Formalisation of an Adaptive State Counting Algorithm

Robert Sachtleben

September 1, 2025

Abstract

This entry provides a formalisation of a refinement of an adaptive state counting algorithm, used to test for reduction between finite state machines. The algorithm has been originally presented by Hierons in [2] and was slightly refined by Sachtleben et al. in [3]. Definitions for finite state machines and adaptive test cases are given and many useful theorems are derived from these. The algorithm is formalised using mutually recursive functions, for which it is proven that the generated test suite is sufficient to test for reduction against finite state machines of a certain fault domain. Additionally, the algorithm is specified in a simple WHILE-language and its correctness is shown using Hoare-logic.

Contents

theory *FSM*
imports
Transition-Systems-and-Automata.Sequence-Zip
Transition-Systems-and-Automata.Transition-System
Transition-Systems-and-Automata.Transition-System-Extra
Transition-Systems-and-Automata.Transition-System-Construction
begin

1 Finite state machines

We formalise finite state machines as a 4-tuples, omitting the explicit formulation of the state set,as it can easily be calculated from the successor function. This definition does not require the successor function to be restricted to the input or output alphabet, which is later expressed by the property `well_formed`, together with the finiteness of the state set.

record (*'in*, *'out*, *'state*) *FSM* =
succ :: (*'in* × *'out*) ⇒ *'state* ⇒ *'state set*
inputs :: *'in set*
outputs :: *'out set*
initial :: *'state*

1.1 FSMs as transition systems

We interpret FSMs as transition systems with a singleton initial state set, based on [1].

global-interpretation *FSM* : *transition-system-initial*
λ *a p. snd a* — execute
λ *a p. snd a* ∈ *succ A* (*fst a*) *p* — enabled
λ *p. p* = *initial A* — initial
for *A*
defines *path* = *FSM.path*
and *run* = *FSM.run*
and *reachable* = *FSM.reachable*
and *nodes* = *FSM.nodes*
by *this*

abbreviation *size-FSM M* ≡ *card* (*nodes M*)
notation
size-FSM (‹(|-|)›)

1.2 Language

The following definitions establish basic notions for FSMs similarly to those of nondeterministic finite automata as defined in [1].

In particular, the language of an FSM state are the IO-parts of the paths in the FSM enabled from that state.

abbreviation *target* ≡ *FSM.target*
abbreviation *states* ≡ *FSM.states*

```
abbreviation trace ≡ FSM.trace

abbreviation successors :: ('in, 'out, 'state, 'more) FSM-scheme ⇒ 'state ⇒ 'state set where
  successors ≡ FSM.successors TYPE('in) TYPE('out) TYPE('more)

lemma states-alt-def: states r p = map snd r
  by (induct r arbitrary: p) (auto)
lemma trace-alt-def: trace r p = smap snd r
  by (coinduction arbitrary: r p) (auto)


definition language-state :: ('in, 'out, 'state) FSM ⇒ 'state
                          ⇒ ('in × 'out) list set (‹LS›)
  where
  language-state M q ≡ {map fst r |r . path M r q}
```

The language of an FSM is the language of its initial state.

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


lemma language-state-alt-def : LS M q = {io | io tr . path M (io || tr) q ∧ length io = length tr}
proof −
  have LS M q ⊆ { io | io tr . path M (io || tr) q ∧ length io = length tr }
  proof
    fix xr assume xr-assm : xr ∈ LS M q
    then obtain r where r-def : map fst r = xr path M r q
      unfolding language-state-def by auto
    then obtain xs ys where xr-split : xr = xs || ys
                                       length xs = length ys
                                       length xs = length xr
      by (metis length-map zip-map-fst-snd)
    then have (xs || ys) ∈ { io | io tr . path M (io || tr) q ∧ length io = length tr }
    proof −
      have f1: xs || ys = map fst r
        by (simp add: r-def(1) xr-split(1))
      then have f2: path M ((xs || ys) || take (min (length (xs || ys)) (length (map snd r)))
                                               (map snd r)) q
        by (simp add: r-def(2))
      have length (xs || ys) = length
                             (take (min (length (xs || ys)) (length (map snd r))) (map snd r))
        using f1 by force
      then show ?thesis
        using f2 by blast
    qed
    then show xr ∈ { io | io tr . path M (io || tr) q ∧ length io = length tr }
      using xr-split by metis
  qed
  moreover have { io | io tr . path M (io || tr) q ∧ length io = length tr } ⊆ LS M q
  proof
    fix xs assume xs-assm : xs ∈ { io | io tr . path M (io || tr) q ∧ length io = length tr }
    then obtain ys where ys-def : path M (xs || ys) q length xs = length ys
      by auto
    then have xs = map fst (xs || ys)
      by auto
    then show xs ∈ LS M q
      using ys-def unfolding language-state-def by blast
  qed
  ultimately show ?thesis
    by auto
qed


lemma language-state[intro]:
  assumes path M (w || r) q length w = length r
  shows w ∈ LS M q
  using assms unfolding language-state-def by force
```

```
lemma language-state-elim[elim]:
  assumes w ∈ LS M q
  obtains r
  where path M (w || r) q length w = length r
  using assms unfolding language-state-def by (force iff: split-zip-ex)

lemma language-state-split:
  assumes w1 @ w2 ∈ LS M q
  obtains tr1 tr2
  where path M (w1 || tr1) q length w1 = length tr1
        path M (w2 || tr2) (target (w1 || tr1) q) length w2 = length tr2
proof −
  obtain tr where tr-def : path M ((w1 @ w2) || tr) q length (w1 @ w2) = length tr
    using assms by blast
  let ?tr1 = take (length w1) tr
  let ?tr2 = drop (length w1) tr
  have tr-split : ?tr1 @ ?tr2 = tr
    by auto
  then show ?thesis
  proof −
    have f1: length w1 + length w2 = length tr
      using tr-def(2) by auto
    then have f2: length w2 = length tr − length w1
      by presburger
    then have length w1 = length (take (length w1) tr)
      using f1 by (metis (no-types) tr-split diff-add-inverse2 length-append length-drop)
    then show ?thesis
      using f2 by (metis (no-types) FSM.path-append-elim length-drop that tr-def(1) zip-append1)
  qed
qed

lemma language-state-prefix :
  assumes w1 @ w2 ∈ LS M q
shows w1 ∈ LS M q
  using assms by (meson language-state language-state-split)


lemma succ-nodes :
  fixes A :: ('a,'b,'c) FSM
  and   w :: ('a × 'b)
  assumes q2 ∈ succ A w q1
  and     q1 ∈ nodes A
shows q2 ∈ nodes A
proof −
  obtain x y where w = (x,y)
    by (meson surj-pair)
  then have q2 ∈ successors A q1
    using assms  by auto
  then have q2 ∈ reachable A q1
    by blast
  then have q2 ∈ reachable A (initial A)
    using assms by blast
  then show q2 ∈ nodes A
    by blast
qed

lemma states-target-index :
  assumes i < length p
  shows (states p q1) ! i = target (take (Suc i) p) q1
  using assms by auto
```

1.3 Product machine for language intersection

The following describes the construction of a product machine from two FSMs `M1` and `M2` such that the language of the product machine is the intersection of the language of `M1` and the language of `M2`.

definition *product* :: $('in, 'out, 'state1)$ *FSM* $\Rightarrow$ $('in, 'out, 'state2)$ *FSM* $\Rightarrow$
$('in, 'out, 'state1 \times 'state2)$ *FSM* **where**
product A B $\equiv$
$($|
succ = λ *a* (p_1, p_2). *succ A a* $p_1 \times$ *succ B a* p_2,
inputs = *inputs A* $\cup$ *inputs B*,
outputs = *outputs A* $\cup$ *outputs B*,
initial = (*initial A*, *initial B*)
$|)$

lemma *product-simps*[*simp*]:
succ (*product A B*) *a* (p_1, p_2) = *succ A a* $p_1 \times$ *succ B a* p_2
inputs (*product A B*) = *inputs A* $\cup$ *inputs B*
outputs (*product A B*) = *outputs A* $\cup$ *outputs B*
initial (*product A B*) = (*initial A*, *initial B*)
unfolding *product-def* **by** *simp+*

lemma *product-target*[*simp*]:
assumes *length w* = *length* r_1 *length* r_1 = *length* r_2
shows *target* $(w \mid\mid r_1 \mid\mid r_2)$ (p_1, p_2) = (*target* $(w \mid\mid r_1)$ p_1, *target* $(w \mid\mid r_2)$ p_2)
using *assms* **by** (*induct arbitrary*: p_1 p_2 *rule*: *list-induct3*) (*auto*)
lemma *product-path*[*iff*]:
assumes *length w* = *length* r_1 *length* r_1 = *length* r_2
shows *path* (*product A B*) $(w \mid\mid r_1 \mid\mid r_2)$ (p_1, p_2) $\longleftrightarrow$ *path A* $(w \mid\mid r_1)$ p_1 $\wedge$ *path B* $(w \mid\mid r_2)$ p_2
using *assms* **by** (*induct arbitrary*: p_1 p_2 *rule*: *list-induct3*) (*auto*)

lemma *product-language-state*[*simp*]: *LS* (*product A B*) (*q1*,*q2*) = *LS A q1* $\cap$ *LS B q2*
by (*fastforce iff*: *split-zip*)

lemma *product-nodes* :
nodes (*product A B*) $\subseteq$ *nodes A* $\times$ *nodes B*
proof
fix *q* **assume** *q* $\in$ *nodes* (*product A B*)
then show *q* $\in$ *nodes A* $\times$ *nodes B*
proof (*induction rule*: *FSM.nodes.induct*)
case (*initial p*)
then show *?case* **by** *auto*
next
case (*execute p a*)
then have *fst p* $\in$ *nodes A* *snd p* $\in$ *nodes B*
by *auto*

have *snd a* $\in$ (*succ A* (*fst a*) (*fst p*)) $\times$ (*succ B* (*fst a*) (*snd p*))
using *execute* **by** *auto*
then have *fst* (*snd a*) $\in$ *succ A* (*fst a*) (*fst p*)
snd (*snd a*) $\in$ *succ B* (*fst a*) (*snd p*)
by *auto*

have *fst* (*snd a*) $\in$ *nodes A*
using ‹*fst p* $\in$ *nodes A*› ‹*fst* (*snd a*) $\in$ *succ A* (*fst a*) (*fst p*)›
by (*metis FSM.nodes.simps fst-conv snd-conv*)
moreover have *snd* (*snd a*) $\in$ *nodes B*
using ‹*snd p* $\in$ *nodes B*› ‹*snd* (*snd a*) $\in$ *succ B* (*fst a*) (*snd p*)›
by (*metis FSM.nodes.simps fst-conv snd-conv*)
ultimately show *?case*
by (*simp add*: *mem-Times-iff*)

qed
qed

1.4 Required properties

FSMs used by the adaptive state counting algorithm are required to satisfy certain properties which are introduced in here. Most notably, the observability property (see function `observable`) implies the uniqueness of certain paths and hence allows for several stronger variations of previous results.

```
fun finite-FSM :: ('in, 'out, 'state) FSM ⇒ bool where
  finite-FSM M = (finite (nodes M)
                  ∧ finite (inputs M)
                  ∧ finite (outputs M))

fun observable :: ('in, 'out, 'state) FSM ⇒ bool where
  observable M = (∀ t . ∀ s1 . ((succ M) t s1 = {})
                              ∨ (∃ s2 . (succ M) t s1 = {s2}))

fun completely-specified :: ('in, 'out, 'state) FSM ⇒ bool where
  completely-specified M = (∀ s1 ∈ nodes M . ∀ x ∈ inputs M .
                              ∃ y ∈ outputs M .
                                ∃ s2 . s2 ∈ (succ M) (x,y) s1)

fun well-formed :: ('in, 'out, 'state) FSM ⇒ bool where
  well-formed M = (finite-FSM M
                   ∧ (∀ s1 x y . (x ∉ inputs M ∨ y ∉ outputs M)
                                 ⟶ succ M (x,y) s1 = {})
                   ∧ inputs M ≠ {}
                   ∧ outputs M ≠ {})

abbreviation OFSM M ≡ well-formed M
                      ∧ observable M
                      ∧ completely-specified M

lemma OFSM-props[elim!] :
  assumes OFSM M
shows well-formed M
      observable M
      completely-specified M using assms by auto

lemma set-of-succs-finite :
  assumes well-formed M
  and     q ∈ nodes M
shows finite (succ M io q)
proof (rule ccontr)
  assume infinite (succ M io q)
  moreover have succ M io q ⊆ nodes M
    using assms by (simp add: subsetI succ-nodes)
  ultimately have infinite (nodes M)
    using infinite-super by blast
  then show False
    using assms by auto
qed

lemma well-formed-path-io-containment :
  assumes well-formed M
  and     path M p q
shows set (map fst p) ⊆ (inputs M × outputs M)
using assms proof (induction p arbitrary: q)
case Nil
  then show ?case by auto
next
  case (Cons a p)
  have fst a ∈ (inputs M × outputs M)
  proof (rule ccontr)
    assume fst a ∉ inputs M × outputs M
    then have fst (fst a) ∉ inputs M ∨ snd (fst a) ∉ outputs M
      by (metis SigmaI prod.collapse)
    then have succ M (fst a) q = {}
      using Cons by (metis prod.collapse well-formed.elims(2))
    moreover have (snd a) ∈ succ M (fst a) q
      using Cons by auto
    ultimately show False
      by auto
  qed
```

```
  moreover have set (map fst p) ⊆ (inputs M × outputs M)
    using Cons by blast
  ultimately show ?case
    by auto
qed


lemma path-input-containment :
  assumes well-formed M
  and     path M p q
shows set (map fst (map fst p)) ⊆ inputs M
using assms proof (induction p arbitrary: q rule: rev-induct)
  case Nil
  then show ?case by auto
next
  case (snoc a p)
  have set (map fst (p @ [a])) ⊆ (inputs M × outputs M)
    using well-formed-path-io-containment[OF snoc.prems] by assumption
  then have (fst a) ∈ (inputs M × outputs M)
    by auto
  then have fst (fst a) ∈ inputs M
    by auto
  moreover have set (map fst (map fst p)) ⊆ inputs M
    using snoc.IH[OF snoc.prems(1)]
    using snoc.prems(2) by blast
  ultimately show ?case
    by simp
qed

lemma path-state-containment :
  assumes path M p q
  and     q ∈ nodes M
shows set (map snd p) ⊆ nodes M
  using assms by (metis FSM.nodes-states states-alt-def)


lemma language-state-inputs :
  assumes well-formed M
  and     io ∈ language-state M q
shows set (map fst io) ⊆ inputs M
proof −
  obtain tr where path M (io || tr) q length tr = length io
    using assms(2) by auto
  show ?thesis
    by (metis (no-types)
        ‹⋀thesis. (⋀tr. ⟦path M (io || tr) q; length tr = length io⟧ ⟹ thesis) ⟹ thesis›
        assms(1) map-fst-zip path-input-containment)
qed


lemma set-of-paths-finite :
  assumes well-formed M
  and     q1 ∈ nodes M
shows finite { p . path M p q1 ∧ target p q1 = q2 ∧ length p ≤ k }
proof −

  let ?trs = { tr . set tr ⊆ nodes M ∧ length tr ≤ k }
  let ?ios = { io . set io ⊆ inputs M × outputs M ∧ length io ≤ k }
  let ?iotrs = image (λ (io,tr) . io || tr) (?ios × ?trs)

  let ?paths = { p . path M p q1 ∧ target p q1 = q2 ∧ length p ≤ k }

  have finite (inputs M × outputs M)
    using assms by auto
  then have finite ?ios
    using assms by (simp add: finite-lists-length-le)
```

```
  moreover have finite ?trs
    using assms by (simp add: finite-lists-length-le)
  ultimately have finite ?iotrs
    by auto

  moreover have ?paths ⊆ ?iotrs
  proof
    fix p assume p-assm : p ∈ { p . path M p q1 ∧ target p q1 = q2 ∧ length p ≤ k }
    then obtain io tr where p-split : p = io || tr ∧ length io = length tr
      using that by (metis (no-types) length-map zip-map-fst-snd)
    then have io ∈ ?ios
      using well-formed-path-io-containment
    proof −
      have f1: path M p q1 ∧ target p q1 = q2 ∧ length p ≤ k
        using p-assm by force
      then have set io ⊆ inputs M × outputs M
        by (metis (no-types) assms(1) map-fst-zip p-split well-formed-path-io-containment)
      then show ?thesis
        using f1 by (simp add: p-split)
    qed

    moreover have tr ∈ ?trs using p-split
    proof −
      have f1: path M (io || tr) q1 ∧ target (io || tr) q1 = q2
                ∧ length (io || tr) ≤ k using ‹p ∈ {p. path M p q1
                ∧ target p q1 = q2 ∧ length p ≤ k}› p-split by force
      then have f2: length tr ≤ k by (simp add: p-split)
      have set tr ⊆ nodes M
        using f1 by (metis (no-types) assms(2) length-map p-split path-state-containment
                      zip-eq zip-map-fst-snd)
      then show ?thesis
        using f2 by blast
    qed
    ultimately show p ∈ ?iotrs
      using p-split by auto
  qed

  ultimately show ?thesis
    using Finite-Set.finite-subset by blast
qed


lemma non-distinct-duplicate-indices :
  assumes ¬ distinct xs
shows ∃ i1 i2 . i1 ≠ i2 ∧ xs ! i1 = xs ! i2 ∧ i1 ≤ length xs ∧ i2 ≤ length xs
  using assms by (meson distinct-conv-nth less-imp-le)

lemma reaching-path-without-repetition :
  assumes well-formed M
  and     q2 ∈ reachable M q1
  and     q1 ∈ nodes M
shows ∃ p . path M p q1 ∧ target p q1 = q2 ∧ distinct (q1 # states p q1)
proof −
  have shorten-nondistinct : ∀ p. (path M p q1 ∧ target p q1 = q2 ∧ ¬ distinct (q1 # states p q1))
            ⟶ (∃ p' . path M p' q1 ∧ target p' q1 = q2 ∧ length p' < length p)
  proof
    fix p
    show (path M p q1 ∧ target p q1 = q2 ∧ ¬ distinct (q1 # states p q1))
            ⟶ (∃ p' . path M p' q1 ∧ target p' q1 = q2 ∧ length p' < length p)
    proof
      assume assm : path M p q1 ∧ target p q1 = q2 ∧ ¬ distinct (q1 # states p q1)
      then show (∃p'. path M p' q1 ∧ target p' q1 = q2 ∧ length p' < length p)
      proof (cases q1 ∈ set (states p q1))
        case True
        have ∃ i1 . target (take i1 p) q1 = q1 ∧ i1 ≤ length p ∧ i1 > 0
        proof (rule ccontr)
```

```
      assume ¬ (∃ i1. target (take i1 p) q1 = q1 ∧ i1 ≤ length p ∧ i1 > 0)
      then have ¬ (∃ i1 . (states p q1) ! i1 = q1 ∧ i1 ≤ length (states p q1))
        by (metis True in-set-conv-nth less-eq-Suc-le scan-length scan-nth zero-less-Suc)
      then have q1 ∉ set (states p q1)
        by (meson in-set-conv-nth less-imp-le)
      then show False
        using True by auto
    qed
    then obtain i1 where i1-def : target (take i1 p) q1 = q1 ∧ i1 ≤ length p ∧ i1 > 0
      by auto

    then have path M (take i1 p) q1
      using assm by (metis FSM.path-append-elim append-take-drop-id)
    moreover have path M (drop i1 p) q1
      using i1-def by (metis FSM.path-append-elim append-take-drop-id assm)
    ultimately have path M (drop i1 p) q1 ∧ (target (drop i1 p) q1 = q2)
      using i1-def by (metis (no-types) append-take-drop-id assm fold-append o-apply)

    moreover have length (drop i1 p) < length p
      using i1-def by auto
    ultimately show ?thesis
      using assms by blast

  next
    case False
    then have assm' : path M p q1 ∧ target p q1 = q2 ∧ ¬ distinct (states p q1)
      using assm by auto

    have ∃ i1 i2 . i1 ≠ i2 ∧ target (take i1 p) q1 = target (take i2 p) q1
                ∧ i1 ≤ length p ∧ i2 ≤ length p
    proof (rule ccontr)
      assume ¬ (∃ i1 i2 . i1 ≠ i2 ∧ target (take i1 p) q1 = target (take i2 p) q1
                  ∧ i1 ≤ length p ∧ i2 ≤ length p)
      then have ¬ (∃ i1 i2 . i1 ≠ i2 ∧ (states p q1) ! i1 = (states p q1) ! i2
                  ∧ i1 ≤ length (states p q1) ∧ i2 ≤ length (states p q1))
        by (metis (no-types, lifting) Suc-leI assm' distinct-conv-nth nat.inject
            scan-length scan-nth)

      then have distinct (states p q1)
        using non-distinct-duplicate-indices by blast
      then show False
        using assm' by auto
    qed
    then obtain i1 i2 where i-def : i1 < i2 ∧ target (take i1 p) q1 = target (take i2 p) q1
                                 ∧ i1 ≤ length p ∧ i2 ≤ length p
      by (metis nat-neq-iff)

    then have path M (take i1 p) q1
      using assm by (metis FSM.path-append-elim append-take-drop-id)
    moreover have path M (drop i2 p) (target (take i2 p) q1)
      by (metis FSM.path-append-elim append-take-drop-id assm)
    ultimately have path M ((take i1 p) @ (drop i2 p)) q1
                    ∧ (target ((take i1 p) @ (drop i2 p)) q1 = q2)
      using i-def assm
      by (metis FSM.path-append append-take-drop-id fold-append o-apply)

    moreover have length ((take i1 p) @ (drop i2 p)) < length p
      using i-def by auto

    ultimately have path M ((take i1 p) @ (drop i2 p)) q1
                    ∧ target ((take i1 p) @ (drop i2 p)) q1 = q2
                    ∧ length ((take i1 p) @ (drop i2 p)) < length p
      by simp

    then show ?thesis
      using assms by blast
```

```
      qed
    qed
  qed

  obtain p where p-def : path M p q1 ∧ target p q1 = q2
    using assms by auto

  let ?paths = {p' . (path M p' q1 ∧ target p' q1 = q2 ∧ length p' ≤ length p)}
  let ?minPath = arg-min length (λ io . io ∈ ?paths)

  have ?paths ≠ empty
    using p-def by auto
  moreover have finite ?paths
    using assms by (simp add: set-of-paths-finite)
  ultimately have minPath-def : ?minPath ∈ ?paths ∧ (∀ p' ∈ ?paths . length ?minPath ≤ length p')
    by (meson arg-min-nat-lemma equals0I)

  moreover have distinct (q1 # states ?minPath q1)
  proof (rule ccontr)
    assume ¬ distinct (q1 # states ?minPath q1)
    then have ∃ p' . path M p' q1 ∧ target p' q1 = q2 ∧ length p' < length ?minPath
      using shorten-nondistinct minPath-def by blast
    then show False
      using minPath-def using arg-min-nat-le dual-order.strict-trans1 by auto
  qed

  ultimately show ?thesis by auto
qed





lemma observable-path-unique[simp] :
  assumes io ∈ LS M q
  and     observable M
  and     path M (io || tr1) q length io = length tr1
  and     path M (io || tr2) q length io = length tr2
shows tr1 = tr2
proof (rule ccontr)
  assume tr-assm : tr1 ≠ tr2
  then have state-diff : (states (io || tr1) q ) ≠ (states (io || tr2) q)
    by (metis assms(4) assms(6) map-snd-zip states-alt-def)
  show False
  using assms tr-assm proof (induction io arbitrary: q tr1 tr2)
    case Nil
    then show ?case using Nil
      by simp
  next
    case (Cons io-hd io-tl)
    then obtain tr1-hd tr1-tl tr2-hd tr2-tl where tr-split : tr1 = tr1-hd # tr1-tl
                                                             ∧ tr2 = tr2-hd # tr2-tl
      by (metis length-0-conv neq-Nil-conv)

    have p1: path M ([io-hd] || [tr1-hd]) q
      using Cons.prems tr-split by auto
    have p2: path M ([io-hd] || [tr2-hd]) q
      using Cons.prems tr-split by auto
    have tr-hd-eq : tr1-hd = tr2-hd
      using Cons.prems unfolding observable.simps
    proof −
      assume ∀ t s1. succ M t s1 = {} ∨ (∃ s2. succ M t s1 = {s2})
      then show ?thesis
        by (metis (no-types) p1 p2 FSM.path-cons-elim empty-iff prod.sel(1) prod.sel(2) singletonD
            zip-Cons-Cons)
    qed
```

```
    then show ?thesis
      using Cons.IH Cons.prems(3) Cons.prems(4) Cons.prems(5) Cons.prems(6) Cons.prems(7) assms(2)
            tr-split by auto
  qed
qed


lemma observable-path-unique-ex[elim] :
  assumes observable M
  and     io ∈ LS M q
obtains tr
where { t . path M (io || t) q ∧ length io = length t } = { tr }
proof −
  obtain tr where tr-def : path M (io || tr) q length io = length tr
    using assms by auto
  then have { t . path M (io || t) q ∧ length io = length t } ≠ {}
    by blast
  moreover have ∀ t ∈ { t . path M (io || t) q ∧ length io = length t } . t = tr
    using assms tr-def by auto
  ultimately show ?thesis
    using that by auto
qed

lemma well-formed-product[simp] :
  assumes well-formed M1
  and     well-formed M2
shows well-formed (product M2 M1) (is well-formed ?PM)
unfolding well-formed.simps proof
  have finite (nodes M1) finite (nodes M2)
    using assms by auto
  then have finite (nodes M2 × nodes M1)
    by simp

  moreover have nodes ?PM ⊆ nodes M2 × nodes M1
    using product-nodes assms by blast
  ultimately show finite-FSM ?PM
    using infinite-subset assms by auto
next
  have inputs ?PM = inputs M2 ∪ inputs M1
       outputs ?PM = outputs M2 ∪ outputs M1
    by auto
  then show (∀ s1 x y. x ∉ inputs ?PM ∨ y ∉ outputs ?PM ⟶ succ ?PM (x, y) s1 = {})
                                                  ∧ inputs ?PM ≠ {} ∧ outputs ?PM ≠ {}
    using assms by auto
qed
```

1.5 States reached by a given IO-sequence

Function `io_targets` collects all states of an FSM reached from a given state by a given IO-sequence. Notably, for any observable FSM, this set contains at most one state.

```
fun io-targets :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ ('in × 'out) list ⇒ 'state set where
  io-targets M q io = { target (io || tr) q | tr . path M (io || tr) q ∧ length io = length tr }

lemma io-target-implies-L :
  assumes q ∈ io-targets M (initial M) io
  shows io ∈ L M
proof −
  obtain tr where path M (io || tr) (initial M)
                  length tr = length io
                  target (io || tr) (initial M) = q
    using assms by auto
  then show ?thesis by auto
qed
```

lemma *io-target-from-path* :
  assumes *path M* (*w* || *tr*) *q*
  and *length w* = *length tr*
shows *target* (*w* || *tr*) *q* ∈ *io-targets M q w*
  using *assms* **by** *auto*

lemma *io-targets-observable-singleton-ex* :
  assumes *observable M*
  and *io* ∈ *LS M q1*
shows ∃ *q2* . *io-targets M q1 io* = { *q2* }
proof −
  obtain *tr* **where** *tr-def* : { *t* . *path M* (*io* || *t*) *q1* ∧ *length io* = *length t* } = { *tr* }
    using *assms observable-path-unique-ex* **by** (*metis* (*mono-tags*, *lifting*))
  then have *io-targets M q1 io* = { *target* (*io* || *tr*) *q1* }
    by *fastforce*
  then show *?thesis*
    by *blast*
qed

lemma *io-targets-observable-singleton-ob* :
  assumes *observable M*
  and *io* ∈ *LS M q1*
obtains *q2*
  where *io-targets M q1 io* = { *q2* }
proof −
  obtain *tr* **where** *tr-def* : { *t* . *path M* (*io* || *t*) *q1* ∧ *length io* = *length t* } = { *tr* }
    using *assms observable-path-unique-ex* **by** (*metis* (*mono-tags*, *lifting*))
  then have *io-targets M q1 io* = { *target* (*io* || *tr*) *q1* }
    by *fastforce*
  then show *?thesis* **using** *that* **by** *blast*
qed

lemma *io-targets-elim*[*elim*] :
  assumes *p* ∈ *io-targets M q io*
obtains *tr*
where *target* (*io* || *tr*) *q* = *p* ∧ *path M* (*io* || *tr*) *q* ∧ *length io* = *length tr*
  using *assms* **unfolding** *io-targets.simps* **by** *force*

lemma *io-targets-reachable* :
  assumes *q2* ∈ *io-targets M q1 io*
  shows *q2* ∈ *reachable M q1*
  using *assms* **unfolding** *io-targets.simps* **by** *blast*

lemma *io-targets-nodes* :
  assumes *q2* ∈ *io-targets M q1 io*
  and *q1* ∈ *nodes M*
shows *q2* ∈ *nodes M*
  using *assms* **by** *auto*

lemma *observable-io-targets-split* :
  assumes *observable M*
  and *io-targets M q1* (*vs* @ *xs*) = {*q3*}
  and *io-targets M q1 vs* = {*q2*}
shows *io-targets M q2 xs* = {*q3*}
proof −
  have *vs* @ *xs* ∈ *LS M q1*
    using *assms*(*2*) **by** *force*
  then obtain *trV trX* **where** *tr-def* :
      path M (*vs* || *trV*) *q1 length vs* = *length trV*
      path M (*xs* || *trX*) (*target* (*vs* || *trV*) *q1*) *length xs* = *length trX*
    using *language-state-split*[*of vs xs M q1*] **by** *auto*
  then have *tgt-V* : *target* (*vs* || *trV*) *q1* = *q2*
    using *assms*(*3*) **by** *auto*
  then have *path-X* : *path M* (*xs* || *trX*) *q2* ∧ *length xs* = *length trX*

    using *tr-def* **by** *auto*

  have *tgt-all* : *target* (*vs* @ *xs* || *trV* @ *trX*) *q1* = *q3*
  proof −
    have *f1*: ∃ *cs*. *q3* = *target* (*vs* @ *xs* || *cs*) *q1*
                ∧ *path M* (*vs* @ *xs* || *cs*) *q1* ∧ *length* (*vs* @ *xs*) = *length cs*
      using *assms*(*2*) **by** *auto*
    have *length* (*vs* @ *xs*) = *length trV* + *length trX*
      by (*simp add*: *tr-def*(*2*) *tr-def*(*4*))
    then have *length* (*vs* @ *xs*) = *length* (*trV* @ *trX*)
      by *simp*
    then show *?thesis*
      using *f1* **by** (*metis FSM.path-append* ‹*vs* @ *xs* ∈ *LS M q1*› *assms*(*1*) *observable-path-unique*
                tr-def(*1*) *tr-def*(*2*) *tr-def*(*3*) *zip-append*)
  qed
  then have *target* ((*vs* || *trV*) @ (*xs* || *trX*)) *q1* = *q3*
    using *tr-def* **by** *simp*
  then have *target* (*xs* || *trX*) *q2* = *q3*
    using *tgt-V* **by** *auto*
  then have *q3* ∈ *io-targets M q2 xs*
    using *path-X* **by** *auto*
  then show *?thesis*
    by (*metis* (*no-types*) ‹*observable M*› *path-X insert-absorb io-targets-observable-singleton-ex*
      language-state singleton-insert-inj-eq′)
qed

lemma *observable-io-target-unique-target* :
  assumes *observable M*
  and   *io-targets M q1 io* = {*q2*}
  and   *path M* (*io* || *tr*) *q1*
  and   *length io* = *length tr*
shows *target* (*io* || *tr*) *q1* = *q2*
  using *assms* **by** *auto*

lemma *target-in-states* :
  assumes *length io* = *length tr*
  and   *length io* > *0*
  shows *last* (*states* (*io* || *tr*) *q*) = *target* (*io* || *tr*) *q*
proof −
  have *0* < *length tr*
    using *assms*(*1*) *assms*(*2*) **by** *presburger*
  then show *?thesis*
    by (*simp add*: *FSM.target-alt-def assms*(*1*) *states-alt-def*)
qed

lemma *target-alt-def* :
  assumes *length io* = *length tr*
  shows *length io* = *0* ⟹ *target* (*io* || *tr*) *q* = *q*
      length io > *0* ⟹ *target* (*io* || *tr*) *q* = *last tr*
proof −
  show *length io* = *0* ⟹ *target* (*io* || *tr*) *q* = *q* **by** *simp*
  show *length io* > *0* ⟹ *target* (*io* || *tr*) *q* = *last tr*
    by (*metis assms last-ConsR length-greater-0-conv map-snd-zip scan-last states-alt-def*)
qed

lemma *obs-target-is-io-targets* :
  assumes *observable M*
  and   *path M* (*io* || *tr*) *q*
  and   *length io* = *length tr*
shows *io-targets M q io* = {*target* (*io* || *tr*) *q*}
  by (*metis assms*(*1*) *assms*(*2*) *assms*(*3*) *io-targets-observable-singleton-ex language-state*
    observable-io-target-unique-target)

```
lemma io-target-target :
  assumes io-targets M q1 io = {q2}
  and     path M (io || tr) q1
  and     length io = length tr
shows target (io || tr) q1 = q2
proof −
  have target (io || tr) q1 ∈ io-targets M q1 io using assms(2) assms(3) by auto
  then show ?thesis using assms(1) by blast
qed


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

lemma path-last-io-target :
  assumes path M (xs || tr) q
  and     length xs = length tr
  and     length xs > 0
shows last tr ∈ io-targets M q xs
proof −
  have last tr = target (xs || tr) q
    by (metis assms(2) assms(3) map-snd-zip states-alt-def target-in-states)
  then show ?thesis using assms(1) assms(2) by auto
qed


lemma path-prefix-io-targets :
  assumes path M (xs || tr) q
  and     length xs = length tr
  and     length xs > 0
shows last (take (Suc i) tr) ∈ io-targets M q (take (Suc i) xs)
proof −
  have path M (take (Suc i) xs || take (Suc i) tr) q
    by (metis (no-types) FSM.path-append-elim append-take-drop-id assms(1) take-zip)
  then show ?thesis
    using assms(2) assms(3) path-last-io-target by fastforce
qed


lemma states-index-io-target :
  assumes i < length xs
  and     path M (xs || tr) q
  and     length xs = length tr
  and     length xs > 0
shows (states (xs || tr) q) ! i ∈ io-targets M q (take (Suc i) xs)
proof −
  have (states (xs || tr) q) ! i = last (take (Suc i) (states (xs || tr) q))
    by (metis assms(1) assms(3) map-snd-zip states-alt-def index-last-take)
  then have (states (xs || tr) q) ! i = last (states (take (Suc i) xs || take (Suc i) tr) q)
    by (simp add: take-zip)
  then have (states (xs || tr) q) ! i = last (take (Suc i) tr)
    by (simp add: assms(3) states-alt-def)
  moreover have last (take (Suc i) tr) ∈ io-targets M q (take (Suc i) xs)
    by (meson assms(2) assms(3) assms(4) path-prefix-io-targets)
  ultimately show ?thesis
    by simp
qed


lemma observable-io-targets-append :
  assumes observable M
  and io-targets M q1 vs = {q2}
  and io-targets M q2 xs = {q3}
```

```
shows io-targets M q1 (vs@xs) = {q3}
proof -
  obtain trV where path M (vs || trV) q1 ∧ length trV = length vs ∧ target (vs || trV) q1 = q2
    by (metis assms(2) io-targets-elim singletonI)
  moreover obtain trX where path M (xs || trX) q2 ∧ length trX = length xs
                             ∧ target (xs || trX) q2 = q3
    by (metis assms(3) io-targets-elim singletonI)
  ultimately have path M (vs @ xs || trV @ trX) q1 ∧ length (trV @ trX) = length (vs @ xs)
                   ∧ target (vs @ xs || trV @ trX) q1 = q3
    by auto
  then show ?thesis
    by (metis assms(1) obs-target-is-io-targets)
qed


lemma io-path-states-prefix :
  assumes observable M
  and path M (io1 || tr1) q
  and length tr1 = length io1
  and path M (io2 || tr2) q
  and length tr2 = length io2
  and prefix io1 io2
shows tr1 = take (length tr1) tr2
proof -
  let ?tr1' = take (length tr1) tr2
  let ?io1' = take (length tr1) io2
  have path M (?io1' || ?tr1') q
    by (metis FSM.path-append-elim append-take-drop-id assms(4) take-zip)
  have length ?tr1' = length ?io1'
    using assms (5) by auto

  have ?io1' = io1
  proof -
    have ∀ps psa. ¬ prefix (ps::('a × 'b) list) psa ∨ length ps ≤ length psa
      using prefix-length-le by blast
    then have length (take (length tr1) io2) = length io1
      using assms(3) assms(6) min.absorb2 by auto
    then show ?thesis
      by (metis assms(6) min.cobounded2 min-def-raw prefix-length-prefix
          prefix-order.dual-order.antisym take-is-prefix)
  qed

  show tr1 = ?tr1'
    by (metis ‹length (take (length tr1) tr2) = length (take (length tr1) io2)›
        ‹path M (take (length tr1) io2 || take (length tr1) tr2) q› ‹take (length tr1) io2 = io1›
        assms(1) assms(2) assms(3) language-state observable-path-unique)
qed



lemma observable-io-targets-suffix :
  assumes observable M
  and io-targets M q1 vs = {q2}
  and io-targets M q1 (vs@xs) = {q3}
shows io-targets M q2 xs = {q3}
proof -
  have prefix vs (vs@xs)
    by auto

  obtain trV where path M (vs || trV) q1 ∧ length trV = length vs ∧ target (vs || trV) q1 = q2
    by (metis assms(2) io-targets-elim singletonI)
  moreover obtain trVX where path M (vs@xs || trVX) q1
                              ∧ length trVX = length (vs@xs) ∧ target (vs@xs || trVX) q1 = q3
    by (metis assms(3) io-targets-elim singletonI)
```

ultimately have *trV* = *take* (*length trV*) *trVX*
using *io-path-states-prefix*[*OF assms*(*1*) - - - - ‹*prefix vs* (*vs*@*xs*)›, *of trV q1 trVX*] **by** *auto*
show *?thesis*
by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *observable-io-targets-split*)
qed

lemma *observable-io-target-is-singleton*[*simp*] :
assumes *observable M*
and $p \in$ *io-targets M q io*
shows *io-targets M q io* = {*p*}
proof −
have *io* ∈ *LS M q*
using *assms*(*2*) **by** *auto*
then obtain *p*′ **where** *io-targets M q io* = {*p*′}
using *assms*(*1*) **by** (*meson io-targets-observable-singleton-ex*)
then show *?thesis*
using *assms*(*2*) **by** *simp*
qed

lemma *observable-path-prefix* :
assumes *observable M*
and *path M* (*io* || *tr*) *q*
and *length io* = *length tr*
and *path M* (*ioP* || *trP*) *q*
and *length ioP* = *length trP*
and *prefix ioP io*
shows *trP* = *take* (*length ioP*) *tr*
proof −
have *ioP-def* : *ioP* = *take* (*length ioP*) *io*
using *assms*(*6*) **by** (*metis append-eq-conv-conj prefixE*)
then have *take* (*length ioP*) (*io* || *tr*) = *take* (*length ioP*) *io* || *take* (*length ioP*) *tr*
using *take-zip* **by** *blast*
moreover have *path M* (*take* (*length ioP*) (*io* || *tr*)) *q*
using *assms* **by** (*metis FSM.path-append-elim append-take-drop-id*)
ultimately have *path M* (*take* (*length ioP*) *io* || *take* (*length ioP*) *tr*) *q*
∧ *length* (*take* (*length ioP*) *io*) = *length* (*take* (*length ioP*) *tr*)
using *assms*(*3*) **by** *auto*
then have *path M* (*ioP* || *take* (*length ioP*) *tr*) *q* ∧ *length ioP* = *length* (*take* (*length ioP*) *tr*)
using *assms*(*3*) **using** *ioP-def* **by** *auto*
then show *?thesis*
by (*meson assms*(*1*) *assms*(*4*) *assms*(*5*) *language-state observable-path-unique*)
qed

lemma *io-targets-succ* :
assumes *q2* ∈ *io-targets M q1* [*xy*]
shows *q2* ∈ *succ M xy q1*
proof −
obtain *tr* **where** *tr-def* : *target* ([*xy*] || *tr*) *q1* = *q2*
path M ([*xy*] || *tr*) *q1*
length [*xy*] = *length tr*
using *assms* **by** *auto*

have *length tr* = *Suc 0*
using ‹*length* [*xy*] = *length tr*› **by** *auto*
then obtain *q2*′ **where** *tr* = [*q2*′]
by (*metis Suc-length-conv length-0-conv*)
then have *target* ([*xy*] || *tr*) *q1* = *q2*′
by *auto*
then have *q2*′ = *q2*
using ‹*target* ([*xy*] || *tr*) *q1* = *q2*› **by** *simp*
then have *path M* ([*xy*] || [*q2*]) *q1*
using *tr-def*(*2*) ‹*tr* = [*q2*′]› **by** *auto*
then have *path M* [(*xy*,*q2*)] *q1*

```
    by auto

  show ?thesis
  proof (cases rule: FSM.path.cases[of M [(xy,q2)] q1])
    case nil
    show ?case
      using ‹path M [(xy,q2)] q1› by simp
  next
    case cons
    show snd (xy, q2) ∈ succ M (fst (xy, q2)) q1 ⟹ path M [] (snd (xy, q2))
            ⟹ q2 ∈ succ M xy q1
      by auto
  qed
qed
```

1.6 D-reachability

A state of some FSM is d-reached (deterministically reached) by some input sequence if any sequence in the language of the FSM with this input sequence reaches that state. That state is then called d-reachable.

