹card (UNIV :: 'a fset set) = card (fset ` (UNIV :: 'a fset set))›
      by (metis ‹finite (UNIV :: 'a set set)› card-mono subset-UNIV)
    moreover have CARD('a fset) ≥ CARD('a set)
      unfolding ‹(UNIV :: 'a set set) = fset ` (UNIV :: 'a fset set)›
      using ‹CARD('a::type fset) = card (range fset)› by linarith
    ultimately have CARD('a fset) = CARD('a set)
      by linarith
    then show CARD('a fset) = (2::nat) ^ CARD('a)
      by (simp add: ‹CARD('a::type set) = (2::nat) ^ CARD('a::type)›)
  qed

  show infinite (UNIV :: 'a set) ⟶ infinite (UNIV :: 'a fset set)
    by (simp add: finiteness-fset-UNIV)
qed

instantiation fset :: (card-UNIV) card-UNIV begin
definition card-UNIV = Phantom('a fset)
  (let c = of-phantom (card-UNIV :: 'a card-UNIV) in if c = 0 then 0 else 2 ^ c)
instance by intro-classes (simp add: card-UNIV-fset-def card-UNIV-fset card-UNIV)
end

derive (choose) mapping-impl fset

lemma uint64-range : range nat-of-uint64 = {..<2 ^ 64}
proof
  show {..<2 ^ 64} ⊆ range nat-of-uint64
    using uint64-nat-bij
    by (metis lessThan-iff range-eqI subsetI)
  have ⋀ x . nat-of-uint64 x < 2^64
    apply transfer using take-bit-nat-eq-self
    by (metis uint64.size-eq-length unsigned-less)
  then show range nat-of-uint64 ⊆ {..<2 ^ 64}
    by auto
qed

lemma card-UNIV-uint64: CARD(uint64) = 2^64
proof −
  have inj nat-of-uint64
    apply transfer
    by simp
  then have bij-betw nat-of-uint64 (UNIV :: uint64 set) {..<2^64}
    using uint64-range
    unfolding bij-betw-def by blast
  then show ?thesis
```

by (*simp add*: *bij-betw-same-card*)
qed

lemma *nat-of-uint64-bij-betw* : *bij-betw nat-of-uint64* (*UNIV* :: *uint64 set*) {..<2 ^ 64}
unfolding *bij-betw-def*
using *uint64-range*
by *transfer* (*auto*)

lemma *uint64-UNIV* : (*UNIV* :: *uint64 set*) = *uint64-of-nat* ‘ {..<2 ^ 64}
using *nat-of-uint64-bij-betw*
by (*metis UNIV-I UNIV-eq-I bij-betw-def card-UNIV-uint64 imageI image-eqI inj-on-contraD lessThan-iff rangeI uint64-nat-bij uint64-range*)

lemma *uint64-of-nat-bij-betw* : *bij-betw uint64-of-nat* {..<2 ^ 64} (*UNIV* :: *uint64 set*)
unfolding *bij-betw-def*
proof
show *inj-on uint64-of-nat* {..<2 ^ 64}
using *nat-of-uint64-bij-betw uint64-nat-bij*
by (*metis inj-on-inverseI lessThan-iff*)
show *uint64-of-nat* ‘ {..<2 ^ 64} = *UNIV*
using *uint64-UNIV* **by** *blast*
qed

lemma *uint64-finite* : *finite* (*UNIV* :: *uint64 set*)
unfolding *uint64-UNIV*
by *simp*

instantiation *uint64* :: *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*uint64*) *True*
instance apply *intro-classes*
by (*simp add*: *finite-UNIV-uint64-def uint64-finite*)
end

instantiation *uint64* :: *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*uint64*) (*2^64*)
instance
by *intro-classes* (*simp add*: *card-UNIV-uint64-def card-UNIV-uint64 card-UNIV*)
end

instantiation *uint64* :: *compare*
begin
definition *compare-uint64* :: *uint64* ⇒ *uint64* ⇒ *order* **where**

compare-uint64 x y = (*case* (*x* < *y*, *x* = *y*) *of* (*True*,-) ⇒ *Lt* | (*False*,*True*) ⇒ *Eq* | (*False*,*False*) ⇒ *Gt*)

instance
apply *intro-classes*
proof
show ⋀*x y*::*uint64*. *invert-order* (*compare x y*) = *compare y x*
proof −
fix *x y*::*uint64* **show** *invert-order* (*compare x y*) = *compare y x*
proof (*cases x* = *y*)
case *True*
then show *?thesis* **unfolding** *compare-uint64-def* **by** *auto*
next
case *False*
then show *?thesis* **proof** (*cases x* < *y*)
case *True*
then show *?thesis* **unfolding** *compare-uint64-def* **using** *False*
using *order-less-not-sym* **by** *fastforce*
next
case *False*
then show *?thesis* **unfolding** *compare-uint64-def* **using** ‹*x* ≠ *y*›
using *linorder-less-linear* **by** *fastforce*
qed
qed
qed

show ⋀*x y*::*uint64*. *compare x y* = *Eq* ⟹ *x* = *y*
unfolding *compare-uint64-def*
by (*metis* (*mono-tags*) *case-prod-conv old.bool.simps*(*3*) *old.bool.simps*(*4*) *order.distinct*(*1*) *order.distinct*(*3*))

show ⋀*x y z*::*uint64*. *compare x y* = *Lt* ⟹ *compare y z* = *Lt* ⟹ *compare x z* = *Lt*
unfolding *compare-uint64-def*
by (*metis* (*full-types*, *lifting*) *case-prod-conv old.bool.simps*(*3*) *old.bool.simps*(*4*) *order.distinct*(*5*) *order-less-trans*)
qed
end

instantiation *uint64* :: *ccompare*
begin
definition *ccompare-uint64* :: (*uint64* ⇒ *uint64* ⇒ *order*) *option* **where**
ccompare-uint64 = *Some compare*

instance by (*intro-classes*; *simp add*: *ccompare-uint64-def comparator-compare*)
end

derive (*eq*) *ceq uint64*
derive (*no*) *cenum uint64*