```
abbreviation  d-reached-by M p xs q tr ys ≡
                 ((length xs = length ys ∧ length xs = length tr
                 ∧ (path M ((xs || ys) || tr) p) ∧ target ((xs || ys) || tr) p = q)
                 ∧ (∀ ys2 tr2 .  (length xs = length ys2 ∧ length xs = length tr2
                 ∧ path M ((xs || ys2) || tr2) p) ⟶ target ((xs || ys2) || tr2) p = q))

fun d-reaches :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ 'in list ⇒ 'state ⇒ bool where
  d-reaches M p xs q = (∃ tr ys . d-reached-by M p xs q tr ys)

fun d-reachable :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ 'state set where
  d-reachable M p = { q . (∃ xs . d-reaches M p xs q) }


lemma d-reaches-unique[elim] :
  assumes d-reaches M p xs q1
  and     d-reaches M p xs q2
shows q1 = q2
using assms unfolding d-reaches.simps by blast

lemma d-reaches-unique-cases[simp] : { q . d-reaches M (initial M) xs q } = {}
                                  ∨ (∃ q2 . { q . d-reaches M (initial M) xs q } = { q2 })
  unfolding d-reaches.simps by blast

lemma d-reaches-unique-obtain[simp] :
  assumes d-reaches M (initial M) xs q
shows { p . d-reaches M (initial M) xs p } = { q }
  using assms unfolding d-reaches.simps by blast

lemma d-reaches-io-target :
  assumes d-reaches M p xs q
  and     length ys = length xs
shows io-targets M p (xs || ys) ⊆ {q}
proof
  fix q' assume q' ∈ io-targets M p (xs || ys)
  then obtain trQ where path M ((xs || ys) || trQ) p ∧ length (xs || ys) = length trQ
    by auto
  moreover obtain trD ysD where d-reached-by M p xs q trD ysD using assms(1)
    by auto
  ultimately have target ((xs || ys) || trQ) p = q
    by (simp add: assms(2))
  then show q' ∈ {q}
    using ‹d-reached-by M p xs q trD ysD› ‹q' ∈ io-targets M p (xs || ys)› assms(2) by auto
qed

lemma d-reachable-reachable : d-reachable M p ⊆ reachable M p
  unfolding d-reaches.simps d-reachable.simps by blast
```

1.7 Deterministic state cover

The deterministic state cover of some FSM is a minimal set of input sequences such that every d-reachable state of the FSM is d-reached by a sequence in the set and the set contains the empty sequence (which d-reaches the initial state).

```
fun is-det-state-cover-ass :: ('in, 'out, 'state) FSM ⇒ ('state ⇒ 'in list) ⇒ bool where
  is-det-state-cover-ass M f = (f (initial M) = [] ∧ (∀ s ∈ d-reachable M (initial M) .
                                                   d-reaches M (initial M) (f s) s))

lemma det-state-cover-ass-dist :
  assumes is-det-state-cover-ass M f
  and     s1 ∈ d-reachable M (initial M)
  and     s2 ∈ d-reachable M (initial M)
  and     s1 ≠ s2
shows ¬(d-reaches M (initial M) (f s2) s1)
  by (meson assms(1) assms(3) assms(4) d-reaches-unique is-det-state-cover-ass.simps)


lemma det-state-cover-ass-diff :
  assumes is-det-state-cover-ass M f
  and     s1 ∈ d-reachable M (initial M)
  and     s2 ∈ d-reachable M (initial M)
  and     s1 ≠ s2
shows f s1 ≠ f s2
  by (metis assms det-state-cover-ass-dist is-det-state-cover-ass.simps)


fun is-det-state-cover :: ('in, 'out, 'state) FSM ⇒ 'in list set ⇒ bool where
  is-det-state-cover M V = (∃ f . is-det-state-cover-ass M f
                              ∧ V = image f (d-reachable M (initial M)))

lemma det-state-cover-d-reachable[elim] :
  assumes is-det-state-cover M V
  and     v ∈ V
obtains q
where d-reaches M (initial M) v q
  by (metis (no-types, opaque-lifting) assms(1) assms(2) image-iff is-det-state-cover.simps
     is-det-state-cover-ass.elims(2))



lemma det-state-cover-card[simp] :
  assumes is-det-state-cover M V
  and     finite (nodes M)
shows   card (d-reachable M (initial M)) = card V
proof –
  obtain f where f-def : is-det-state-cover-ass M f ∧ V = image f (d-reachable M (initial M))
    using assms unfolding is-det-state-cover.simps by blast
  then have card-f : card V = card (image f (d-reachable M (initial M)))
    by simp

  have d-reachable M (initial M) ⊆ nodes M
    unfolding d-reachable.simps d-reaches.simps using d-reachable-reachable by blast
  then have dr-finite : finite (d-reachable M (initial M))
    using assms infinite-super by blast

  then have card-le : card (image f (d-reachable M (initial M))) ≤ card (d-reachable M (initial M))
    using card-image-le by blast

  have card (image f (d-reachable M (initial M))) = card (d-reachable M (initial M))
    by (meson card-image det-state-cover-ass-diff f-def inj-onI)

  then show ?thesis using card-f by auto
qed

lemma det-state-cover-finite :
```

assumes *is-det-state-cover M V*
and *finite (nodes M)*
shows *finite V*
proof −
have *d-reachable M (initial M) ⊆ nodes M*
by *auto*
show *finite V* **using** *det-state-cover-card[OF assms]*
by (*metis ‹d-reachable M (initial M) ⊆ nodes M› assms(1) assms(2) finite-imageI infinite-super is-det-state-cover.simps*)
qed

lemma *det-state-cover-initial* :
assumes *is-det-state-cover M V*
shows [] ∈ *V*
proof −
have *d-reached-by M (initial M)* [] *(initial M)* [] []
by (*simp add*: *FSM.nil*)
then have *d-reaches M (initial M)* [] *(initial M)*
by *auto*

have *initial M* ∈ *d-reachable M (initial M)*
by (*metis (no-types) ‹d-reaches M (initial M)* [] *(initial M)› d-reachable.simps mem-Collect-eq*)
then show *?thesis*
by (*metis (no-types, lifting) assms image-iff is-det-state-cover.elims(2) is-det-state-cover-ass.simps*)
qed

lemma *det-state-cover-empty* :
assumes *is-det-state-cover M V*
shows [] ∈ *V*
proof −
obtain *f* **where** *f-def* : *is-det-state-cover-ass M f* ∧ *V* = *f* ‘ *d-reachable M (initial M)*
using *assms* **by** *auto*
then have *f (initial M)* = []
by *auto*
moreover have *initial M* ∈ *d-reachable M (initial M)*
proof −
have *d-reaches M (initial M)* [] *(initial M)*
by *auto*
then show *?thesis*
by (*metis d-reachable.simps mem-Collect-eq*)
qed
moreover have *f (initial M)* ∈ *V*
using *f-def calculation* **by** *blast*
ultimately show *?thesis*
by *auto*
qed

1.8 IO reduction

An FSM is a reduction of another, if its language is a subset of the language of the latter FSM.

fun *io-reduction* :: (*'in*, *'out*, *'state*) *FSM* ⇒ (*'in*, *'out*, *'state*) *FSM*
⇒ *bool* (**infix** ‹⪯› *200*)
where
M1 ⪯ *M2* = (*LS M1 (initial M1)* ⊆ *LS M2 (initial M2)*)

lemma *language-state-inclusion-of-state-reached-by-same-sequence* :
assumes *LS M1 q1* ⊆ *LS M2 q2*
and *observable M1*
and *observable M2*
and *io-targets M1 q1 io* = { *q1t* }
and *io-targets M2 q2 io* = { *q2t* }
shows *LS M1 q1t* ⊆ *LS M2 q2t*
proof

```
  fix x assume x ∈ LS M1 q1t
  obtain q1x where io-targets M1 q1t x = {q1x}
    by (meson ‹x ∈ LS M1 q1t› assms(2) io-targets-observable-singleton-ex)
  have io ∈ LS M1 q1
    using assms(4) by auto
  have io@x ∈ LS M1 q1
    using observable-io-targets-append[OF assms(2) ‹io-targets M1 q1 io = { q1t }›
          ‹io-targets M1 q1t x = {q1x}›]
    by (metis io-targets-elim language-state singletonI)
  then have io@x ∈ LS M2 q2
    using assms(1) by blast
  then obtain q2x where io-targets M2 q2 (io@x) = {q2x}
    by (meson assms(3) io-targets-observable-singleton-ex)
  show x ∈ LS M2 q2t
    using observable-io-targets-split[OF assms(3) ‹io-targets M2 q2 (io @ x) = {q2x}› assms(5)]
    by auto
qed
```

1.9 Language subsets for input sequences

The following definitions describe restrictions of languages to only those IO-sequences that exhibit a certain input sequence or whose input sequence is contained in a given set of input sequences. This allows to define the notion that some FSM is a reduction of another over a given set of input sequences, but not necessarily over the entire language of the latter FSM.

```
fun language-state-for-input ::
  ('in, 'out, 'state) FSM ⇒ 'state ⇒ 'in list ⇒ ('in × 'out) list set where
  language-state-for-input M q xs = {(xs || ys) | ys . (length xs = length ys ∧ (xs || ys) ∈ LS M q)}

fun language-state-for-inputs ::
  ('in, 'out, 'state) FSM ⇒ 'state ⇒ 'in list set ⇒ ('in × 'out) list set
    (‹(LS_in - - -)› [1000,1000,1000]) where
  language-state-for-inputs M q ISeqs = {(xs || ys) | xs ys . (xs ∈ ISeqs
                                                  ∧ length xs = length ys
                                                  ∧ (xs || ys) ∈ LS M q)}

abbreviation L_in M TS ≡ LS_in M (initial M) TS

abbreviation io-reduction-on M1 TS M2 ≡ (L_in M1 TS ⊆ L_in M2 TS)
notation
  io-reduction-on (‹(- ≼⟦-⟧ -)› [1000,0,0] 61)
notation (latex output)
  io-reduction-on (‹(- ≼- -)› [1000,0,0] 61)


lemma language-state-for-input-alt-def :
  language-state-for-input M q xs = LS_in M q {xs}
  unfolding language-state-for-input.simps language-state-for-inputs.simps by blast

lemma language-state-for-inputs-alt-def :
  LS_in M q ISeqs = ⋃ (image (language-state-for-input M q) ISeqs)
  by auto

lemma language-state-for-inputs-in-language-state :
  LS_in M q T ⊆ language-state M q
  unfolding language-state-for-inputs.simps language-state-def
  by blast

lemma language-state-for-inputs-map-fst :
  assumes io ∈ language-state M q
  and     map fst io ∈ T
shows io ∈ LS_in M q T
proof −
  let ?xs = map fst io
  let ?ys = map snd io
  have ?xs ∈ T ∧ length ?xs = length ?ys ∧ ?xs || ?ys ∈ language-state M q
```

```
    using assms(2,1) by auto
  then have ?xs || ?ys ∈ LS_in M q T
    unfolding language-state-for-inputs.simps by blast
  then show ?thesis
    by simp
qed

lemma language-state-for-inputs-nonempty :
  assumes set xs ⊆ inputs M
  and     completely-specified M
  and     q ∈ nodes M
shows LS_in M q {xs} ≠ {}
using assms proof (induction xs arbitrary: q)
  case Nil
  then show ?case by auto
next
  case (Cons x xs)
  then have x ∈ inputs M
    by simp
  then obtain y q' where x-step : q' ∈ succ M (x,y) q
    using Cons(3,4) unfolding completely-specified.simps by blast
  then have path M ([(x,y)] || [q']) q ∧ length [q] = length [(x,y)]
          target ([(x,y)] || [q']) q = q'
    by auto
  then have q' ∈ nodes M
    using Cons(4) by (metis FSM.nodes-target)
  then have LS_in M q' {xs} ≠ {}
    using Cons.prems Cons.IH by auto
  then obtain ys where length xs = length ys ∧ (xs || ys) ∈ LS M q'
    by auto
  then obtain tr where path M ((xs || ys) || tr) q' ∧ length tr = length (xs || ys)
    by auto
  then have path M ([(x,y)] @ (xs || ys) || [q'] @ tr) q
            ∧ length ([q'] @ tr) = length ([(x,y)] @ (xs || ys))
    by (simp add: FSM.path.intros(2) x-step)
  then have path M ((x#xs || y#ys) || [q'] @ tr) q ∧ length ([q'] @ tr) = length (x#xs || y#ys)
    by auto
  then have (x#xs || y#ys) ∈ LS M q
    by (metis language-state)
  moreover have length (x#xs) = length (y#ys)
    by (simp add: ‹length xs = length ys ∧ xs || ys ∈ LS M q'›)
  ultimately have (x#xs || y#ys) ∈ LS_in M q {x # xs}
    unfolding language-state-for-inputs.simps by blast
  then show ?case by blast
qed

lemma language-state-for-inputs-map-fst-contained :
  assumes vs ∈ LS_in M q V
shows map fst vs ∈ V
proof −
  have (map fst vs) || (map snd vs) = vs
    by auto
  then have (map fst vs) || (map snd vs) ∈ LS_in M q V
    using assms by auto
  then show ?thesis by auto
qed

lemma language-state-for-inputs-empty :
  assumes [] ∈ V
  shows [] ∈ LS_in M q V
proof −
  have [] ∈ language-state-for-input M q [] by auto
  then show ?thesis using language-state-for-inputs-alt-def by (metis UN-I assms)
qed

lemma language-state-for-input-empty[simp] :
```

language-state-for-input M q [] = {[]}
by *auto*

lemma *language-state-for-input-take* :
assumes *io* ∈ *language-state-for-input M q xs*
shows *take n io* ∈ *language-state-for-input M q* (*take n xs*)
proof −
obtain *ys* **where** *io* = *xs* || *ys length xs* = *length ys xs* || *ys* ∈ *language-state M q*
using *assms* **by** *auto*
then obtain *p* **where** *length p* = *length xs path M* ((*xs* || *ys*) || *p*) *q*
by *auto*
then have *path M* (*take n* ((*xs* || *ys*) || *p*)) *q*
by (*metis FSM.path-append-elim append-take-drop-id*)
then have *take n* (*xs* || *ys*) ∈ *language-state M q*
by (*simp add*: ‹*length p* = *length xs*› ‹*length xs* = *length ys*› *language-state take-zip*)
then have (*take n xs*) || (*take n ys*) ∈ *language-state M q*
by (*simp add*: *take-zip*)

have *take n io* = (*take n xs*) || (*take n ys*)
using ‹*io* = *xs* || *ys*› *take-zip* **by** *blast*
moreover have *length* (*take n xs*) = *length* (*take n ys*)
by (*simp add*: ‹*length xs* = *length ys*›)
ultimately show *?thesis*
using ‹(*take n xs*) || (*take n ys*) ∈ *language-state M q*›
unfolding *language-state-for-input.simps* **by** *blast*
qed

lemma *language-state-for-inputs-prefix* :
assumes *vs*@*xs* ∈ L_{in} *M1* {*vs′*@*xs′*}
and *length vs* = *length vs′*
shows *vs* ∈ L_{in} *M1* {*vs′*}
proof −
have *vs*@*xs* ∈ *L M1*
using *assms*(*1*) **by** *auto*
then have *vs* ∈ *L M1*
by (*meson language-state-prefix*)
then have *vs* ∈ L_{in} *M1* {*map fst vs*}
by (*meson insertI1 language-state-for-inputs-map-fst*)
moreover have *vs′* = *map fst vs*
by (*metis append-eq-append-conv assms*(*1*) *assms*(*2*) *language-state-for-inputs-map-fst-contained length-map map-append singletonD*)
ultimately show *?thesis*
by *blast*
qed

lemma *language-state-for-inputs-union* :
shows LS_{in} *M q T1* ∪ LS_{in} *M q T2* = LS_{in} *M q* (*T1* ∪ *T2*)
unfolding *language-state-for-inputs.simps* **by** *blast*

lemma *io-reduction-on-subset* :
assumes *io-reduction-on M1 T M2*
and *T′* ⊆ *T*
shows *io-reduction-on M1 T′ M2*
proof (*rule ccontr*)
assume ¬ *io-reduction-on M1 T′ M2*
then obtain *xs′* **where** *xs′* ∈ *T′* ¬ L_{in} *M1* {*xs′*} ⊆ L_{in} *M2* {*xs′*}
proof −
have *f1*: ∀ *ps P Pa*. (*ps*::(*′a* × *′b*) *list*) ∉ *P* ∨ ¬ *P* ⊆ *Pa* ∨ *ps* ∈ *Pa*
by *blast*
obtain *pps* :: (*′a* × *′b*) *list set* ⇒ (*′a* × *′b*) *list set* ⇒ (*′a* × *′b*) *list* **where**
∀ *x0 x1*. (∃ *v2*. *v2* ∈ *x1* ∧ *v2* ∉ *x0*) = (*pps x0 x1* ∈ *x1* ∧ *pps x0 x1* ∉ *x0*)
by *moura*
then have *f2*: ∀ *P Pa*. *pps Pa P* ∈ *P* ∧ *pps Pa P* ∉ *Pa* ∨ *P* ⊆ *Pa*
by (*meson subsetI*)
have *f3*: ∀ *ps f c A*. (*ps*::(*′a* × *′b*) *list*) ∉ LS_{in} *f* (*c*::*′c*) *A* ∨ *map fst ps* ∈ *A*

    by (*meson language-state-for-inputs-map-fst-contained*)
  then have L_{in} *M1 T′* ⊆ L_{in} *M1 T*
    using *f2* **by** (*meson assms*(*2*) *language-state-for-inputs-in-language-state*
      language-state-for-inputs-map-fst set-rev-mp)
  then show *?thesis*
    using *f3 f2 f1* **by** (*meson* ‹¬ *io-reduction-on M1 T′ M2*› *assms*(*1*)
      language-state-for-inputs-in-language-state
      language-state-for-inputs-map-fst)
qed
then have *xs′* ∈ *T*
  using *assms*(*2*) **by** *blast*

have ¬ *io-reduction-on M1 T M2*
proof −
  have *f1*: ∀ *as. as* ∉ *T′* ∨ *as* ∈ *T*
    using *assms*(*2*) **by** *auto*
  obtain *pps* :: (*′a* × *′b*) *list set* ⇒ (*′a* × *′b*) *list set* ⇒ (*′a* × *′b*) *list* **where**
    ∀ *x0 x1*. (∃ *v2*. *v2* ∈ *x1* ∧ *v2* ∉ *x0*) = (*pps x0 x1* ∈ *x1* ∧ *pps x0 x1* ∉ *x0*)
    by *moura*
  then have ∀ *P Pa*. (¬ *P* ⊆ *Pa* ∨ (∀ *ps. ps* ∉ *P* ∨ *ps* ∈ *Pa*))
      ∧ (*P* ⊆ *Pa* ∨ *pps Pa P* ∈ *P* ∧ *pps Pa P* ∉ *Pa*)
    by *blast*
  then show *?thesis*
    using *f1* **by** (*meson* ‹¬ *io-reduction-on M1 T′ M2*› *language-state-for-inputs-in-language-state*
      language-state-for-inputs-map-fst language-state-for-inputs-map-fst-contained)
qed

then show *False*
  using *assms*(*1*) **by** *auto*
qed

1.10 Sequences to failures

A sequence to a failure for FSMs `M1` and `M2` is a sequence such that any proper prefix of it is contained in the languages of both `M1` and `M2`, while the sequence itself is contained only in the language of A.

That is, if a sequence to a failure for `M1` and `M2` exists, then `M1` is not a reduction of `M2`.

fun *sequence-to-failure* ::
  (*′in*,*′out*,*′state*) *FSM* ⇒ (*′in*,*′out*,*′state*) *FSM* ⇒ (*′in* × *′out*) *list* ⇒ *bool* **where**
  sequence-to-failure M1 M2 xs = (
    (*butlast xs*) ∈ (*language-state M2* (*initial M2*) ∩ *language-state M1* (*initial M1*))
    ∧ *xs* ∈ (*language-state M1* (*initial M1*) − *language-state M2* (*initial M2*)))

lemma *sequence-to-failure-ob* :
  assumes ¬ *M1* ≼ *M2*
  and *well-formed M1*
  and *well-formed M2*
obtains *io*
where *sequence-to-failure M1 M2 io*
proof −
  let *?diff* = { *io* . *io* ∈ *language-state M1* (*initial M1*) ∧ *io* ∉ *language-state M2* (*initial M2*)}
  have *?diff* ≠ *empty*
    using *assms* **by** *auto*
  moreover obtain *io* **where** *io-def*[*simp*] : *io* = *arg-min length* (λ *io* . *io* ∈ *?diff*)
    using *assms* **by** *auto*
  ultimately have *io-diff* : *io* ∈ *?diff*
    using *assms* **by** (*meson all-not-in-conv arg-min-natI*)

  then have *io* ≠ []
    using *assms io-def language-state* **by** *auto*
  then obtain *io-init io-last* **where** *io-split*[*simp*] : *io* = *io-init* @ [*io-last*]
    by (*metis append-butlast-last-id*)

  have *io-init-inclusion* : *io-init* ∈ *language-state M1* (*initial M1*)
      ∧ *io-init* ∈ *language-state M2* (*initial M2*)

```
  proof (rule ccontr)
    assume assm : ¬ (io-init ∈ language-state M1 (initial M1)
                      ∧ io-init ∈ language-state M2 (initial M2))

    have io-init @ [io-last] ∈ language-state M1 (initial M1)
      using io-diff io-split by auto
    then have io-init ∈ language-state M1 (initial M1)
      by (meson language-state language-state-split)
    moreover have io-init ∉ language-state M2 (initial M2)
      using assm calculation by auto
    ultimately have io-init ∈ ?diff
      by auto
    moreover have length io-init < length io
      using io-split by auto
    ultimately have io ≠ arg-min length (λ io . io ∈ ?diff)
    proof −
      have ∃ps. ps ∈ {ps ∈ language-state M1 (initial M1).
                           ps ∉ language-state M2 (initial M2)} ∧ ¬ length io ≤ length ps
        using ‹io-init ∈ {io∈ language-state M1 (initial M1). io ∉ language-state M2 (initial M2)}›
              ‹length io-init < length io› linorder-not-less
        by blast
      then show ?thesis
        by (meson arg-min-nat-le)
    qed
    then show False using io-def by simp
  qed

  then have sequence-to-failure M1 M2 io
    using io-split io-diff by auto
  then show ?thesis
    using that by auto
qed

lemma sequence-to-failure-succ :
  assumes sequence-to-failure M1 M2 io
  shows ∀ q ∈ io-targets M2 (initial M2) (butlast io) . succ M2 (last io) q = {}
proof
  have io ≠ []
    using assms by auto
  fix q assume q ∈ io-targets M2 (initial M2) (butlast io)
  then obtain tr where q = target (butlast io || tr) (initial M2)
                 and   path M2 (butlast io || tr) (initial M2)
                 and   length (butlast io) = length tr
    unfolding io-targets.simps by auto

  show succ M2 (last io) q = {}
  proof (rule ccontr)
    assume succ M2 (last io) q ≠ {}
    then obtain q' where q' ∈ succ M2 (last io) q
      by blast
    then have path M2 [(last io, q')] (target (butlast io || tr) (initial M2))
      using ‹q = target (butlast io || tr) (initial M2)› by auto

    have path M2 ((butlast io || tr) @ [(last io, q')]) (initial M2)
      using ‹path M2 (butlast io || tr) (initial M2)›
            ‹path M2 [(last io, q')] (target (butlast io || tr) (initial M2))› by auto

    have butlast io @ [last io] = io
      by (meson ‹io ≠ []› append-butlast-last-id)

    have path M2 (io || (tr@[q'])) (initial M2)
    proof −
      have path M2 ((butlast io || tr) @ ([last io] || [q'])) (initial M2)
        by (simp add: FSM.path-append ‹path M2 (butlast io || tr) (initial M2)›
              ‹path M2 [(last io, q')] (target (butlast io || tr) (initial M2))›)
      then show ?thesis
```

by (*metis* (*no-types*) ‹*butlast io* @ [*last io*] = *io*›
‹*length* (*butlast io*) = *length tr*› *zip-append*)
qed

have *io* ∈ *L M2*
proof −
have *length tr* + (*0* + *Suc 0*) = *length io*
by (*metis* ‹*butlast io* @ [*last io*] = *io*› ‹*length* (*butlast io*) = *length tr*›
length-append list.size(*3*) *list.size*(*4*))
then show *?thesis*
using ‹*path M2* (*io* || *tr* @ [*q'*]) (*initial M2*)› **by** *fastforce*
qed
then show *False*
using *assms* **by** *auto*
qed
qed

lemma *sequence-to-failure-non-nil* :
assumes *sequence-to-failure M1 M2 xs*
shows $xs \neq []$
proof
assume $xs = []$
then have *xs* ∈ *L M1* ∩ *L M2*
by *auto*
then show *False* **using** *assms* **by** *auto*
qed

lemma *sequence-to-failure-from-arbitrary-failure* :
assumes *vs@xs* ∈ *L M1* − *L M2*
and *vs* ∈ *L M2* ∩ *L M1*
shows ∃ *xs'* . *prefix xs' xs* ∧ *sequence-to-failure M1 M2* (*vs@xs'*)
using *assms* **proof** (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)

have *vs* @ *xs* ∈ *L M1*
using *snoc.prems*(*1*) **by** (*metis Diff-iff append.assoc language-state-prefix*)

show *?case*
proof (*cases vs@xs* ∈ *L M2*)
case *True*
have *butlast* (*vs@xs@*[*x*]) ∈ *L M2* ∩ *L M1*
using *True* ‹*vs* @ *xs* ∈ *L M1*› **by** (*simp add*: *butlast-append*)
then show *?thesis*
using *sequence-to-failure.simps snoc.prems* **by** *blast*
next
case *False*
then have *vs@xs* ∈ *L M1* − *L M2*
using ‹*vs* @ *xs* ∈ *L M1*› **by** *blast*
then obtain *xs'* **where** *prefix xs' xs sequence-to-failure M1 M2* (*vs@xs'*)
using *snoc.prems*(*2*) *snoc.IH* **by** *blast*
then show *?thesis*
using *prefix-snoc* **by** *auto*
qed
qed

The following lemma shows that if `M1` is not a reduction of `M2`, then a minimal sequence to a failure exists that is of length at most the number of states in `M1` times the number of states in `M2`.

lemma *sequence-to-failure-length* :
assumes *well-formed M1*
and *well-formed M2*
and *observable M1*
and *observable M2*
and ¬ *M1* ⪯ *M2*

shows ∃ *xs . sequence-to-failure M1 M2 xs* ∧ *length xs* ≤ |*M2*| ∗ |*M1*|
proof −

obtain *seq* **where** *sequence-to-failure M1 M2 seq*
using *assms sequence-to-failure-ob* **by** *blast*
then have *seq* ≠ []
by *auto*

let *?bls* = *butlast seq*
have *?bls* ∈ *L M1 ?bls* ∈ *L M2*
using ‹*sequence-to-failure M1 M2 seq*› **by** *auto*

then obtain *tr1b tr2b* **where**
path M1 (*?bls* || *tr1b*) (*initial M1*)
length tr1b = *length ?bls*
path M2 (*?bls* || *tr2b*) (*initial M2*)
length ?bls = *length tr2b*
by *fastforce*
then have *length tr2b* = *length tr1b*
by *auto*

let *?PM* = *product M2 M1*
have *well-formed ?PM*
using *well-formed-product*[*OF assms*(*1*,*2*)] **by** *assumption*

have *path ?PM* (*?bls* || *tr2b* || *tr1b*) (*initial M2*, *initial M1*)
using *product-path*[*OF* ‹*length ?bls* = *length tr2b*› ‹*length tr2b* = *length tr1b*›,
of M2 M1 initial M2 initial M1]
using ‹*path M1* (*butlast seq* || *tr1b*) (*initial M1*)›
‹*path M2* (*butlast seq* || *tr2b*) (*initial M2*)›
by *blast*

let *?q1b* = *target* (*?bls* || *tr1b*) (*initial M1*)
let *?q2b* = *target* (*?bls* || *tr2b*) (*initial M2*)

have *io-targets M2* (*initial M2*) *?bls* = {*?q2b*}
by (*metis* ‹*length* (*butlast seq*) = *length tr2b*› ‹*path M2* (*butlast seq* || *tr2b*) (*initial M2*)›
assms(*4*) *obs-target-is-io-targets*)
have *io-targets M1* (*initial M1*) *?bls* = {*?q1b*}
by (*metis* ‹*length tr1b* = *length* (*butlast seq*)› ‹*path M1* (*butlast seq* || *tr1b*) (*initial M1*)›
assms(*3*) *obs-target-is-io-targets*)

have (*?q2b*, *?q1b*) ∈ *reachable* (*product M2 M1*) (*initial M2*, *initial M1*)
proof −
have *target* (*butlast seq* || *tr2b* || *tr1b*) (*initial M2*, *initial M1*)
∈ *reachable* (*product M2 M1*) (*initial M2*, *initial M1*)
using ‹*path* (*product M2 M1*) (*butlast seq* || *tr2b* || *tr1b*) (*initial M2*, *initial M1*)› **by** *blast*
then show *?thesis*
using ‹*length* (*butlast seq*) = *length tr2b*› ‹*length tr2b* = *length tr1b*› **by** *auto*
qed

have (*initial M2*, *initial M1*) ∈ *nodes* (*product M2 M1*)
by (*simp add*: *FSM.nodes.initial*)

obtain *p* **where** *repFreePath* : *path* (*product M2 M1*) *p* (*initial M2*, *initial M1*) ∧
target p (*initial M2*, *initial M1*) =
(*?q2b*,*?q1b*)
distinct ((*initial M2*, *initial M1*) # *states p* (*initial M2*, *initial M1*))
using *reaching-path-without-repetition*[*OF* ‹*well-formed ?PM*›
‹(*?q2b*, *?q1b*) ∈ *reachable* (*product M2 M1*) (*initial M2*, *initial M1*)›

```
      ‹(initial M2, initial M1) ∈ nodes (product M2 M1)›]
  by blast

then have set (states p (initial M2, initial M1)) ⊆ nodes ?PM
  by (simp add: FSM.nodes-states ‹(initial M2, initial M1) ∈ nodes (product M2 M1)›)
moreover have (initial M2, initial M1) ∉ set (states p (initial M2, initial M1))
  using ‹distinct ((initial M2, initial M1) # states p (initial M2, initial M1))› by auto
ultimately have set (states p (initial M2, initial M1)) ⊆ nodes ?PM − {(initial M2,initial M1)}
  by blast
moreover have finite (nodes ?PM)
  using ‹well-formed ?PM› by auto
ultimately have card (set (states p (initial M2, initial M1))) < card (nodes ?PM)
  by (metis ‹(initial M2, initial M1) ∈ nodes (product M2 M1)›
      ‹(initial M2, initial M1) ∉ set (states p (initial M2, initial M1))›
      ‹set (states p (initial M2, initial M1)) ⊆ nodes (product M2 M1)›
      psubsetI psubset-card-mono)

moreover have card (set (states p (initial M2, initial M1)))
                = length (states p (initial M2, initial M1))
  using distinct-card repFreePath(2) by fastforce
ultimately have length (states p (initial M2, initial M1)) < |?PM|
  by linarith
then have length p < |?PM|
  by auto



let ?p1 = map (snd ∘ snd) p
let ?p2 = map (fst ∘ snd) p
let ?pIO = map fst p

have p = ?pIO || ?p2 || ?p1
  by (metis map-map zip-map-fst-snd)


have path M2 (?pIO || ?p2) (initial M2)
     path M1 (?pIO || ?p1) (initial M1)
  using product-path[of ?pIO ?p2 ?p1 M2 M1]
  using ‹p = ?pIO || ?p2 || ?p1› repFreePath(1) by auto

have (?q2b, ?q1b) = (target (?pIO || ?p2 || ?p1) (initial M2, initial M1))
  using ‹p = ?pIO || ?p2 || ?p1› repFreePath(1) by auto

then have ?q2b = target (?pIO || ?p2) (initial M2)
          ?q1b = target (?pIO || ?p1) (initial M1)
  by auto

have io-targets M2 (initial M2) ?pIO = {?q2b}
  by (metis ‹path M2 (map fst p || map (fst ∘ snd) p) (initial M2)›
      ‹target (?bls || tr2b) (initial M2) = target (map fst p || map (fst ∘ snd) p) (initial M2)›
      assms(4) length-map obs-target-is-io-targets)

have io-targets M1 (initial M1) ?pIO = {?q1b}
  by (metis ‹path M1 (map fst p || map (snd ∘ snd) p) (initial M1)›
      ‹target (?bls || tr1b) (initial M1) = target (map fst p || map (snd ∘ snd) p) (initial M1)›
      assms(3) length-map obs-target-is-io-targets)



have seq ∈ L M1 seq ∉ L M2
  using ‹sequence-to-failure M1 M2 seq› by auto

have io-targets M1 (initial M1) ?bls = {?q1b}
  by (metis ‹length tr1b = length (butlast seq)› ‹path M1 (butlast seq || tr1b) (initial M1)›
      assms(3) obs-target-is-io-targets)
```

obtain *q1s* **where** *io-targets M1* (*initial M1*) *seq* = {*q1s*}
  by (*meson* ‹*seq* ∈ *L M1*› *assms*(*3*) *io-targets-observable-singleton-ob*)

moreover have *seq* = (*butlast seq*)@[*last seq*]
  using ‹*seq* ≠ []› **by** *auto*
ultimately have *io-targets M1* (*initial M1*) ((*butlast seq*)@[*last seq*]) = {*q1s*}
  by *auto*

have *io-targets M1 ?q1b* [*last seq*] = {*q1s*}
  using *observable-io-targets-suffix*[*OF assms*(*3*) ‹*io-targets M1* (*initial M1*) *?bls* = {*?q1b*}›
    ‹*io-targets M1* (*initial M1*) ((*butlast seq*)@[*last seq*]) = {*q1s*}›] **by** *assumption*
then obtain *tr1s* **where** *q1s* = *target* ([*last seq*] || *tr1s*) *?q1b*
        path M1 ([*last seq*] || *tr1s*) *?q1b*
        length [*last seq*] = *length tr1s*
  by *auto*

have *path M1* ([*last seq*] || [*q1s*]) *?q1b*
  by (*metis* (*no-types*) ‹*length* [*last seq*] = *length tr1s*›
    ‹*path M1* ([*last seq*] || *tr1s*) (*target* (*butlast seq* || *tr1b*) (*initial M1*))›
    ‹*q1s* = *target* ([*last seq*] || *tr1s*) (*target* (*butlast seq* || *tr1b*) (*initial M1*))›
    append-Nil append-butlast-last-id butlast.simps(*2*) *length-butlast length-greater-0-conv*
    not-Cons-self2 target-alt-def(*2*))
then have *q1s* ∈ *succ M1* (*last seq*) *?q1b*
  by *auto*

have *succ M2* (*last seq*) *?q2b* = {}
proof (*rule ccontr*)
  assume *succ M2* (*last seq*) (*target* (*butlast seq* || *tr2b*) (*initial M2*)) ≠ {}
  then obtain *q2f* **where** *q2f* ∈ *succ M2* (*last seq*) *?q2b*
    by *blast*
  then have *target* ([*last seq*] || [*q2f*]) *?q2b* = *q2f*
      path M2 ([*last seq*] || [*q2f*]) *?q2b*
      length [*q2f*] = *length* [*last seq*]
    by *auto*
  then have *q2f* ∈ *io-targets M2 ?q2b* [*last seq*]
    by (*metis io-target-from-path*)
  then have *io-targets M2 ?q2b* [*last seq*] = {*q2f*}
    using *assms*(*4*) **by** (*meson observable-io-target-is-singleton*)

  have *io-targets M2* (*initial M2*) (*butlast seq* @ [*last seq*]) = {*q2f*}
    using *observable-io-targets-append*[*OF assms*(*4*) ‹*io-targets M2* (*initial M2*) *?bls* = {*?q2b*}›
      ‹*io-targets M2 ?q2b* [*last seq*] = {*q2f*}›] **by** *assumption*
  then have *seq* ∈ *L M2*
    using ‹*seq* = *butlast seq* @ [*last seq*]› **by** *auto*
  then show *False*
    using ‹*seq* ∉ *L M2*› **by** *blast*
qed

have *?pIO* ∈ *L M1 ?pIO* ∈ *L M2*
  using ‹*path M1* (*?pIO* || *?p1*) (*initial M1*)› ‹*path M2* (*?pIO* || *?p2*) (*initial M2*)› **by** *auto*
then have *butlast* (*?pIO*@[*last seq*]) ∈ *L M1* ∩ *L M2*
  by *auto*

have *?pIO*@[*last seq*] ∈ *L M1*
  using *observable-io-targets-append*[*OF assms*(*3*) ‹*io-targets M1* (*initial M1*) *?pIO* = {*?q1b*}›
    ‹*io-targets M1 ?q1b* [*last seq*] = {*q1s*}›]
  by (*metis all-not-in-conv insert-not-empty io-targets-elim language-state*)

moreover have *?pIO*@[*last seq*] ∉ *L M2*
proof
  assume *?pIO*@[*last seq*] ∈ *L M2*
  then obtain *q2f* **where** *io-targets M2* (*initial M2*) (*?pIO*@[*last seq*]) = {*q2f*}
    by (*meson assms*(*4*) *io-targets-observable-singleton-ob*)

```
    have io-targets M2 ?q2b [last seq] = {q2f}
      using observable-io-targets-split[OF assms(4)
            ‹io-targets M2 (initial M2) (?pIO@[last seq]) = {q2f}›
            ‹io-targets M2 (initial M2) (map fst p) = {?q2b}›] by assumption

    then have q2f ∈ succ M2 (last seq) ?q2b
      by (simp add: io-targets-succ)
    then show False
      using ‹succ M2 (last seq) ?q2b = {}› by auto
  qed

  ultimately have ?pIO@[last seq] ∈ L M1 − L M2
    by auto


  have sequence-to-failure M1 M2 (?pIO@[last seq])
    using ‹butlast (?pIO@[last seq]) ∈ L M1 ∩ L M2› ‹?pIO@[last seq] ∈ L M1 − L M2› by auto

  have length (?pIO@[last seq]) = Suc (length ?pIO)
    by auto
  then have length (?pIO@[last seq]) ≤ |?PM|
    using ‹length p < |?PM|› by auto


  have card (nodes M2 × nodes M1) ≤ |M2| ∗ |M1|
    by (simp add: card-cartesian-product)

  have finite (nodes M2 × nodes M1)
  proof
    show finite (nodes M2)
      using assms by auto
    show finite (nodes M1)
      using assms by auto
  qed

  have |?PM| ≤ |M2| ∗ |M1|
    by (meson ‹card (nodes M2 × nodes M1) ≤ |M2| ∗ |M1|› ‹finite (nodes M2 × nodes M1)›
        card-mono dual-order.trans product-nodes)

  then have length (?pIO@[last seq]) ≤ |M2| ∗ |M1|
    using ‹length (?pIO@[last seq]) ≤ |?PM|› by auto

  then have sequence-to-failure M1 M2 (?pIO@[last seq]) ∧ length (?pIO@[last seq]) ≤ |M2| ∗ |M1|
    using ‹sequence-to-failure M1 M2 (?pIO@[last seq])› by auto
  then show ?thesis
    by blast
qed
```

1.11 Minimal sequence to failure extending

A minimal sequence to a failure extending some some set of IO-sequences is a sequence to a failure of minimal length such that a prefix of that sequence is contained in the set.

```
fun minimal-sequence-to-failure-extending ::
  'in list set ⇒ ('in,'out,'state) FSM ⇒ ('in,'out,'state) FSM ⇒ ('in × 'out) list
    ⇒ ('in × 'out) list ⇒ bool where
  minimal-sequence-to-failure-extending V M1 M2 v' io = (
   v' ∈ L_in M1 V ∧ sequence-to-failure M1 M2 (v' @ io)
             ∧ ¬ (∃ io' . ∃ w' ∈ L_in M1 V . sequence-to-failure M1 M2 (w' @ io')
                                                ∧ length io' < length io))


lemma minimal-sequence-to-failure-extending-det-state-cover-ob :
  assumes well-formed M1
  and     well-formed M2
  and     observable M2
```

and *is-det-state-cover M2 V*
and $\neg$ *M1* $\preceq$ *M2*
obtains *vs xs*
where *minimal-sequence-to-failure-extending V M1 M2 vs xs*
proof –
— set of all IO-sequences that extend some reaction of M1 to V to a failure
let *?exts* = $\{xs.\ \exists\, vs' \in L_{in}\ M1\ V.\ \textit{sequence-to-failure M1 M2}\ (vs'@xs)\}$

— arbitrary sequence to failure
— must be contained in ?exts as V contains the empty sequence
obtain *stf* **where** *sequence-to-failure M1 M2 stf*
using *assms sequence-to-failure-ob* **by** *blast*
then have *sequence-to-failure M1 M2* ([] @ *stf*)
by *simp*
moreover have [] $\in L_{in}$ *M1 V*
by (*meson assms*(*4*) *det-state-cover-initial language-state-for-inputs-empty*)
ultimately have *stf* $\in$ *?exts*
by *blast*

— the minimal length sequence of ?exts
— is a minimal sequence to a failure extending V by construction
let *?xsMin* = *arg-min length* ($\lambda xs.\ xs \in$ *?exts*)
have *xsMin-def* : *?xsMin* $\in$ *?exts*
$\wedge$ ($\forall$ *xs* $\in$ *?exts*. *length ?xsMin* $\leq$ *length xs*)
by (*metis* (*no-types*, *lifting*) ‹*stf* $\in$ *?exts*› *arg-min-nat-lemma*)
then obtain *vs* **where** *vs* $\in L_{in}$ *M1 V*
$\wedge$ *sequence-to-failure M1 M2* (*vs* @ *?xsMin*)
by *blast*
moreover have $\neg(\exists$ *xs* . $\exists$ *ws* $\in L_{in}$ *M1 V*. *sequence-to-failure M1 M2* (*ws*@*xs*)
$\wedge$ *length xs* < *length ?xsMin*)
using *leD xsMin-def* **by** *blast*
ultimately have *minimal-sequence-to-failure-extending V M1 M2 vs ?xsMin*
by *auto*
then show *?thesis*
using *that* **by** *auto*
qed

lemma *mstfe-prefix-input-in-V* :
assumes *minimal-sequence-to-failure-extending V M1 M2 vs xs*
shows (*map fst vs*) $\in$ *V*
proof –
have *vs* $\in L_{in}$ *M1 V*
using *assms* **by** *auto*
then show *?thesis*
using *language-state-for-inputs-map-fst-contained* **by** *auto*
qed

1.12 Complete test suite derived from the product machine

The classical result of testing FSMs for language inclusion : Any failure can be observed by a sequence of length at most n*m where n is the number of states of the reference model (here FSM `M2`) and m is an upper bound on the number of states of the SUT (here FSM `M1`).

lemma *product-suite-soundness* :
assumes *well-formed M1*
and *well-formed M2*
and *observable M1*
and *observable M2*
and *inputs M2* = *inputs M1*
and $|M1| \leq m$
shows $\neg$ *M1* $\preceq$ *M2* $\longrightarrow \neg$ *M1* $\preceq[\![\{xs\ .\ \textit{set xs} \subseteq \textit{inputs M2} \wedge \textit{length xs} \leq |M2| * m\}]\!]$ *M2*
(**is** $\neg$ *M1* $\preceq$ *M2* $\longrightarrow \neg$ *M1* $\preceq[\![?TS]\!]$ *M2*)
proof
assume $\neg$ *M1* $\preceq$ *M2*
obtain *stf* **where** *sequence-to-failure M1 M2 stf* $\wedge$ *length stf* $\leq |M2| * |M1|$
using *sequence-to-failure-length*[*OF assms*(*1*–*4*) ‹$\neg$ *M1* $\preceq$ *M2*›] **by** *blast*

```
  then have sequence-to-failure M1 M2 stf length stf ≤ |M2| ∗ |M1|
    by auto

  then have stf ∈ L M1
    by auto
  let ?xs = map fst stf
  have set ?xs ⊆ inputs M1
    by (meson ‹stf ∈ L M1› assms(1) language-state-inputs)
  then have set ?xs ⊆ inputs M2
    using assms(5) by auto

  have length ?xs ≤ |M2| ∗ |M1|
    using ‹length stf ≤ |M2| ∗ |M1|› by auto
  have length ?xs ≤ |M2| ∗ m
  proof −
    show ?thesis
      by (metis (no-types) ‹length (map fst stf) ≤ |M2| ∗ |M1|› ‹|M1| ≤ m›
          dual-order.trans mult.commute mult-le-mono1)
  qed

  have stf ∈ L_in M1 {?xs}
    by (meson ‹stf ∈ L M1› insertI1 language-state-for-inputs-map-fst)
  have ?xs ∈ ?TS
    using ‹set ?xs ⊆ inputs M2› ‹length ?xs ≤ |M2| ∗ m› by blast
  have stf ∈ L_in M1 ?TS
    by (metis (no-types, lifting) ‹map fst stf ∈ {xs. set xs ⊆ inputs M2 ∧ length xs ≤ |M2| ∗ m}›
        ‹stf ∈ L M1› language-state-for-inputs-map-fst)

  have stf ∉ L M2
    using ‹sequence-to-failure M1 M2 stf› by auto
  then have stf ∉ L_in M2 ?TS
    by auto


  show ¬ M1 ≼⟦?TS⟧ M2
    using ‹stf ∈ L_in M1 ?TS› ‹stf ∉ L_in M2 ?TS› by blast
qed


lemma product-suite-completeness :
  assumes well-formed M1
  and     well-formed M2
  and     observable M1
  and     observable M2
  and     inputs M2 = inputs M1
  and     |M1| ≤ m
shows     M1 ≼ M2 ⟷ M1 ≼⟦{xs . set xs ⊆ inputs M2 ∧ length xs ≤ |M2| ∗ m}⟧ M2
  (is M1 ≼ M2 ⟷ M1 ≼⟦?TS⟧ M2)
proof
  show M1 ≼ M2 ⟹ M1 ≼⟦?TS⟧ M2 — soundness holds trivially
    unfolding language-state-for-inputs.simps io-reduction.simps by blast
  show M1 ≼⟦?TS⟧ M2 ⟹ M1 ≼ M2
    using product-suite-soundness[OF assms] by auto
qed


end
theory FSM-Product
imports FSM
begin
```

2 Product machines with an additional fail state

We extend the product machine for language intersection presented in theory FSM by an additional state that is reached only by sequences such that any proper prefix of the sequence is in the language intersection, whereas the full sequence is only contained in the language of the machine `B` for which we want to check whether it is a reduction of some machine `A`.

To allow for free choice of the FAIL state, we define the following property that holds iff `AB` is the product machine of `A` and `B` extended with fail state `FAIL`.

fun *productF* :: (*'in*, *'out*, *'state1*) *FSM* ⇒ (*'in*, *'out*, *'state2*) *FSM* ⇒ (*'state1* × *'state2*)
  ⇒ (*'in*, *'out*, *'state1* ×*'state2*) *FSM* ⇒ *bool* **where**
  productF A B FAIL AB = (
    (*inputs A* = *inputs B*)
  ∧ (*fst FAIL* ∉ *nodes A*)
  ∧ (*snd FAIL* ∉ *nodes B*)
  ∧ *AB* = ⦇
      succ = (λ *a* (*p1*,*p2*) . (*if* (*p1* ∈ *nodes A* ∧ *p2* ∈ *nodes B* ∧ (*fst a* ∈ *inputs A*)
                ∧ (*snd a* ∈ *outputs A* ∪ *outputs B*))
              then (*if* (*succ A a p1* = {} ∧ *succ B a p2* ≠ {})
                then {*FAIL*}
                else (*succ A a p1* × *succ B a p2*))
              else {})),
      inputs = *inputs A*,
      outputs = *outputs A* ∪ *outputs B*,
      initial = (*initial A*, *initial B*)
    ⦈)

lemma *productF-simps*[*simp*]:
  productF A B FAIL AB ⟹ *succ AB a* (*p1*,*p2*) = (*if* (*p1* ∈ *nodes A* ∧ *p2* ∈ *nodes B*
                ∧ (*fst a* ∈ *inputs A*) ∧ (*snd a* ∈ *outputs A* ∪ *outputs B*))
              then (*if* (*succ A a p1* = {} ∧ *succ B a p2* ≠ {})
                then {*FAIL*}
                else (*succ A a p1* × *succ B a p2*))
              else {})
  productF A B FAIL AB ⟹ *inputs AB* = *inputs A*
  productF A B FAIL AB ⟹ *outputs AB* = *outputs A* ∪ *outputs B*
  productF A B FAIL AB ⟹ *initial AB* = (*initial A*, *initial B*)
  unfolding *productF.simps* **by** *simp*+

lemma *fail-next-productF* :
  assumes *well-formed M1*
  and *well-formed M2*
  and *productF M2 M1 FAIL PM*
shows *succ PM a FAIL* = {}
proof (*cases* ((*fst FAIL*) ∈ *nodes M2* ∧ (*snd FAIL*) ∈ *nodes M1*))
  case *True*
  then show *?thesis*
    using *assms* **by** *auto*
next
  case *False*
  then show *?thesis*
    using *assms* **by** (*cases* (*succ M2 a* (*fst FAIL*) = {} ∧ (*fst a* ∈ *inputs M2*)
                ∧ (*snd a* ∈ *outputs M2*)); *auto*)
qed

lemma *nodes-productF* :
  assumes *well-formed M1*
  and *well-formed M2*
  and *productF M2 M1 FAIL PM*
shows *nodes PM* ⊆ *insert FAIL* (*nodes M2* × *nodes M1*)
proof
  fix *q* **assume** *q-assm* : *q* ∈ *nodes PM*
  then show *q* ∈ *insert FAIL* (*nodes M2* × *nodes M1*)
  using *assms* **proof** (*cases*)
    case *initial*
    then show *?thesis* **using** *assms* **by** *auto*
  next

```
    case (execute p a)
    then obtain p1 p2 x y q1 q2  where p-a-split[simp] : p = (p1,p2)
                                                     a = ((x,y),q)
                                                     q = (q1,q2)
      by (metis eq-snd-iff)
    have subnodes : p1 ∈ nodes M2 ∧ p2 ∈ nodes M1 ∧ x ∈ inputs M2 ∧ y ∈ outputs M2 ∪ outputs M1
    proof (rule ccontr)
      assume ¬ (p1 ∈ nodes M2 ∧ p2 ∈ nodes M1  ∧ x ∈ inputs M2 ∧ y ∈ outputs M2 ∪ outputs M1)
      then have succ PM (x,y) (p1,p2) = {}
        using assms(3) by auto
      then show False
        using execute by auto
    qed

    show ?thesis proof (cases (succ M2 (x,y) p1 = {} ∧ succ M1 (x,y) p2 ≠ {}))
      case True
      then have q = FAIL
        using subnodes assms(3) execute by auto
      then show ?thesis
        by auto
    next
      case False
      then have succ PM (fst a) p = succ M2 (x,y) p1 × succ M1 (x,y) p2
        using subnodes assms(3) execute by auto
      then have q ∈ (succ M2 (x,y) p1 × succ M1 (x,y) p2)
        using execute by blast
      then have q-succ : (q1,q2) ∈ (succ M2 (x,y) p1 × succ M1 (x,y) p2)
        by simp

      have q1 ∈ succ M2 (x,y) p1
        using q-succ by simp
      then have q1 ∈ successors M2 p1
        by auto
      then have q1 ∈ reachable M2 p1
        by blast
      then have q1 ∈ reachable M2 (initial M2)
        using subnodes by blast
      then have nodes1 : q1 ∈ nodes M2
        by blast

      have q2 ∈ succ M1 (x,y) p2
        using q-succ by simp
      then have q2 ∈ successors M1 p2
        by auto
      then have q2 ∈ reachable M1 p2
        by blast
      then have q2 ∈ reachable M1 (initial M1)
        using subnodes by blast
      then have nodes2 : q2 ∈ nodes M1
        by blast

      show ?thesis
        using nodes1 nodes2 by auto
    qed
  qed
qed
```

```
lemma well-formed-productF[simp] :
  assumes well-formed M1
  and     well-formed M2
  and     productF M2 M1 FAIL PM
shows well-formed PM
unfolding well-formed.simps proof
```

have *finite* (*nodes M1*) *finite* (*nodes M2*)
  using *assms* **by** *auto*
then have *finite* (*insert FAIL* (*nodes M2* × *nodes M1*))
  by *simp*
moreover have *nodes PM* ⊆ *insert FAIL* (*nodes M2* × *nodes M1*)
  using *nodes-productF assms* **by** *blast*
moreover have *inputs PM* = *inputs M2 outputs PM* = *outputs M2* ∪ *outputs M1*
  using *assms* **by** *auto*
ultimately show *finite-FSM PM*
  using *infinite-subset assms* **by** *auto*
next
have *inputs PM* = *inputs M2 outputs PM* = *outputs M2* ∪ *outputs M1*
  using *assms* **by** *auto*
then show (∀ *s1 x y*. *x* ∉ *inputs PM* ∨ *y* ∉ *outputs PM* ⟶ *succ PM* (*x*, *y*) *s1* = {})
  ∧ *inputs PM* ≠ {} ∧ *outputs PM* ≠ {}
  using *assms* **by** *auto*
qed

lemma *observable-productF*[*simp*] :
assumes *observable M1*
and *observable M2*
and *productF M2 M1 FAIL PM*
shows *observable PM*
unfolding *observable.simps*
proof −
have ∀ *t s* . *succ M1 t* (*fst s*) = {} ∨ (∃ *s2*. *succ M1 t* (*fst s*) = {*s2*})
  using *assms* **by** *auto*
moreover have ∀ *t s* . *succ M2 t* (*snd s*) = {} ∨ (∃ *s2*. *succ M2 t* (*snd s*) = {*s2*})
  using *assms* **by** *auto*
ultimately have *sub-succs* : ∀ *t s* . *succ M2 t* (*fst s*) × *succ M1 t* (*snd s*) = {}
  ∨ (∃ *s2* . *succ M2 t* (*fst s*) × *succ M1 t* (*snd s*) = {*s2*})
  by *fastforce*
moreover have *succ-split* : ∀ *t s* . *succ PM t s* = {}
  ∨ *succ PM t s* = {*FAIL*}
  ∨ *succ PM t s* = *succ M2 t* (*fst s*) × *succ M1 t* (*snd s*)
  using *assms* **by** *auto*
ultimately show ∀ *t s*. *succ PM t s* = {} ∨ (∃ *s2*. *succ PM t s* = {*s2*})
  by *metis*
qed

lemma *no-transition-after-FAIL* :
assumes *productF A B FAIL AB*
shows *succ AB io FAIL* = {}
using *assms* **by** *auto*

lemma *no-prefix-targets-FAIL* :
assumes *productF M2 M1 FAIL PM*
and *path PM p q*
and *k* < *length p*
shows *target* (*take k p*) *q* ≠ *FAIL*
proof
assume *assm* : *target* (*take k p*) *q* = *FAIL*
have *path PM* (*take k p* @ *drop k p*) *q*
  using *assms* **by** *auto*
then have *path PM* (*drop k p*) (*target* (*take k p*) *q*)
  by *blast*
then have *path-from-FAIL* : *path PM* (*drop k p*) *FAIL*
  using *assm* **by** *auto*

have *length* (*drop k p*) ≠ *0*
  using *assms* **by** *auto*
then obtain *io q* **where** *drop k p* = (*io*,*q*) # (*drop* (*Suc k*) *p*)
  by (*metis Cons-nth-drop-Suc assms*(*3*) *prod-cases3*)
then have *succ PM io FAIL* ≠ {}

```
    using path-from-FAIL by auto

  then show False
    using no-transition-after-FAIL assms by auto
qed


lemma productF-path-inclusion :
  assumes length w = length r1 length r1 = length r2
  and     productF A B FAIL AB
  and     well-formed A
  and     well-formed B
  and     path A (w || r1) p1 ∧ path B (w || r2) p2
  and     p1 ∈ nodes A
  and     p2 ∈ nodes B
shows path (AB) (w || r1 || r2) (p1, p2)
using assms proof (induction w r1 r2 arbitrary: p1 p2 rule: list-induct3)
  case Nil
  then show ?case by auto
next
  case (Cons w ws r1 r1s r2 r2s)
  then have path A ([w] || [r1]) p1 ∧ path B ([w] || [r2]) p2
    by auto
  then have succs : r1 ∈ succ A w p1 ∧ r2 ∈ succ B w p2
    by auto
  then have succ A w p1 ≠ {}
    by force
  then have w-elem : fst w ∈ inputs A ∧ snd w ∈ outputs A
    using Cons by (metis assms(4) prod.collapse well-formed.elims(2))
  then have (r1,r2) ∈ succ AB w (p1,p2)
    using Cons succs by auto
  then have path-head : path AB ([w] || [(r1,r2)]) (p1,p2)
    by auto

  have path A (ws || r1s) r1 ∧ path B (ws || r2s) r2
    using Cons by auto
  moreover have r1 ∈ nodes A ∧ r2 ∈ nodes B
    using succs Cons.prems succ-nodes[of r1 A w p1] succ-nodes[of r2 B w p2] by auto
  ultimately have path AB (ws || r1s || r2s) (r1,r2)
    using Cons by blast

  then show ?case
    using path-head by auto
qed

lemma productF-path-forward :
  assumes length w = length r1 length r1 = length r2
  and     productF A B FAIL AB
  and     well-formed A
  and     well-formed B
  and     (path A (w || r1) p1 ∧ path B (w || r2) p2)
          ∨ (target (w || r1 || r2) (p1, p2) = FAIL
            ∧ length w > 0
            ∧ path A (butlast (w || r1)) p1
            ∧ path B (butlast (w || r2)) p2
            ∧ succ A (last w) (target (butlast (w || r1)) p1) = {}
            ∧ succ B (last w) (target (butlast (w || r2)) p2) ≠ {})
  and     p1 ∈ nodes A
  and     p2 ∈ nodes B
shows path (AB) (w || r1 || r2) (p1, p2)
using assms proof (induction w r1 r2 arbitrary: p1 p2 rule: list-induct3)
  case Nil
  then show ?case by auto
next
  case (Cons w ws r1 r1s r2 r2s)
  then show ?case
```

```
proof (cases (path A (w # ws || r1 # r1s) p1 ∧ path B (w # ws || r2 # r2s) p2))
  case True
  then show ?thesis
    using Cons productF-path-inclusion[of w # ws r1 # r1s r2 # r2s A B FAIL AB p1 p2]
    by auto
next
  case False
  then have fail-prop : target (w # ws || r1 # r1s || r2 # r2s) (p1, p2) = FAIL ∧
            0 < length (w # ws) ∧
            path A (butlast (w # ws || r1 # r1s)) p1 ∧
            path B (butlast (w # ws || r2 # r2s)) p2 ∧
            succ A (last (w # ws)) (target (butlast (w # ws || r1 # r1s)) p1) = {} ∧
            succ B (last (w # ws)) (target (butlast (w # ws || r2 # r2s)) p2) ≠ {}
    using Cons.prems by fastforce


  then show ?thesis
  proof (cases length ws)
    case 0
    then have empty[simp] : ws = [] r1s = [] r2s = []
      using Cons.hyps by auto
    then have fail-prop-0 : target ( [w] || [r1] || [r2]) (p1, p2) = FAIL ∧
            0 < length ([w]) ∧
            path A [] p1 ∧
            path B [] p2 ∧
            succ A w p1 = {} ∧
            succ B w p2 ≠ {}
      using fail-prop by auto
    then have fst w ∈ inputs B ∧ snd w ∈ outputs B
      using Cons.prems by (metis prod.collapse well-formed.elims(2))
    then have inputs-0 : fst w ∈ inputs A ∧ snd w ∈ outputs B
      using Cons.prems by auto

    moreover have fail-elems-0 : (r1,r2) = FAIL
      using fail-prop by auto
    ultimately have succ AB w (p1,p2) = {FAIL}
      using fail-prop-0 Cons.prems by auto

    then have path AB ( [w] || [r1] || [r2]) (p1, p2)
      using Cons.prems fail-elems-0 by auto
    then show ?thesis
      by auto
  next
    case (Suc nat)

    then have path-r1 : path A ([w] || [r1]) p1
      using fail-prop
      by (metis Cons.hyps(1) FSM.nil FSM.path.intros(2) FSM.path-cons-elim Suc-neq-Zero
          butlast.simps(2) length-0-conv zip-Cons-Cons zip-Nil zip-eq)
    then have path-r1s : path A (butlast (ws || r1s)) r1
      using Suc
      by (metis (no-types, lifting) Cons.hyps(1) FSM.path-cons-elim Suc-neq-Zero butlast.simps(2)
          fail-prop length-0-conv snd-conv zip.simps(1) zip-Cons-Cons zip-eq)

    have path-r2 : path B ([w] || [r2]) p2
      using Suc fail-prop
      by (metis Cons.hyps(1) Cons.hyps(2) FSM.nil FSM.path.intros(2) FSM.path-cons-elim
          Suc-neq-Zero butlast.simps(2) length-0-conv zip-Cons-Cons zip-Nil zip-eq)
    then have path-r2s : path B (butlast (ws || r2s)) r2
      using Suc
      by (metis (no-types, lifting) Cons.hyps(1) Cons.hyps(2) FSM.path-cons-elim Suc-neq-Zero
          butlast.simps(2) fail-prop length-0-conv snd-conv zip.simps(1) zip-Cons-Cons zip-eq)

    have target (ws || r1s || r2s) (r1, r2) = FAIL
      using fail-prop by auto
    moreover have r1 ∈ nodes A
```

```
      using Cons.prems path-r1 by (metis FSM.path-cons-elim snd-conv succ-nodes zip-Cons-Cons)
    moreover have r2 ∈ nodes B
      using Cons.prems path-r2 by (metis FSM.path-cons-elim snd-conv succ-nodes zip-Cons-Cons)
    moreover have succ A (last ws) (target (butlast (ws || r1s)) r1) = {}
      by (metis (no-types, lifting) Cons.hyps(1) Suc Suc-neq-Zero butlast.simps(2) fail-prop
          fold-simps(2) last-ConsR list.size(3) snd-conv zip-Cons-Cons zip-Nil zip-eq)
    moreover have succ B (last ws) (target (butlast (ws || r2s)) r2) ≠ {}
      by (metis (no-types, lifting) Cons.hyps(1) Cons.hyps(2) Suc Suc-neq-Zero butlast.simps(2)
          fail-prop fold-simps(2) last-ConsR list.size(3) snd-conv zip-Cons-Cons zip-Nil zip-eq)

    have path AB (ws || r1s || r2s) (r1, r2)
      using Cons.IH Suc ‹succ B (last ws) (target (butlast (ws || r2s)) r2) ≠ {}›
            assms(3) assms(4) assms(5) calculation(1−4) path-r1s path-r2s zero-less-Suc
      by presburger
    moreover have path AB ([w] || [r1] || [r2]) (p1,p2)
      using path-r1 path-r2 productF-path-inclusion[of [w] [r1] [r2] A B FAIL AB p1 p2]
            Cons.prems
      by auto
    ultimately show ?thesis
      by auto
  qed
 qed
qed


lemma butlast-zip-cons : length ws = length r1s ⟹ ws ≠ []
                           ⟹ butlast (w # ws || r1 # r1s) = ((w,r1) # (butlast (ws || r1s)))
proof −
assume a1: length ws = length r1s
assume a2: ws ≠ []
 have length (w # ws) = length r1s + Suc 0
   using a1 by (metis list.size(4))
 then have f3: length (w # ws) = length (r1 # r1s)
   by (metis list.size(4))
 have f4: ws @ w # ws ≠ w # ws
   using a2 by (meson append-self-conv2)
 have length (ws @ w # ws) = length (r1s @ r1 # r1s)
   using a1 by auto
 then have ws @ w # ws || r1s @ r1 # r1s ≠ w # ws || r1 # r1s
   using f4 f3 by (meson zip-eq)
 then show ?thesis
   using a1 by simp
qed


lemma productF-succ-fail-imp :
 assumes productF A B FAIL AB
 and     FAIL ∈ succ AB w (p1,p2)
 and     well-formed A
 and     well-formed B
shows p1 ∈ nodes A ∧ p2 ∈ nodes B ∧ (fst w ∈ inputs A) ∧ (snd w ∈ outputs A ∪ outputs B)
      ∧ succ AB w (p1,p2) = {FAIL} ∧ succ A w p1 = {} ∧ succ B w p2 ≠ {}
proof −
 have path-head : path AB ([w] || [FAIL]) (p1,p2)
   using assms by auto
 then have succ-nonempty : succ AB w (p1,p2) ≠ {}
   by force
 then have succ-if-1 : p1 ∈ nodes A ∧ p2 ∈ nodes B ∧ (fst w ∈ inputs A)
                        ∧ (snd w ∈ outputs A ∪ outputs B)
   using assms by auto
 then have (p1,p2) ≠ FAIL
   using assms by auto
```

```
  have succ A w p1 ⊆ nodes A
    using assms succ-if-1 by (simp add: subsetI succ-nodes)
  moreover have succ B w p2 ⊆ nodes B
    using assms succ-if-1 by (simp add: subsetI succ-nodes)
  ultimately have FAIL ∉ (succ A w p1 × succ B w p2)
    using assms by auto
  then have succ-no-inclusion : succ AB w (p1,p2) ≠ (succ A w p1 × succ B w p2)
    using assms succ-if-1 by blast
  moreover have succ AB w (p1,p2) = {} ∨ succ AB w (p1,p2) = {FAIL}
                ∨ succ AB w (p1,p2) = (succ A w p1 × succ B w p2)
    using assms by simp
  ultimately have succ-fail : succ AB w (p1,p2) = {FAIL}
    using succ-nonempty by simp

  have succ A w p1 = {} ∧ succ B w p2 ≠ {}
  proof (rule ccontr)
    assume ¬ (succ A w p1 = {} ∧ succ B w p2 ≠ {})
    then have succ AB w (p1,p2) = (succ A w p1 × succ B w p2)
      using assms by auto
    then show False
      using succ-no-inclusion by simp
  qed

  then show ?thesis
    using succ-if-1 succ-fail by simp
qed



lemma productF-path-reverse :
  assumes length w = length r1 length r1 = length r2
  and     productF A B FAIL AB
  and     well-formed A
  and     well-formed B
  and     path AB (w || r1 || r2) (p1, p2)
  and     p1 ∈ nodes A
  and     p2 ∈ nodes B
shows (path A (w || r1) p1 ∧ path B (w || r2) p2)
        ∨ (target (w || r1 || r2) (p1, p2) = FAIL
          ∧ length w > 0
          ∧ path A (butlast (w || r1)) p1
          ∧ path B (butlast (w || r2)) p2
          ∧ succ A (last w) (target (butlast (w || r1)) p1) = {}
          ∧ succ B (last w) (target (butlast (w || r2)) p2) ≠ {})
using assms  proof (induction w r1 r2 arbitrary: p1 p2 rule: list-induct3)
  case Nil
  then show ?case by auto
next
  case (Cons w ws r1 r1s r2 r2s)

  have path-head : path AB ([w] || [(r1,r2)]) (p1,p2) using Cons by auto
  then have succ-nonempty : succ AB w (p1,p2) ≠ {} by force
  then have succ-if-1 : p1 ∈ nodes A ∧ p2 ∈ nodes B ∧ (fst w ∈ inputs A)
                      ∧ (snd w ∈ outputs A ∪ outputs B)
    using Cons by fastforce
  then have (p1,p2) ≠ FAIL
    using Cons by auto

  have path-tail : path AB (ws || r1s || r2s) (r1,r2)
    using path-head Cons  by auto

  show ?case
  proof (cases (r1,r2) = FAIL)
    case True
    have r1s = []
    proof (rule ccontr)
```

```
    assume ¬ (r1s = [])
    then have (¬ (ws = [])) ∧ (¬ (r1s = [])) ∧ (¬ (r2s = []))
      using Cons.hyps by auto
    moreover have path AB (ws || r1s || r2s) FAIL
      using True path-tail by simp
    ultimately have path AB ([hd ws] @ tl ws || [hd r1s] @ tl r1s || [hd r2s] @ tl r2s) FAIL
      by simp
    then have path AB ([hd ws] || [hd r1s] || [hd r2s]) FAIL
      by auto
    then have succ AB (hd ws) FAIL ≠ {}
      by auto
    then show False using no-transition-after-FAIL
      using Cons.prems by auto
  qed
  then have tail-nil : ws = [] ∧ r1s = [] ∧ r2s = []
    using Cons.hyps by simp

  have succ-fail : FAIL ∈ succ AB w (p1,p2)
    using path-head True by auto

  then have succs : succ A w p1 = {} ∧ succ B w p2 ≠ {}
    using Cons.prems by (meson productF-succ-fail-imp)

  have  target (w # ws || r1 # r1s || r2 # r2s) (p1, p2) = FAIL
    using True tail-nil by simp
  moreover have 0 < length (w # ws)
    by simp
  moreover have path A (butlast (w # ws || r1 # r1s)) p1
    using tail-nil by auto
  moreover have path B (butlast (w # ws || r2 # r2s)) p2
    using tail-nil by auto
  moreover have succ A (last (w # ws)) (target (butlast (w # ws || r1 # r1s)) p1) = {}
    using succs tail-nil by simp
  moreover have succ B (last (w # ws)) (target (butlast (w # ws || r2 # r2s)) p2) ≠ {}
    using succs tail-nil by simp
  ultimately show ?thesis
    by simp
next
  case False

  have (r1,r2) ∈ succ AB w (p1,p2)
    using path-head by auto
  then have succ-not-fail : succ AB w (p1,p2) ≠ {FAIL}
    using succ-nonempty False by auto

  have ¬ (succ A w p1 = {} ∧ succ B w p2 ≠ {})
  proof (rule ccontr)
    assume ¬ ¬ (succ A w p1 = {} ∧ succ B w p2 ≠ {})
    then have succ AB w (p1,p2) = {FAIL}
      using succ-if-1 Cons by auto
    then show False
      using succ-not-fail by simp
  qed

  then have succ AB w (p1,p2) = (succ A w p1 × succ B w p2)
    using succ-if-1 Cons by auto
  then have (r1,r2) ∈ (succ A w p1 × succ B w p2)
    using Cons by auto
  then have succs-next : r1 ∈ succ A w p1 ∧ r2 ∈ succ B w p2
    by auto
  then have nodes-next : r1 ∈ nodes A ∧ r2 ∈ nodes B
    using Cons succ-nodes by metis

  moreover have path-tail : path AB (ws || r1s || r2s) (r1,r2)
    using Cons by auto
  ultimately have prop-tail :
```

```
      path A (ws || r1s) r1 ∧ path B (ws || r2s) r2 ∨
        target (ws || r1s || r2s) (r1, r2) = FAIL ∧
        0 < length ws ∧
        path A (butlast (ws || r1s)) r1 ∧
        path B (butlast (ws || r2s)) r2 ∧
        succ A (last ws) (target (butlast (ws || r1s)) r1) = {} ∧
        succ B (last ws) (target (butlast (ws || r2s)) r2) ≠ {}
    using Cons.IH[of r1 r2] Cons.prems by auto

  moreover have path A ([w] || [r1]) p1 ∧ path B ([w] || [r2]) p2
    using succs-next by auto
  then show ?thesis
  proof (cases path A (ws || r1s) r1 ∧ path B (ws || r2s) r2)
    case True
    moreover have paths-head : path A ([w] || [r1]) p1 ∧ path B ([w] || [r2]) p2
      using succs-next by auto
    ultimately show ?thesis
      by (metis (no-types) FSM.path.simps FSM.path-cons-elim True eq-snd-iff
          paths-head zip-Cons-Cons)
  next
    case False

    then have fail-prop : target (ws || r1s || r2s) (r1, r2) = FAIL ∧
          0 < length ws ∧
          path A (butlast (ws || r1s)) r1 ∧
          path B (butlast (ws || r2s)) r2 ∧
          succ A (last ws) (target (butlast (ws || r1s)) r1) = {} ∧
          succ B (last ws) (target (butlast (ws || r2s)) r2) ≠ {}
      using prop-tail by auto

    then have paths-head : path A ([w] || [r1]) p1 ∧ path B ([w] || [r2]) p2
      using succs-next by auto

    have (last (w # ws)) = last ws
      using fail-prop by simp
    moreover have (target (butlast (w # ws || r1 # r1s)) p1) = (target (butlast (ws || r1s)) r1)
      using fail-prop Cons.hyps(1) butlast-zip-cons by auto
    moreover have (target (butlast (w # ws || r2 # r2s)) p2) = (target (butlast (ws || r2s)) r2)
      using fail-prop Cons.hyps(1) Cons.hyps(2) butlast-zip-cons by auto
    ultimately have succ A (last (w # ws)) (target (butlast (w # ws || r1 # r1s)) p1) = {}
                ∧ succ B (last (w # ws)) (target (butlast (w # ws || r2 # r2s)) p2) ≠ {}
      using fail-prop by auto
    moreover have path A (butlast (w # ws || r1 # r1s)) p1
      using fail-prop paths-head by auto
    moreover have path B (butlast (w # ws || r2 # r2s)) p2
      using fail-prop paths-head by auto
    moreover have target (w # ws || r1 # r1s || r2 # r2s) (p1, p2) = FAIL
      using fail-prop paths-head by auto
    ultimately show ?thesis
      by simp
  qed

 qed
qed

lemma butlast-zip[simp] :
  assumes length xs = length ys
  shows butlast (xs || ys) = (butlast xs || butlast ys)
  using assms by (metis (no-types, lifting) map-butlast map-fst-zip map-snd-zip zip-map-fst-snd)



lemma productF-path-reverse-ob :
  assumes length w = length r1 length r1 = length r2
  and     productF A B FAIL AB
  and     well-formed A
```

```
  and     well-formed B
  and     path AB (w || r1 || r2) (p1, p2)
  and     p1 ∈ nodes A
  and     p2 ∈ nodes B
obtains r2'
where path B (w || r2') p2 ∧ length w = length r2'
proof −
  have path-prop : (path A (w || r1) p1 ∧ path B (w || r2) p2)
                    ∨ (target (w || r1 || r2) (p1, p2) = FAIL
                      ∧ length w > 0
                      ∧ path A (butlast (w || r1)) p1
                      ∧ path B (butlast (w || r2)) p2
                      ∧ succ A (last w) (target (butlast (w || r1)) p1) = {}
                      ∧ succ B (last w) (target (butlast (w || r2)) p2) ≠ {})
    using assms productF-path-reverse[of w r1 r2 A B FAIL AB p1 p2] by simp
  have ∃ r1' . path B (w || r1') p2 ∧ length w = length r1'
  proof (cases path A (w || r1) p1 ∧ path B (w || r2) p2)
    case True
    then show ?thesis
      using assms by auto
  next
    case False
    then have B-prop : length w > 0
                ∧ path B (butlast (w || r2)) p2
                ∧ succ B (last w) (target (butlast (w || r2)) p2) ≠ {}
      using path-prop by auto
    then obtain rx where rx ∈ succ B (last w) (target (butlast (w || r2)) p2)
      by auto

    then have path B ([last w] || [rx]) (target (butlast (w || r2)) p2)
      using B-prop by auto
    then have path B ((butlast (w || r2)) @ ([last w] || [rx])) p2
      using B-prop by auto
    moreover have butlast (w || r2) = (butlast w || butlast r2)
      using assms by simp

    ultimately have path B ((butlast w) @ [last w] || (butlast r2) @ [rx]) p2
      using assms B-prop by auto
    moreover have (butlast w) @ [last w] = w
      using B-prop by simp
    moreover have length ((butlast r2) @ [rx]) = length w
      using assms B-prop by auto
    ultimately show ?thesis
      by auto
  qed
  then obtain r1' where path B (w || r1') p2 ∧ length w = length r1'
    by blast
  then show ?thesis
    using that by blast
qed
```

The following lemma formalizes the property of paths of the product machine as described in the section introduction.

```
lemma productF-path[iff] :
  assumes length w = length r1 length r1 = length r2
  and     productF A B FAIL AB
  and     well-formed A
  and     well-formed B
  and     p1 ∈ nodes A
  and     p2 ∈ nodes B
shows path AB (w || r1 || r2) (p1, p2) ⟷ ((path A (w || r1) p1 ∧ path B (w || r2) p2)
            ∨ (target (w || r1 || r2) (p1, p2) = FAIL
              ∧ length w > 0
              ∧ path A (butlast (w || r1)) p1
              ∧ path B (butlast (w || r2)) p2
              ∧ succ A (last w) (target (butlast (w || r1)) p1) = {}
```

∧ *succ B* (*last w*) (*target* (*butlast* (*w* || *r2*)) *p2*) ≠ {})) (**is** *?path* ⟷ *?paths*)
proof
assume *?path*
then show *?paths* **using** *assms productF-path-reverse*[*of w r1 r2 A B FAIL AB p1 p2*] **by** *simp*
next
assume *?paths*
then show *?path* **using** *assms productF-path-forward*[*of w r1 r2 A B FAIL AB p1 p2*] **by** *simp*
qed

lemma *path-last-succ* :
assumes *path A* (*ws* || *r1s*) *p1*
and *length r1s* = *length ws*
and *length ws* > *0*
shows *last r1s* ∈ *succ A* (*last ws*) (*target* (*butlast* (*ws* || *r1s*)) *p1*)
proof −
have *path A* (*butlast* (*ws* || *r1s*)) *p1*
∧ *path A* [*last* (*ws* || *r1s*)] (*target* (*butlast* (*ws* || *r1s*)) *p1*)
by (*metis FSM.path-append-elim append-butlast-last-id assms length-greater-0-conv list.size*(*3*) *zip-Nil zip-eq*)

then have *snd* (*last* (*ws* || *r1s*)) ∈
succ A (*fst* (*last* (*ws* || *r1s*))) (*target* (*butlast* (*ws* || *r1s*)) *p1*)
by *auto*
moreover have *ws* || *r1s* ≠ []
using *assms*(*3*) *assms*(*2*) **by** (*metis length-zip list.size*(*3*) *min.idem neq0-conv*)
ultimately have *last r1s* ∈ *succ A* (*last ws*) (*target* (*butlast* (*ws* || *r1s*)) *p1*)
by (*simp add*: *assms*(*2*))
then show *?thesis*
by *auto*
qed

lemma *zip-last* :
assumes *length r1* > *0*
and *length r1* = *length r2*
shows *last* (*r1* || *r2*) = (*last r1*, *last r2*)
by (*metis* (*no-types*) *assms*(*1*) *assms*(*2*) *less-nat-zero-code list.size*(*3*) *map-fst-zip zip-Nil zip-last*)

lemma *productF-path-reverse-ob-2* :
assumes *length w* = *length r1 length r1* = *length r2*
and *productF A B FAIL AB*
and *well-formed A*
and *well-formed B*
and *path AB* (*w* || *r1* || *r2*) (*p1*, *p2*)
and *p1* ∈ *nodes A*
and *p2* ∈ *nodes B*
and *w* ∈ *language-state A p1*
and *observable A*
shows *path A* (*w* || *r1*) *p1* ∧ *length w* = *length r1 path B* (*w* || *r2*) *p2* ∧ *length w* = *length r2*
target (*w* || *r1*) *p1* = *fst* (*target* (*w* || *r1* || *r2*) (*p1*,*p2*))
target (*w* || *r2*) *p2* = *snd* (*target* (*w* || *r1* || *r2*) (*p1*,*p2*))
proof −

have (*path A* (*w* || *r1*) *p1* ∧ *path B* (*w* || *r2*) *p2*)
∨ (*target* (*w* || *r1* || *r2*) (*p1*, *p2*) = *FAIL*
∧ *length w* > *0*
∧ *path A* (*butlast* (*w* || *r1*)) *p1*
∧ *path B* (*butlast* (*w* || *r2*)) *p2*
∧ *succ A* (*last w*) (*target* (*butlast* (*w* || *r1*)) *p1*) = {}
∧ *succ B* (*last w*) (*target* (*butlast* (*w* || *r2*)) *p2*) ≠ {})
using *productF-path*[*of w r1 r2 A B FAIL AB p1 p2*] *assms* **by** *blast*

moreover have *path A* (*butlast* (*w* || *r1*)) *p1*

∧ *succ A* (*last w*) (*target* (*butlast* (*w* || *r1*)) *p1*) = {}
∧ *length w* > *0* ⟹ *False*
proof −
assume *assm* : *path A* (*butlast* (*w* || *r1*)) *p1*
∧ *succ A* (*last w*) (*target* (*butlast* (*w* || *r1*)) *p1*) = {}
∧ *length w* > *0*
obtain *r1'* **where** *r1'-def* : *path A* (*w* || *r1'*) *p1* ∧ *length r1'* = *length w*
using *assms*(*9*) **by** *auto*
then have *path A* (*butlast* (*w* || *r1'*)) *p1* ∧ *length* (*butlast r1'*) = *length* (*butlast w*)
by (*metis FSM.path-append-elim append-butlast-last-id butlast.simps*(*1*) *length-butlast*)
moreover have *path A* (*butlast* (*w* || *r1*)) *p1* ∧ *length* (*butlast r1*) = *length* (*butlast w*)
using *assm assms*(*1*) **by** *auto*
ultimately have *butlast r1* = *butlast r1'*
by (*metis assms*(*1*) *assms*(*10*) *butlast-zip language-state observable-path-unique r1'-def*)

then have *butlast* (*w* || *r1*) = *butlast* (*w* || *r1'*)
using *assms*(*1*) *r1'-def* **by** *simp*
moreover have *succ A* (*last w*) (*target* (*butlast* (*w* || *r1'*)) *p1*) ≠ {}
by (*metis* (*no-types*) *assm empty-iff path-last-succ r1'-def*)
ultimately show *False*
using *assm* **by** *auto*
qed

ultimately have *paths* : (*path A* (*w* || *r1*) *p1* ∧ *path B* (*w* || *r2*) *p2*)
by *auto*

show *path A* (*w* || *r1*) *p1* ∧ *length w* = *length r1*
using *assms*(*1*) *paths* **by** *simp*
show *path B* (*w* || *r2*) *p2* ∧ *length w* = *length r2*
using *assms*(*1*) *assms*(*2*) *paths* **by** *simp*

have *length w* = *0* ⟹ *target* (*w* || *r1* || *r2*) (*p1*,*p2*) = (*p1*,*p2*)
by *simp*
moreover have *length w* > *0* ⟹ *target* (*w* || *r1* || *r2*) (*p1*,*p2*) = *last* (*r1* || *r2*)
proof −
assume *length w* > *0*
moreover have *length w* = *length* (*r1* || *r2*)
using *assms*(*1*) *assms*(*2*) **by** *simp*
ultimately show *?thesis*
using *target-alt-def*(*2*)[*of w r1* || *r2* (*p1*,*p2*)] **by** *simp*
qed

ultimately have *target* (*w* || *r1*) *p1* = *fst* (*target* (*w* || *r1* || *r2*) (*p1*, *p2*))
∧ *target* (*w* || *r2*) *p2* = *snd* (*target* (*w* || *r1* || *r2*) (*p1*, *p2*))
proof (*cases length w*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc nat*)
then have *length w* > *0* **by** *simp*

have *target* (*w* || *r1* || *r2*) (*p1*,*p2*) = *last* (*r1* || *r2*)
proof −
have *length w* = *length* (*r1* || *r2*)
using *assms*(*1*) *assms*(*2*) **by** *simp*
then show *?thesis*
using ‹*length w* > *0*› *target-alt-def*(*2*)[*of w r1* || *r2* (*p1*,*p2*)] **by** *simp*
qed
moreover have *target* (*w* || *r1*) *p1* = *last r1*
using ‹*length w* > *0*› *target-alt-def*(*2*)[*of w r1 p1*] *assms*(*1*) **by** *simp*
moreover have *target* (*w* || *r2*) *p2* = *last r2*
using ‹*length w* > *0*› *target-alt-def*(*2*)[*of w r2 p2*] *assms*(*1*) *assms*(*2*) **by** *simp*
moreover have *last* (*r1* || *r2*) = (*last r1*, *last r2*)
using ‹*length w* > *0*› *assms*(*1*) *assms*(*2*) *zip-last*[*of r1 r2*] **by** *simp*
ultimately show *?thesis*
by *simp*

```
  qed

  then show target (w || r1) p1 = fst (target (w || r1 || r2) (p1,p2))
            target (w || r2) p2 = snd (target (w || r1 || r2) (p1,p2))
    by simp+
qed

lemma productF-path-unzip :
  assumes productF A B FAIL AB
  and     path AB (w || tr) q
  and     length tr = length w
shows path AB (w || (map fst tr || map snd tr)) q
proof –
  have map fst tr || map snd tr = tr
    by auto
  then show ?thesis
    using assms by auto
qed

lemma productF-path-io-targets :
  assumes productF A B FAIL AB
  and     io-targets AB (qA,qB) w = {(pA,pB)}
  and     w ∈ language-state A qA
  and     w ∈ language-state B qB
  and     observable A
  and     observable B
  and     well-formed A
  and     well-formed B
  and     qA ∈ nodes A
  and     qB ∈ nodes B
shows pA ∈ io-targets A qA w pB ∈ io-targets B qB w
proof –
  obtain tr where tr-def : target (w || tr) (qA,qB) = (pA,pB)
                           ∧ path AB (w || tr) (qA,qB)
                           ∧ length w = length tr using assms(2)
    by blast
  have path-A : path A (w || map fst tr) qA ∧ length w = length (map fst tr)
    using productF-path-reverse-ob-2[of w map fst tr map snd tr A B FAIL AB qA qB]
          assms tr-def by auto
  have path-B : path B (w || map snd tr) qB ∧ length w = length (map snd tr)
    using productF-path-reverse-ob-2[of w map fst tr map snd tr A B FAIL AB qA qB]
          assms tr-def by auto

  have targets : target (w || map fst tr) qA = pA ∧ target (w || map snd tr) qB = pB
  proof (cases tr)
    case Nil
    then have qA = pA ∧ qB = pB
      using tr-def by auto
    then show ?thesis
      by (simp add: local.Nil)
  next
    case (Cons a list)
    then have last tr = (pA,pB)
      using tr-def by (simp add: tr-def FSM.target-alt-def states-alt-def)

    moreover have target (w || map fst tr) qA = last (map fst tr)
      using Cons by (simp add: FSM.target-alt-def states-alt-def tr-def)
    moreover have last (map fst tr) = fst (last tr)
      using last-map Cons by blast
```

moreover have *target* (*w* || *map snd tr*) *qB* = *last* (*map snd tr*)
using *Cons* **by** (*simp add*: *FSM.target-alt-def states-alt-def tr-def*)
moreover have *last* (*map snd tr*) = *snd* (*last tr*)
using *last-map Cons* **by** *blast*

ultimately show *?thesis*
by *simp*
qed

show *pA* ∈ *io-targets A qA w*
using *path-A targets* **by** *auto*
show *pB* ∈ *io-targets B qB w*
using *path-B targets* **by** *auto*
qed

lemma *productF-path-io-targets-reverse* :
assumes *productF A B FAIL AB*
and *pA* ∈ *io-targets A qA w*
and *pB* ∈ *io-targets B qB w*
and *w* ∈ *language-state A qA*
and *w* ∈ *language-state B qB*
and *observable A*
and *observable B*
and *well-formed A*
and *well-formed B*
and *qA* ∈ *nodes A*
and *qB* ∈ *nodes B*
shows *io-targets AB* (*qA*,*qB*) *w* = {(*pA*,*pB*)}
proof −
obtain *trA* **where** *path A* (*w* || *trA*) *qA*
length w = *length trA*
target (*w* || *trA*) *qA* = *pA*
using *assms*(*2*) **by** *auto*
obtain *trB* **where** *path B* (*w* || *trB*) *qB*
length trA = *length trB*
target (*w* || *trB*) *qB* = *pB*
using ‹*length w* = *length trA*› *assms*(*3*) **by** *auto*

have *path AB* (*w* || *trA* || *trB*) (*qA*,*qB*)
length (*trA* || *trB*) = *length w*
using *productF-path-inclusion*
[*OF* ‹*length w* = *length trA*› ‹*length trA* = *length trB*› *assms*(*1*) *assms*(*8*,*9*) - *assms*(*10*,*11*)]
by (*simp add*: ‹*length trA* = *length trB*› ‹*length w* = *length trA*› ‹*path A* (*w* || *trA*) *qA*›
‹*path B* (*w* || *trB*) *qB*›)+

have *target* (*w* || *trA* || *trB*) (*qA*,*qB*) = (*pA*,*pB*)
by (*simp add*: ‹*length trA* = *length trB*› ‹*length w* = *length trA*› ‹*target* (*w* || *trA*) *qA* = *pA*›
‹*target* (*w* || *trB*) *qB* = *pB*›)

have (*pA*,*pB*) ∈ *io-targets AB* (*qA*,*qB*) *w*
by (*metis* ‹*length* (*trA* || *trB*) = *length w*› ‹*path AB* (*w* || *trA* || *trB*) (*qA*, *qB*)›
‹*target* (*w* || *trA* || *trB*) (*qA*, *qB*) = (*pA*, *pB*)› *io-target-from-path*)

have *observable AB*
by (*metis* (*no-types*) *assms*(*1*) *assms*(*6*) *assms*(*7*) *observable-productF*)

show *?thesis*
by (*meson* ‹(*pA*, *pB*) ∈ *io-targets AB* (*qA*, *qB*) *w*› ‹*observable AB*›
observable-io-target-is-singleton)
qed

2.1 Sequences to failure in the product machine

A sequence to a failure for `A` and `B` reaches the fail state of any product machine of `A` and `B` with added fail state.