```
derive (rbt) set-impl uint64
derive (rbt) mapping-impl uint64


instantiation uint64 :: proper-interval begin
fun proper-interval-uint64 :: uint64 proper-interval
  where
    proper-interval-uint64 None None = True |
    proper-interval-uint64 None (Some y) = (y > 0)|
    proper-interval-uint64 (Some x) None = (x ≠ uint64-of-nat (2^64−1)) |
    proper-interval-uint64 (Some x) (Some y) = (x < y ∧ x+1 < y)

instance apply intro-classes
proof −
  show proper-interval None (None :: uint64 option) = True by auto

  show ⋀y. proper-interval None (Some (y::uint64)) = (∃ z. z < y)
    unfolding proper-interval-uint64.simps
    apply transfer
    using word-gt-a-gt-0 by auto

  have ⋀ x . (x ≠ uint64-of-nat (2^64−1)) = (nat-of-uint64 x ≠ 2^64−1)
  proof
    fix x
    show (x ≠ uint64-of-nat (2^64−1)) ⟹ (nat-of-uint64 x ≠ 2^64−1)
      apply transfer
      by (metis Word.of-nat-unat ucast-id)
    show nat-of-uint64 x ≠ 2 ^ 64 − 1 ⟹ x ≠ uint64-of-nat (2 ^ 64 − 1)
      by (meson diff-less pos2 uint64-nat-bij zero-less-one zero-less-power)
  qed
  then show ⋀x. proper-interval (Some (x::uint64)) None = (∃ z. x < z)
    unfolding proper-interval-uint64.simps
    apply transfer
   by (metis uint64.size-eq-length unat-minus-one-word word-le-less-eq word-le-not-less
word-order.extremum)