```
lemma fail-reachable-by-sequence-to-failure :
  assumes sequence-to-failure M1 M2 io
  and     well-formed M1
  and     well-formed M2
  and productF M2 M1 FAIL PM
obtains p
where path PM (io||p) (initial PM) ∧ length p = length io ∧ target (io||p) (initial PM) = FAIL
proof −
  have io ≠ []
    using assms by auto
  then obtain io-init io-last where io-split[simp] : io = io-init @ [io-last]
    by (metis append-butlast-last-id)
  have io-init-inclusion : io-init ∈ language-state M1 (initial M1)
                         ∧ io-init ∈ language-state M2 (initial M2)
    using assms by auto

  have io-init @ [io-last] ∈ language-state M1 (initial M1)
    using assms by auto
  then obtain tr1-init tr1-last where tr1-def :
    path M1 (io-init @ [io-last] || tr1-init @ [tr1-last]) (initial M1)
      ∧ length (tr1-init @ [tr1-last]) = length (io-init @ [io-last])
    by (metis append-butlast-last-id language-state-elim length-0-conv length-append-singleton
        nat.simps(3))

  then have path-init-1 : path M1 (io-init || tr1-init) (initial M1)
                         ∧ length tr1-init = length io-init
    by auto
  then have path M1 ([io-last] || [tr1-last]) (target (io-init || tr1-init) (initial M1))
    using tr1-def by auto
  then have succ-1 : succ M1 io-last (target (io-init || tr1-init) (initial M1)) ≠ {}
    by auto

  obtain tr2 where tr2-def : path M2 (io-init || tr2) (initial M2) ∧ length tr2 = length io-init
    using io-init-inclusion by auto
  have succ-2 : succ M2 io-last (target (io-init || tr2) (initial M2)) = {}
  proof (rule ccontr)
    assume succ M2 io-last (target (io-init || tr2) (initial M2)) ≠ {}
    then obtain tr2-last where tr2-last ∈ succ M2 io-last (target (io-init || tr2) (initial M2))
      by auto
    then have path M2 ([io-last] || [tr2-last]) (target (io-init || tr2) (initial M2))
      by auto
    then have io-init @ [io-last] ∈ language-state M2 (initial M2)
      by (metis FSM.path-append language-state length-Cons length-append list.size(3) tr2-def
          zip-append)
    then show False
      using assms io-split by simp
  qed

  have fail-lengths : length (io-init @ [io-last]) = length (tr2 @ [fst FAIL])
                    ∧ length (tr2 @ [fst FAIL]) = length (tr1-init @ [snd FAIL])
    using assms tr2-def tr1-def by auto
  then have fail-tgt : target (io-init @ [io-last] || tr2 @ [fst FAIL] || tr1-init @ [snd FAIL])
                            (initial M2, initial M1) = FAIL
    by auto

  have fail-butlast-simp[simp] :
    butlast (io-init @ [io-last] || tr2 @ [fst FAIL]) = io-init || tr2
    butlast (io-init @ [io-last] || tr1-init @ [snd FAIL]) = io-init || tr1-init
    using fail-lengths by simp+

  have path M2 (butlast (io-init @ [io-last] || tr2 @ [fst FAIL])) (initial M2)
      ∧ path M1 (butlast (io-init @ [io-last] || tr1-init @ [snd FAIL])) (initial M1)
```

using *tr1-def tr2-def* **by** *auto*
moreover have *succ M2* (*last* (*io-init* @ [*io-last*]))
(*target* (*butlast* (*io-init* @ [*io-last*] || *tr2* @ [*fst FAIL*])) (*initial M2*)) = {}
using *succ-2* **by** *simp*
moreover have *succ M1* (*last* (*io-init* @ [*io-last*]))
(*target* (*butlast* (*io-init* @ [*io-last*] || *tr1-init* @ [*snd FAIL*])) (*initial M1*))
≠ {}
using *succ-1* **by** *simp*
moreover have *initial M2* ∈ *nodes M2* ∧ *initial M1* ∈ *nodes M1*
by *auto*
ultimately have *path PM* (*io-init* @ [*io-last*] || *tr2* @ [*fst FAIL*] || *tr1-init* @ [*snd FAIL*])
(*initial M2*, *initial M1*)
using *fail-lengths fail-tgt assms path-init-1 tr2-def productF-path-forward*
[*of io-init* @ [*io-last*] *tr2* @ [*fst FAIL*] *tr1-init* @ [*snd FAIL*] *M2 M1 FAIL PM*
initial M2 initial M1]
by *simp*

moreover have *initial PM* = (*initial M2*, *initial M1*)
using *assms*(*4*) *productF-simps*(*4*) **by** *blast*

ultimately have
path PM (*io-init* @ [*io-last*] || *tr2* @ [*fst FAIL*] || *tr1-init* @ [*snd FAIL*]) (*initial PM*)
∧ *length* (*tr2* @ [*fst FAIL*] || *tr1-init* @ [*snd FAIL*]) = *length* (*io-init* @ [*io-last*])
∧ *target* (*io-init* @ [*io-last*] || *tr2* @ [*fst FAIL*] || *tr1-init* @ [*snd FAIL*]) (*initial PM*)= *FAIL*
using *fail-lengths fail-tgt* **by** *auto*
then show *?thesis* **using** *that*
using *io-split* **by** *blast*
qed

lemma *fail-reachable* :
assumes ¬ *M1* ≼ *M2*
and *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
shows *FAIL* ∈ *reachable PM* (*initial PM*)
proof −
obtain *io* **where** *sequence-to-failure M1 M2 io*
using *sequence-to-failure-ob assms* **by** *blast*
then show *?thesis*
using *assms fail-reachable-by-sequence-to-failure*[*of M1 M2 io FAIL PM*]
by (*metis FSM.reachable.reflexive FSM.reachable-target*)
qed

lemma *fail-reachable-ob* :
assumes ¬ *M1* ≼ *M2*
and *well-formed M1*
and *well-formed M2*
and *observable M2*
and *productF M2 M1 FAIL PM*
obtains *p*
where *path PM p* (*initial PM*) *target p* (*initial PM*) = *FAIL*
using *assms fail-reachable* **by** (*metis FSM.reachable-target-elim*)

lemma *fail-reachable-reverse* :
assumes *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *FAIL* ∈ *reachable PM* (*initial PM*)
and *observable M2*
shows ¬ *M1* ≼ *M2*
proof −
obtain *pathF* **where** *pathF-def* : *path PM pathF* (*initial PM*) ∧ *target pathF* (*initial PM*) = *FAIL*
using *assms* **by** *auto*

```
let ?io = map fst pathF
let ?tr2 = map fst (map snd pathF)
let ?tr1 = map snd (map snd pathF)

have initial PM ≠ FAIL
  using assms by auto
then have pathF ≠ []
  using pathF-def by auto
moreover have initial PM = (initial M2, initial M1)
  using assms by simp
ultimately have path M2 (?io || ?tr2) (initial M2) ∧ path M1 (?io || ?tr1) (initial M1) ∨
                  target (?io || ?tr2 || ?tr1) (initial M2, initial M1) = FAIL ∧
                  0 < length (?io) ∧
                  path M2 (butlast (?io || ?tr2)) (initial M2) ∧
                  path M1 (butlast (?io || ?tr1)) (initial M1) ∧
                  succ M2 (last (?io)) (target (butlast (?io || ?tr2)) (initial M2)) = {} ∧
                  succ M1 (last (?io)) (target (butlast (?io || ?tr1)) (initial M1)) ≠ {}
  using productF-path-reverse[of ?io ?tr2 ?tr1 M2 M1 FAIL PM initial M2 initial M1]
  using assms pathF-def
proof −
  have f1: path PM (?io || ?tr2 || ?tr1) (initial M2, initial M1)
    by (metis (no-types) ‹initial PM = (initial M2, initial M1)› pathF-def zip-map-fst-snd)
  have f2: length (?io) = length pathF ⟶ length (?io) = length (?tr2)
    by auto
  have length (?io) = length pathF ∧ length (?tr2) = length (?tr1)
    by auto
  then show ?thesis
    using f2 f1 ‹productF M2 M1 FAIL PM› ‹well-formed M1› ‹well-formed M2› by blast
qed

moreover have ¬ (path M2 (?io || ?tr2) (initial M2) ∧ path M1 (?io || ?tr1) (initial M1))
proof (rule ccontr)
  assume ¬ ¬ (path M2 (?io || ?tr2) (initial M2) ∧
        path M1 (?io || ?tr1) (initial M1))
  then have path M2 (?io || ?tr2) (initial M2)
    by simp
  then have target (?io || ?tr2) (initial M2) ∈ nodes M2
    by auto
  then have target (?io || ?tr2) (initial M2) ≠ fst FAIL
    using assms by auto
  then show False
    using pathF-def
  proof −
    have FAIL = target (map fst pathF || map fst (map snd pathF) || map snd (map snd pathF))
                    (initial M2, initial M1)
      by (metis (no-types) ‹initial PM = (initial M2, initial M1)›
          ‹path PM pathF (initial PM) ∧ target pathF (initial PM) = FAIL› zip-map-fst-snd)
    then show ?thesis
      using ‹target (map fst pathF || map fst (map snd pathF)) (initial M2) ≠ fst FAIL› by auto
  qed
qed

ultimately have fail-prop :
        target (?io || ?tr2 || ?tr1) (initial M2, initial M1) = FAIL ∧
          0 < length (?io) ∧
          path M2 (butlast (?io || ?tr2)) (initial M2) ∧
          path M1 (butlast (?io || ?tr1)) (initial M1) ∧
          succ M2 (last (?io)) (target (butlast (?io || ?tr2)) (initial M2)) = {} ∧
          succ M1 (last (?io)) (target (butlast (?io || ?tr1)) (initial M1)) ≠ {}
  by auto

then have ?io ∈ language-state M1 (initial M1)
proof −
  have f1: path PM (map fst pathF || map fst (map snd pathF) || map snd (map snd pathF))
                  (initial M2, initial M1)
```

```
      by (metis (no-types) ‹initial PM = (initial M2, initial M1)› pathF-def zip-map-fst-snd)
    have ∀ c f. c ≠ initial (f::('a, 'b, 'c) FSM) ∨ c ∈ nodes f
      by blast
    then show ?thesis
      using f1 by (metis (no-types) assms(1) assms(2) assms(3) language-state length-map
                productF-path-reverse-ob)
  qed

  moreover have ?io ∉ language-state M2 (initial M2)
  proof (rule ccontr)
    assume ¬ ?io ∉ language-state M2 (initial M2)
    then have assm : ?io ∈ language-state M2 (initial M2)
      by simp
    then obtain tr2' where tr2'-def : path M2 (?io || tr2') (initial M2)
                                      ∧ length ?io = length tr2'
      by auto
    then obtain tr2'-init tr2'-last where tr2'-split : tr2' = tr2'-init @ [tr2'-last]
      using fail-prop by (metis ‹pathF ≠ []› append-butlast-last-id length-0-conv map-is-Nil-conv)

    have butlast ?io ∈ language-state M2 (initial M2)
      using fail-prop by auto
    then have {t. path M2 (butlast ?io || t) (initial M2) ∧ length (butlast ?io) = length t}
              = {butlast ?tr2}
      using assms(5) observable-path-unique[of butlast ?io M2 initial M2 butlast ?tr2]
          fail-prop by fastforce
    then have ∀ t ts . path M2 ((butlast ?io) @ [last ?io] || ts @ [t]) (initial M2)
                       ∧ length ((butlast ?io) @ [last ?io]) = length (ts @ [t])
                      ⟶ ts = butlast ?tr2
      by (metis (no-types, lifting) FSM.path-append-elim
          ‹butlast (map fst pathF) ∈ language-state M2 (initial M2)› assms(5) butlast-snoc
          butlast-zip fail-prop length-butlast length-map observable-path-unique zip-append)

    then have tr2'-init = butlast ?tr2
      using tr2'-def tr2'-split ‹pathF ≠ []› by auto
    then have path M2 ((butlast ?io) @ [last ?io] || (butlast ?tr2) @ [tr2'-last]) (initial M2)
              ∧ length ((butlast ?io) @ [last ?io]) = length ((butlast ?tr2) @ [tr2'-last])
      using tr2'-def fail-prop tr2'-split by auto
    then have path M2 ([last ?io] || [tr2'-last])
                      (target (butlast ?io || butlast ?tr2) (initial M2))
              ∧ length [last ?io] = length [tr2'-last]
      by auto
    then have tr2'-last ∈ succ M2 (last (?io)) (target (butlast (?io || ?tr2)) (initial M2))
      by auto
    then show False
      using fail-prop by auto
  qed

  ultimately show ?thesis by auto
qed



lemma fail-reachable-iff[iff] :
  assumes well-formed M1
  and     well-formed M2
  and     productF M2 M1 FAIL PM
  and     observable M2
shows FAIL ∈ reachable PM (initial PM) ⟷ ¬ M1 ≼ M2
proof
  show FAIL ∈ reachable PM (initial PM) ⟹ ¬ M1 ≼ M2
    using assms fail-reachable-reverse by blast
  show ¬ M1 ≼ M2 ⟹ FAIL ∈ reachable PM (initial PM)
    using assms fail-reachable by blast
qed
```

```
lemma reaching-path-length :
  assumes productF A B FAIL AB
  and     well-formed A
  and     well-formed B
  and     q2 ∈ reachable AB q1
  and     q2 ≠ FAIL
  and     q1 ∈ nodes AB
shows ∃ p . path AB p q1 ∧ target p q1 = q2 ∧ length p < card (nodes A) ∗ card (nodes B)
proof −
  obtain p where p-def : path AB p q1 ∧ target p q1 = q2 ∧ distinct (q1 # states p q1)
    using assms reaching-path-without-repetition by (metis well-formed-productF)

  have FAIL ∉ set (q1 # states p q1)
  proof(cases p)
    case Nil
    then have q1 = q2
      using p-def by auto
    then have q1 ≠ FAIL
      using assms by auto
    then show ?thesis
      using Nil by auto
  next
    case (Cons a list)
    have FAIL ∉ set (butlast (q1 # states p q1))
    proof (rule ccontr)
      assume assm : ¬ FAIL ∉ set (butlast (q1 # states p q1))
      then obtain i where i-def : i < length (butlast (q1 # states p q1))
                                  ∧ butlast (q1 # states p q1) ! i = FAIL
        by (metis distinct-Ex1 distinct-butlast p-def)
      then have i < Suc (length (butlast p))
        using local.Cons by fastforce
      then have i < length p
        by (metis append-butlast-last-id length-append-singleton list.simps(3) local.Cons)

      then have butlast (q1 # states p q1) ! i = target (take i p) q1
      using i-def assm proof (induction i)
        case 0
        then show ?case by auto
      next
        case (Suc i)
        then show ?case by (metis Suc-lessD nth-Cons-Suc nth-butlast states-target-index)
      qed

      then have target (take i p) q1 = FAIL using i-def by auto
      moreover have ∀ k . k < length p ⟶ target (take k p) q1 ≠ FAIL
        using no-prefix-targets-FAIL[of A B FAIL AB p q1] assms p-def by auto
      ultimately show False
        by (metis assms(5) linorder-neqE-nat nat-less-le order-refl p-def take-all)
    qed

    moreover have last (q1 # states p q1) ≠ FAIL
      using assms(5) local.Cons p-def transition-system-universal.target-alt-def by force
    ultimately show ?thesis
      by (metis (no-types, lifting) UnE append-butlast-last-id list.set(1) list.set(2)
          list.simps(3) set-append singletonD)
  qed

  moreover have set (q1 # states p q1) ⊆ nodes AB
    using assms by (metis FSM.nodes-states insert-subset list.simps(15) p-def)
  ultimately have states-subset : set (q1 # states p q1) ⊆ nodes A × nodes B
    using nodes-productF assms by blast

  have finite-nodes : finite (nodes A × nodes B)
```

using *assms(2) assms(3)* **by** *auto*
have *length p ≤ length (states p q1)*
by *simp*
then have *length p < card (nodes A) ∗ card (nodes B)*
by (*metis (no-types) finite-nodes states-subset card-cartesian-product card-mono distinct-card impossible-Cons less-le-trans not-less p-def*)

then show *?thesis*
using *p-def* **by** *blast*
qed

lemma *reaching-path-fail-length* :
assumes *productF A B FAIL AB*
and *well-formed A*
and *well-formed B*
and *q2 ∈ reachable AB q1*
and *q1 ∈ nodes AB*
shows *∃ p . path AB p q1 ∧ target p q1 = q2 ∧ length p ≤ card (nodes A) ∗ card (nodes B)*
proof (*cases q2 = FAIL*)
case *True*

then have *q2-def : q2 = FAIL*
by *simp*
then show *?thesis*
proof (*cases q1 = q2*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then obtain *px* **where** *px-def : path AB px q1 ∧ target px q1 = q2*
using *assms* **by** *auto*
then have *px-nonempty : px ≠ []*
using *q2-def False* **by** *auto*
let *?qx = target (butlast px) q1*
have *?qx ∈ reachable AB q1*
using *px-def px-nonempty*
by (*metis FSM.path-append-elim FSM.reachable.reflexive FSM.reachable-target append-butlast-last-id*)
moreover have *?qx ≠ FAIL*
using *False q2-def assms*
by (*metis One-nat-def Suc-pred butlast-conv-take length-greater-0-conv lessI no-prefix-targets-FAIL px-def px-nonempty*)
ultimately obtain *px′* **where** *px′-def : path AB px′ q1*
∧ target px′ q1 = ?qx
∧ length px′ < card (nodes A) ∗ card (nodes B)
using *assms reaching-path-length[of A B FAIL AB ?qx q1]* **by** *blast*

have *px-split : path AB ((butlast px) @ [last px]) q1*
∧ target ((butlast px) @ [last px]) q1 = q2
using *px-def px-nonempty* **by** *auto*
then have *path AB [last px] ?qx ∧ target [last px] ?qx = q2*
using *px-nonempty*
proof −
have *target [last px] (target (butlast px) q1) = q2*
using *px-split* **by** *force*
then show *?thesis*
using *px-split* **by** *blast*
qed

then have *path AB (px′ @ [last px]) q1 ∧ target (px′ @ [last px]) q1 = q2*
using *px′-def* **by** *auto*
moreover have *length (px′ @ [last px]) ≤ card (nodes A) ∗ card (nodes B)*
using *px′-def* **by** *auto*
ultimately show *?thesis*
by *blast*

  qed
next
  case *False*
  then show *?thesis*
    using *assms reaching-path-length* **by** (*metis less-imp-le*)
qed

lemma *productF-language* :
  assumes *productF A B FAIL AB*
  and *well-formed A*
  and *well-formed B*
  and *io* ∈ *L A* ∩ *L B*
shows *io* ∈ *L AB*
proof −
  obtain *trA trB* **where** *tr-def* : *path A* (*io* || *trA*) (*initial A*) ∧ *length io* = *length trA*
  path B (*io* || *trB*) (*initial B*) ∧ *length io* = *length trB*
    using *assms* **by** *blast*
  then have *path AB* (*io* || *trA* || *trB*) (*initial A*, *initial B*)
    using *assms* **by** (*metis FSM.nodes.initial productF-path-inclusion*)
  then show *?thesis*
    using *tr-def* **by** (*metis assms*(*1*) *language-state length-zip min.idem productF-simps*(*4*))
qed

lemma *productF-language-state-intermediate* :
  assumes *vs* @ *xs* ∈ *L M2* ∩ *L M1*
  and *productF M2 M1 FAIL PM*
  and *observable M2*
  and *well-formed M2*
  and *observable M1*
  and *well-formed M1*
obtains *q2 q1 tr*
where *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
  path PM (*xs* || *tr*) (*q2*,*q1*)
  length xs = *length tr*
proof −
  have *vs* @ *xs* ∈ *L PM*
    using *productF-language*[*OF assms*(*2*,*4*,*6*,*1*)] **by** *simp*
  then obtain *trVX* **where** *path PM* (*vs*@*xs* || *trVX*) (*initial PM*) ∧ *length trVX* = *length* (*vs*@*xs*)
    by *auto*
  then have *tgt-VX* : *io-targets PM* (*initial PM*) (*vs*@*xs*) = {*target* (*vs*@*xs* || *trVX*) (*initial PM*)}
    by (*metis assms*(*2*) *assms*(*3*) *assms*(*5*) *obs-target-is-io-targets observable-productF*)

  have *vs* ∈ *L PM* **using** ‹*vs*@*xs* ∈ *L PM*›
    by (*meson language-state-prefix*)
  then obtain *trV* **where** *path PM* (*vs* || *trV*) (*initial PM*) ∧ *length trV* = *length vs*
    by *auto*
  then have *tgt-V* : *io-targets PM* (*initial PM*) *vs* = {*target* (*vs* || *trV*) (*initial PM*)}
    by (*metis assms*(*2*) *assms*(*3*) *assms*(*5*) *obs-target-is-io-targets observable-productF*)

  let *?q2* = *fst* (*target* (*vs* || *trV*) (*initial PM*))
  let *?q1* = *snd* (*target* (*vs* || *trV*) (*initial PM*))

  have *observable PM*
    by (*meson assms*(*2*,*3*,*5*) *observable-productF*)

  have *io-targets PM* (*?q2*,*?q1*) *xs* = {*target* (*vs* @ *xs* || *trVX*) (*initial PM*)}
    using *observable-io-targets-split*[*OF* ‹*observable PM*› *tgt-VX tgt-V*] **by** *simp*

  then have *xs* ∈ *language-state PM* (*?q2*,*?q1*)
    by *auto*

  then obtain *tr* **where** *path PM* (*xs* || *tr*) (*?q2*,*?q1*)
  length xs = *length tr*
    by *auto*

then show *?thesis*
by (*metis prod.collapse tgt-V that*)
qed

lemma *sequence-to-failure-reaches-FAIL* :
assumes *sequence-to-failure M1 M2 io*
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows *FAIL* ∈ *io-targets PM* (*initial PM*) *io*
proof −
obtain *p* **where** *path PM* (*io* || *p*) (*initial PM*)
∧ *length p* = *length io*
∧ *target* (*io* || *p*) (*initial PM*) = *FAIL*
using *fail-reachable-by-sequence-to-failure*[*OF assms*(*1*)]
using *assms*(*2*) *assms*(*3*) *assms*(*4*) **by** *blast*
then show *?thesis*
by *auto*
qed

lemma *sequence-to-failure-reaches-FAIL-ob* :
assumes *sequence-to-failure M1 M2 io*
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows *io-targets PM* (*initial PM*) *io* = {*FAIL*}
proof −
have *FAIL* ∈ *io-targets PM* (*initial PM*) *io*
using *sequence-to-failure-reaches-FAIL*[*OF assms*(*1*−*4*)] **by** *assumption*
have *observable PM*
by (*meson assms*(*2*) *assms*(*3*) *assms*(*4*) *observable-productF*)
show *?thesis*
by (*meson* ‹*FAIL* ∈ *io-targets PM* (*initial PM*) *io*› ‹*observable PM*›
observable-io-target-is-singleton)
qed

lemma *sequence-to-failure-alt-def* :
assumes *io-targets PM* (*initial PM*) *io* = {*FAIL*}
and *OFSM M1*
and *OFSM M2*
and *productF M2 M1 FAIL PM*
shows *sequence-to-failure M1 M2 io*
proof −
obtain *p* **where** *path PM* (*io* || *p*) (*initial PM*)
length p = *length io*
target (*io* || *p*) (*initial PM*) = *FAIL*
using *assms*(*1*) **by** (*metis io-targets-elim singletonI*)
have *io* ≠ []
proof
assume *io* = []
then have *io-targets PM* (*initial PM*) *io* = {*initial PM*}
by *auto*
moreover have *initial PM* ≠ *FAIL*
proof −
have *initial PM* = (*initial M2*, *initial M1*)
using *assms*(*4*) **by** *auto*
then have *initial PM* ∈ (*nodes M2* × *nodes M1*)
by (*simp add*: *FSM.nodes.initial*)
moreover have *FAIL* ∉ (*nodes M2* × *nodes M1*)
using *assms*(*4*) **by** *auto*
ultimately show *?thesis*
by *auto*

qed
ultimately show *False*
using *assms(1)* **by** *blast*
qed
then have *0 < length io*
by *blast*

have *target (butlast (io||p)) (initial PM) ≠ FAIL*
using *no-prefix-targets-FAIL[OF assms(4) ‹path PM (io || p) (initial PM)›, of (length io) − 1]*
by *(metis (no-types, lifting) ‹0 < length io› ‹length p = length io› butlast-conv-take diff-less length-map less-numeral-extra(1) map-fst-zip)*
have *target (butlast (io||p)) (initial PM) ∈ nodes PM*
by *(metis FSM.nodes.initial FSM.nodes-target FSM.path-append-elim ‹path PM (io || p) (initial PM)› append-butlast-last-id butlast.simps(1))*
moreover have *nodes PM ⊆ insert FAIL (nodes M2 × nodes M1)*
using *nodes-productF[OF - - assms(4)] assms(2) assms(3)* **by** *linarith*
ultimately have *target (butlast (io||p)) (initial PM) ∈ insert FAIL (nodes M2 × nodes M1)*
by *blast*

have *target (butlast (io||p)) (initial PM) ∈ (nodes M2 × nodes M1)*
using *‹target (butlast (io || p)) (initial PM) ∈ insert FAIL (nodes M2 × nodes M1)›*
‹target (butlast (io || p)) (initial PM) ≠ FAIL›
by *blast*
then obtain *s2 s1* **where** *target (butlast (io||p)) (initial PM) = (s2,s1)*
s2 ∈ nodes M2 s1 ∈ nodes M1
by *blast*

have *length (butlast io) = length (map fst (butlast p))*
length (map fst (butlast p)) = length (map snd (butlast p))
by *(simp add: ‹length p = length io›)+*

have *path PM (butlast (io||p)) (initial PM)*
by *(metis FSM.path-append-elim ‹path PM (io || p) (initial PM)› append-butlast-last-id butlast.simps(1))*
then have *path PM ((butlast io) || (map fst (butlast p)) || (map snd (butlast p))) (initial M2, initial M1)*
using *‹length p = length io› assms(4)* **by** *auto*
have *target (butlast io || map fst (butlast p) || map snd (butlast p)) (initial M2, initial M1) ≠ FAIL*
using *‹length p = length io› ‹target (butlast (io || p)) (initial PM) ≠ FAIL› assms(4)*
by *auto*

have *path M2 (butlast io || map fst (butlast p)) (initial M2) ∧*
path M1 (butlast io || map snd (butlast p)) (initial M1) ∨
target (butlast io || map fst (butlast p) || map snd (butlast p)) (initial M2, initial M1) = FAIL
using *productF-path-reverse*
[OF ‹length (butlast io) = length (map fst (butlast p))›
‹length (map fst (butlast p)) = length (map snd (butlast p))›
assms(4) - -
‹path PM ((butlast io) || (map fst (butlast p)) || (map snd (butlast p)))
(initial M2, initial M1)› - -]
using *assms(2) assms(3)* **by** *auto*
then have *path M2 (butlast io || map fst (butlast p)) (initial M2)*
path M1 (butlast io || map snd (butlast p)) (initial M1)
using *‹target (butlast io || map fst (butlast p) || map snd (butlast p)) (initial M2, initial M1) ≠ FAIL›*
by *auto*

then have *butlast io ∈ L M2 ∩ L M1*
using *‹length (butlast io) = length (map fst (butlast p))›* **by** *auto*

have *path PM (io || map fst p || map snd p) (initial M2, initial M1)*
using *‹path PM (io || p) (initial PM)› assms(4)* **by** *auto*
have *length io = length (map fst p)*
length (map fst p) = length (map snd p)

```
    by (simp add: ‹length p = length io›)+

  obtain p1' where path M1 (io || p1') (initial M1) ∧ length io = length p1'
    using productF-path-reverse-ob
          [OF ‹length io = length (map fst p)›
              ‹length (map fst p) = length (map snd p)› assms(4) - -
              ‹path PM (io || map fst p || map snd p) (initial M2, initial M1)›]
    using assms(2) assms(3) by blast
  then have io ∈ L M1
    by auto

  moreover have io ∉ L M2
  proof
    assume io ∈ L M2 — only possible if io does not target FAIL
    then obtain p2' where path M2 (io || p2') (initial M2) length io = length p2'
      by auto
    then have length p2' = length p1'
      using ‹path M1 (io || p1') (initial M1) ∧ length io = length p1'›
      by auto

    have path PM (io || p2' || p1') (initial M2, initial M1)
      using productF-path-inclusion[OF ‹length io = length p2'› ‹length p2' = length p1'› assms(4),
                                    of initial M2 initial M1]
            ‹path M1 (io || p1') (initial M1) ∧ length io = length p1'›
            ‹path M2 (io || p2') (initial M2)› assms(2) assms(3)
      by blast


    have target (io || p2' || p1') (initial M2, initial M1) ∈ (nodes M2 × nodes M1)
      using ‹length io = length p2'› ‹path M1 (io || p1') (initial M1) ∧ length io = length p1'›
            ‹path M2 (io || p2') (initial M2)›
      by auto
    moreover have FAIL ∉ (nodes M2 × nodes M1)
      using assms(4) by auto
    ultimately have target (io || p2' || p1') (initial M2, initial M1) ≠ FAIL
      by blast

    have length io = length (p2' || p1')
      by (simp add: ‹length io = length p2'› ‹length p2' = length p1'›)
    have target (io || p2' || p1') (initial M2, initial M1)
            ∈ io-targets PM (initial M2, initial M1) io
      using ‹path PM (io || p2' || p1') (initial M2, initial M1)› ‹length io = length (p2' || p1')›
      unfolding io-targets.simps by blast

    have io-targets PM (initial PM) io ≠ {FAIL}
      using ‹target (io || p2' || p1') (initial M2, initial M1)
              ∈ io-targets PM (initial M2, initial M1) io›
            ‹target (io || p2' || p1') (initial M2, initial M1) ≠ FAIL› assms(4)
      by auto
    then show False
      using assms(1) by blast
  qed

  ultimately have io ∈ L M1 − L M2
    by blast

  show sequence-to-failure M1 M2 io
    using ‹butlast io ∈ L M2 ∩ L M1› ‹io ∈ L M1 − L M2› by auto
qed



end
theory ATC
imports ../FSM/FSM
begin
```

3 Adaptive test cases

Adaptive test cases (ATCs) are tree-like structures that label nodes with inputs and edges with outputs such that applying an ATC to some FSM is performed by applying the label of its root node and then applying the ATC connected to the root node by an edge labeled with the observed output of the FSM. The result of such an application is here called an ATC-reaction.

ATCs are here modelled to have edges for every possible output from each non-leaf node. This is not a restriction on the definition of ATCs by Hierons [2] as a missing edge can be expressed by an edge to a leaf.

```
datatype ('in, 'out) ATC = Leaf | Node 'in 'out ⇒ ('in, 'out) ATC

inductive atc-reaction :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ ('in, 'out) ATC
                          ⇒ ('in × 'out) list ⇒ bool
  where
  leaf[intro!]: atc-reaction M q1 Leaf [] |
  node[intro!]: q2 ∈ succ M (x,y) q1
               ⟹ atc-reaction M q2 (f y) io
               ⟹ atc-reaction M q1 (Node x f) ((x,y)#io)

inductive-cases leaf-elim[elim!] : atc-reaction M q1 Leaf []
inductive-cases node-elim[elim!] : atc-reaction M q1 (Node x f) ((x,y)#io)
```

3.1 Properties of ATC-reactions

```
lemma atc-reaction-empty[simp] :
  assumes atc-reaction M q t []
  shows t = Leaf
using assms atc-reaction.simps by force

lemma atc-reaction-nonempty-no-leaf :
  assumes atc-reaction M q t (Cons a io)
  shows t ≠ Leaf
using assms
proof –
  have ⋀f c a ps. ¬ atc-reaction f (c::'c) (a::('a, 'b) ATC) ps ∨ a ≠ Leaf ∨ a ≠ Leaf ∨ ps = []
    using atc-reaction.simps by fastforce
  then show ?thesis
    using assms by blast
qed

lemma atc-reaction-nonempty[elim] :
  assumes atc-reaction M q1 t (Cons (x,y) io)
  obtains q2 f
  where t = Node x f q2 ∈ succ M (x,y) q1  atc-reaction M q2 (f y) io
proof –
  obtain x2 f where t = Node x2 f
    using assms by (metis ATC.exhaust atc-reaction-nonempty-no-leaf)
  moreover have x = x2
    using assms calculation atc-reaction.cases by fastforce
  ultimately show ?thesis
    using assms using that by blast
qed

lemma atc-reaction-path-ex :
  assumes atc-reaction M q1 t io
  shows ∃ tr . path M (io || tr) q1 ∧ length io = length tr
using assms proof (induction io arbitrary: q1 t rule: list.induct)
  case Nil
  then show ?case by (simp add: FSM.nil)
next
  case (Cons io-hd io-tl)
  then obtain x y where io-hd-def : io-hd = (x,y)
    by (meson surj-pair)
  then obtain f where f-def : t = (Node x f)
    using Cons atc-reaction-nonempty by metis
  then obtain q2 where q2-def : q2 ∈ succ M (x,y) q1 atc-reaction M q2 (f y) io-tl
```

using *Cons io-hd-def atc-reaction-nonempty* **by** *auto*
then obtain *tr-tl* **where** *tr-tl-def* : *path M* (*io-tl* || *tr-tl*) *q2 length io-tl* = *length tr-tl*
using *Cons.IH*[*of q2 f y*] **by** *blast*
then have *path M* (*io-hd* # *io-tl* || *q2* # *tr-tl*) *q1*
using *Cons q2-def* **by** (*simp add*: *FSM.path.intros*(*2*) *io-hd-def*)
then show *?case* **using** *tr-tl-def* **by** *fastforce*
qed

lemma *atc-reaction-path*[*elim*] :
assumes *atc-reaction M q1 t io*
obtains *tr*
where *path M* (*io* || *tr*) *q1 length io* = *length tr*
by (*meson assms atc-reaction-path-ex*)

3.2 Applicability

An ATC can be applied to an FSM if each node-label is contained in the input alphabet of the FSM.

inductive *subtest* :: ('*in*, '*out*) *ATC* ⇒ ('*in*, '*out*) *ATC* ⇒ *bool* **where**
$t \in$ *range f* ⟹ *subtest t* (*Node x f*)

lemma *accp-subtest* : *Wellfounded.accp subtest t*
proof (*induction t*)
case *Leaf*
then show *?case* **by** (*meson ATC.distinct*(*1*) *accp.simps subtest.cases*)
next
case (*Node x f*)
have *IH*: *Wellfounded.accp subtest t* **if** $t \in$ *range f* **for** *t*
using *Node*[*of t*] **and** *that* **by** (*auto simp*: *eq-commute*)
show *?case* **by** (*rule accpI*) (*auto intro*: *IH elim*!: *subtest.cases*)
qed

definition *subtest-rel* **where** *subtest-rel* = {(*t*, *Node x f*) |*f x t*. $t \in$ *range f*}

lemma *subtest-rel-altdef*: *subtest-rel* = {(*s*, *t*) |*s t*. *subtest s t*}
by (*auto simp*: *subtest-rel-def subtest.simps*)

lemma *subtest-relI* [*intro*]: $t \in$ *range f* ⟹ (*t*, *Node x f*) ∈ *subtest-rel*
by (*simp add*: *subtest-rel-def*)

lemma *subtest-relI'* [*intro*]: *t* = *f y* ⟹ (*t*, *Node x f*) ∈ *subtest-rel*
by (*auto simp*: *subtest-rel-def ran-def*)

lemma *wf-subtest-rel* [*simp*, *intro*]: *wf subtest-rel*
using *accp-subtest* **unfolding** *subtest-rel-altdef accp-eq-acc wf-iff-acc*
by *auto*

function *inputs-atc* :: ('*a*,'*b*) *ATC* ⇒ '*a set* **where**
inputs-atc Leaf = {} |
inputs-atc (*Node x f*) = *insert x* (⋃ (*image inputs-atc* (*range f*)))
by *pat-completeness auto*
termination by (*relation subtest-rel*) *auto*

fun *applicable* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in*, '*out*) *ATC* ⇒ *bool* **where**
applicable M t = (*inputs-atc t* ⊆ *inputs M*)

fun *applicable-set* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in*, '*out*) *ATC set* ⇒ *bool* **where**
applicable-set M Ω = (∀ *t* ∈ Ω . *applicable M t*)

lemma *applicable-subtest* :
assumes *applicable M* (*Node x f*)
shows *applicable M* (*f y*)
using *assms inputs-atc.simps*
by (*simp add*: *Sup-le-iff*)

3.3 Application function IO

Function IO collects all ATC-reactions of some FSM to some ATC.

fun *IO* :: ('*in*, '*out*, '*state*) *FSM* ⇒ '*state* ⇒ ('*in*, '*out*) *ATC* ⇒ ('*in* × '*out*) *list set* **where**
  IO M q t = { *tr* . *atc-reaction M q t tr* }

fun *IO-set* :: ('*in*, '*out*, '*state*) *FSM* ⇒ '*state* ⇒ ('*in*, '*out*) *ATC set* ⇒ ('*in* × '*out*) *list set*
  where
  IO-set M q Ω = ⋃ {*IO M q t* | *t* . *t* ∈ Ω}

lemma *IO-language* : *IO M q t* ⊆ *language-state M q*
  by (*metis atc-reaction-path IO.elims language-state mem-Collect-eq subsetI*)

lemma *IO-leaf*[*simp*] : *IO M q Leaf* = {[]}
proof
  show *IO M q Leaf* ⊆ {[]}
  proof (*rule ccontr*)
    assume *assm* : ¬ *IO M q Leaf* ⊆ {[]}
    then obtain *io-hd io-tl* **where** *elem-ex* : *Cons io-hd io-tl* ∈ *IO M q Leaf*
      by (*metis* (*no-types*, *opaque-lifting*) *insertI1 neq-Nil-conv subset-eq*)
    then show *False*
      using *atc-reaction-nonempty-no-leaf assm* **by** (*metis IO.simps mem-Collect-eq*)
  qed
next
  show {[]} ⊆ *IO M q Leaf* **by** *auto*
qed

lemma *IO-applicable-nonempty* :
  assumes *applicable M t*
  and *completely-specified M*
  and *q1* ∈ *nodes M*
  shows *IO M q1 t* ≠ {}
using *assms* **proof** (*induction t arbitrary*: *q1*)
  case *Leaf*
  then show *?case* **by** *auto*
next
  case (*Node x f*)
  then have *x* ∈ *inputs M* **by** *auto*
  then obtain *y q2* **where** *x-appl* : *q2* ∈ *succ M* (*x*, *y*) *q1*
    using *Node* **unfolding** *completely-specified.simps* **by** *blast*
  then have *applicable M* (*f y*)
    using *applicable-subtest Node* **by** *metis*
  moreover have *q2* ∈ *nodes M*
    using *Node*(*4*) ‹*q2* ∈ *succ M* (*x*, *y*) *q1*› *FSM.nodes.intros*(*2*)[*of q1 M* ((*x*,*y*),*q2*)] **by** *auto*
  ultimately have *IO M q2* (*f y*) ≠ {}
    using *Node* **by** *auto*
  then show *?case* **unfolding** *IO.simps*
    using *x-appl* **by** *blast*
qed

lemma *IO-in-language* :
  IO M q t ⊆ *LS M q*
  unfolding *IO.simps* **by** *blast*

lemma *IO-set-in-language* :
  IO-set M q Ω ⊆ *LS M q*
  using *IO-in-language*[*of M q*] **unfolding** *IO-set.simps* **by** *blast*

3.4 R-distinguishability

A non-empty ATC r-distinguishes two states of some FSM if there exists no shared ATC-reaction.

fun *r-dist* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in*, '*out*) *ATC* ⇒ '*state* ⇒ '*state* ⇒ *bool* **where**
r-dist M t s1 s2 = (*t* ≠ *Leaf* ∧ *IO M s1 t* ∩ *IO M s2 t* = {})

fun *r-dist-set* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in*, '*out*) *ATC set* ⇒ '*state* ⇒ '*state* ⇒ *bool* **where**
r-dist-set M T s1 s2 = (∃ *t* ∈ *T* . *r-dist M t s1 s2*)

lemma *r-dist-dist* :
assumes *applicable M t*
and *completely-specified M*
and *r-dist M t q1 q2*
and *q1* ∈ *nodes M*
shows *q1* ≠ *q2*
proof (*rule ccontr*)
assume ¬(*q1* ≠ *q2*)
then have *q1* = *q2*
by *simp*
then have *IO M q1 t* = {}
using *assms* **by** *simp*
moreover have *IO M q1 t* ≠ {}
using *assms IO-applicable-nonempty* **by** *auto*
ultimately show *False*
by *simp*
qed

lemma *r-dist-set-dist* :
assumes *applicable-set M* Ω
and *completely-specified M*
and *r-dist-set M* Ω *q1 q2*
and *q1* ∈ *nodes M*
shows *q1* ≠ *q2*
using *assms r-dist-dist* **by** (*metis applicable-set.elims*(*2*) *r-dist-set.elims*(*2*))

lemma *r-dist-set-dist-disjoint* :
assumes *applicable-set M* Ω
and *completely-specified M*
and ∀ *t1* ∈ *T1* . ∀ *t2* ∈ *T2* . *r-dist-set M* Ω *t1 t2*
and *T1* ⊆ *nodes M*
shows *T1* ∩ *T2* = {}
by (*metis assms disjoint-iff-not-equal r-dist-set-dist subsetCE*)

3.5 Response sets

The following functions calculate the sets of all ATC-reactions observed by applying some set of ATCs on every state reached in some FSM using a given set of IO-sequences.

fun *B* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in* ∗ '*out*) *list* ⇒ ('*in*, '*out*) *ATC set*
⇒ ('*in* ∗ '*out*) *list set* **where**
B M io Ω = ⋃ (*image* (λ *s* . *IO-set M s* Ω) (*io-targets M* (*initial M*) *io*))

fun *D* :: ('*in*, '*out*, '*state*) *FSM* ⇒ '*in list set* ⇒ ('*in*, '*out*) *ATC set*
⇒ ('*in* ∗ '*out*) *list set set* **where**
D M ISeqs Ω = *image* (λ *io* . *B M io* Ω) (LS_{in} *M* (*initial M*) *ISeqs*)

fun *append-io-B* :: ('*in*, '*out*, '*state*) *FSM* ⇒ ('*in* ∗ '*out*) *list* ⇒ ('*in*, '*out*) *ATC set*
⇒ ('*in* ∗ '*out*) *list set* **where**
append-io-B M io Ω = { *io*@*res* | *res* . *res* ∈ *B M io* Ω }

lemma *B-dist'* :
assumes *df*: *B M io1* Ω ≠ *B M io2* Ω
shows (*io-targets M* (*initial M*) *io1*) ≠ (*io-targets M* (*initial M*) *io2*)
using *assms* **by** *force*

lemma *B-dist* :

assumes *io-targets M* (*initial M*) *io1* = {*q1*}
and *io-targets M* (*initial M*) *io2* = {*q2*}
and *B M io1* Ω ≠ *B M io2* Ω
shows *q1* ≠ *q2*
using *assms* **by** *force*

lemma *D-bound* :
assumes *wf*: *well-formed M*
and *ob*: *observable M*
and *fi*: *finite ISeqs*
shows *finite* (*D M ISeqs* Ω) *card* (*D M ISeqs* Ω) ≤ *card* (*nodes M*)
proof −
have *D M ISeqs* Ω ⊆ *image* (λ *s* . *IO-set M s* Ω) (*nodes M*)
proof
fix *RS* **assume** *RS-def* : *RS* ∈ *D M ISeqs* Ω
then obtain *xs ys* **where** *RS-tr* : *RS* = *B M* (*xs* || *ys*) Ω
(*xs* ∈ *ISeqs* ∧ *length xs* = *length ys*
∧ (*xs* || *ys*) ∈ *language-state M* (*initial M*))
by *auto*
then obtain *qx* **where** *qx-def* : *io-targets M* (*initial M*) (*xs* || *ys*) = { *qx* }
by (*meson io-targets-observable-singleton-ex ob*)
then have *RS* = *IO-set M qx* Ω
using *RS-tr* **by** *auto*
moreover have *qx* ∈ *nodes M*
by (*metis FSM.nodes.initial io-targets-nodes qx-def singletonI*)
ultimately show *RS* ∈ *image* (λ *s* . *IO-set M s* Ω) (*nodes M*)
by *auto*
qed
moreover have *finite* (*nodes M*)
using *assms* **by** *auto*
ultimately show *finite* (*D M ISeqs* Ω) *card* (*D M ISeqs* Ω) ≤ *card* (*nodes M*)
by (*meson finite-imageI infinite-super surj-card-le*)+
qed

lemma *append-io-B-in-language* :
append-io-B M io Ω ⊆ *L M*
proof
fix *x* **assume** *x* ∈ *append-io-B M io* Ω
then obtain *res* **where** *x* = *io*@*res res* ∈ *B M io* Ω
unfolding *append-io-B.simps* **by** *blast*
then obtain *q* **where** *q* ∈ *io-targets M* (*initial M*) *io res* ∈ *IO-set M q* Ω
unfolding *B.simps* **by** *blast*
then have *res* ∈ *LS M q*
using *IO-set-in-language*[*of M q* Ω] **by** *blast*

obtain *pIO* **where** *path M* (*io* || *pIO*) (*initial M*)
length pIO = *length io target* (*io* || *pIO*) (*initial M*) = *q*
using ‹*q* ∈ *io-targets M* (*initial M*) *io*› **by** *auto*
moreover obtain *pRes* **where** *path M* (*res* || *pRes*) *q length pRes* = *length res*
using ‹*res* ∈ *LS M q*› **by** *auto*
ultimately have *io*@*res* ∈ *L M*
using *FSM.path-append*[*of M io*||*pIO initial M res*||*pRes*]
by (*metis language-state length-append zip-append*)
then show *x* ∈ *L M*
using ‹*x* = *io*@*res*› **by** *blast*
qed

lemma *append-io-B-nonempty* :
assumes *applicable-set M* Ω
and *completely-specified M*
and *io* ∈ *language-state M* (*initial M*)
and Ω ≠ {}
shows *append-io-B M io* Ω ≠ {}

```
proof –
  obtain t where t ∈ Ω
    using assms(4) by blast
  then have applicable M t
    using assms(1) by simp
  moreover obtain tr where path M (io || tr) (initial M) ∧ length tr = length io
    using assms(3) by auto
  moreover have target (io || tr) (initial M) ∈ nodes M
    using calculation(2) by blast
  ultimately have IO M (target (io || tr) (initial M)) t ≠ {}
    using assms(2) IO-applicable-nonempty by simp
  then obtain io' where io' ∈ IO M (target (io || tr) (initial M)) t
    by blast
  then have io' ∈ IO-set M (target (io || tr) (initial M)) Ω
    using ‹t ∈ Ω› unfolding IO-set.simps by blast
  moreover have (target (io || tr) (initial M)) ∈ io-targets M (initial M) io
    using ‹path M (io || tr) (initial M) ∧ length tr = length io› by auto
  ultimately have io' ∈ B M io Ω
    unfolding B.simps by blast
  then have io@io' ∈ append-io-B M io Ω
    unfolding append-io-B.simps by blast
  then show ?thesis by blast
qed
```

```
lemma append-io-B-prefix-in-language :
  assumes append-io-B M io Ω ≠ {}
  shows io ∈ L M
proof –
  obtain res where io @ res ∈ append-io-B M io Ω ∧ res ∈ B M io Ω
    using assms by auto
  then have io-targets M (initial M) io ≠ {}
    by auto
  then obtain q where q ∈ io-targets M (initial M) io
    by blast
  then obtain tr where target (io || tr) (initial M) = q ∧ path M (io || tr) (initial M)
                        ∧ length tr = length io by auto
  then show ?thesis by auto
qed
```

3.6 Characterizing sets

A set of ATCs is a characterizing set for some FSM if for every pair of r-distinguishable states it contains an ATC that r-distinguishes them.

```
fun characterizing-atc-set :: ('in, 'out, 'state) FSM ⇒ ('in, 'out) ATC set ⇒ bool where
characterizing-atc-set M Ω = (applicable-set M Ω ∧ (∀ s1 ∈ (nodes M) . ∀ s2 ∈ (nodes M) .
    (∃ td . r-dist M td s1 s2) ⟶ (∃ tt ∈ Ω . r-dist M tt s1 s2)))
```

3.7 Reduction over ATCs

Some state is a an ATC-reduction of another over some set of ATCs if for every contained ATC every ATC-reaction to it of the former state is also an ATC-reaction of the latter state.

```
fun atc-reduction :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ ('in, 'out, 'state) FSM ⇒ 'state
                     ⇒ ('in, 'out) ATC set ⇒ bool where
  atc-reduction M2 s2 M1 s1 Ω = (∀ t ∈ Ω . IO M2 s2 t ⊆ IO M1 s1 t)
```

— r-distinguishability holds for atc-reductions

```
lemma atc-rdist-dist[intro] :
  assumes wf2   : well-formed M2
  and     cs2   : completely-specified M2
  and     ap2   : applicable-set M2 Ω
  and     el-t1 : t1 ∈ nodes M2
  and     red1  : atc-reduction M2 t1 M1 s1 Ω
```

and *red2* : *atc-reduction M2 t2 M1 s2* Ω
and *rdist* : *r-dist-set M1* Ω *s1 s2*
and *t1* ∈ *nodes M2*
shows *r-dist-set M2* Ω *t1 t2*
proof −
obtain *td* **where** *td-def* : *td* ∈ Ω ∧ *r-dist M1 td s1 s2*
using *rdist* **by** *auto*
then have *IO M1 s1 td* ∩ *IO M1 s2 td* = {}
using *td-def* **by** *simp*
moreover have *IO M2 t1 td* ⊆ *IO M1 s1 td*
using *red1 td-def* **by** *auto*
moreover have *IO M2 t2 td* ⊆ *IO M1 s2 td*
using *red2 td-def* **by** *auto*
ultimately have *no-inter* : *IO M2 t1 td* ∩ *IO M2 t2 td* = {}
by *blast*

then have *td* ≠ *Leaf*
by *auto*
then have *IO M2 t1 td* ≠ {}
by (*meson ap2 IO-applicable-nonempty applicable-set.elims*(*2*) *cs2 td-def assms*(*8*))
then have *IO M2 t1 td* ≠ *IO M2 t2 td*
using *no-inter* **by** *auto*
then show *?thesis*
using *no-inter td-def* **by** *auto*
qed

3.8 Reduction over ATCs applied after input sequences

The following functions check whether some FSM is a reduction of another over a given set of input sequences while furthermore the response sets obtained by applying a set of ATCs after every input sequence to the first FSM are subsets of the analogously constructed response sets of the second FSM.

fun *atc-io-reduction-on* :: (*'in*, *'out*, *'state1*) *FSM* ⇒ (*'in*, *'out*, *'state2*) *FSM* ⇒ *'in list*
⇒ (*'in*, *'out*) *ATC set* ⇒ *bool* **where**
atc-io-reduction-on M1 M2 iseq Ω = (L_{in} *M1* {*iseq*} ⊆ L_{in} *M2* {*iseq*}
∧ (∀ *io* ∈ L_{in} *M1* {*iseq*} . *B M1 io* Ω ⊆ *B M2 io* Ω))

fun *atc-io-reduction-on-sets* :: (*'in*, *'out*, *'state1*) *FSM* ⇒ *'in list set* ⇒ (*'in*, *'out*) *ATC set*
⇒ (*'in*, *'out*, *'state2*) *FSM* ⇒ *bool* **where**
atc-io-reduction-on-sets M1 TS Ω *M2* = (∀ *iseq* ∈ *TS* . *atc-io-reduction-on M1 M2 iseq* Ω)

notation
atc-io-reduction-on-sets (‹(- ≼⟦-.-⟧ -)› [*1000*,*1000*,*1000*,*1000*])

lemma *io-reduction-from-atc-io-reduction* :
assumes *atc-io-reduction-on-sets M1 T* Ω *M2*
and *finite T*
shows *io-reduction-on M1 T M2*
using *assms*(*2*,*1*) **proof** (*induction T*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert t T*)
then have *atc-io-reduction-on M1 M2 t* Ω
by *auto*
then have L_{in} *M1* {*t*} ⊆ L_{in} *M2* {*t*}
using *atc-io-reduction-on.simps* **by** *blast*

have L_{in} *M1 T* ⊆ L_{in} *M2 T*
using *insert.IH*
proof −
have *atc-io-reduction-on-sets M1 T* Ω *M2*
by (*meson contra-subsetD insert.prems atc-io-reduction-on-sets.simps subset-insertI*)
then show *?thesis*

    using *insert.IH* **by** *blast*
  qed
  then have L_{in} *M1 T* ⊆ L_{in} *M2* (*insert t T*)
    by (*meson insert-iff language-state-for-inputs-in-language-state*
      language-state-for-inputs-map-fst language-state-for-inputs-map-fst-contained
      subsetCE subsetI)
  moreover have L_{in} *M1* {*t*} ⊆ L_{in} *M2* (*insert t T*)
  proof −
    obtain *pps* :: ('*a* × '*b*) *list set* ⇒ ('*a* × '*b*) *list set* ⇒ ('*a* × '*b*) *list* **where**
      ∀ *x0 x1*. (∃ *v2*. *v2* ∈ *x1* ∧ *v2* ∉ *x0*) = (*pps x0 x1* ∈ *x1* ∧ *pps x0 x1* ∉ *x0*)
      by *moura*
    then have ∀ *P Pa*. *pps Pa P* ∈ *P* ∧ *pps Pa P* ∉ *Pa* ∨ *P* ⊆ *Pa*
      by *blast*
    moreover
    { assume *map fst* (*pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* {*t*})) ∉ *insert t T*
      then have *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* {*t*}) ∉ L_{in} *M1* {*t*}
          ∨ *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* {*t*}) ∈ L_{in} *M2* (*insert t T*)
        by (*metis* (*no-types*) *insertI1 language-state-for-inputs-map-fst-contained singletonD*) **}**
    ultimately show *?thesis*
      by (*meson* ‹L_{in} *M1* {*t*} ⊆ L_{in} *M2* {*t*}› *language-state-for-inputs-in-language-state*
        language-state-for-inputs-map-fst set-rev-mp)
  qed

  ultimately show *?case*
  proof −
    have *f1*: ∀ *ps P Pa*. (*ps*::('*a* × '*b*) *list*) ∉ *P* ∨ ¬ *P* ⊆ *Pa* ∨ *ps* ∈ *Pa*
      by *blast*
    obtain *pps* :: ('*a* × '*b*) *list set* ⇒ ('*a* × '*b*) *list set* ⇒ ('*a* × '*b*) *list* **where**
      ∀ *x0 x1*. (∃ *v2*. *v2* ∈ *x1* ∧ *v2* ∉ *x0*) = (*pps x0 x1* ∈ *x1* ∧ *pps x0 x1* ∉ *x0*)
      by *moura*
    moreover
    { assume *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*))
          ∉ L_{in} *M1* {*t*}
      moreover
      { assume *map fst* (*pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*)))
            ∉ {*t*}
        then have *map fst* (*pps* (L_{in} *M2* (*insert t T*))
              (L_{in} *M1* (*insert t T*))) ≠ *t*
          by *blast*
        then have *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*))
              ∉ L_{in} *M1* (*insert t T*)
            ∨ *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*))
              ∈ L_{in} *M2* (*insert t T*)
          using *f1* **by** (*meson* ‹L_{in} *M1 T* ⊆ L_{in} *M2* (*insert t T*)›
              insertE language-state-for-inputs-in-language-state
              language-state-for-inputs-map-fst
              language-state-for-inputs-map-fst-contained) **}**
      ultimately have *io-reduction-on M1* (*insert t T*) *M2*
              ∨ *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*))
                ∉ L_{in} *M1* (*insert t T*)
              ∨ *pps* (L_{in} *M2* (*insert t T*)) (L_{in} *M1* (*insert t T*))
                ∈ L_{in} *M2* (*insert t T*)
        using *f1* **by** (*meson language-state-for-inputs-in-language-state*
            language-state-for-inputs-map-fst) **}**
      ultimately show *?thesis*
        using *f1* **by** (*meson* ‹L_{in} *M1* {*t*} ⊆ L_{in} *M2* (*insert t T*)› *subsetI*)
  qed
qed

lemma *atc-io-reduction-on-subset* :
  assumes *atc-io-reduction-on-sets M1 T* Ω *M2*
  and   *T'* ⊆ *T*
shows *atc-io-reduction-on-sets M1 T'* Ω *M2*
  using *assms* **unfolding** *atc-io-reduction-on-sets.simps* **by** *blast*

```
lemma atc-reaction-reduction[intro] :
  assumes ls : language-state M1 q1 ⊆ language-state M2 q2
  and     el1 : q1 ∈ nodes M1
  and     el2 : q2 ∈ nodes M2
  and     rct : atc-reaction M1 q1 t io
  and     ob2 : observable M2
  and     ob1 : observable M1
shows atc-reaction M2 q2 t io
using assms proof (induction t arbitrary: io q1 q2)
  case Leaf
  then have io = []
    by (metis atc-reaction-nonempty-no-leaf list.exhaust)
  then show ?case
    by (simp add: leaf)
next
  case (Node x f)
  then obtain io-hd io-tl where io-split : io = io-hd # io-tl
    by (metis ATC.distinct(1) atc-reaction-empty list.exhaust)
  moreover obtain y where y-def : io-hd = (x,y)
    using Node calculation by (metis ATC.inject atc-reaction-nonempty surj-pair)
  ultimately  obtain q1x where q1x-def : q1x ∈ succ M1 (x,y) q1 atc-reaction M1 q1x (f y) io-tl
    using Node.prems(4) by blast

  then have pt1 : path M1 ([(x,y)] || [q1x]) q1
    by auto
  then have ls1 : [(x,y)] ∈ language-state M1 q1
    unfolding language-state-def path-def using list.simps(9) by force
  moreover have q1x ∈ io-targets M1 q1 [(x,y)]
    unfolding io-targets.simps
  proof −
    have f1: length [(x, y)] = length [q1x]
      by simp
    have q1x = target ([(x, y)] || [q1x]) q1
      by simp
    then show q1x ∈ {target ([(x, y)] || cs) q1 |cs. path M1 ([(x, y)] || cs) q1
                                                  ∧ length [(x, y)] = length cs}
      using f1 pt1 by blast
  qed
  ultimately have tgt1 : io-targets M1 q1 [(x,y)] = {q1x}
    using Node.prems io-targets-observable-singleton-ex q1x-def
    by (metis (no-types, lifting) singletonD)


  then have ls2 : [(x,y)] ∈ language-state M2 q2
    using Node.prems(1) ls1 by auto
  then obtain q2x where q2x-def : q2x ∈ succ M2 (x,y) q2
    unfolding language-state-def path-def
    using transition-system.path.cases by fastforce
  then have pt2 : path M2 ([(x,y)] || [q2x]) q2
    by auto
  then have q2x ∈ io-targets M2 q2 [(x,y)]
    using ls2 unfolding io-targets.simps
  proof −
    have f1: length [(x, y)] = length [q2x]
      by simp
    have q2x = target ([(x, y)] || [q2x]) q2
      by simp
    then show q2x ∈ {target ([(x, y)] || cs) q2 |cs. path M2 ([(x, y)] || cs) q2
                                                  ∧ length [(x, y)] = length cs}
      using f1 pt2 by blast
  qed

  then have tgt2 : io-targets M2 q2 [(x,y)] = {q2x}
    using Node.prems io-targets-observable-singleton-ex ls2 q2x-def
    by (metis (no-types, lifting) singletonD)
```

then have *language-state M1 q1x* ⊆ *language-state M2 q2x*
  using *language-state-inclusion-of-state-reached-by-same-sequence*
    [*of M1 q1 M2 q2* [(*x*,*y*)] *q1x q2x*]
    tgt1 tgt2 Node.prems **by** *auto*
moreover have *q1x* ∈ *nodes M1*
  using *q1x-def*(*1*) *Node.prems*(*2*) **by** (*metis insertI1 io-targets-nodes tgt1*)
moreover have *q2x* ∈ *nodes M2*
  using *q2x-def*(*1*) *Node.prems*(*3*) **by** (*metis insertI1 io-targets-nodes tgt2*)
ultimately have *q2x* ∈ *succ M2* (*x*,*y*) *q2* ∧ *atc-reaction M2 q2x* (*f y*) *io-tl*
  using *Node.IH*[*of f y q1x q2x io-tl*] *ob1 ob2 q1x-def*(*2*) *q2x-def* **by** *blast*

then show *atc-reaction M2 q2* (*Node x f*) *io* **using** *io-split y-def* **by** *blast*
qed

lemma *IO-reduction* :
  assumes *ls* : *language-state M1 q1* ⊆ *language-state M2 q2*
  and *el1* : *q1* ∈ *nodes M1*
  and *el2* : *q2* ∈ *nodes M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *IO M1 q1 t* ⊆ *IO M2 q2 t*
  using *assms atc-reaction-reduction* **unfolding** *IO.simps* **by** *auto*

lemma *IO-set-reduction* :
  assumes *ls* : *language-state M1 q1* ⊆ *language-state M2 q2*
  and *el1* : *q1* ∈ *nodes M1*
  and *el2* : *q2* ∈ *nodes M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *IO-set M1 q1* Ω ⊆ *IO-set M2 q2* Ω
proof −
  have ∀ *t* ∈ Ω . *IO M1 q1 t* ⊆ *IO M2 q2 t*
    using *assms IO-reduction* **by** *metis*
  then show *?thesis*
    unfolding *IO-set.simps* **by** *blast*
qed

lemma *B-reduction* :
  assumes *red* : *M1* ⪯ *M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *B M1 io* Ω ⊆ *B M2 io* Ω
proof
  fix *xy* **assume** *xy-assm* : *xy* ∈ *B M1 io* Ω
  then obtain *q1x* **where** *q1x-def* : *q1x* ∈ (*io-targets M1* (*initial M1*) *io*) ∧ *xy* ∈ *IO-set M1 q1x* Ω
    unfolding *B.simps* **by** *auto*
  then obtain *tr1* **where** *tr1-def* : *path M1* (*io* || *tr1*) (*initial M1*) ∧ *length io* = *length tr1*
    by *auto*

  then have *q1x-ob* : *io-targets M1* (*initial M1*) *io* = {*q1x*}
    using *assms*
    by (*metis io-targets-observable-singleton-ex language-state q1x-def singleton-iff*)

  then have *ls1* : *io* ∈ *language-state M1* (*initial M1*)
    by *auto*
  then have *ls2* : *io* ∈ *language-state M2* (*initial M2*)
    using *red* **by** *auto*

  then obtain *tr2* **where** *tr2-def* : *path M2* (*io* || *tr2*) (*initial M2*) ∧ *length io* = *length tr2*
    by *auto*
  then obtain *q2x* **where** *q2x-def* : *q2x* ∈ (*io-targets M2* (*initial M2*) *io*)
    by *auto*

then have *q2x-ob* : *io-targets M2* (*initial M2*) *io* = {*q2x*}
  using *tr2-def assms*
  by (*metis io-targets-observable-singleton-ex language-state singleton-iff*)

then have *language-state M1 q1x* ⊆ *language-state M2 q2x*
  using *q1x-ob assms* **unfolding** *io-reduction.simps*
  by (*simp add*: *language-state-inclusion-of-state-reached-by-same-sequence*)
then have *IO-set M1 q1x* Ω ⊆ *IO-set M2 q2x* Ω
  using *assms IO-set-reduction* **by** (*metis FSM.nodes.initial io-targets-nodes q1x-def q2x-def*)
moreover have *B M1 io* Ω = *IO-set M1 q1x* Ω
  using *q1x-ob* **by** *auto*
moreover have *B M2 io* Ω = *IO-set M2 q2x* Ω
  using *q2x-ob* **by** *auto*
ultimately have *B M1 io* Ω ⊆ *B M2 io* Ω
  by *simp*
then show *xy* ∈ *B M2 io* Ω **using** *xy-assm*
  by *blast*
qed

lemma *append-io-B-reduction* :
  assumes *red* : *M1* ⪯ *M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *append-io-B M1 io* Ω ⊆ *append-io-B M2 io* Ω
proof
  fix *ioR* **assume** *ioR-assm* : *ioR* ∈ *append-io-B M1 io* Ω
  then obtain *res* **where** *res-def* : *ioR* = *io* @ *res res* ∈ *B M1 io* Ω
    by *auto*
  then have *res* ∈ *B M2 io* Ω
    using *assms B-reduction* **by** (*metis* (*no-types*, *opaque-lifting*) *subset-iff*)
  then show *ioR* ∈ *append-io-B M2 io* Ω
    using *ioR-assm res-def* **by** *auto*
qed

lemma *atc-io-reduction-on-reduction*[*intro*] :
  assumes *red* : *M1* ⪯ *M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *atc-io-reduction-on M1 M2 iseq* Ω
unfolding *atc-io-reduction-on.simps* **proof**
  show L_{in} *M1* {*iseq*} ⊆ L_{in} *M2* {*iseq*}
    using *red* **by** *auto*
next
  show ∀ *io*∈L_{in} *M1* {*iseq*}. *B M1 io* Ω ⊆ *B M2 io* Ω
    using *B-reduction assms* **by** *blast*
qed

lemma *atc-io-reduction-on-sets-reduction*[*intro*] :
  assumes *red* : *M1* ⪯ *M2*
  and *ob1* : *observable M1*
  and *ob2* : *observable M2*
shows *atc-io-reduction-on-sets M1 TS* Ω *M2*
  using *assms atc-io-reduction-on-reduction* **by** (*metis atc-io-reduction-on-sets.elims*(*3*))

lemma *atc-io-reduction-on-sets-via-*LS_{in} :
  assumes *atc-io-reduction-on-sets M1 TS* Ω *M2*
  shows (L_{in} *M1 TS* ∪ (⋃ *io*∈L_{in} *M1 TS*. *B M1 io* Ω))
    ⊆ (L_{in} *M2 TS* ∪ (⋃ *io*∈L_{in} *M2 TS*. *B M2 io* Ω))
proof −
  have ∀ *iseq* ∈ *TS* . (L_{in} *M1* {*iseq*} ⊆ L_{in} *M2* {*iseq*}
      ∧ (∀ *io* ∈ L_{in} *M1* {*iseq*} . *B M1 io* Ω ⊆ *B M2 io* Ω))

```
    using assms by auto
  then have ∀ iseq ∈ TS . (⋃io∈L_in M1 {iseq}. B M1 io Ω)
                          ⊆ (⋃io∈L_in M2 {iseq}. B M2 io Ω)
    by blast
  moreover have ∀ iseq ∈ TS . (⋃io∈L_in M2 {iseq}. B M2 io Ω)
                              ⊆ (⋃io∈L_in M2 TS. B M2 io Ω)
    unfolding language-state-for-inputs.simps by blast
  ultimately have elem-subset : ∀ iseq ∈ TS .
                                 (⋃io∈L_in M1 {iseq}. B M1 io Ω)
                                   ⊆ (⋃io∈L_in M2 TS. B M2 io Ω)
    by blast

  show ?thesis
  proof
    fix x assume x ∈ L_in M1 TS ∪ (⋃io∈L_in M1 TS. B M1 io Ω)
    then show x ∈ L_in M2 TS ∪ (⋃io∈L_in M2 TS. B M2 io Ω)
    proof (cases x ∈ L_in M1 TS)
      case True
      then obtain iseq where iseq ∈ TS x∈ L_in M1 {iseq}
        unfolding language-state-for-inputs.simps by blast
      then have atc-io-reduction-on M1 M2 iseq Ω
        using assms by auto
      then have L_in M1 {iseq} ⊆ L_in M2 {iseq}
        by auto
      then have x ∈ L_in M2 TS
        by (metis (no-types, lifting) UN-I
            ‹⋀thesis. (⋀iseq. ⟦iseq ∈ TS; x ∈ L_in M1 {iseq}⟧ ⟹ thesis) ⟹ thesis›
            ‹∀ iseq∈TS. L_in M1 {iseq} ⊆ L_in M2 {iseq} ∧ (∀ io∈L_in M1 {iseq}. B M1 io Ω ⊆ B M2 io Ω)›
            language-state-for-input-alt-def language-state-for-inputs-alt-def set-rev-mp)
      then show ?thesis
        by blast
    next
      case False
      then have x ∈ (⋃io∈L_in M1 TS. B M1 io Ω)
        using ‹x ∈ L_in M1 TS ∪ (⋃io∈L_in M1 TS. B M1 io Ω)› by blast
      then obtain io where io ∈ L_in M1 TS x ∈ B M1 io Ω
        by blast
      then obtain iseq where iseq ∈ TS io∈L_in M1 {iseq}
        unfolding language-state-for-inputs.simps by blast
      have x ∈ (⋃io∈L_in M1 {iseq}. B M1 io Ω)
        using ‹io ∈ L_in M1 {iseq}› ‹x ∈ B M1 io Ω› by blast
      then have x ∈ (⋃io∈L_in M2 TS. B M2 io Ω)
        using ‹iseq ∈ TS› elem-subset by blast
      then show ?thesis
        by blast
    qed
  qed
qed
```

end
theory *ASC-LB*
imports *../ATC/ATC ../FSM/FSM-Product*
begin

4 The lower bound function

This theory defines the lower bound function LB and its properties.

Function LB calculates a lower bound on the number of states of some FSM in order for some sequence to not contain certain repetitions.

4.1 Permutation function Perm

Function `Perm` calculates all possible reactions of an FSM to a set of inputs sequences such that every set in the calculated set of reactions contains exactly one reaction for each input sequence.

```
fun Perm :: 'in list set ⇒ ('in, 'out, 'state) FSM ⇒ ('in × 'out) list set set where
  Perm V M = {image f V | f . ∀ v ∈ V . f v ∈ language-state-for-input M (initial M) v }

lemma perm-empty :
  assumes is-det-state-cover M2 V
  and V'' ∈ Perm V M1
shows [] ∈ V''
proof −
  have init-seq : [] ∈ V using det-state-cover-empty assms by simp
  obtain f where f-def : V'' = image f V
                        ∧ (∀ v ∈ V . f v ∈ language-state-for-input M1 (initial M1) v)
    using assms by auto
  then have f [] = []
    using init-seq by (metis language-state-for-input-empty singleton-iff)
  then show ?thesis
    using init-seq f-def by (metis image-eqI)
qed

lemma perm-elem-finite :
  assumes is-det-state-cover M2 V
  and     well-formed M2
  and     V'' ∈ Perm V M1
  shows finite V''
proof −
  obtain f where is-det-state-cover-ass M2 f ∧ V = f ‘ d-reachable M2 (initial M2)
    using assms by auto
  moreover have finite (d-reachable M2 (initial M2))
  proof −
    have finite (nodes M2)
      using assms by auto
    moreover have nodes M2 = reachable M2 (initial M2)
      by auto
    ultimately have finite (reachable M2 (initial M2))
      by simp
    moreover have d-reachable M2 (initial M2) ⊆ reachable M2 (initial M2)
      by auto
    ultimately show ?thesis
      using infinite-super by blast
  qed
  ultimately have finite V
    by auto
  moreover obtain f'' where V'' = image f'' V
                        ∧ (∀ v ∈ V . f'' v ∈ language-state-for-input M1 (initial M1) v)
    using assms(3) by auto
  ultimately show ?thesis
    by simp
qed

lemma perm-inputs :
  assumes V'' ∈ Perm V M
  and     vs ∈ V''
shows map fst vs ∈ V
proof −
  obtain f where f-def : V'' = image f V
                        ∧ (∀ v ∈ V . f v ∈ language-state-for-input M (initial M) v)
    using assms by auto
  then obtain v where v-def : v ∈ V ∧ f v = vs
    using assms by auto
  then have vs ∈ language-state-for-input M (initial M) v
    using f-def by auto
  then show ?thesis
    using v-def unfolding language-state-for-input.simps by auto
```

qed

lemma *perm-inputs-diff* :
assumes $V'' \in Perm\ V\ M$
and $vs1 \in V''$
and $vs2 \in V''$
and $vs1 \neq vs2$
shows *map fst vs1* $\neq$ *map fst vs2*
proof −
obtain *f* **where** *f-def* : $V'' = image\ f\ V$
$\wedge$ ($\forall$ $v \in V$. $f\ v \in$ *language-state-for-input M* (*initial M*) *v*)
using *assms* **by** *auto*
then obtain *v1 v2* **where** *v-def* : $v1 \in V \wedge f\ v1 = vs1 \wedge v2 \in V \wedge f\ v2 = vs2$
using *assms* **by** *auto*
then have $vs1 \in$ *language-state-for-input M* (*initial M*) *v1*
$vs2 \in$ *language-state-for-input M* (*initial M*) *v2*
using *f-def* **by** *auto*
moreover have $v1 \neq v2$
using *v-def assms*(*4*) **by** *blast*
ultimately show *?thesis*
by *auto*
qed

lemma *perm-language* :
assumes $V'' \in Perm\ V\ M$
and $vs \in V''$
shows $vs \in L\ M$
proof −
obtain *f* **where** *f-def* : $image\ f\ V = V''$
$\wedge$ ($\forall$ $v \in V$. $f\ v \in$ *language-state-for-input M* (*initial M*) *v*)
using *assms*(*1*) **by** *auto*
then have $\exists$ *v* . $f\ v = vs \wedge f\ v \in$ *language-state-for-input M* (*initial M*) *v*
using *assms*(*2*) **by** *blast*
then show *?thesis*
by *auto*
qed

4.2 Helper predicates

The following predicates are used to combine often repeated assumption.

abbreviation *asc-fault-domain M2 M1 m* $\equiv$ (*inputs M2* = *inputs M1* $\wedge$ *card* (*nodes M1*) $\leq$ *m*)

lemma *asc-fault-domain-props*[*elim*!] :
assumes *asc-fault-domain M2 M1 m*
shows *inputs M2* = *inputs M1*
card (*nodes M1*) $\leq$ *m***using** *assms* **by** *auto*

abbreviation
test-tools M2 M1 FAIL PM V $\Omega \equiv$ (
productF M2 M1 FAIL PM
$\wedge$ *is-det-state-cover M2 V*
$\wedge$ *applicable-set M2* Ω
)

lemma *test-tools-props*[*elim*] :
assumes *test-tools M2 M1 FAIL PM V* Ω
and *asc-fault-domain M2 M1 m*
shows *productF M2 M1 FAIL PM*
is-det-state-cover M2 V
applicable-set M2 Ω
applicable-set M1 Ω
proof −
show *productF M2 M1 FAIL PM* **using** *assms*(*1*) **by** *blast*
show *is-det-state-cover M2 V* **using** *assms*(*1*) **by** *blast*
show *applicable-set M2* Ω **using** *assms*(*1*) **by** *blast*
then show *applicable-set M1* Ω

unfolding *applicable-set.simps applicable.simps*
using *asc-fault-domain-props*(*1*)[*OF assms*(*2*)] **by** *simp*
qed

lemma *perm-nonempty* :
assumes *is-det-state-cover M2 V*
and *OFSM M1*
and *OFSM M2*
and *inputs M1* = *inputs M2*
shows *Perm V M1* ≠ {}
proof −
have *finite* (*nodes M2*)
using *assms*(*3*) **by** *auto*
moreover have *d-reachable M2* (*initial M2*) ⊆ *nodes M2*
by *auto*
ultimately have *finite V*
using *det-state-cover-card*[*OF assms*(*1*)]
by (*metis assms*(*1*) *finite-imageI infinite-super is-det-state-cover.elims*(*2*))

have [] ∈ *V*
using *assms*(*1*) *det-state-cover-empty* **by** *blast*

have ⋀ *VS* . *VS* ⊆ *V* ∧ *VS* ≠ {} ⟹ *Perm VS M1* ≠ {}
proof −
fix *VS* **assume** *VS* ⊆ *V* ∧ *VS* ≠ {}
then have *finite VS* **using** ‹*finite V*›
using *infinite-subset* **by** *auto*
then show *Perm VS M1* ≠ {}
using ‹*VS* ⊆ *V* ∧ *VS* ≠ {}› ‹*finite VS*›
proof (*induction VS*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert vs F*)
then have *vs* ∈ *V* **by** *blast*

obtain *q2* **where** *d-reaches M2* (*initial M2*) *vs q2*
using *det-state-cover-d-reachable*[*OF assms*(*1*) ‹*vs* ∈ *V*›] **by** *blast*
then obtain *vs′ vsP* **where** *io-path* : *length vs* = *length vs′*
∧ *length vs* = *length vsP*
∧ (*path M2* ((*vs* || *vs′*) || *vsP*) (*initial M2*))
∧ *target* ((*vs* || *vs′*) || *vsP*) (*initial M2*) = *q2*
by *auto*

have *well-formed M2*
using *assms* **by** *auto*

have *map fst* (*map fst* ((*vs* || *vs′*) || *vsP*)) = *vs*
proof −
have *length* (*vs* || *vs′*) = *length vsP*
using *io-path* **by** *simp*
then show *?thesis*
using *io-path* **by** *auto*
qed
moreover have *set* (*map fst* (*map fst* ((*vs* || *vs′*) || *vsP*))) ⊆ *inputs M2*
using *path-input-containment*[*OF* ‹*well-formed M2*›, *of* (*vs* || *vs′*) || *vsP initial M2*]
io-path
by *linarith*
ultimately have *set vs* ⊆ *inputs M2*
by *presburger*

then have *set vs* ⊆ *inputs M1*

```
      using assms by auto

    then have L_in M1 {vs} ≠ {}
      using assms(2) language-state-for-inputs-nonempty
      by (metis FSM.nodes.initial)
    then have language-state-for-input M1 (initial M1) vs ≠ {}
      by auto
    then obtain vs' where vs' ∈ language-state-for-input M1 (initial M1) vs
      by blast

    show ?case
    proof (cases F = {})
      case True
      moreover obtain f where f vs = vs'
        by force
      ultimately have image f (insert vs F) ∈ Perm (insert vs F) M1
        using Perm.simps ‹vs' ∈ language-state-for-input M1 (initial M1) vs› by blast
      then show ?thesis by blast
    next
      case False
      then obtain F'' where F'' ∈ Perm F M1
        using insert.IH insert.hyps(1) insert.prems(1) by blast
      then obtain f where F'' = image f F
                          (∀ v ∈ F . f v ∈ language-state-for-input M1 (initial M1) v)
        by auto
      let ?f = f(vs := vs')
      have ∀ v ∈ (insert vs F) . ?f v ∈ language-state-for-input M1 (initial M1) v
      proof
        fix v assume v ∈ insert vs F
        then show ?f v ∈ language-state-for-input M1 (initial M1) v
        proof (cases v = vs)
          case True
          then show ?thesis
            using ‹vs' ∈ language-state-for-input M1 (initial M1) vs› by auto
        next
          case False
          then have v ∈ F
            using ‹v ∈ insert vs F› by blast
          then show ?thesis
            using False ‹∀ v∈F. f v ∈ language-state-for-input M1 (initial M1) v› by auto
        qed
      qed
      then have image ?f (insert vs F) ∈ Perm (insert vs F) M1
        using Perm.simps by blast
      then show ?thesis
        by blast
    qed
  qed
qed

then show ?thesis
  using ‹[] ∈ V› by blast
qed



lemma perm-elem :
  assumes is-det-state-cover M2 V
  and OFSM M1
  and OFSM M2
  and inputs M1 = inputs M2
  and vs ∈ V
  and vs' ∈ language-state-for-input M1 (initial M1) vs
obtains V''
where V'' ∈ Perm V M1 vs' ∈ V''
proof -
```

```
  obtain V'' where V'' ∈ Perm V M1
    using perm-nonempty[OF assms(1−4)] by blast
  then obtain f where V'' = image f V
                    (∀ v ∈ V . f v ∈ language-state-for-input M1 (initial M1) v)
    by auto

  let ?f = f(vs := vs')

  have ∀ v ∈ V . (?f v) ∈ (language-state-for-input M1 (initial M1) v)
    using ‹∀ v∈V. (f v) ∈ (language-state-for-input M1 (initial M1) v)› assms(6) by fastforce

  then have (image ?f V) ∈ Perm V M1
    unfolding Perm.simps by blast
  moreover have vs' ∈ image ?f V
    by (metis assms(5) fun-upd-same imageI)
  ultimately show ?thesis
    using that by blast
qed
```

4.3 Function R

Function R calculates the set of suffixes of a sequence that reach a given state if applied after a given other sequence.