  show ⋀x y. proper-interval (Some x) (Some (y::uint64)) = (∃ z>x. z < y)
    unfolding proper-interval-uint64.simps
    apply transfer
    using inc-le less-is-non-zero-p1 word-overflow
    by fastforce
qed
end



instantiation uint64 :: cproper-interval begin
definition cproper-interval = (proper-interval :: uint64 proper-interval)
```

instance
apply *intro-classes*
apply (*simp add*: *cproper-interval-uint64-def ord-defs ccompare-uint64-def ID-Some proper-interval-class.axioms uint64-finite*)
proof

fix *x y* :: *uint64*

show *proper-interval None* (*None* :: *uint64 option*) = *True*
by *auto*

have (∃ *z*. *lt-of-comp compare z y*) = (∃ *z*. *z* < *y*)
unfolding *compare-uint64-def lt-of-comp-def*
by (*metis bool.case-eq-if case-prod-conv order.simps*(*7*) *order.simps*(*8*) *order.simps*(*9*))
then show *proper-interval None* (*Some y*) = (∃ *z*. *lt-of-comp compare z y*)
using *proper-interval-simps*(*2*) **by** *blast*

have (∃ *z*. *lt-of-comp compare x z*) = (∃ *z*. *x* < *z*)
unfolding *compare-uint64-def lt-of-comp-def*
by (*metis bool.case-eq-if case-prod-conv order.simps*(*7*) *order.simps*(*8*) *order.simps*(*9*))
then show *proper-interval* (*Some x*) *None* = (∃ *z*. *lt-of-comp compare x z*)
using *proper-interval-simps*(*3*) **by** *blast*

have (∃ *z*. *lt-of-comp compare x z* ∧ *lt-of-comp compare z y*) ⟹ (∃ *z*>*x*. *z* < *y*)
unfolding *compare-uint64-def lt-of-comp-def*
by (*metis bool.case-eq-if case-prod-conv order.simps*(*7*) *order.simps*(*9*))
moreover have (∃ *z*>*x*. *z* < *y*) ⟹ (∃ *z*. *lt-of-comp compare x z* ∧ *lt-of-comp compare z y*)
unfolding *compare-uint64-def lt-of-comp-def*
unfolding *proper-interval-simps*(*4*)[*symmetric*]
using *compare-uint64-def*
by (*metis* (*mono-tags*, *lifting*) ‹⋀*y x*. (∃ *z*>*x*. *z* < *y*) = *proper-interval* (*Some x*) (*Some y*)› *case-prod-conv old.bool.simps*(*3*) *order.simps*(*8*))
ultimately show *proper-interval* (*Some x*) (*Some y*) = (∃ *z*. *lt-of-comp compare x z* ∧ *lt-of-comp compare z y*)
using *proper-interval-simps*(*4*) **by** *blast*
qed
end

46.4 Exports

fun *fsm-from-list-uint64* :: *uint64* ⇒ (*uint64* × *uint64* × *uint64* × *uint64*) *list* ⇒ (*uint64*, *uint64*, *uint64*) *fsm*
where *fsm-from-list-uint64 q ts* = *fsm-from-list q ts*

fun *fsm-from-list-integer* :: *integer* ⇒ (*integer* × *integer* × *integer* × *integer*) *list* ⇒ (*integer*, *integer*, *integer*) *fsm*

where *fsm-from-list-integer q ts* = *fsm-from-list q ts*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts

spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *Haskell* **module-name** *GeneratedCode* **file-prefix** *haskell-export*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input

spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *Scala* **module-name** *GeneratedCode* **file-prefix** *scala-export* (*case-insensitive*)

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input
h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts

h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *SML* **module-name** *GeneratedCode* **file-prefix** *sml-export*

export-code *Inl*
fsm-from-list
fsm-from-list-uint64
fsm-from-list-integer
size
to-prime
make-observable
rename-states
index-states
restrict-to-reachable-states
integer-of-nat
generate-reduction-test-suite-naive
generate-reduction-test-suite-greedy
w-method-via-h-framework-ts
w-method-via-h-framework-input
w-method-via-h-framework-2-ts
w-method-via-h-framework-2-input
w-method-via-spy-framework-ts
w-method-via-spy-framework-input
w-method-via-pair-framework-ts
w-method-via-pair-framework-input
wp-method-via-h-framework-ts
wp-method-via-h-framework-input
wp-method-via-spy-framework-ts
wp-method-via-spy-framework-input
hsi-method-via-h-framework-ts
hsi-method-via-h-framework-input
hsi-method-via-spy-framework-ts
hsi-method-via-spy-framework-input
hsi-method-via-pair-framework-ts
hsi-method-via-pair-framework-input
h-method-via-h-framework-ts
h-method-via-h-framework-input
h-method-via-pair-framework-ts
h-method-via-pair-framework-input

h-method-via-pair-framework-2-ts
h-method-via-pair-framework-2-input
h-method-via-pair-framework-3-ts
h-method-via-pair-framework-3-input
spy-method-via-h-framework-ts
spy-method-via-h-framework-input
spy-method-via-spy-framework-ts
spy-method-via-spy-framework-input
spyh-method-via-h-framework-ts
spyh-method-via-h-framework-input
spyh-method-via-spy-framework-ts
spyh-method-via-spy-framework-input
partial-s-method-via-h-framework-ts
partial-s-method-via-h-framework-input
in *OCaml* **module-name** *GeneratedCode* **file-prefix** *ocaml-export*

end

References

[1] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–186, Mar. 1978.

[2] R. Dorofeeva, K. El-Fakih, and N. Yevtushenko. An improved conformance testing method. In F. Wang, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, volume 3731 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2005. ISBN 3-540-29189-X. doi: 10.1007/11562436_16. URL https://doi.org/10.1007/11562436_16.

[3] G. Luo, A. Petrenko, and G. v. Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. In T. Mizuno, T. Higashino, and N. Shiratori, editors, *Protocol Test Systems: 7th workshop 7th IFIP WG 6.1 international workshop on protocol text systems*, IFIP - The International Federation for Information Processing, pages 95–110. Springer US. ISBN 978-0-387-34883-4. doi: 10.1007/978-0-387-34883-4_6. URL https://doi.org/10.1007/978-0-387-34883-4_6.

[4] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. *IEEE Transactions on Software Engineering*, 20(2): 149–162, 1994. ISSN 0098-5589. doi: 10.1109/32.265636.

[5] J. Peleska and W.-l. Huang. *Test Automation - Foundations and Applications of Model-based Testing.* University of Bremen, January 2019.

Lecture notes, available under http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf.

[6] A. Petrenko and N. Yevtushenko. Adaptive testing of nondeterministic systems with FSM. In *15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014*, pages 224–228, 2014. doi: 10.1109/HASE.2014.39. URL http://dx.doi.org/10.1109/HASE.2014.39.

[7] R. Sachtleben. Formalisation of an adaptive state counting algorithm. *Archive of Formal Proofs*, Aug. 2019. ISSN 2150-914x. http://isa-afp.org/entries/Adaptive_State_Counting.html, Formal proof development.

[8] R. Sachtleben. An approach for the verification and synthesis of complete test generation algorithms for finite state machines. 2022. doi: 10.26092/elib/1665.

[9] R. Sachtleben, R. M. Hierons, W.-l. Huang, and J. Peleska. A mechanised proof of an adaptive state counting algorithm. In C. Gaston, N. Kosmatov, and P. Le Gall, editors, *Testing Software and Systems*, pages 176–193, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31280-0.

[10] A. Simão, A. Petrenko, and N. Yevtushenko. On reducing test length for FSMs with extra states. *Software Testing, Verification and Reliability*, 22(6):435–454, Sept. 2012. ISSN 1099-1689. doi: 10.1002/stvr.452. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.452.

[11] M. Soucha and K. Bogdanov. SPYH-method: An improvement in testing of finite-state machines. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 194–203, 2018. doi: 10.1109/ICSTW.2018.00050.