```
fun R :: ('in, 'out, 'state) FSM ⇒ 'state ⇒ ('in × 'out) list
         ⇒ ('in × 'out) list ⇒ ('in × 'out) list set
  where
  R M s vs xs = { vs@xs' | xs' . xs' ≠ []
                              ∧ prefix xs' xs
                              ∧ s ∈ io-targets M (initial M) (vs@xs') }

lemma finite-R : finite (R M s vs xs)
proof −
  have R M s vs xs ⊆ { vs @ xs' | xs' .prefix xs' xs }
    by auto
  then have R M s vs xs ⊆ image (λ xs' . vs @ xs') {xs' . prefix xs' xs}
    by auto
  moreover have {xs' . prefix xs' xs} = {take n xs | n . n ≤ length xs}
  proof
    show {xs'. prefix xs' xs} ⊆ {take n xs |n. n ≤ length xs}
    proof
      fix xs' assume xs' ∈ {xs'. prefix xs' xs}
      then obtain zs' where xs' @ zs' = xs
        by (metis (full-types) mem-Collect-eq prefixE)
      then obtain i where xs' = take i xs ∧ i ≤ length xs
        by (metis (full-types) append-eq-conv-conj le-cases take-all)
      then show xs' ∈ {take n xs |n. n ≤ length xs}
        by auto
    qed
    show {take n xs |n. n ≤ length xs} ⊆ {xs'. prefix xs' xs}
      using take-is-prefix by force
  qed
  moreover have finite {take n xs | n . n ≤ length xs}
    by auto
  ultimately show ?thesis
    by auto
qed



lemma card-union-of-singletons :
  assumes ∀ S ∈ SS . (∃ t . S = {t})
shows card (⋃ SS) = card SS
proof −
  let ?f = λ x . {x}
  have bij-betw ?f (⋃ 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 ∀ *S1* ∈ *SS* . ∀ *S2* ∈ *SS* . *S1* = *S2* ∨ *f S1* ∩ *f S2* = {}
and *finite SS*
and ∀ *S* ∈ *SS* . *f S* ≠ {}
shows *card* (*image f SS*) = *card SS*
proof −
from *assms*(*2*) **have** ∀ *S1* ∈ *SS* . ∀ *S2* ∈ *SS* . *S1* = *S2* ∨ *f S1* ∩ *f S2* = {}
⟹ ∀ *S* ∈ *SS* . *f S* ≠ {} ⟹ *?thesis*
proof (*induction SS*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert x F*)
then have ¬ (∃ *y* ∈ *F* . *f y* = *f x*)
by *auto*
then have *f x* ∉ *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 *R-count* :
assumes (*vs* @ *xs*) ∈ *L M1* ∩ *L M2*
and *observable M1*
and *observable M2*
and *well-formed M1*
and *well-formed M2*
and *s* ∈ *nodes M2*
and *productF M2 M1 FAIL PM*
and *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
and *path PM* (*xs* || *tr*) (*q2*,*q1*)
and *length xs* = *length tr*
and *distinct* (*states* (*xs* || *tr*) (*q2*,*q1*))
shows *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*R M2 s vs xs*))) = *card* (*R M2 s vs xs*)
— each sequence in the set calculated by R reaches a different state in M1
proof −

— Proof sketch: - states of PM reached by the sequences calculated by R can differ only in their second value - the sequences in the set calculated by R reach different states in PM due to distinctness

have *obs-PM* : *observable PM* **using** *observable-productF assms*(*2*) *assms*(*3*) *assms*(*7*) **by** *blast*

have *state-component-2* : ∀ *io* ∈ (*R M2 s vs xs*) . *io-targets M2* (*initial M2*) *io* = {*s*}
proof
fix *io* **assume** *io* ∈ *R M2 s vs xs*
then have *s* ∈ *io-targets M2* (*initial M2*) *io*
by *auto*
moreover have *io* ∈ *language-state M2* (*initial M2*)
using *calculation* **by** *auto*
ultimately show *io-targets M2* (*initial M2*) *io* = {*s*}
using *assms*(*3*) *io-targets-observable-singleton-ex* **by** (*metis singletonD*)
qed

```
moreover have ∀ io ∈ R M2 s vs xs . io-targets PM (initial PM) io
                = io-targets M2 (initial M2) io × io-targets M1 (initial M1) io
proof
  fix io assume io-assm : io ∈ R M2 s vs xs
  then have io-prefix : prefix io (vs @ xs)
    by auto
  then have io-lang-subs : io ∈ L M1 ∧ io ∈ L M2
    using assms(1) unfolding prefix-def by (metis IntE language-state language-state-split)
  then have io-lang-inter : io ∈ L M1 ∩ L M2
    by simp
  then have io-lang-pm : io ∈ L PM
    using productF-language assms by blast
  moreover obtain p2 p1 where (p2,p1) ∈ io-targets PM (initial PM) io
    by (metis assms(2) assms(3) assms(7) calculation insert-absorb insert-ident insert-not-empty
        io-targets-observable-singleton-ob observable-productF singleton-insert-inj-eq subrelI)
  ultimately have targets-pm : io-targets PM (initial PM) io = {(p2,p1)}
    using assms io-targets-observable-singleton-ex singletonD by (metis observable-productF)
  then obtain trP where trP-def : target (io || trP) (initial PM) = (p2,p1)
                                  ∧ path PM (io || trP) (initial PM)
                                  ∧ length io = length trP
    proof −
      assume a1: ⋀trP. target (io || trP) (initial PM) = (p2, p1)
                      ∧ path PM (io || trP) (initial PM)
                      ∧ length io = length trP ⟹ thesis
      have ∃ps. target (io || ps) (initial PM) = (p2, p1)
              ∧ path PM (io || ps) (initial PM) ∧ length io = length ps
        using ‹(p2, p1) ∈ io-targets PM (initial PM) io› by auto
      then show ?thesis
        using a1 by blast
    qed
    then have trP-unique : { tr . path PM (io || tr) (initial PM) ∧ length io = length tr }
                          = { trP }
      using observable-productF observable-path-unique-ex[of PM io initial PM]
          io-lang-pm assms(2) assms(3) assms(7)
    proof −
      obtain pps :: ('d × 'c) list where
        f1: {ps. path PM (io || ps) (initial PM) ∧ length io = length ps} = {pps}
            ∨ ¬ observable PM
        by (metis (no-types) ‹⋀thesis. ⟦observable PM; io ∈ L PM; ⋀tr.
                                {t. path PM (io || t) (initial PM)
                                ∧ length io = length t} = {tr} ⟹ thesis⟧ ⟹ thesis›
          io-lang-pm)
      have f2: observable PM
        by (meson ‹observable M1› ‹observable M2› ‹productF M2 M1 FAIL PM› observable-productF)
      then have trP ∈ {pps}
        using f1 trP-def by blast
      then show ?thesis
        using f2 f1 by force
    qed


  obtain trIO2 where trIO2-def : {tr . path M2 (io||tr) (initial M2) ∧ length io = length tr}
                                = { trIO2 }
    using observable-path-unique-ex[of M2 io initial M2] io-lang-subs assms(3) by blast
  obtain trIO1 where trIO1-def : {tr . path M1 (io||tr) (initial M1) ∧ length io = length tr}
                                = { trIO1 }
    using observable-path-unique-ex[of M1 io initial M1] io-lang-subs assms(2) by blast

  have path PM (io || trIO2 || trIO1) (initial M2, initial M1)
      ∧ length io = length trIO2
      ∧ length trIO2 = length trIO1
  proof −
    have f1: path M2 (io || trIO2) (initial M2) ∧ length io = length trIO2
      using trIO2-def by auto
    have f2: path M1 (io || trIO1) (initial M1) ∧ length io = length trIO1
```

using *trIO1-def* **by** *auto*
then have *length trIO2 = length trIO1*
using *f1* **by** *presburger*
then show *?thesis*
using *f2 f1 assms(4) assms(5) assms(7)* **by** *blast*
qed
then have *trP-split* : *path PM (io || trIO2 || trIO1) (initial PM)*
$\wedge$ *length io = length trIO2*
$\wedge$ *length trIO2 = length trIO1*
using *assms(7)* **by** *auto*
then have *trP-zip* : *trIO2 || trIO1 = trP*
using *trP-def trP-unique* **using** *length-zip* **by** *fastforce*

have *target (io || trIO2) (initial M2) = p2*
$\wedge$ *path M2 (io || trIO2) (initial M2)*
$\wedge$ *length io = length trIO2*
using *trP-zip trP-split assms(7) trP-def trIO2-def* **by** *auto*
then have *p2* $\in$ *io-targets M2 (initial M2) io*
by *auto*
then have *targets-2* : *io-targets M2 (initial M2) io = {p2}*
by *(metis state-component-2 io-assm singletonD)*

have *target (io || trIO1) (initial M1) = p1*
$\wedge$ *path M1 (io || trIO1) (initial M1)*
$\wedge$ *length io = length trIO1*
using *trP-zip trP-split assms(7) trP-def trIO1-def* **by** *auto*
then have *p1* $\in$ *io-targets M1 (initial M1) io*
by *auto*
then have *targets-1* : *io-targets M1 (initial M1) io = {p1}*
by *(metis io-lang-subs assms(2) io-targets-observable-singleton-ex singletonD)*

have *io-targets M2 (initial M2) io* $\times$ *io-targets M1 (initial M1) io = {(p2,p1)}*
using *targets-2 targets-1* **by** *simp*
then show *io-targets PM (initial PM) io*
= io-targets M2 (initial M2) io $\times$ *io-targets M1 (initial M1) io*
using *targets-pm* **by** *simp*
qed

ultimately have *state-components* : $\forall$ *io* $\in$ *R M2 s vs xs . io-targets PM (initial PM) io*
= {s} $\times$ *io-targets M1 (initial M1) io*
by *auto*

then have $\bigcup$ *(image (io-targets PM (initial PM)) (R M2 s vs xs))*
= $\bigcup$ *(image (*λ *io . {s}* $\times$ *io-targets M1 (initial M1) io) (R M2 s vs xs))*
by *auto*
then have $\bigcup$ *(image (io-targets PM (initial PM)) (R M2 s vs xs))*
= {s} $\times$ $\bigcup$ *(image (io-targets M1 (initial M1)) (R M2 s vs xs))*
by *auto*
then have *card (*$\bigcup$ *(image (io-targets PM (initial PM)) (R M2 s vs xs)))*
= card ($\bigcup$ *(image (io-targets M1 (initial M1)) (R M2 s vs xs)))*
by *(metis (no-types) card-cartesian-product-singleton)*

moreover have *card (*$\bigcup$ *(image (io-targets PM (initial PM)) (R M2 s vs xs)))*
= card (R M2 s vs xs)
proof *(rule ccontr)*
assume *assm* : *card (*$\bigcup$ *(io-targets PM (initial PM) ‘ R M2 s vs xs))* $\neq$ *card (R M2 s vs xs)*

have $\forall$ *io* $\in$ *R M2 s vs xs . io* $\in$ *L PM*
proof
fix *io* **assume** *io-assm* : *io* $\in$ *R M2 s vs xs*
then have *prefix io (vs @ xs)*
by *auto*
then have *io* $\in$ *L M1* $\wedge$ *io* $\in$ *L M2*
using *assms(1)* **unfolding** *prefix-def* **by** *(metis IntE language-state language-state-split)*
then show *io* $\in$ *L PM*
using *productF-language assms* **by** *blast*

qed
then have *singletons* : ∀ *io* ∈ *R M2 s vs xs* . (∃ *t* . *io-targets PM* (*initial PM*) *io* = {*t*})
  using *io-targets-observable-singleton-ex observable-productF assms* **by** *metis*
then have *card-targets* : *card* (⋃(*io-targets PM* (*initial PM*) ‘ *R M2 s vs xs*))
                              = *card* (*image* (*io-targets PM* (*initial PM*)) (*R M2 s vs xs*))
  using *finite-R card-union-of-singletons*
        [*of image* (*io-targets PM* (*initial PM*)) (*R M2 s vs xs*)]
  by *simp*

moreover have *card* (*image* (*io-targets PM* (*initial PM*)) (*R M2 s vs xs*)) ≤ *card* (*R M2 s vs xs*)
  using *finite-R* **by** (*metis card-image-le*)
ultimately have *card-le* : *card* (⋃(*io-targets PM* (*initial PM*) ‘ *R M2 s vs xs*))
                              < *card* (*R M2 s vs xs*)
  using *assm* **by** *linarith*

have ∃ *io1* ∈ (*R M2 s vs xs*) . ∃ *io2* ∈ (*R M2 s vs xs*) . *io1* ≠ *io2*
        ∧ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2* ≠ {}
proof (*rule ccontr*)
  assume ¬ (∃ *io1*∈*R M2 s vs xs*. ∃ *io2*∈*R M2 s vs xs*. *io1* ≠ *io2*
              ∧ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2* ≠ {})
  then have ∀ *io1*∈*R M2 s vs xs*. ∀ *io2*∈*R M2 s vs xs*. *io1* = *io2*
              ∨ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2* = {}
    by *blast*
  moreover have ∀ *io*∈*R M2 s vs xs*. *io-targets PM* (*initial PM*) *io* ≠ {}
    by (*metis insert-not-empty singletons*)
  ultimately have *card* (*image* (*io-targets PM* (*initial PM*)) (*R M2 s vs xs*))
                  = *card* (*R M2 s vs xs*)
    using *finite-R*[*of M2 s vs xs*] *card-union-of-distinct*
          [*of R M2 s vs xs* (*io-targets PM* (*initial PM*))]
    by *blast*
  then show *False*
    using *card-le card-targets* **by** *linarith*
qed

then have ∃ *io1 io2* . *io1* ∈ (*R M2 s vs xs*)
                        ∧ *io2* ∈ (*R M2 s vs xs*)
                        ∧ *io1* ≠ *io2*
                        ∧ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2* ≠ {}
  by *blast*
moreover have ∀ *io1 io2* . (*io1* ∈ (*R M2 s vs xs*) ∧ *io2* ∈ (*R M2 s vs xs*) ∧ *io1* ≠ *io2*)
                          ⟶ *length io1* ≠ *length io2*
proof (*rule ccontr*)
  assume ¬ (∀ *io1 io2*. *io1* ∈ *R M2 s vs xs* ∧ *io2* ∈ *R M2 s vs xs* ∧ *io1* ≠ *io2*
        ⟶ *length io1* ≠ *length io2*)
  then obtain *io1 io2* **where** *io-def* : *io1* ∈ *R M2 s vs xs*
                                  ∧ *io2* ∈ *R M2 s vs xs*
                                  ∧ *io1* ≠ *io2*
                                  ∧ *length io1* = *length io2*
    by *auto*
  then have *prefix io1* (*vs* @ *xs*) ∧ *prefix io2* (*vs* @ *xs*)
    by *auto*
  then have *io1* = *take* (*length io1*) (*vs* @ *xs*) ∧ *io2* = *take* (*length io2*) (*vs* @ *xs*)
    by (*metis append-eq-conv-conj prefixE*)
  then show *False*
    using *io-def* **by** *auto*
qed

ultimately obtain *io1 io2* **where** *rep-ios-def* :
  io1 ∈ (*R M2 s vs xs*)
    ∧ *io2* ∈ (*R M2 s vs xs*)
    ∧ *length io1* < *length io2*
    ∧ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2* ≠ {}
  by (*metis inf-sup-aci*(*1*) *linorder-neqE-nat*)

obtain *rep* **where** (*s*,*rep*) ∈ *io-targets PM* (*initial PM*) *io1* ∩ *io-targets PM* (*initial PM*) *io2*
proof −

```
  assume a1: ⋀rep. (s, rep) ∈ io-targets PM (initial PM) io1 ∩ io-targets PM (initial PM) io2
                 ⟹ thesis
  have ∃f. Sigma {s} f ∩ (io-targets PM (initial PM) io1 ∩ io-targets PM (initial PM) io2)
           ≠ {}
    by (metis (no-types) inf.left-idem rep-ios-def state-components)
  then show ?thesis
    using a1 by blast
qed
then have rep-state : io-targets PM (initial PM) io1 = {(s,rep)}
                      ∧ io-targets PM (initial PM) io2 = {(s,rep)}
  by (metis Int-iff rep-ios-def singletonD singletons)

obtain io1X io2X where rep-ios-split : io1 = vs @ io1X
                                      ∧ prefix io1X xs
                                      ∧ io2 = vs @ io2X
                                      ∧ prefix io2X xs
  using rep-ios-def by auto
then have length io1 > length vs
  using rep-ios-def by auto


— get a path from (initial PM) to (q2,q1)

have vs@xs ∈ L PM
  by (metis (no-types) assms(1) assms(4) assms(5) assms(7) inf-commute productF-language)
then have vs ∈ L PM
  by (meson language-state-prefix)
then obtain trV where trV-def : {tr . path PM (vs || tr) (initial PM) ∧ length vs = length tr}
                                = { trV }
  using observable-path-unique-ex[of PM vs initial PM]
        assms(2) assms(3) assms(7) observable-productF
  by blast
let ?qv = target (vs || trV) (initial PM)

have ?qv ∈ io-targets PM (initial PM) vs
  using trV-def by auto
then have qv-simp[simp] : ?qv = (q2,q1)
  using singletons assms by blast
then have ?qv ∈ nodes PM
  using trV-def assms by blast


— get a path using io1X from the state reached by vs in PM

obtain tr1X-all where tr1X-all-def : path PM (vs @ io1X || tr1X-all) (initial PM)
                                    ∧ length (vs @ io1X) = length tr1X-all
  using rep-ios-def rep-ios-split by auto
let ?tr1X = drop (length vs) tr1X-all
have take (length vs) tr1X-all = trV
proof −
  have path PM (vs || take (length vs) tr1X-all) (initial PM)
        ∧ length vs = length (take (length vs) tr1X-all)
    using tr1X-all-def trV-def
    by (metis (no-types, lifting) FSM.path-append-elim append-eq-conv-conj
        length-take zip-append1)
  then show take (length vs) tr1X-all = trV
    using trV-def by blast
qed
then have tr1X-def : path PM (io1X || ?tr1X) ?qv ∧ length io1X = length ?tr1X
proof −
  have length tr1X-all = length vs + length io1X
    using tr1X-all-def by auto
  then have length io1X = length tr1X-all − length vs
    by presburger
  then show ?thesis
    by (metis (no-types) FSM.path-append-elim ‹take (length vs) tr1X-all = trV›
```

length-drop tr1X-all-def zip-append1)
qed
then have *io1X-lang* : *io1X* ∈ *language-state PM ?qv*
  by *auto*
then obtain *tr1X′* **where** *tr1X′-def* : {*tr* . *path PM* (*io1X* || *tr*) *?qv* ∧ *length io1X* = *length tr*}
  = { *tr1X′* }
  using *observable-path-unique-ex*[*of PM io1X ?qv*]
  assms(*2*) *assms*(*3*) *assms*(*7*) *observable-productF*
  by *blast*
moreover have *?tr1X* ∈ { *tr* . *path PM* (*io1X* || *tr*) *?qv* ∧ *length io1X* = *length tr* }
  using *tr1X-def* **by** *auto*
ultimately have *tr1x-unique* : *tr1X′* = *?tr1X*
  by *simp*

— get a path using io2X from the state reached by vs in PM

obtain *tr2X-all* **where** *tr2X-all-def* : *path PM* (*vs* @ *io2X* || *tr2X-all*) (*initial PM*)
  ∧ *length* (*vs* @ *io2X*) = *length tr2X-all*
  using *rep-ios-def rep-ios-split* **by** *auto*
let *?tr2X* = *drop* (*length vs*) *tr2X-all*
have *take* (*length vs*) *tr2X-all* = *trV*
proof −
  have *path PM* (*vs* || *take* (*length vs*) *tr2X-all*) (*initial PM*)
  ∧ *length vs* = *length* (*take* (*length vs*) *tr2X-all*)
  using *tr2X-all-def trV-def*
  by (*metis* (*no-types*, *lifting*) *FSM.path-append-elim append-eq-conv-conj*
  length-take zip-append1)
  then show *take* (*length vs*) *tr2X-all* = *trV*
  using *trV-def* **by** *blast*
qed
then have *tr2X-def* : *path PM* (*io2X* || *?tr2X*) *?qv* ∧ *length io2X* = *length ?tr2X*
proof −
  have *length tr2X-all* = *length vs* + *length io2X*
  using *tr2X-all-def* **by** *auto*
  then have *length io2X* = *length tr2X-all* − *length vs*
  by *presburger*
  then show *?thesis*
  by (*metis* (*no-types*) *FSM.path-append-elim* ‹*take* (*length vs*) *tr2X-all* = *trV*›
  length-drop tr2X-all-def zip-append1)
qed
then have *io2X-lang* : *io2X* ∈ *language-state PM ?qv* **by** *auto*
then obtain *tr2X′* **where** *tr2X′-def* : {*tr* . *path PM* (*io2X* || *tr*) *?qv* ∧ *length io2X* = *length tr*}
  = { *tr2X′* }
  using *observable-path-unique-ex*[*of PM io2X ?qv*] *assms*(*2*) *assms*(*3*) *assms*(*7*) *observable-productF*
  by *blast*
moreover have *?tr2X* ∈ { *tr* . *path PM* (*io2X* || *tr*) *?qv* ∧ *length io2X* = *length tr* }
  using *tr2X-def* **by** *auto*
ultimately have *tr2x-unique* : *tr2X′* = *?tr2X*
  by *simp*

— both paths reach the same state

have *io-targets PM* (*initial PM*) (*vs* @ *io1X*) = {(*s*,*rep*)}
  using *rep-state rep-ios-split* **by** *auto*
moreover have *io-targets PM* (*initial PM*) *vs* = {*?qv*}
  using *assms*(*8*) **by** *auto*
ultimately have *rep-via-1* : *io-targets PM ?qv io1X* = {(*s*,*rep*)}
  by (*meson obs-PM observable-io-targets-split*)
then have *rep-tgt-1* : *target* (*io1X* || *tr1X′*) *?qv* = (*s*,*rep*)
  using *obs-PM observable-io-target-unique-target*[*of PM ?qv io1X* (*s*,*rep*)] *tr1X′-def* **by** *blast*
have *length-1* : *length* (*io1X* || *tr1X′*) > *0*
  using ‹*length vs* < *length io1*› *rep-ios-split tr1X-def tr1x-unique* **by** *auto*

have *tr1X-alt-def* : *tr1X′* = *take* (*length io1X*) *tr*
  by (*metis* (*no-types*) *assms*(*10*) *assms*(*9*) *obs-PM observable-path-prefix qv-simp*
  rep-ios-split tr1X-def tr1x-unique)

moreover have *io1X = take (length io1X) xs*
using *rep-ios-split* **by** *(metis append-eq-conv-conj prefixE)*
ultimately have *(io1X || tr1X′) = take (length io1X) (xs || tr)*
by *(metis take-zip)*
moreover have *length (xs || tr) ≥ length (io1X || tr1X′)*
by *(metis (no-types) ‹io1X = take (length io1X) xs› assms(10) length-take length-zip nat-le-linear take-all tr1X-def tr1x-unique)*
ultimately have *rep-idx-1 : (states (xs || tr) ?qv) ! ((length io1X) − 1) = (s,rep)*
by *(metis (no-types, lifting) One-nat-def Suc-less-eq Suc-pred rep-tgt-1 length-1 less-Suc-eq-le map-snd-zip scan-length scan-nth states-alt-def tr1X-def tr1x-unique)*

have *io-targets PM (initial PM) (vs @ io2X) = {(s,rep)}*
using *rep-state rep-ios-split* **by** *auto*
moreover have *io-targets PM (initial PM) vs = {?qv}*
using *assms(8)* **by** *auto*
ultimately have *rep-via-2 : io-targets PM ?qv io2X = {(s,rep)}*
by *(meson obs-PM observable-io-targets-split)*
then have *rep-tgt-2 : target (io2X || tr2X′) ?qv = (s,rep)*
using *obs-PM observable-io-target-unique-target[of PM ?qv io2X (s,rep)] tr2X′-def* **by** *blast*
moreover have *length-2 : length (io2X || tr2X′) > 0*
by *(metis ‹length vs < length io1› append.right-neutral length-0-conv length-zip less-asym min.idem neq0-conv rep-ios-def rep-ios-split tr2X-def tr2x-unique)*

have *tr2X-alt-def : tr2X′ = take (length io2X) tr*
by *(metis (no-types) assms(10) assms(9) obs-PM observable-path-prefix qv-simp rep-ios-split tr2X-def tr2x-unique)*
moreover have *io2X = take (length io2X) xs*
using *rep-ios-split* **by** *(metis append-eq-conv-conj prefixE)*
ultimately have *(io2X || tr2X′) = take (length io2X) (xs || tr)*
by *(metis take-zip)*
moreover have *length (xs || tr) ≥ length (io2X || tr2X′)*
using *calculation* **by** *auto*
ultimately have *rep-idx-2 : (states (xs || tr) ?qv) ! ((length io2X) − 1) = (s,rep)*
by *(metis (no-types, lifting) One-nat-def Suc-less-eq Suc-pred rep-tgt-2 length-2 less-Suc-eq-le map-snd-zip scan-length scan-nth states-alt-def tr2X-def tr2x-unique)*

— thus the distinctness assumption is violated

have *length io1X ≠ length io2X*
by *(metis ‹io1X = take (length io1X) xs› ‹io2X = take (length io2X) xs› less-irrefl rep-ios-def rep-ios-split)*
moreover have *(states (xs || tr) ?qv) ! ((length io1X) − 1)*
= (states (xs || tr) ?qv) ! ((length io2X) − 1)
using *rep-idx-1 rep-idx-2* **by** *simp*
ultimately have *¬ (distinct (states (xs || tr) ?qv))*
by *(metis Suc-less-eq ‹io1X = take (length io1X) xs›*
‹io1X || tr1X′ = take (length io1X) (xs || tr)› ‹io2X = take (length io2X) xs›
‹io2X || tr2X′ = take (length io2X) (xs || tr)›
‹length (io1X || tr1X′) ≤ length (xs || tr)› ‹length (io2X || tr2X′) ≤ length (xs || tr)›
assms(10) diff-Suc-1 distinct-conv-nth gr0-conv-Suc le-imp-less-Suc length-1 length-2 length-take map-snd-zip scan-length states-alt-def)
then show *False*
by *(metis assms(11) states-alt-def)*
qed

ultimately show *?thesis*
by *linarith*
qed

lemma *R-state-component-2 :*
assumes *io ∈ (R M2 s vs xs)*
and *observable M2*
shows *io-targets M2 (initial M2) io = {s}*

```
proof -
  have s ∈ io-targets M2 (initial M2) io
    using assms(1) by auto
  moreover have io ∈ language-state M2 (initial M2)
    using calculation by auto
  ultimately show io-targets M2 (initial M2) io = {s}
    using assms(2) io-targets-observable-singleton-ex by (metis singletonD)
qed


lemma R-union-card-is-suffix-length :
  assumes OFSM M2
  and     io@xs ∈ L M2
shows sum (λ q . card (R M2 q io xs)) (nodes M2) = length xs
using assms proof (induction xs rule: rev-induct)
  case Nil
  show ?case
    by (simp add: sum.neutral)
next
  case (snoc x xs)

  have finite (nodes M2)
    using assms by auto

  have R-update : ⋀ q . R M2 q io (xs@[x]) = (if (q ∈ io-targets M2 (initial M2) (io @ xs @ [x]))
                                              then insert (io@xs@[x]) (R M2 q io xs)
                                              else R M2 q io xs)
    by auto

  obtain q where io-targets M2 (initial M2) (io @ xs @ [x]) = {q}
    by (meson assms(1) io-targets-observable-singleton-ex snoc.prems(2))

  then have R M2 q io (xs@[x]) = insert (io@xs@[x]) (R M2 q io xs)
    using R-update by auto
  moreover have (io@xs@[x]) ∉ (R M2 q io xs)
    by auto
  ultimately have card (R M2 q io (xs@[x])) = Suc (card (R M2 q io xs))
    by (metis card-insert-disjoint finite-R)

  have q ∈ nodes M2
    by (metis (full-types) FSM.nodes.initial ‹io-targets M2 (initial M2) (io@xs @ [x]) = {q}›
      insertI1 io-targets-nodes)

  have ∀ q' . q' ≠ q ⟶ R M2 q' io (xs@[x]) = R M2 q' io xs
    using ‹io-targets M2 (initial M2) (io@xs @ [x]) = {q}› R-update
    by auto
  then have ∀ q' . q' ≠ q ⟶ card (R M2 q' io (xs@[x])) = card (R M2 q' io xs)
    by auto

  then have (∑ q∈(nodes M2 - {q}). card (R M2 q io (xs@[x])))
              = (∑ q∈(nodes M2 - {q}). card (R M2 q io xs))
    by auto
  moreover have (∑ q∈nodes M2. card (R M2 q io (xs@[x])))
                  = (∑ q∈(nodes M2 - {q}). card (R M2 q io (xs@[x]))) + (card (R M2 q io (xs@[x])))
                (∑ q∈nodes M2. card (R M2 q io xs))
                  = (∑ q∈(nodes M2 - {q}). card (R M2 q io xs)) + (card (R M2 q io xs))
  proof -
    have ∀ C c f. (infinite C ∨ (c::'c) ∉ C) ∨ sum f C = (f c::nat) + sum f (C - {c})
      by (meson sum.remove)
    then show (∑ q∈nodes M2. card (R M2 q io (xs@[x])))
                = (∑ q∈(nodes M2 - {q}). card (R M2 q io (xs@[x]))) + (card (R M2 q io (xs@[x])))
              (∑ q∈nodes M2. card (R M2 q io xs))
                = (∑ q∈(nodes M2 - {q}). card (R M2 q io xs)) + (card (R M2 q io xs))
      using ‹finite (nodes M2)› ‹q ∈ nodes M2› by presburger+
  qed
  ultimately have (∑ q∈nodes M2. card (R M2 q io (xs@[x])))
```

```
                 = Suc (∑ q∈nodes M2. card (R M2 q io xs))
    using ‹card (R M2 q io (xs@[x])) = Suc (card (R M2 q io xs))›
    by presburger

  have (∑ q∈nodes M2. card (R M2 q io xs)) = length xs
    using snoc.IH snoc.prems language-state-prefix[of io@xs [x] M2 initial M2]
  proof −
    show ?thesis
      by (metis (no-types) ‹(io @ xs) @ [x] ∈ L M2 ⟹ io @ xs ∈ L M2›
          ‹OFSM M2› ‹io @ xs @ [x] ∈ L M2› append.assoc snoc.IH)
  qed

  show ?case
  proof −
    show ?thesis
      by (metis (no-types)
          ‹(∑ q∈nodes M2. card (R M2 q io (xs @ [x]))) = Suc (∑ q∈nodes M2. card (R M2 q io xs))›
          ‹(∑ q∈nodes M2. card (R M2 q io xs)) = length xs› length-append-singleton)
  qed
qed

lemma R-state-repetition-via-long-sequence :
  assumes OFSM M
  and     card (nodes M) ≤ m
  and     Suc (m * m) ≤ length xs
  and     vs@xs ∈ L M
shows ∃ q ∈ nodes M . card (R M q vs xs) > m
proof (rule ccontr)
  assume ¬ (∃ q∈nodes M. m < card (R M q vs xs))
  then have ∀ q ∈ nodes M . card (R M q vs xs) ≤ m
    by auto
  then have sum (λ q . card (R M q vs xs)) (nodes M) ≤ sum (λ q . m) (nodes M)
    by (meson sum-mono)
  moreover have sum (λ q . m) (nodes M) ≤ m * m
    using assms(2) by auto
  ultimately have sum (λ q . card (R M q vs xs)) (nodes M) ≤ m * m
    by presburger

  moreover have Suc (m*m) ≤ sum (λ q . card (R M q vs xs)) (nodes M)
    using R-union-card-is-suffix-length[OF assms(1), of vs xs] assms(4,3) by auto
  ultimately show False by simp
qed

lemma R-state-repetition-distribution :
  assumes OFSM M
  and     Suc (card (nodes M) * m) ≤ length xs
  and     vs@xs ∈ L M
shows ∃ q ∈ nodes M . card (R M q vs xs) > m
proof (rule ccontr)
  assume ¬ (∃ q∈nodes M. m < card (R M q vs xs))
  then have ∀ q ∈ nodes M . card (R M q vs xs) ≤ m
    by auto
  then have sum (λ q . card (R M q vs xs)) (nodes M) ≤ sum (λ q . m) (nodes M)
    by (meson sum-mono)
  moreover have sum (λ q . m) (nodes M) ≤ card (nodes M) * m
    using assms(2) by auto
  ultimately have sum (λ q . card (R M q vs xs)) (nodes M) ≤ card (nodes M) * m
    by presburger

  moreover have Suc (card (nodes M)*m) ≤ sum (λ q . card (R M q vs xs)) (nodes M)
    using R-union-card-is-suffix-length[OF assms(1), of vs xs] assms(3,2) by auto
  ultimately show False
    by simp
qed
```

4.4 Function RP

Function `RP` extends function `MR` by adding all elements from a set of IO-sequences that also reach the given state.

fun *RP* :: $('in, 'out, 'state)$ *FSM* $\Rightarrow$ *'state* $\Rightarrow$ $('in \times 'out)$ *list*
$\Rightarrow$ $('in \times 'out)$ *list* $\Rightarrow$ $('in \times 'out)$ *list set*
$\Rightarrow$ $('in \times 'out)$ *list set*
where
RP M s vs xs $V'' =$ *R M s vs xs*
$\cup$ $\{vs' \in V''$. *io-targets M* (*initial M*) $vs' = \{s\}\}$

lemma *RP-from-R*:
assumes *is-det-state-cover M2 V*
and $V'' \in$ *Perm V M1*
shows *RP M2 s vs xs* $V'' =$ *R M2 s vs xs*
$\vee$ ($\exists$ $vs' \in V''$. $vs' \notin$ *R M2 s vs xs* $\wedge$ *RP M2 s vs xs* $V'' =$ *insert* vs' (*R M2 s vs xs*))
proof (*rule ccontr*)
assume *assm* : $\neg$ (*RP M2 s vs xs* $V'' =$ *R M2 s vs xs* $\vee$
($\exists$ $vs' \in V''$. $vs' \notin$ *R M2 s vs xs* $\wedge$ *RP M2 s vs xs* $V'' =$ *insert* vs' (*R M2 s vs xs*)))

moreover have *R M2 s vs xs* $\subseteq$ *RP M2 s vs xs* V''
by *simp*
moreover have *RP M2 s vs xs* $V'' \subseteq$ *R M2 s vs xs* $\cup$ V''
by *auto*
ultimately obtain *vs1 vs2* **where** *vs-def* :
vs1 $\neq$ *vs2* $\wedge$ *vs1* $\in V''$ $\wedge$ *vs2* $\in V''$
$\wedge$ *vs1* $\notin$ *R M2 s vs xs* $\wedge$ *vs2* $\notin$ *R M2 s vs xs*
$\wedge$ *vs1* $\in$ *RP M2 s vs xs* V'' $\wedge$ *vs2* $\in$ *RP M2 s vs xs* V''
by *blast*

then have *io-targets M2* (*initial M2*) *vs1* $= \{s\}$ $\wedge$ *io-targets M2* (*initial M2*) *vs2* $= \{s\}$
by (*metis* (*mono-tags*, *lifting*) *RP.simps Un-iff mem-Collect-eq*)
then have *io-targets M2* (*initial M2*) *vs1* $=$ *io-targets M2* (*initial M2*) *vs2*
by *simp*

obtain *f* **where** *f-def* : *is-det-state-cover-ass M2 f* $\wedge$ *V* $=$ *f* ‘ *d-reachable M2* (*initial M2*)
using *assms* **by** *auto*
moreover have *V* $=$ *image f* (*d-reachable M2* (*initial M2*))
using *f-def* **by** *blast*
moreover have *map fst vs1* $\in$ *V* $\wedge$ *map fst vs2* $\in$ *V*
using *assms*(*2*) *perm-inputs vs-def* **by** *blast*
ultimately obtain *r1 r2* **where** *r-def* :
f r1 $=$ *map fst vs1* $\wedge$ *r1* $\in$ *d-reachable M2* (*initial M2*)
f r2 $=$ *map fst vs2* $\wedge$ *r2* $\in$ *d-reachable M2* (*initial M2*)
by *force*
then have *d-reaches M2* (*initial M2*) (*map fst vs1*) *r1*
d-reaches M2 (*initial M2*) (*map fst vs2*) *r2*
by (*metis f-def is-det-state-cover-ass.elims*(*2*))+

then have *io-targets M2* (*initial M2*) *vs1* $\subseteq \{r1\}$
using *d-reaches-io-target*[*of M2 initial M2 map fst vs1 r1 map snd vs1*] **by** *simp*
moreover have *io-targets M2* (*initial M2*) *vs2* $\subseteq \{r2\}$
using *d-reaches-io-target*[*of M2 initial M2 map fst vs2 r2 map snd vs2*]
‹*d-reaches M2* (*initial M2*) (*map fst vs2*) *r2*› **by** *auto*
ultimately have *r1* $=$ *r2*
using ‹*io-targets M2* (*initial M2*) *vs1* $= \{s\}$ $\wedge$ *io-targets M2* (*initial M2*) *vs2* $= \{s\}$› **by** *auto*

have *map fst vs1* $\neq$ *map fst vs2*
using *assms*(*2*) *perm-inputs-diff vs-def* **by** *blast*
then have *r1* $\neq$ *r2*
using *r-def*(*1*) *r-def*(*2*) **by** *force*

then show *False*
using ‹*r1* $=$ *r2*› **by** *auto*
qed

```
lemma finite-RP :
  assumes is-det-state-cover M2 V
  and     V'' ∈ Perm V M1
shows finite (RP M2 s vs xs V'')
  using assms RP-from-R finite-R by (metis finite-insert)
```

```
lemma RP-count :
  assumes (vs @ xs) ∈ L M1 ∩ L M2
  and observable M1
  and observable M2
  and well-formed M1
  and well-formed M2
  and s ∈ nodes M2
  and productF M2 M1 FAIL PM
  and io-targets PM (initial PM) vs = {(q2,q1)}
  and path PM (xs || tr) (q2,q1)
  and length xs = length tr
  and distinct (states (xs || tr) (q2,q1))
  and is-det-state-cover M2 V
  and V'' ∈ Perm V M1
  and ∀ s' ∈ set (states (xs || map fst tr) q2) . ¬ (∃ v ∈ V . d-reaches M2 (initial M2) v s')
shows card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) = card (RP M2 s vs xs V'')
  — each sequence in the set calculated by RP reaches a different state in M1
proof −
```

— Proof sketch: - RP calculates either the same set as R or the set of R and an additional element - in the first case, the result for R applies - in the second case, the additional element is not contained in the set calcualted by R due to the assumption that no state reached by a non-empty prefix of xs after vs is also reached by some sequence in V (see the last two assumptions)

```
  have RP-cases : RP M2 s vs xs V'' = R M2 s vs xs
                  ∨ (∃ vs' ∈ V'' . vs' ∉ R M2 s vs xs
                                   ∧ RP M2 s vs xs V'' = insert vs' (R M2 s vs xs))
    using RP-from-R assms by metis
  show ?thesis
  proof (cases RP M2 s vs xs V'' = R M2 s vs xs)
    case True
    then show ?thesis using R-count assms by metis
  next
    case False
    then obtain vs' where vs'-def : vs' ∈ V''
                                   ∧ vs' ∉ R M2 s vs xs
                                   ∧ RP M2 s vs xs V'' = insert vs' (R M2 s vs xs)
      using RP-cases by auto

    have obs-PM : observable PM
      using observable-productF assms(2) assms(3) assms(7) by blast

    have state-component-2 : ∀ io ∈ (R M2 s vs xs) . io-targets M2 (initial M2) io = {s}
    proof
      fix io assume io ∈ R M2 s vs xs
      then have s ∈ io-targets M2 (initial M2) io
        by auto
      moreover have io ∈ language-state M2 (initial M2)
        using calculation by auto
      ultimately show io-targets M2 (initial M2) io = {s}
        using assms(3) io-targets-observable-singleton-ex by (metis singletonD)
    qed

    have vs' ∈ L M1
      using assms(13) perm-language vs'-def by blast
    then obtain s' where s'-def : io-targets M1 (initial M1) vs' = {s'}
```

by (*meson assms*(*2*) *io-targets-observable-singleton-ob*)

moreover have *s′* ∉ ⋃ (*image* (*io-targets M1* (*initial M1*)) (*R M2 s vs xs*))
proof (*rule ccontr*)
assume ¬ *s′* ∉ ⋃(*io-targets M1* (*initial M1*) ‘ *R M2 s vs xs*)
then obtain *xs′* **where** *xs′-def* : *vs* @ *xs′* ∈ *R M2 s vs xs* ∧ *s′* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs′*)
proof −
assume *a1*: ⋀*xs′*. *vs* @ *xs′* ∈ *R M2 s vs xs* ∧ *s′* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs′*)
⟹ *thesis*
obtain *pps* :: (*′a* × *′b*) *list set* ⇒ ((*′a* × *′b*) *list* ⇒ *′c set*) ⇒ *′c* ⇒ (*′a* × *′b*) *list*
where
∀ *x0 x1 x2*. (∃ *v3*. *v3* ∈ *x0* ∧ *x2* ∈ *x1 v3*) = (*pps x0 x1 x2* ∈ *x0* ∧ *x2* ∈ *x1* (*pps x0 x1 x2*))
by *moura*
then have *f2*: *pps* (*R M2 s vs xs*) (*io-targets M1* (*initial M1*)) *s′* ∈ *R M2 s vs xs*
∧ *s′* ∈ *io-targets M1* (*initial M1*) (*pps* (*R M2 s vs xs*)
(*io-targets M1* (*initial M1*)) *s′*)
using ‹¬ *s′* ∉ ⋃(*io-targets M1* (*initial M1*) ‘ *R M2 s vs xs*)› **by** *blast*
then have ∃ *ps*. *pps* (*R M2 s vs xs*) (*io-targets M1* (*initial M1*)) *s′* = *vs* @ *ps*
∧ *ps* ≠ [] ∧ *prefix ps xs* ∧ *s* ∈ *io-targets M2* (*initial M2*) (*vs* @ *ps*)
by *simp*
then show *?thesis*
using *f2 a1* **by** (*metis* (*no-types*))
qed
then obtain *tr′* **where** *tr′-def* : *path M2* (*vs* @ *xs′* || *tr′*) (*initial M2*)
∧ *length tr′* = *length* (*vs* @ *xs′*)
by *auto*

then obtain *trV′ trX′* **where** *tr′-split* : *trV′* = *take* (*length vs*) *tr′*
trX′ = *drop* (*length vs*) *tr′*
tr′ = *trV′* @ *trX′*
by *fastforce*
then have *path M2* (*vs* || *trV′*) (*initial M2*) ∧ *length trV′* = *length vs*
by (*metis* (*no-types*) *FSM.path-append-elim* ‹*trV′* = *take* (*length vs*) *tr′*›
append-eq-conv-conj length-take tr′-def zip-append1)

have *initial PM* = (*initial M2*, *initial M1*)
using *assms*(*7*) **by** *simp*
moreover have *vs* ∈ *L M2 vs* ∈ *L M1*
using *assms*(*1*) *language-state-prefix* **by** *auto*
ultimately have *io-targets M1* (*initial M1*) *vs* = {*q1*}
io-targets M2 (*initial M2*) *vs* = {*q2*}
using *productF-path-io-targets*[*of M2 M1 FAIL PM initial M2 initial M1 vs q2 q1*]
by (*metis FSM.nodes.initial assms*(*7*) *assms*(*8*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*)
io-targets-observable-singleton-ex singletonD)+

then have *target* (*vs* || *trV′*) (*initial M2*) = *q2*
using ‹*path M2* (*vs* || *trV′*) (*initial M2*) ∧ *length trV′* = *length vs*› *io-target-target*
by *metis*
then have *path-xs′* : *path M2* (*xs′* || *trX′*) *q2* ∧ *length trX′* = *length xs′*
by (*metis* (*no-types*) *FSM.path-append-elim*
‹*path M2* (*vs* || *trV′*) (*initial M2*) ∧ *length trV′* = *length vs*›
‹*target* (*vs* || *trV′*) (*initial M2*) = *q2*› *append-eq-conv-conj length-drop tr′-def*
tr′-split(*1*) *tr′-split*(*2*) *zip-append2*)

have *io-targets M2* (*initial M2*) (*vs* @ *xs′*) = {*s*}
using *state-component-2 xs′-def* **by** *blast*
then have *io-targets M2 q2 xs′* = {*s*}
by (*meson assms*(*3*) *observable-io-targets-split* ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}›)
then have *target-xs′* : *target* (*xs′* || *trX′*) *q2* = *s*
using *io-target-target path-xs′* **by** *metis*
moreover have *length xs′* > *0*
using *xs′-def* **by** *auto*

ultimately have *last* (*states* (*xs'* || *trX'*) *q2*) = *s*
  using *path-xs' target-in-states* **by** *metis*
moreover have *length* (*states* (*xs'* || *trX'*) *q2*) > *0*
  using ‹*0* < *length xs'*› *path-xs'* **by** *auto*
ultimately have *states-xs'* : *s* ∈ *set* (*states* (*xs'* || *trX'*) *q2*)
  using *last-in-set* **by** *blast*

have *vs* @ *xs* ∈ *L M2*
  using *assms* **by** *simp*
then obtain *q'* **where** *io-targets M2* (*initial M2*) (*vs@xs*) = {*q'*}
  using *io-targets-observable-singleton-ob*[*of M2 vs@xs initial M2*] *assms*(*3*) **by** *auto*
then have *xs* ∈ *language-state M2 q2*
  using *assms*(*3*) ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}›
    observable-io-targets-split[*of M2 initial M2 vs xs q' q2*]
  by *auto*

moreover have *path PM* (*xs* || *map fst tr* || *map snd tr*) (*q2*,*q1*)
    ∧ *length xs* = *length* (*map fst tr*)
  using *assms*(*7*) *assms*(*9*) *assms*(*10*) *productF-path-unzip* **by** *simp*
moreover have *xs* ∈ *language-state PM* (*q2*,*q1*)
  using *assms*(*9*) *assms*(*10*) **by** *auto*
moreover have *q2* ∈ *nodes M2*
  using ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}› *io-targets-nodes*
  by (*metis FSM.nodes.initial insertI1*)
moreover have *q1* ∈ *nodes M1*
  using ‹*io-targets M1* (*initial M1*) *vs* = {*q1*}› *io-targets-nodes*
  by (*metis FSM.nodes.initial insertI1*)
ultimately have *path-xs* : *path M2* (*xs* || *map fst tr*) *q2*
  using *productF-path-reverse-ob-2*(*1*)[*of xs map fst tr map snd tr M2 M1 FAIL PM q2 q1*]
    assms(*2*,*3*,*4*,*5*,*7*)
  by *simp*

moreover have *prefix xs' xs*
  using *xs'-def* **by** *auto*
ultimately have *trX'* = *take* (*length xs'*) (*map fst tr*)
  using ‹*path PM* (*xs* || *map fst tr* || *map snd tr*) (*q2*, *q1*) ∧ *length xs* = *length* (*map fst tr*)›
    assms(*3*) *path-xs'*
  by (*metis observable-path-prefix*)

then have *states-xs* : *s* ∈ *set* (*states* (*xs* || *map fst tr*) *q2*)
  by (*metis assms*(*10*) *in-set-takeD length-map map-snd-zip path-xs' states-alt-def states-xs'*)

have *d-reaches M2* (*initial M2*) (*map fst vs'*) *s*
proof −
  obtain *fV* **where** *fV-def* : *is-det-state-cover-ass M2 fV*
      ∧ *V* = *fV* ‘ *d-reachable M2* (*initial M2*)
    using *assms*(*12*) **by** *auto*
  moreover have *V* = *image fV* (*d-reachable M2* (*initial M2*))
    using *fV-def* **by** *blast*
  moreover have *map fst vs'* ∈ *V*
    using *perm-inputs vs'-def assms*(*13*) **by** *metis*
  ultimately obtain *qv* **where** *qv-def* : *fV qv* = *map fst vs'* ∧ *qv*∈ *d-reachable M2* (*initial M2*)
    by *force*
  then have *d-reaches M2* (*initial M2*) (*map fst vs'*) *qv*
    by (*metis fV-def is-det-state-cover-ass.elims*(*2*))
  then have *io-targets M2* (*initial M2*) *vs'* ⊆ {*qv*}
    using *d-reaches-io-target*[*of M2 initial M2 map fst vs' qv map snd vs'*] **by** *simp*
  moreover have *io-targets M2* (*initial M2*) *vs'* = {*s*}
    using *vs'-def* **by** (*metis* (*mono-tags*, *lifting*) *RP.simps Un-iff insertI1 mem-Collect-eq*)
  ultimately have *qv* = *s*
    by *simp*
  then show *?thesis*

```
        using ‹d-reaches M2 (initial M2) (map fst vs') qv› by blast
      qed

      then show False  by (meson assms(14) assms(13) perm-inputs states-xs vs'-def)
    qed

    moreover have ⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs)))
                  = insert s' (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs)))
      using s'-def by simp

    moreover have finite (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs)))
    proof
      show finite (R M2 s vs xs)
        using finite-R by simp
      show ⋀a. a ∈ R M2 s vs xs ⟹ finite (io-targets M1 (initial M1) a)
      proof −
        fix a assume a ∈ R M2 s vs xs
        then have prefix a (vs@xs)
          by auto
        then have a ∈ L M1
          using language-state-prefix by (metis IntD1 assms(1) prefix-def)
        then obtain p where io-targets M1 (initial M1) a = {p}
          using assms(2) io-targets-observable-singleton-ob by metis
        then show finite (io-targets M1 (initial M1) a)
          by simp
      qed
    qed

    ultimately have card (⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs))))
                    = Suc (card (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs))))
      by (metis (no-types) card-insert-disjoint)


    moreover have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
                  = card (⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs))))
      using vs'-def by simp

    ultimately have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
                     = Suc (card (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs))))
      by linarith

    then have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
              = Suc (card (R M2 s vs xs))
      using R-count[of vs xs M1 M2 s FAIL PM q2 q1 tr] assms(1,10,11,2−9) by linarith

    moreover have card (RP M2 s vs xs V'') = Suc (card (R M2 s vs xs))
      using vs'-def by (metis card-insert-if finite-R)

    ultimately show ?thesis
      by linarith
  qed
qed


lemma RP-state-component-2 :
  assumes io ∈ (RP M2 s vs xs V'')
  and     observable M2
shows io-targets M2 (initial M2) io = {s}
  by (metis (mono-tags, lifting) RP.simps R-state-component-2 Un-iff assms mem-Collect-eq)


lemma RP-io-targets-split :
  assumes (vs @ xs) ∈ L M1 ∩ L M2
  and observable M1
  and observable M2
  and well-formed M1
```

and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *is-det-state-cover M2 V*
and *V'' ∈ Perm V M1*
and *io ∈ RP M2 s vs xs V''*
shows *io-targets PM* (*initial PM*) *io*
= *io-targets M2* (*initial M2*) *io* × *io-targets M1* (*initial M1*) *io*
proof −
have *RP-cases* : *RP M2 s vs xs V''* = *R M2 s vs xs*
∨ (∃ *vs'* ∈ *V''* . *vs'* ∉ *R M2 s vs xs*
∧ *RP M2 s vs xs V''* = *insert vs'* (*R M2 s vs xs*))
using *RP-from-R assms* **by** *metis*
show *?thesis*
proof (*cases io* ∈ *R M2 s vs xs*)
case *True*
then have *io-prefix* : *prefix io* (*vs* @ *xs*)
by *auto*
then have *io-lang-subs* : *io* ∈ *L M1* ∧ *io* ∈ *L M2*
using *assms*(*1*) **unfolding** *prefix-def* **by** (*metis IntE language-state language-state-split*)
then have *io-lang-inter* : *io* ∈ *L M1* ∩ *L M2*
by *simp*
then have *io-lang-pm* : *io* ∈ *L PM*
using *productF-language assms* **by** *blast*
moreover obtain *p2 p1* **where** (*p2*,*p1*) ∈ *io-targets PM* (*initial PM*) *io*
by (*metis assms*(*2*) *assms*(*3*) *assms*(*6*) *calculation insert-absorb insert-ident insert-not-empty*
io-targets-observable-singleton-ob observable-productF singleton-insert-inj-eq subrelI)
ultimately have *targets-pm* : *io-targets PM* (*initial PM*) *io* = {(*p2*,*p1*)}
using *assms io-targets-observable-singleton-ex singletonD*
by (*metis observable-productF*)
then obtain *trP* **where** *trP-def* : *target* (*io* || *trP*) (*initial PM*) = (*p2*,*p1*)
∧ *path PM* (*io* || *trP*) (*initial PM*) ∧ *length io* = *length trP*
proof −
assume *a1*: ⋀*trP*. *target* (*io* || *trP*) (*initial PM*) = (*p2*, *p1*)
∧ *path PM* (*io* || *trP*) (*initial PM*) ∧ *length io* = *length trP* ⟹ *thesis*
have ∃*ps*. *target* (*io* || *ps*) (*initial PM*) = (*p2*, *p1*) ∧ *path PM* (*io* || *ps*) (*initial PM*)
∧ *length io* = *length ps*
using ‹(*p2*, *p1*) ∈ *io-targets PM* (*initial PM*) *io*› **by** *auto*
then show *?thesis*
using *a1* **by** *blast*
qed
then have *trP-unique* : {*tr* . *path PM* (*io* || *tr*) (*initial PM*) ∧ *length io* = *length tr*} = {*trP*}
using *observable-productF observable-path-unique-ex*[*of PM io initial PM*]
io-lang-pm assms(*2*) *assms*(*3*) *assms*(*7*)
proof −
obtain *pps* :: ('*d* × '*c*) *list* **where**
f1: {*ps*. *path PM* (*io* || *ps*) (*initial PM*) ∧ *length io* = *length ps*} = {*pps*}
∨ ¬ *observable PM*
by (*metis* (*no-types*) ‹⋀*thesis*. ⟦*observable PM*; *io* ∈ *L PM*; ⋀*tr*.
{*t*. *path PM* (*io* || *t*) (*initial PM*) ∧ *length io* = *length t*} = {*tr*}
⟹ *thesis*⟧ ⟹ *thesis*›
io-lang-pm)
have *f2*: *observable PM*
by (*meson* ‹*observable M1*› ‹*observable M2*› ‹*productF M2 M1 FAIL PM*› *observable-productF*)
then have *trP* ∈ {*pps*}
using *f1 trP-def* **by** *blast*
then show *?thesis*
using *f2 f1* **by** *force*
qed

obtain *trIO2* **where** *trIO2-def* : {*tr* . *path M2* (*io* || *tr*) (*initial M2*) ∧ *length io* = *length tr*}
= { *trIO2* }
using *observable-path-unique-ex*[*of M2 io initial M2*] *io-lang-subs assms*(*3*) **by** *blast*
obtain *trIO1* **where** *trIO1-def* : {*tr* . *path M1* (*io* || *tr*) (*initial M1*) ∧ *length io* = *length tr*}
= { *trIO1* }
using *observable-path-unique-ex*[*of M1 io initial M1*] *io-lang-subs assms*(*2*) **by** *blast*

```
have path PM (io || trIO2 || trIO1) (initial M2, initial M1)
      ∧ length io = length trIO2 ∧ length trIO2 = length trIO1
proof −
  have f1: path M2 (io || trIO2) (initial M2) ∧ length io = length trIO2
    using trIO2-def by auto
  have f2: path M1 (io || trIO1) (initial M1) ∧ length io = length trIO1
    using trIO1-def by auto
  then have length trIO2 = length trIO1
    using f1 by presburger
  then show ?thesis
    using f2 f1 assms(4) assms(5) assms(6) by blast
qed
then have trP-split : path PM (io || trIO2 || trIO1) (initial PM)
                    ∧ length io = length trIO2 ∧ length trIO2 = length trIO1
  using assms(6) by auto
then have trP-zip : trIO2 || trIO1 = trP
  using trP-def trP-unique length-zip by fastforce

have target (io || trIO2) (initial M2) = p2
       ∧ path M2 (io || trIO2) (initial M2)
       ∧ length io = length trIO2
  using trP-zip trP-split assms(6) trP-def trIO2-def by auto
then have p2 ∈ io-targets M2 (initial M2) io
  by auto
then have targets-2 : io-targets M2 (initial M2) io = {p2}
  by (meson assms(3) observable-io-target-is-singleton)

have target (io || trIO1) (initial M1) = p1
      ∧ path M1 (io || trIO1) (initial M1)
      ∧ length io = length trIO1
  using trP-zip trP-split assms(6) trP-def trIO1-def by auto
then have p1 ∈ io-targets M1 (initial M1) io
  by auto
then have targets-1 : io-targets M1 (initial M1) io = {p1}
  by (metis io-lang-subs assms(2) io-targets-observable-singleton-ex singletonD)

have io-targets M2 (initial M2) io × io-targets M1 (initial M1) io = {(p2,p1)}
  using targets-2 targets-1 by simp
then show io-targets PM (initial PM) io
          = io-targets M2 (initial M2) io × io-targets M1 (initial M1) io
  using targets-pm by simp

next
case False
then have io ∉ R M2 s vs xs ∧ RP M2 s vs xs V'' = insert io (R M2 s vs xs)
  using RP-cases assms(9) by (metis insertE)

have io ∈ L M1 using assms(8) perm-language assms(9)
  using False by auto
then obtain s' where s'-def : io-targets M1 (initial M1) io = {s'}
  by (meson assms(2) io-targets-observable-singleton-ob)
then obtain tr1 where tr1-def : target (io || tr1) (initial M1) = s'
                              ∧ path M1 (io || tr1) (initial M1) ∧ length tr1 = length io
  by (metis io-targets-elim singletonI)

have io-targets M2 (initial M2) io = {s}
  using assms(9) assms(3) RP-state-component-2 by simp
then obtain tr2 where tr2-def : target (io || tr2) (initial M2) = s
                              ∧ path M2 (io || tr2) (initial M2) ∧ length tr2 = length io
  by (metis io-targets-elim singletonI)
then have paths : path M2 (io || tr2) (initial M2) ∧ path M1 (io || tr1) (initial M1)
  using tr1-def by simp


have length io = length tr2
```

using *tr2-def* **by** *simp*
moreover have *length tr2 = length tr1*
using *tr1-def tr2-def* **by** *simp*
ultimately have *path PM (io || tr2 || tr1) (initial M2, initial M1)*
using *assms(6) assms(5) assms(4) paths*
productF-path-forward[of io tr2 tr1 M2 M1 FAIL PM initial M2 initial M1]
by *blast*

moreover have *target (io || tr2 || tr1) (initial M2, initial M1) = (s,s′)*
by *(simp add: tr1-def tr2-def)*
moreover have *length (tr2 || tr2) = length io*
using *tr1-def tr2-def* **by** *simp*
moreover have *(initial M2, initial M1) = initial PM*
using *assms(6)* **by** *simp*
ultimately have *(s,s′) ∈ io-targets PM (initial PM) io*
by *(metis io-target-from-path length-zip tr1-def tr2-def)*
moreover have *observable PM*
using *assms(2) assms(3) assms(6) observable-productF* **by** *blast*
then have *io-targets PM (initial PM) io = {(s,s′)}*
by *(meson calculation observable-io-target-is-singleton)*

then show *?thesis*
using ‹*io-targets M2 (initial M2) io = {s}*› ‹*io-targets M1 (initial M1) io = {s′}*›
by *simp*
qed
qed

lemma *RP-io-targets-finite-M1* :
assumes *(vs @ xs) ∈ L M1 ∩ L M2*
and *observable M1*
and *is-det-state-cover M2 V*
and *V′′ ∈ Perm V M1*
shows *finite (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V′′)))*
proof
show *finite (RP M2 s vs xs V′′)* **using** *finite-RP assms(3) assms(4)* **by** *simp*
show *⋀a. a ∈ RP M2 s vs xs V′′ ⟹ finite (io-targets M1 (initial M1) a)*
proof −
fix *a* **assume** *a ∈ RP M2 s vs xs V′′*

have *RP-cases : RP M2 s vs xs V′′ = R M2 s vs xs*
∨ (∃ vs′ ∈ V′′ . vs′ ∉ R M2 s vs xs
∧ RP M2 s vs xs V′′ = insert vs′ (R M2 s vs xs))
using *RP-from-R assms* **by** *metis*
have *a ∈ L M1*
proof *(cases a ∈ R M2 s vs xs)*
case *True*
then have *prefix a (vs@xs)*
by *auto*
then show *a ∈ L M1*
using *language-state-prefix* **by** *(metis IntD1 assms(1) prefix-def)*
next
case *False*
then have *a ∈ V′′ ∧ RP M2 s vs xs V′′ = insert a (R M2 s vs xs)*
using *RP-cases* ‹*a ∈ RP M2 s vs xs V′′*› **by** *(metis insertE)*
then show *a ∈ L M1*
by *(meson assms(4) perm-language)*
qed
then obtain *p* **where** *io-targets M1 (initial M1) a = {p}*
using *assms(2) io-targets-observable-singleton-ob* **by** *metis*
then show *finite (io-targets M1 (initial M1) a)*
by *simp*
qed

qed

lemma *RP-io-targets-finite-PM* :
  assumes $(vs \ @ \ xs) \in L \ M1 \cap L \ M2$
  and *observable M1*
  and *observable M2*
  and *well-formed M1*
  and *well-formed M2*
  and *productF M2 M1 FAIL PM*
  and *is-det-state-cover M2 V*
  and $V'' \in Perm \ V \ M1$
shows *finite* ($\bigcup$ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s vs xs* V'')))
proof −
  have $\forall$ *io* $\in$ *RP M2 s vs xs* V'' . *io-targets PM* (*initial PM*) *io*
    = $\{s\} \times$ *io-targets M1* (*initial M1*) *io*
  proof
    fix *io* **assume** *io* $\in$ *RP M2 s vs xs* V''
    then have *io-targets PM* (*initial PM*) *io*
      = *io-targets M2* (*initial M2*) *io* $\times$ *io-targets M1* (*initial M1*) *io*
      using *assms RP-io-targets-split*[*of vs xs M1 M2 FAIL PM V* V'' *io s*] **by** *simp*
    moreover have *io-targets M2* (*initial M2*) *io* = $\{s\}$
      using ‹*io* $\in$ *RP M2 s vs xs* V''› *assms*(*3*) *RP-state-component-2*[*of io M2 s vs xs* V'']
      by *blast*
    ultimately show *io-targets PM* (*initial PM*) *io* = $\{s\} \times$ *io-targets M1* (*initial M1*) *io*
      by *auto*
  qed
  then have $\bigcup$ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s vs xs* V''))
      = $\bigcup$ (*image* (λ *io* . $\{s\} \times$ *io-targets M1* (*initial M1*) *io*) (*RP M2 s vs xs* V''))
    by *simp*
  moreover have $\bigcup$ (*image* (λ *io* . $\{s\} \times$ *io-targets M1* (*initial M1*) *io*) (*RP M2 s vs xs* V''))
      = $\{s\} \times \bigcup$ (*image* (λ *io* . *io-targets M1* (*initial M1*) *io*) (*RP M2 s vs xs* V''))
    by *blast*
  ultimately have $\bigcup$ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s vs xs* V''))
      = $\{s\} \times \bigcup$ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs* V''))
    by *auto*
  moreover have *finite* ($\{s\} \times \bigcup$ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs* V'')))
    using *assms*(*1*,*2*,*7*,*8*) *RP-io-targets-finite-M1*[*of vs xs M1 M2 V* V'' *s*] **by** *simp*
  ultimately show *?thesis*
    by *simp*
qed

lemma *RP-union-card-is-suffix-length* :
  assumes *OFSM M2*
  and *io*@*xs* $\in$ *L M2*
  and *is-det-state-cover M2 V*
  and $V'' \in Perm \ V \ M1$
shows $\bigwedge$ *q* . *card* (*R M2 q io xs*) $\leq$ *card* (*RP M2 q io xs* V'')
    sum (λ *q* . *card* (*RP M2 q io xs* V'')) (*nodes M2*) $\geq$ *length xs*
proof −
  have *sum* (λ *q* . *card* (*R M2 q io xs*)) (*nodes M2*) = *length xs*
    using *R-union-card-is-suffix-length*[*OF assms*(*1*,*2*)] **by** *assumption*
  show $\bigwedge$ *q* . *card* (*R M2 q io xs*) $\leq$ *card* (*RP M2 q io xs* V'')
    by (*metis RP-from-R assms*(*3*) *assms*(*4*) *card-insert-le eq-iff finite-R*)
  show *sum* (λ *q* . *card* (*RP M2 q io xs* V'')) (*nodes M2*) $\geq$ *length xs*
    by (*metis* (*no-types, lifting*) ‹($\sum$ *q*$\in$*nodes M2*. *card* (*R M2 q io xs*)) = *length xs*›
      ‹$\bigwedge$*q*. *card* (*R M2 q io xs*) $\leq$ *card* (*RP M2 q io xs* V'')› *sum-mono*)
qed

lemma *RP-state-repetition-distribution-productF* :
  assumes *OFSM M2*
  and *OFSM M1*
  and (*card* (*nodes M2*) $*$ *m*) $\leq$ *length xs*
  and *card* (*nodes M1*) $\leq$ *m*
  and *vs*@*xs* $\in$ *L M2* $\cap$ *L M1*

```
  and     is-det-state-cover M2 V
  and     V'' ∈ Perm V M1
shows ∃ q ∈ nodes M2 . card (RP M2 q vs xs V'') > m
proof −
  have finite (nodes M1)
       finite (nodes M2)
    using assms(1,2) by auto
  then have card(nodes M2 × nodes M1) = card (nodes M2) ∗ card (nodes M1)
    using card-cartesian-product by blast

  have nodes (product M2 M1) ⊆ nodes M2 × nodes M1
    using product-nodes by auto

  have card (nodes (product M2 M1)) ≤ card (nodes M2) ∗ card (nodes M1)
    by (metis (no-types) ‹card (nodes M2 × nodes M1) = |M2| ∗ |M1|› ‹finite (nodes M1)›
      ‹finite (nodes M2)› ‹nodes (product M2 M1) ⊆ nodes M2 × nodes M1›
      card-mono finite-cartesian-product)


  have (∀ q ∈ nodes M2 . card (R M2 q vs xs) = m) ∨ (∃ q ∈ nodes M2 . card (R M2 q vs xs) > m)
  proof (rule ccontr)
    assume ¬ ((∀ q∈nodes M2. card (R M2 q vs xs) = m) ∨ (∃ q∈nodes M2. m < card (R M2 q vs xs)))

    then have ∀ q ∈ nodes M2 . card (R M2 q vs xs) ≤ m
      by auto
    moreover obtain q' where q'∈nodes M2 card (R M2 q' vs xs) < m
      using ‹¬ ((∀ q∈nodes M2. card (R M2 q vs xs) = m) ∨ (∃ q∈nodes M2. m < card (R M2 q vs xs)))›
            nat-neq-iff
      by blast

    have sum (λ q . card (R M2 q vs xs)) (nodes M2)
          = sum (λ q . card (R M2 q vs xs)) (nodes M2 − {q'})
            + sum (λ q . card (R M2 q vs xs)) {q'}
      using ‹q'∈nodes M2›
      by (meson ‹finite (nodes M2)› empty-subsetI insert-subset sum.subset-diff)
    moreover have sum (λ q . card (R M2 q vs xs)) (nodes M2 − {q'})
                  ≤ sum (λ q . m) (nodes M2 − {q'})
      using ‹∀ q ∈ nodes M2 . card (R M2 q vs xs) ≤ m›
      by (meson sum-mono DiffD1)
    moreover have sum (λ q . card (R M2 q vs xs)) {q'} < m
      using ‹card (R M2 q' vs xs) < m› by auto
    ultimately have sum (λ q . card (R M2 q vs xs)) (nodes M2) < sum (λ q . m) (nodes M2)
    proof −
      have ∀ C c f. infinite C ∨ (c::'c) ∉ C ∨ sum f C = (f c::nat) + sum f (C − {c})
        by (meson sum.remove)
      then have (∑ c∈nodes M2. m) = m + (∑ c∈nodes M2 − {q'}. m)
        using ‹finite (nodes M2)› ‹q' ∈ nodes M2› by blast
      then show ?thesis
        using ‹(∑ q∈nodes M2 − {q'}. card (R M2 q vs xs)) ≤ (∑ q∈nodes M2 − {q'}. m)›
              ‹(∑ q∈nodes M2. card (R M2 q vs xs)) = (∑ q∈nodes M2 − {q'}. card (R M2 q vs xs))
                + (∑ q∈{q'}. card (R M2 q vs xs))›
              ‹(∑ q∈{q'}. card (R M2 q vs xs)) < m›
        by linarith
    qed


    moreover have sum (λ q . m) (nodes M2) ≤ card (nodes M2) ∗ m
      using assms(2) by auto
    ultimately have sum (λ q . card (R M2 q vs xs)) (nodes M2) < card (nodes M2) ∗ m
      by presburger

    moreover have Suc (card (nodes M2)∗m) ≤ sum (λ q . card (R M2 q vs xs)) (nodes M2)
      using R-union-card-is-suffix-length[OF assms(1), of vs xs] assms(5,3)
      by (metis Int-iff ‹vs @ xs ∈ L M2 ⟹ (∑ q∈nodes M2. card (R M2 q vs xs)) = length xs›
        ‹vs @ xs ∈ L M2 ∩ L M1› ‹|M2| ∗ m ≤ length xs› calculation less-eq-Suc-le not-less-eq-eq)
```

```
    ultimately show False by simp
  qed
  then show ?thesis
  proof
    let ?q = initial M2

    assume ∀ q∈nodes M2. card (R M2 q vs xs) = m
    then have card (R M2 ?q vs xs) = m
      by auto

    have [] ∈ V''
      by (meson assms(6) assms(7) perm-empty)
    then have [] ∈ RP M2 ?q vs xs V''
      by auto
    have [] ∉ R M2 ?q vs xs
      by auto
    have card (RP M2 ?q vs xs V'') ≥ card (R M2 ?q vs xs)
      using finite-R[of M2 ?q vs xs] finite-RP[OF assms(6,7),of ?q vs xs] unfolding RP.simps
      by (simp add: card-mono)

    have card (RP M2 ?q vs xs V'') > card (R M2 ?q vs xs)
    proof −
      have f1: ∀ n na. (¬ (n::nat) ≤ na ∨ n = na) ∨ n < na
        by (meson le-neq-trans)
      have RP M2 (initial M2) vs xs V'' ≠ R M2 (initial M2) vs xs
        using ‹[] ∈ RP M2 (initial M2) vs xs V''› ‹[] ∉ R M2 (initial M2) vs xs› by blast
      then show ?thesis
        using f1 by (metis (no-types) RP-from-R
                ‹card (R M2 (initial M2) vs xs) ≤ card (RP M2 (initial M2) vs xs V'')›
                assms(6) assms(7) card-insert-disjoint finite-R le-simps(2))
    qed

    then show ?thesis
      using ‹card (R M2 ?q vs xs) = m›
      by blast
  next
    assume ∃ q∈nodes M2. m < card (R M2 q vs xs)
    then obtain q where q∈nodes M2 m < card (R M2 q vs xs)
      by blast
    moreover have card (RP M2 q vs xs V'') ≥ card (R M2 q vs xs)
      using finite-R[of M2 q vs xs] finite-RP[OF assms(6,7),of q vs xs] unfolding RP.simps
      by (simp add: card-mono)
    ultimately have m < card (RP M2 q vs xs V'')
      by simp

    show ?thesis
      using ‹q ∈ nodes M2›  ‹m < card (RP M2 q vs xs V'')› by blast
  qed
qed
```

4.5 Conditions for the result of LB to be a valid lower bound

The following predicates describe the assumptions necessary to show that the value calculated by `LB` is a lower bound on the number of states of a given FSM.

```
fun Prereq :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM ⇒ ('in × 'out) list
          ⇒ ('in × 'out) list ⇒ 'in list set ⇒ 'state1 set ⇒ ('in, 'out) ATC set
          ⇒ ('in × 'out) list set ⇒ bool
  where
  Prereq M2 M1 vs xs T S Ω V'' = (
    (finite T)
    ∧ (vs @ xs) ∈ L M2 ∩ L M1
    ∧ S ⊆ nodes M2
    ∧ (∀  s1 ∈ S . ∀  s2 ∈ S . s1 ≠ s2
        ⟶ (∀  io1 ∈ RP M2 s1 vs xs V'' .
              ∀  io2 ∈ RP M2 s2 vs xs V'' .
```

B M1 io1 Ω ≠ B M1 io2 Ω)))

fun *Rep-Pre* :: (*'in*, *'out*, *'state1*) *FSM* ⇒ (*'in*, *'out*, *'state2*) *FSM* ⇒ (*'in* × *'out*) *list*
⇒ (*'in* × *'out*) *list* ⇒ *bool* **where**
Rep-Pre M2 M1 vs xs = (∃ *xs1 xs2* . *prefix xs1 xs2* ∧ *prefix xs2 xs* ∧ *xs1* ≠ *xs2*
∧ (∃ *s2* . *io-targets M2* (*initial M2*) (*vs* @ *xs1*) = {*s2*}
∧ *io-targets M2* (*initial M2*) (*vs* @ *xs2*) = {*s2*})
∧ (∃ *s1* . *io-targets M1* (*initial M1*) (*vs* @ *xs1*) = {*s1*}
∧ *io-targets M1* (*initial M1*) (*vs* @ *xs2*) = {*s1*}))

fun *Rep-Cov* :: (*'in*, *'out*, *'state1*) *FSM* ⇒ (*'in*, *'out*, *'state2*) *FSM* ⇒ (*'in* × *'out*) *list set*
⇒ (*'in* × *'out*) *list* ⇒ (*'in* × *'out*) *list* ⇒ *bool* **where**
Rep-Cov M2 M1 V'' vs xs = (∃ *xs' vs'* . *xs'* ≠ [] ∧ *prefix xs' xs* ∧ *vs'* ∈ *V''*
∧ (∃ *s2* . *io-targets M2* (*initial M2*) (*vs* @ *xs'*) = {*s2*}
∧ *io-targets M2* (*initial M2*) (*vs'*) = {*s2*})
∧ (∃ *s1* . *io-targets M1* (*initial M1*) (*vs* @ *xs'*) = {*s1*}
∧ *io-targets M1* (*initial M1*) (*vs'*) = {*s1*}))

lemma *distinctness-via-Rep-Pre* :
assumes ¬ *Rep-Pre M2 M1 vs xs*
and *productF M2 M1 FAIL PM*
and *observable M1*
and *observable M2*
and *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
and *path PM* (*xs* || *tr*) (*q2*,*q1*)
and *length xs* = *length tr*
and (*vs* @ *xs*) ∈ *L M1* ∩ *L M2*
and *well-formed M1*
and *well-formed M2*
shows *distinct* (*states* (*xs* || *tr*) (*q2*, *q1*))
proof (*rule ccontr*)
assume *assm* : ¬ *distinct* (*states* (*xs* || *tr*) (*q2*, *q1*))
then obtain *i1 i2* **where** *index-def* :
i1 ≠ *0*
∧ *i1* ≠ *i2*
∧ *i1* < *length* (*states* (*xs* || *tr*) (*q2*, *q1*))
∧ *i2* < *length* (*states* (*xs* || *tr*) (*q2*, *q1*))
∧ (*states* (*xs* || *tr*) (*q2*, *q1*)) ! *i1* = (*states* (*xs* || *tr*) (*q2*, *q1*)) ! *i2*
by (*metis distinct-conv-nth*)
then have *length xs* > *0* **by** *auto*

let *?xs1* = *take* (*Suc i1*) *xs*
let *?xs2* = *take* (*Suc i2*) *xs*
let *?tr1* = *take* (*Suc i1*) *tr*
let *?tr2* = *take* (*Suc i2*) *tr*
let *?st* = (*states* (*xs* || *tr*) (*q2*, *q1*)) ! *i1*

have *obs-PM* : *observable PM*
using *observable-productF assms*(*2*) *assms*(*3*) *assms*(*4*) **by** *blast*

have *initial PM* = (*initial M2*, *initial M1*)
using *assms*(*2*) **by** *simp*
moreover have *vs* ∈ *L M2 vs* ∈ *L M1*
using *assms*(*8*) *language-state-prefix* **by** *auto*
ultimately have *io-targets M1* (*initial M1*) *vs* = {*q1*} *io-targets M2* (*initial M2*) *vs* = {*q2*}
using *productF-path-io-targets*[*of M2 M1 FAIL PM initial M2 initial M1 vs q2 q1*]
by (*metis FSM.nodes.initial assms*(*2*) *assms*(*3*) *assms*(*4*) *assms*(*5*) *assms*(*9*) *assms*(*10*)
io-targets-observable-singleton-ex singletonD)+

— paths for ?xs1

have (*states* (*xs* || *tr*) (*q2*, *q1*)) ! *i1* ∈ *io-targets PM* (*q2*, *q1*) *?xs1*

by (*metis* ‹*0 < length xs*› *assms*(*6*) *assms*(*7*) *index-def map-snd-zip states-alt-def states-index-io-target*)
then have *io-targets PM* (*q2*, *q1*) *?xs1* = {*?st*}
using *obs-PM* **by** (*meson observable-io-target-is-singleton*)

have *path PM* (*?xs1* || *?tr1*) (*q2*,*q1*)
by (*metis FSM.path-append-elim append-take-drop-id assms*(*6*) *assms*(*7*) *length-take zip-append*)
then have *path PM* (*?xs1* || *map fst ?tr1* || *map snd ?tr1*) (*q2*,*q1*)
by *auto*

have *vs* @ *?xs1* ∈ *L M2*
by (*metis* (*no-types*) *IntD2 append-assoc append-take-drop-id assms*(*8*) *language-state-prefix*)
then obtain *q2'* **where** *io-targets M2* (*initial M2*) (*vs*@*?xs1*) = {*q2'*}
using *io-targets-observable-singleton-ob*[*of M2 vs*@*?xs1 initial M2*] *assms*(*4*) **by** *auto*
then have *q2'* ∈ *io-targets M2 q2 ?xs1*
using *assms*(*4*) ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}›
observable-io-targets-split[*of M2 initial M2 vs ?xs1 q2' q2*]
by *simp*
then have *?xs1* ∈ *language-state M2 q2*
by *auto*
moreover have *length ?xs1* = *length* (*map snd ?tr1*)
using *assms*(*7*) **by** *auto*
moreover have *length* (*map fst ?tr1*) = *length* (*map snd ?tr1*)
by *auto*
moreover have *q2* ∈ *nodes M2*
using ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}› *io-targets-nodes*
by (*metis FSM.nodes.initial insertI1*)
moreover have *q1* ∈ *nodes M1*
using ‹*io-targets M1* (*initial M1*) *vs* = {*q1*}› *io-targets-nodes*
by (*metis FSM.nodes.initial insertI1*)
ultimately have
path M1 (*?xs1* || *map snd ?tr1*) *q1*
path M2 (*?xs1* || *map fst ?tr1*) *q2*
target (*?xs1* || *map snd ?tr1*) *q1* = *snd* (*target* (*?xs1* || *map fst ?tr1* || *map snd ?tr1*) (*q2*,*q1*))
target (*?xs1* || *map fst ?tr1*) *q2* = *fst* (*target* (*?xs1* || *map fst ?tr1* || *map snd ?tr1*) (*q2*,*q1*))
using *assms*(*2*) *assms*(*9*) *assms*(*10*) ‹*path PM* (*?xs1* || *map fst ?tr1* || *map snd ?tr1*) (*q2*,*q1*)›
assms(*4*)
productF-path-reverse-ob-2[*of ?xs1 map fst ?tr1 map snd ?tr1 M2 M1 FAIL PM q2 q1*]
by *simp+*
moreover have *target* (*?xs1* || *map fst ?tr1* || *map snd ?tr1*) (*q2*,*q1*) = *?st*
by (*metis* (*no-types*) *index-def scan-nth take-zip zip-map-fst-snd*)
ultimately have
target (*?xs1* || *map snd ?tr1*) *q1* = *snd ?st*
target (*?xs1* || *map fst ?tr1*) *q2* = *fst ?st*
by *simp+*

— paths for ?xs2

have (*states* (*xs* || *tr*) (*q2*, *q1*)) ! *i2* ∈ *io-targets PM* (*q2*, *q1*) *?xs2*
by (*metis* ‹*0 < length xs*› *assms*(*6*) *assms*(*7*) *index-def map-snd-zip states-alt-def states-index-io-target*)
then have *io-targets PM* (*q2*, *q1*) *?xs2* = {*?st*}
using *obs-PM* **by** (*metis index-def observable-io-target-is-singleton*)

have *path PM* (*?xs2* || *?tr2*) (*q2*,*q1*)
by (*metis FSM.path-append-elim append-take-drop-id assms*(*6*) *assms*(*7*) *length-take zip-append*)
then have *path PM* (*?xs2* || *map fst ?tr2* || *map snd ?tr2*) (*q2*,*q1*)
by *auto*

have *vs* @ *?xs2* ∈ *L M2*
by (*metis* (*no-types*) *IntD2 append-assoc append-take-drop-id assms*(*8*) *language-state-prefix*)
then obtain *q2''* **where** *io-targets M2* (*initial M2*) (*vs*@*?xs2*) = {*q2''*}
using *io-targets-observable-singleton-ob*[*of M2 vs*@*?xs2 initial M2*] *assms*(*4*)
by *auto*
then have *q2''* ∈ *io-targets M2 q2 ?xs2*
using *assms*(*4*) ‹*io-targets M2* (*initial M2*) *vs* = {*q2*}›
observable-io-targets-split[*of M2 initial M2 vs ?xs2 q2'' q2*]

by *simp*
then have *?xs2 ∈ language-state M2 q2*
by *auto*
moreover have *length ?xs2 = length (map snd ?tr2)* **using** *assms(7)*
by *auto*
moreover have *length (map fst ?tr2) = length (map snd ?tr2)*
by *auto*
moreover have *q2 ∈ nodes M2*
using *‹io-targets M2 (initial M2) vs = {q2}› io-targets-nodes*
by *(metis FSM.nodes.initial insertI1)*
moreover have *q1 ∈ nodes M1*
using *‹io-targets M1 (initial M1) vs = {q1}› io-targets-nodes*
by *(metis FSM.nodes.initial insertI1)*
ultimately have
path M1 (?xs2 || map snd ?tr2) q1
path M2 (?xs2 || map fst ?tr2) q2
target (?xs2 || map snd ?tr2) q1 = snd(target (?xs2 || map fst ?tr2 || map snd ?tr2) (q2,q1))
target (?xs2 || map fst ?tr2) q2 = fst(target (?xs2 || map fst ?tr2 || map snd ?tr2) (q2,q1))
using *assms(2) assms(9) assms(10) ‹path PM (?xs2 || map fst ?tr2 || map snd ?tr2) (q2,q1)›*
assms(4)
productF-path-reverse-ob-2[of ?xs2 map fst ?tr2 map snd ?tr2 M2 M1 FAIL PM q2 q1]
by *simp+*
moreover have *target (?xs2 || map fst ?tr2 || map snd ?tr2) (q2,q1) = ?st*
by *(metis (no-types) index-def scan-nth take-zip zip-map-fst-snd)*
ultimately have
target (?xs2 || map snd ?tr2) q1 = snd ?st
target (?xs2 || map fst ?tr2) q2 = fst ?st
by *simp+*

have *io-targets M1 q1 ?xs1 = {snd ?st}*
using *‹path M1 (?xs1 || map snd ?tr1) q1› ‹target (?xs1 || map snd ?tr1) q1 = snd ?st›*
‹length ?xs1 = length (map snd ?tr1)› assms(3) obs-target-is-io-targets[of M1 ?xs1
map snd ?tr1 q1]
by *simp*
then have *tgt-1-1 : io-targets M1 (initial M1) (vs @ ?xs1) = {snd ?st}*
by *(meson ‹io-targets M1 (initial M1) vs = {q1}› assms(3) observable-io-targets-append)*

have *io-targets M2 q2 ?xs1 = {fst ?st}*
using *‹path M2 (?xs1 || map fst ?tr1) q2› ‹target (?xs1 || map fst ?tr1) q2 = fst ?st›*
‹length ?xs1 = length (map snd ?tr1)› assms(4)
obs-target-is-io-targets[of M2 ?xs1 map fst ?tr1 q2]
by *simp*
then have *tgt-1-2 : io-targets M2 (initial M2) (vs @ ?xs1) = {fst ?st}*
by *(meson ‹io-targets M2 (initial M2) vs = {q2}› assms(4) observable-io-targets-append)*

have *io-targets M1 q1 ?xs2 = {snd ?st}*
using *‹path M1 (?xs2 || map snd ?tr2) q1› ‹target (?xs2 || map snd ?tr2) q1 = snd ?st›*
‹length ?xs2 = length (map snd ?tr2)› assms(3)
obs-target-is-io-targets[of M1 ?xs2 map snd ?tr2 q1]
by *simp*
then have *tgt-2-1 : io-targets M1 (initial M1) (vs @ ?xs2) = {snd ?st}*
by *(meson ‹io-targets M1 (initial M1) vs = {q1}› assms(3) observable-io-targets-append)*

have *io-targets M2 q2 ?xs2 = {fst ?st}*
using *‹path M2 (?xs2 || map fst ?tr2) q2› ‹target (?xs2 || map fst ?tr2) q2 = fst ?st›*
‹length ?xs2 = length (map snd ?tr2)› assms(4)
obs-target-is-io-targets[of M2 ?xs2 map fst ?tr2 q2]
by *simp*
then have *tgt-2-2 : io-targets M2 (initial M2) (vs @ ?xs2) = {fst ?st}*
by *(meson ‹io-targets M2 (initial M2) vs = {q2}› assms(4) observable-io-targets-append)*

have *?xs1 ≠ []* **using** *‹0 < length xs›*
by *auto*
have *prefix ?xs1 xs*

```
    using take-is-prefix by blast
  have prefix ?xs2 xs
    using take-is-prefix by blast
  have ?xs1 ≠ ?xs2
  proof −
    have f1: ∀ n na. ¬ n < na ∨ Suc n ≤ na
      by presburger
    have f2: Suc i1 ≤ length xs
      using index-def by force
    have Suc i2 ≤ length xs
      using f1 by (metis index-def length-take map-snd-zip-take min-less-iff-conj states-alt-def)
    then show ?thesis
      using f2 by (metis (no-types) index-def length-take min.absorb2 nat.simps(1))
  qed
  have Rep-Pre M2 M1 vs xs
  proof (cases length ?xs1 < length ?xs2)
    case True
    then have prefix ?xs1 ?xs2
      by (meson ‹prefix (take (Suc i1) xs) xs› ‹prefix (take (Suc i2) xs) xs› leD prefix-length-le
          prefix-same-cases)
    show ?thesis
      by (meson Rep-Pre.elims(3) ‹prefix (take (Suc i1) xs) (take (Suc i2) xs)›
          ‹prefix (take (Suc i2) xs) xs› ‹take (Suc i1) xs ≠ take (Suc i2) xs›
          tgt-1-1 tgt-1-2 tgt-2-1 tgt-2-2)
  next
    case False
    moreover have length ?xs1 ≠ length ?xs2
      by (metis (no-types) ‹take (Suc i1) xs ≠ take (Suc i2) xs› append-eq-conv-conj
          append-take-drop-id)
    ultimately have length ?xs2 < length ?xs1
      by auto
    then have prefix ?xs2 ?xs1
      using ‹prefix (take (Suc i1) xs) xs› ‹prefix (take (Suc i2) xs) xs› less-imp-le-nat
            prefix-length-prefix
      by blast
    show ?thesis
      by (metis Rep-Pre.elims(3) ‹prefix (take (Suc i1) xs) xs›
          ‹prefix (take (Suc i2) xs) (take (Suc i1) xs)› ‹take (Suc i1) xs ≠ take (Suc i2) xs›
          tgt-1-1 tgt-1-2 tgt-2-1 tgt-2-2)
  qed

  then show False
    using assms(1) by simp
qed




lemma RP-count-via-Rep-Cov :
  assumes (vs @ xs) ∈ L M1 ∩ L M2
  and observable M1
  and observable M2
  and well-formed M1
  and well-formed M2
  and s ∈ nodes M2
  and productF M2 M1 FAIL PM
  and io-targets PM (initial PM) vs = {(q2,q1)}
  and path PM (xs || tr) (q2,q1)
  and length xs = length tr
  and distinct (states (xs || tr) (q2,q1))
  and is-det-state-cover M2 V
  and V'' ∈ Perm V M1
  and ¬ Rep-Cov M2 M1 V'' vs xs
shows card (⋃(image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) = card (RP M2 s vs xs V'')
proof −
```

```
have RP-cases : RP M2 s vs xs V'' = R M2 s vs xs
                ∨ (∃ vs' ∈ V'' . vs' ∉ R M2 s vs xs
                                 ∧ RP M2 s vs xs V'' = insert vs' (R M2 s vs xs))
  using RP-from-R assms by metis
show ?thesis
proof (cases RP M2 s vs xs V'' = R M2 s vs xs)
  case True
  then show ?thesis
    using R-count assms by metis
next
  case False
  then obtain vs' where vs'-def : vs' ∈ V''
                                 ∧ vs' ∉ R M2 s vs xs
                                 ∧ RP M2 s vs xs V'' = insert vs' (R M2 s vs xs)
    using RP-cases by auto

  have state-component-2 : ∀ io ∈ (R M2 s vs xs) . io-targets M2 (initial M2) io = {s}
  proof
    fix io assume io ∈ R M2 s vs xs
    then have s ∈ io-targets M2 (initial M2) io
      by auto
    moreover have io ∈ language-state M2 (initial M2)
      using calculation by auto
    ultimately show io-targets M2 (initial M2) io = {s}
      using assms(3) io-targets-observable-singleton-ex by (metis singletonD)
  qed

  have vs' ∈ L M1
    using assms(13) perm-language vs'-def by blast
  then obtain s' where s'-def : io-targets M1 (initial M1) vs' = {s'}
    by (meson assms(2) io-targets-observable-singleton-ob)

  moreover have s' ∉ ⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs))
  proof (rule ccontr)
    assume ¬ s' ∉ ⋃(io-targets M1 (initial M1) ‘ R M2 s vs xs)
    then obtain xs' where xs'-def : vs @ xs' ∈ R M2 s vs xs
                                   ∧ s' ∈ io-targets M1 (initial M1) (vs @ xs')
    proof −
      assume a1: ⋀xs'. vs @ xs' ∈ R M2 s vs xs
                 ∧ s' ∈ io-targets M1 (initial M1) (vs @ xs') ⟹ thesis
      obtain pps :: ('a × 'b) list set ⇒ (('a × 'b) list ⇒ 'c set) ⇒ 'c ⇒ ('a × 'b) list
        where
        ∀x0 x1 x2. (∃v3. v3 ∈ x0 ∧ x2 ∈ x1 v3) = (pps x0 x1 x2 ∈ x0 ∧ x2 ∈ x1 (pps x0 x1 x2))
        by moura
      then have f2: pps (R M2 s vs xs) (io-targets M1 (initial M1)) s' ∈ R M2 s vs xs
                ∧ s' ∈ io-targets M1 (initial M1)
                               (pps (R M2 s vs xs) (io-targets M1 (initial M1)) s')
        using ‹¬ s' ∉ ⋃(io-targets M1 (initial M1) ‘ R M2 s vs xs)› by blast
      then have ∃ps. pps (R M2 s vs xs) (io-targets M1 (initial M1)) s' = vs @ ps ∧ ps ≠ []
                ∧ prefix ps xs ∧ s ∈ io-targets M2 (initial M2) (vs @ ps)
        by simp
      then show ?thesis
        using f2 a1 by (metis (no-types))
    qed

    have vs @ xs' ∈ L M1
      using xs'-def by blast
    then have io-targets M1 (initial M1) (vs@xs') = {s'}
      by (metis assms(2) io-targets-observable-singleton-ob singletonD xs'-def)
    moreover have io-targets M1 (initial M1) (vs') = {s'}
      using s'-def by blast
    moreover have io-targets M2 (initial M2) (vs @ xs') = {s}
      using state-component-2 xs'-def by blast
    moreover have io-targets M2 (initial M2) (vs') = {s}
      by (metis (mono-tags, lifting) RP.simps Un-iff insertI1 mem-Collect-eq vs'-def)
    moreover have xs' ≠ []
```

```
      using xs'-def by simp
    moreover have prefix xs' xs
      using xs'-def by simp
    moreover have vs' ∈ V''
      using vs'-def by simp
    ultimately have Rep-Cov M2 M1 V'' vs xs
      by auto

    then show False
      using assms(14) by simp
  qed

  moreover have ⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs)))
                = insert s' (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs)))
    using s'-def by simp

  moreover have finite (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs)))
  proof
    show finite (R M2 s vs xs)
      using finite-R by simp
    show ⋀a. a ∈ R M2 s vs xs ⟹ finite (io-targets M1 (initial M1) a)
    proof −
      fix a assume a ∈ R M2 s vs xs
      then have prefix a (vs@xs)
        by auto
      then have a ∈ L M1
        using language-state-prefix by (metis IntD1 assms(1) prefix-def)
      then obtain p where io-targets M1 (initial M1) a = {p}
        using assms(2) io-targets-observable-singleton-ob by metis
      then show finite (io-targets M1 (initial M1) a)
        by simp
    qed
  qed

  ultimately have card (⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs))))
                  = Suc (card (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs))))
    by (metis (no-types) card-insert-disjoint)


  moreover have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
                = card (⋃ (image (io-targets M1 (initial M1)) (insert vs' (R M2 s vs xs))))
    using vs'-def by simp

  ultimately have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
                  = Suc (card (⋃ (image (io-targets M1 (initial M1)) (R M2 s vs xs))))
    by linarith

  then have card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))
             = Suc (card (R M2 s vs xs))
    using R-count[of vs xs M1 M2 s FAIL PM q2 q1 tr] using assms(1,10,11,2−9)
    by linarith

  moreover have card (RP M2 s vs xs V'') = Suc (card (R M2 s vs xs))
    using vs'-def by (metis card-insert-if finite-R)

  ultimately show ?thesis
    by linarith
 qed
qed


lemma RP-count-alt-def :
 assumes (vs @ xs) ∈ L M1 ∩ L M2
 and observable M1
 and observable M2
 and well-formed M1
```

and *well-formed M2*
and *s* ∈ *nodes M2*
and *productF M2 M1 FAIL PM*
and *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
and *path PM* (*xs* || *tr*) (*q2*,*q1*)
and *length xs* = *length tr*
and ¬ *Rep-Pre M2 M1 vs xs*
and *is-det-state-cover M2 V*
and *V''* ∈ *Perm V M1*
and ¬ *Rep-Cov M2 M1 V'' vs xs*
shows *card* (⋃(*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs V''*))) = *card* (*RP M2 s vs xs V''*)
proof −
have *distinct* (*states* (*xs* || *tr*) (*q2*,*q1*))
using *distinctness-via-Rep-Pre*[*of M2 M1 vs xs FAIL PM q2 q1 tr*] *assms* **by** *simp*
then show *?thesis*
using *RP-count-via-Rep-Cov*[*of vs xs M1 M2 s FAIL PM q2 q1 tr V V''*]
using *assms*(*1*,*10*,*12*−*14*,*2*−*9*) **by** *blast*
qed

4.6 Function LB

`LB` adds together the number of elements in sets calculated via RP for a given set of states and the number of ATC-reaction known to exist but not produced by a state reached by any of the above elements.

fun *LB* :: ('*in*, '*out*, '*state1*) *FSM* ⇒ ('*in*, '*out*, '*state2*) *FSM*
⇒ ('*in* × '*out*) *list* ⇒ ('*in* × '*out*) *list* ⇒ '*in list set*
⇒ '*state1 set* ⇒ ('*in*, '*out*) *ATC set*
⇒ ('*in* × '*out*) *list set* ⇒ *nat*
where
LB M2 M1 vs xs T S Ω *V''* =
(*sum* (λ *s* . *card* (*RP M2 s vs xs V''*)) *S*)
+ *card* ((*D M1 T* Ω) −
{*B M1 xs'* Ω | *xs' s'* . *s'* ∈ *S* ∧ *xs'* ∈ *RP M2 s' vs xs V''*})

lemma *LB-count-helper-RP-disjoint-and-cards* :
assumes (*vs* @ *xs*) ∈ *L M1* ∩ *L M2*
and *observable M1*
and *observable M2*
and *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *is-det-state-cover M2 V*
and *V''* ∈ *Perm V M1*
and *s1* ≠ *s2*
shows ⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s1 vs xs V''*))
∩ ⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s2 vs xs V''*)) = {}
card (⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s1 vs xs V''*)))
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*)))
card (⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s2 vs xs V''*)))
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)))
proof −
have ∀ *io* ∈ *RP M2 s1 vs xs V''* . *io-targets PM* (*initial PM*) *io*
= {*s1*} × *io-targets M1* (*initial M1*) *io*
proof
fix *io* **assume** *io* ∈ *RP M2 s1 vs xs V''*
then have *io-targets PM* (*initial PM*) *io*
= *io-targets M2* (*initial M2*) *io* × *io-targets M1* (*initial M1*) *io*
using *assms RP-io-targets-split*[*of vs xs M1 M2 FAIL PM V V'' io s1*] **by** *simp*
moreover have *io-targets M2* (*initial M2*) *io* = {*s1*}
using ‹*io* ∈ *RP M2 s1 vs xs V''*› *assms*(*3*) *RP-state-component-2*[*of io M2 s1 vs xs V''*]
by *blast*
ultimately show *io-targets PM* (*initial PM*) *io* = {*s1*} × *io-targets M1* (*initial M1*) *io*
by *auto*
qed

```
  then have ⋃ (image (io-targets PM (initial PM)) (RP M2 s1 vs xs V''))
            = ⋃ (image (λ io . {s1} × io-targets M1 (initial M1) io) (RP M2 s1 vs xs V''))
    by simp
  moreover have ⋃ (image (λ io . {s1} × io-targets M1 (initial M1) io) (RP M2 s1 vs xs V''))
                = {s1} × ⋃ (image (λ io . io-targets M1 (initial M1) io) (RP M2 s1 vs xs V''))
    by blast
  ultimately have image-split-1 :
    ⋃ (image (io-targets PM (initial PM)) (RP M2 s1 vs xs V'') )
      = {s1} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))
    by simp
  then show card (⋃ (image (io-targets PM (initial PM)) (RP M2 s1 vs xs V'')))
            = card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V'')))
    by (metis (no-types) card-cartesian-product-singleton)



  have ∀ io ∈ RP M2 s2 vs xs V'' . io-targets PM (initial PM) io
                                        = {s2} × io-targets M1 (initial M1) io
  proof
    fix io assume io ∈ RP M2 s2 vs xs V''
    then have io-targets PM (initial PM) io
                = io-targets M2 (initial M2) io × io-targets M1 (initial M1) io
      using assms RP-io-targets-split[of vs xs M1 M2 FAIL PM V V'' io s2] by simp
    moreover have io-targets M2 (initial M2) io = {s2}
      using ‹io ∈ RP M2 s2 vs xs V''› assms(3) RP-state-component-2[of io M2 s2 vs xs V'']
      by blast
    ultimately show io-targets PM (initial PM) io = {s2} × io-targets M1 (initial M1) io
      by auto
  qed
  then have ⋃ (image (io-targets PM (initial PM)) (RP M2 s2 vs xs V''))
            = ⋃ (image (λ io . {s2} × io-targets M1 (initial M1) io) (RP M2 s2 vs xs V''))
    by simp
  moreover have ⋃ (image (λ io . {s2} × io-targets M1 (initial M1) io) (RP M2 s2 vs xs V''))
                = {s2} × ⋃ (image (λ io . io-targets M1 (initial M1) io) (RP M2 s2 vs xs V''))
    by blast
  ultimately have image-split-2 :
    ⋃ (image (io-targets PM (initial PM)) (RP M2 s2 vs xs V''))
      = {s2} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')) by simp
  then show card (⋃ (image (io-targets PM (initial PM)) (RP M2 s2 vs xs V'')))
            = card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')))
    by (metis (no-types) card-cartesian-product-singleton)


  have ⋃ (image (io-targets PM (initial PM)) (RP M2 s1 vs xs V''))
         ∩ ⋃ (image (io-targets PM (initial PM)) (RP M2 s2 vs xs V''))
       = {s1} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))
         ∩ {s2} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V''))
    using image-split-1 image-split-2 by blast
  moreover have {s1} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V''))
                ∩ {s2} × ⋃ (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')) = {}
    using assms(9) by auto
  ultimately show ⋃ (image (io-targets PM (initial PM)) (RP M2 s1 vs xs V''))
                  ∩ ⋃ (image (io-targets PM (initial PM)) (RP M2 s2 vs xs V'')) = {}
    by presburger
qed


lemma LB-count-helper-RP-disjoint-card-M1 :
  assumes (vs @ xs) ∈ L M1 ∩ L M2
  and observable M1
  and observable M2
  and well-formed M1
  and well-formed M2
  and productF M2 M1 FAIL PM
  and is-det-state-cover M2 V
```

and *V'' ∈ Perm V M1*
and *s1 ≠ s2*
shows *card* (⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s1 vs xs V''*))
∪ ⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s2 vs xs V''*)))
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*)))
+ *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)))
proof −
have *finite* (⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s1 vs xs V''*)))
using *RP-io-targets-finite-PM*[*OF assms*(*1−8*)] **by** *simp*
moreover have *finite* (⋃ (*image* (*io-targets PM* (*initial PM*)) (*RP M2 s2 vs xs V''*)))
using *RP-io-targets-finite-PM*[*OF assms*(*1−8*)] **by** *simp*
ultimately show *?thesis*
using *LB-count-helper-RP-disjoint-and-cards*[*OF assms*]
by (*metis* (*no-types*) *card-Un-disjoint*)
qed

lemma *LB-count-helper-RP-disjoint-M1-pair* :
assumes (*vs @ xs*) ∈ *L M1* ∩ *L M2*
and *observable M1*
and *observable M2*
and *well-formed M1*
and *well-formed M2*
and *productF M2 M1 FAIL PM*
and *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
and *path PM* (*xs* || *tr*) (*q2*,*q1*)
and *length xs* = *length tr*
and ¬ *Rep-Pre M2 M1 vs xs*
and *is-det-state-cover M2 V*
and *V'' ∈ Perm V M1*
and ¬ *Rep-Cov M2 M1 V'' vs xs*
and *Prereq M2 M1 vs xs T S Ω V''*
and *s1 ≠ s2*
and *s1 ∈ S*
and *s2 ∈ S*
and *applicable-set M1 Ω*
and *completely-specified M1*
shows *card* (*RP M2 s1 vs xs V''*) + *card* (*RP M2 s2 vs xs V''*)
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*)))
+ *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)))
⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*))
∩ ⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*))
= {}
proof −
have *s1 ∈ nodes M2*
using *assms*(*14*,*16*) **unfolding** *Prereq.simps* **by** *blast*
have *s2 ∈ nodes M2*
using *assms*(*14*,*17*) **unfolding** *Prereq.simps* **by** *blast*
have *card* (*RP M2 s1 vs xs V''*)
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*)))
using *RP-count-alt-def*[*OF assms*(*1−5*) ‹*s1 ∈ nodes M2*› *assms*(*6−13*)]
by *linarith*
moreover have *card* (*RP M2 s2 vs xs V''*)
= *card* (⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)))
using *RP-count-alt-def*[*OF assms*(*1−5*) ‹*s2 ∈ nodes M2*› *assms*(*6−13*)]
by *linarith*

moreover show ⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*))
∩ ⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)) = {}
proof (*rule ccontr*)
assume ⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s1 vs xs V''*))
∩ ⋃ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s2 vs xs V''*)) ≠ {}
then obtain *io1 io2 t* **where** *shared-elem-def* :
io1 ∈ (*RP M2 s1 vs xs V''*)
io2 ∈ (*RP M2 s2 vs xs V''*)
t ∈ *io-targets M1* (*initial M1*) *io1*
t ∈ *io-targets M1* (*initial M1*) *io2*

```
    by blast


  have dist-prop: (∀ s1 ∈ S . ∀ s2 ∈ S . s1 ≠ s2
        ⟶ (∀ io1 ∈ RP M2 s1 vs xs V'' .
              ∀ io2 ∈ RP M2 s2 vs xs V'' .
                B M1 io1 Ω ≠ B M1 io2 Ω ))
    using assms(14) by simp

  have io-targets M1 (initial M1) io1 ∩ io-targets M1 (initial M1) io2 = {}
  proof (rule ccontr)
    assume io-targets M1 (initial M1) io1 ∩ io-targets M1 (initial M1) io2 ≠ {}
    then have io-targets M1 (initial M1) io1 ≠ {} io-targets M1 (initial M1) io2 ≠ {}
      by blast+

    then obtain s1 s2 where s1 ∈ io-targets M1 (initial M1) io1
                          s2 ∈ io-targets M1 (initial M1) io2
      by blast

    then have io-targets M1 (initial M1) io1 = {s1}
          io-targets M1 (initial M1) io2 = {s2}
      by (meson assms(2) observable-io-target-is-singleton)+

    then have s1 = s2
      using ‹io-targets M1 (initial M1) io1 ∩ io-targets M1 (initial M1) io2 ≠ {}›
      by auto

    then have B M1 io1 Ω = B M1 io2 Ω
      using ‹io-targets M1 (initial M1) io1 = {s1}› ‹io-targets M1 (initial M1) io2 = {s2}›
      by auto
    then show False
      using assms(15−17) dist-prop shared-elem-def(1,2) by blast
  qed
  then show False
    using shared-elem-def(3,4) by blast
 qed

 ultimately show card (RP M2 s1 vs xs V'') + card (RP M2 s2 vs xs V'')
     = card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s1 vs xs V'')))
       + card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s2 vs xs V'')))
   by linarith
qed





lemma LB-count-helper-RP-card-union :
 assumes observable M2
 and     s1 ≠ s2
shows RP M2 s1 vs xs V'' ∩ RP M2 s2 vs xs V'' = {}
proof (rule ccontr)
 assume RP M2 s1 vs xs V'' ∩ RP M2 s2 vs xs V'' ≠ {}
 then obtain io where io ∈ RP M2 s1 vs xs V'' ∧ io ∈ RP M2 s2 vs xs V''
   by blast
 then have s1 ∈ io-targets M2 (initial M2) io
         s2 ∈ io-targets M2 (initial M2) io
   by auto
 then have s1 = s2
   using assms(1) by (metis observable-io-target-is-singleton singletonD)
 then show False
   using assms(2) by simp
qed
```

```
lemma LB-count-helper-RP-inj :
obtains f
where ∀ q ∈ (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S)) .
          f q ∈ nodes M1
      inj-on f (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S))
proof −
  let ?f =
    λ q . if (q ∈ (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S)))
      then q
      else (initial M1)

  have (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S)) ⊆ nodes M1
    by blast

  then have ∀ q ∈ (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S)) .
            ?f q ∈ nodes M1
    by (metis Un-iff sup.order-iff)

  moreover have inj-on ?f (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1))
                                               (RP M2 s vs xs V''))) S))
  proof
    fix x assume x ∈ (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S))
    then have ?f x = x
      by presburger

    fix y assume y ∈ (⋃ (image (λ s . ⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V''))) S))
    then have ?f y = y
      by presburger

    assume ?f x = ?f y
    then show x = y using ‹?f x = x› ‹?f y = y›
      by presburger
  qed

  ultimately show ?thesis
    using that by presburger
qed

abbreviation (input) UNION :: 'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set
  where UNION A f ≡ ⋃ (f ‘ A)


lemma LB-count-helper-RP-card-union-sum :
  assumes (vs @ xs) ∈ L M2 ∩ L M1
  and     OFSM M1
  and     OFSM M2
  and     asc-fault-domain M2 M1 m
  and     test-tools M2 M1 FAIL PM V Ω
  and     V'' ∈ Perm V M1
  and     Prereq M2 M1 vs xs T S Ω V''
  and     ¬ Rep-Pre M2 M1 vs xs
  and     ¬ Rep-Cov M2 M1 V'' vs xs
shows sum (λ s . card (RP M2 s vs xs V'')) S
        = sum (λ s . card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))) S
using assms proof −
  have finite (nodes M2)
    using assms(3) by auto
  moreover have S ⊆ nodes M2
    using assms(7) by simp
  ultimately have finite S
    using infinite-super by blast

  then have sum (λ s . card (RP M2 s vs xs V'')) S
            = sum (λ s . card (⋃ (image (io-targets M1 (initial M1)) (RP M2 s vs xs V'')))) S
  using assms proof (induction S)
```

case *empty*
show *?case* **by** *simp*
next
case (*insert s S*)

have (*insert s S*) ⊆ *nodes M2*
using *insert.prems*(*7*) **by** *simp*
then have *s* ∈ *nodes M2*
by *simp*

have *Prereq M2 M1 vs xs T S* Ω *V''*
using ‹*Prereq M2 M1 vs xs T* (*insert s S*) Ω *V''*› **by** *simp*
then have ($\sum$ *s*∈*S*. *card* (*RP M2 s vs xs V''*))
= ($\sum$ *s*∈*S*. *card* ($\bigcup$ *a*∈*RP M2 s vs xs V''*. *io-targets M1* (*initial M1*) *a*))
using *insert.IH*[*OF insert.prems*(*1*−*6*) - *assms*(*8*,*9*)] **by** *metis*
moreover have ($\sum$ *s'*∈(*insert s S*). *card* (*RP M2 s' vs xs V''*))
= ($\sum$ *s'*∈*S*. *card* (*RP M2 s' vs xs V''*)) + *card* (*RP M2 s vs xs V''*)
by (*simp add*: *add.commute insert.hyps*(*1*) *insert.hyps*(*2*))
ultimately have *S-prop* : ($\sum$ *s'*∈(*insert s S*). *card* (*RP M2 s' vs xs V''*))
= ($\sum$ *s*∈*S*. *card* ($\bigcup$ *a*∈*RP M2 s vs xs V''*. *io-targets M1* (*initial M1*) *a*))
+ *card* (*RP M2 s vs xs V''*)
by *presburger*

have *vs*@*xs* ∈ *L M1* ∩ *L M2*
using *insert.prems*(*1*) **by** *simp*

obtain *q2 q1 tr* **where** *suffix-path* : *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
path PM (*xs* || *tr*) (*q2*,*q1*)
length xs = *length tr*
using *productF-language-state-intermediate*[*OF insert.prems*(*1*)
test-tools-props(*1*)[*OF insert.prems*(*5*,*4*)] *OFSM-props*(*2*,*1*)[*OF insert.prems*(*3*)]
OFSM-props(*2*,*1*)[*OF insert.prems*(*2*)]]
by *blast*

have *card* (*RP M2 s vs xs V''*)
= *card* ($\bigcup$ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs V''*)))
using *OFSM-props*(*2*,*1*)[*OF insert.prems*(*3*)] *OFSM-props*(*2*,*1*)[*OF insert.prems*(*2*)]
RP-count-alt-def[*OF* ‹*vs*@*xs* ∈ *L M1* ∩ *L M2*› - - - -
‹*s*∈*nodes M2*› *test-tools-props*(*1*)[*OF insert.prems*(*5*,*4*)]
suffix-path insert.prems(*8*)
test-tools-props(*2*)[*OF insert.prems*(*5*,*4*)] *assms*(*6*) *insert.prems*(*9*)]
by *linarith*

show ($\sum$ *s*∈*insert s S*. *card* (*RP M2 s vs xs V''*)) =
($\sum$ *s*∈*insert s S*. *card* (*UNION* (*RP M2 s vs xs V''*) (*io-targets M1* (*initial M1*))))
proof −
have ($\sum$ *c*∈*insert s S*. *card* (*UNION* (*RP M2 c vs xs V''*) (*io-targets M1* (*initial M1*))))
= *card* (*UNION* (*RP M2 s vs xs V''*) (*io-targets M1* (*initial M1*)))
+ ($\sum$ *c*∈*S*. *card* (*UNION* (*RP M2 c vs xs V''*) (*io-targets M1* (*initial M1*))))
by (*meson insert.hyps*(*1*) *insert.hyps*(*2*) *sum.insert*)
then show *?thesis*
using ‹($\sum$ *s'*∈*insert s S*. *card* (*RP M2 s' vs xs V''*))
= ($\sum$ *s*∈*S*. *card* ($\bigcup$ *a*∈*RP M2 s vs xs V''*. *io-targets M1* (*initial M1*) *a*))
+ *card* (*RP M2 s vs xs V''*)›
‹*card* (*RP M2 s vs xs V''*)
= *card* (*UNION* (*RP M2 s vs xs V''*) (*io-targets M1* (*initial M1*)))›
by *presburger*
qed
qed

then show *?thesis*
using *assms* **by** *blast*
qed

lemma *finite-insert-card* :
  assumes *finite* ($\bigcup$ *SS*)
  and *finite S*
  and $S \cap (\bigcup SS) = \{\}$
shows *card* ($\bigcup$ (*insert S SS*)) = *card* ($\bigcup$ *SS*) + *card S*
  by (*simp add*: *assms*(*1*) *assms*(*2*) *assms*(*3*) *card-Un-disjoint*)

lemma *LB-count-helper-RP-disjoint-M1-union* :
  assumes (*vs* @ *xs*) ∈ *L M2* ∩ *L M1*
  and *OFSM M1*
  and *OFSM M2*
  and *asc-fault-domain M2 M1 m*
  and *test-tools M2 M1 FAIL PM V* Ω
  and V'' ∈ *Perm V M1*
  and *Prereq M2 M1 vs xs T S* Ω V''
  and ¬ *Rep-Pre M2 M1 vs xs*
  and ¬ *Rep-Cov M2 M1* V'' *vs xs*
shows *sum* (λ *s* . *card* (*RP M2 s vs xs* V'')) *S*
    = *card* ($\bigcup$ (*image* (λ *s* . $\bigcup$ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs* V''))) *S*))
using *assms* **proof** −
  have *finite* (*nodes M2*)
    using *assms*(*3*) **by** *auto*
  moreover have *S* ⊆ *nodes M2*
    using *assms*(*7*) **by** *simp*
  ultimately have *finite S*
    using *infinite-super* **by** *blast*

  then show *sum* (λ *s* . *card* (*RP M2 s vs xs* V'')) *S*
      = *card* ($\bigcup$ (*image* (λ *s* . $\bigcup$ (*image* (*io-targets M1* (*initial M1*)) (*RP M2 s vs xs* V''))) *S*))
  using *assms* **proof** (*induction S*)
    case *empty*
    show *?case* **by** *simp*
  next
    case (*insert s S*)

    have (*insert s S*) ⊆ *nodes M2*
      using *insert.prems*(*7*) **by** *simp*
    then have *s* ∈ *nodes M2*
      by *simp*

    have *Prereq M2 M1 vs xs T S* Ω V''
      using ‹*Prereq M2 M1 vs xs T* (*insert s S*) Ω V''› **by** *simp*
    then have *applied-IH* : ($\sum$ *s*∈*S*. *card* (*RP M2 s vs xs* V''))
        = *card* ($\bigcup$ *s*∈*S*. $\bigcup$ *a*∈*RP M2 s vs xs* V''. *io-targets M1* (*initial M1*) *a*)
      using *insert.IH*[*OF insert.prems*(*1*−*6*) - *insert.prems*(*8*,*9*)] **by** *metis*

    obtain *q2 q1 tr* **where** *suffix-path* : *io-targets PM* (*initial PM*) *vs* = {(*q2*,*q1*)}
        path PM (*xs* || *tr*) (*q2*,*q1*)
        length xs = *length tr*
      using *productF-language-state-intermediate*
        [*OF insert.prems*(*1*) *test-tools-props*(*1*)[*OF insert.prems*(*5*,*4*)]
          OFSM-props(*2*,*1*)[*OF insert.prems*(*3*)] *OFSM-props*(*2*,*1*)[*OF insert.prems*(*2*)]]
      by *blast*

    have *s* ∈ *insert s S*
      by *simp*

    have *vs*@*xs* ∈ *L M1* ∩ *L M2*
      using *insert.prems*(*1*) **by** *simp*

    have ∀ s' ∈ *S* . ($\bigcup$ *a*∈*RP M2 s vs xs* V''. *io-targets M1* (*initial M1*) *a*)
        ∩ ($\bigcup$ *a*∈*RP M2* s' *vs xs* V''. *io-targets M1* (*initial M1*) *a*) = {}
    proof
      fix s' **assume** s' ∈ *S*

have *s* ≠ *s′*
  using *insert.hyps*(*2*) ‹*s′* ∈ *S*› **by** *blast*
have *s′* ∈ *insert s S*
  using ‹*s′* ∈ *S*› **by** *simp*

show (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
    ∩ (⋃ *a*∈*RP M2 s′ vs xs V′′*. *io-targets M1* (*initial M1*) *a*) = {}
  using *OFSM-props*(*2*,*1*)[*OF assms*(*3*)] *OFSM-props*(*2*,*1*,*3*)[*OF assms*(*2*)]
    LB-count-helper-RP-disjoint-M1-pair(*2*)
      [*OF* ‹*vs*@*xs* ∈ *L M1* ∩ *L M2*› - - - - *test-tools-props*(*1*)[*OF insert.prems*(*5*,*4*)]
        suffix-path insert.prems(*8*) *test-tools-props*(*2*)[*OF insert.prems*(*5*,*4*)]
        insert.prems(*6*,*9*,*7*) ‹*s* ≠ *s′*› ‹*s* ∈ *insert s S*› ‹*s′* ∈ *insert s S*›
        test-tools-props(*4*)[*OF insert.prems*(*5*,*4*)]]
  by *linarith*
qed
then have *disj-insert* : (⋃ *s*∈*S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
    ∩ (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*) = {}
  by *blast*
have *finite-S* : *finite* (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
  using *RP-io-targets-finite-M1*[*OF insert.prems*(*1*)]
  by (*meson RP-io-targets-finite-M1* ‹*vs* @ *xs* ∈ *L M1* ∩ *L M2*› *assms*(*2*) *assms*(*5*) *insert.prems*(*6*))
have *finite-s* : *finite* (⋃ *s*∈*S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
  by (*meson RP-io-targets-finite-M1* ‹*vs* @ *xs* ∈ *L M1* ∩ *L M2*› *assms*(*2*) *assms*(*5*)
    finite-UN-I insert.hyps(*1*) *insert.prems*(*6*))

have *card* (⋃ *s*∈*insert s S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
    = *card* (⋃ *s*∈*S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
    + *card* (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
proof −
  have *f1*: *insert* (*UNION* (*RP M2 s vs xs V′′*) (*io-targets M1* (*initial M1*)))
      ((λ*c*. *UNION* (*RP M2 c vs xs V′′*) (*io-targets M1* (*initial M1*))) ‘ *S*)
    = (λ*c*. *UNION* (*RP M2 c vs xs V′′*) (*io-targets M1* (*initial M1*))) ‘ *insert s S*
    by *blast*
  have ∀ *c*. *c* ∈ *S* ⟶ *UNION* (*RP M2 s vs xs V′′*) (*io-targets M1* (*initial M1*))
      ∩ *UNION* (*RP M2 c vs xs V′′*) (*io-targets M1* (*initial M1*)) = {}
    by (*meson* ‹∀ *s′*∈*S*. (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
      ∩ (⋃ *a*∈*RP M2 s′ vs xs V′′*. *io-targets M1* (*initial M1*) *a*) = {}›)
  then have *UNION* (*RP M2 s vs xs V′′*) (*io-targets M1* (*initial M1*))
      ∩ (⋃ *c*∈*S*. *UNION* (*RP M2 c vs xs V′′*) (*io-targets M1* (*initial M1*))) = {}
    by *blast*
  then show *?thesis*
    using *f1* **by** (*metis finite-S finite-insert-card finite-s*)
qed

have *card* (*RP M2 s vs xs V′′*)
    = *card* (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
  using *assms*(*2*) *assms*(*3*)
    RP-count-alt-def[*OF* ‹*vs*@*xs* ∈ *L M1* ∩ *L M2*› - - - - ‹*s* ∈ *nodes M2*›
      test-tools-props(*1*)[*OF insert.prems*(*5*,*4*)] *suffix-path*
      insert.prems(*8*) *test-tools-props*(*2*)[*OF insert.prems*(*5*,*4*)]
      insert.prems(*6*,*9*)]
  by *metis*

show *?case*
proof −
  have (∑ *c*∈*insert s S*. *card* (*RP M2 c vs xs V′′*))
      = *card* (*RP M2 s vs xs V′′*) + (∑ *c*∈*S*. *card* (*RP M2 c vs xs V′′*))
    by (*meson insert.hyps*(*1*) *insert.hyps*(*2*) *sum.insert*)
  then show *?thesis*
    using ‹*card* (*RP M2 s vs xs V′′*)
        = *card* (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)›
      ‹*card* (⋃ *s*∈*insert s S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
        = *card* (⋃ *s*∈*S*. ⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)
        + *card* (⋃ *a*∈*RP M2 s vs xs V′′*. *io-targets M1* (*initial M1*) *a*)› *applied-IH*
    by *presburger*

qed
qed
qed

lemma *LB-count-helper-LB1* :
assumes $(vs\ @\ xs) \in L\ M2 \cap L\ M1$
and *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and *test-tools M2 M1 FAIL PM V* Ω
and $V'' \in$ *Perm V M1*
and *Prereq M2 M1 vs xs T S* Ω V''
and $\neg$ *Rep-Pre M2 M1 vs xs*
and $\neg$ *Rep-Cov M2 M1* V'' *vs xs*
shows (*sum* (λ *s* . *card* (*RP M2 s vs xs* V'')) *S*) $\leq$ *card* (*nodes M1*)
proof −
have ($\bigcup s{\in}S$. *UNION* (*RP M2 s vs xs* V'') (*io-targets M1* (*initial M1*))) $\subseteq$ *nodes M1*
by *blast*
moreover have *finite* (*nodes M1*)
using *assms*(*2*) *OFSM-props*(*1*) **unfolding** *well-formed.simps finite-FSM.simps* **by** *simp*
ultimately have *card* ($\bigcup s{\in}S$. *UNION* (*RP M2 s vs xs* V'') (*io-targets M1* (*initial M1*)))
$\leq$ *card* (*nodes M1*)
by (*meson card-mono*)

moreover have ($\sum s{\in}S$. *card* (*RP M2 s vs xs* V''))
= *card* ($\bigcup s{\in}S$. *UNION* (*RP M2 s vs xs* V'') (*io-targets M1* (*initial M1*)))
using *LB-count-helper-RP-disjoint-M1-union*[*OF assms*]
by *linarith*

ultimately show *?thesis*
by *linarith*
qed

lemma *LB-count-helper-D-states* :
assumes *observable M*
and *RS* $\in$ (*D M T* Ω)
obtains *q*
where *q* $\in$ *nodes M* $\wedge$ *RS* = *IO-set M q* Ω
proof −
have *RS* $\in$ *image* (λ *io* . *B M io* Ω) (LS_{in} *M* (*initial M*) *T*)
using *assms* **by** *simp*
then obtain *io* **where** *RS* = *B M io* Ω *io* $\in$ LS_{in} *M* (*initial M*) *T*
by *blast*
then have *io* $\in$ *language-state M* (*initial M*)
using *language-state-for-inputs-in-language-state*[*of M initial M T*] **by** *blast*
then obtain *q* **where** {*q*} = *io-targets M* (*initial M*) *io*
by (*metis assms*(*1*) *io-targets-observable-singleton-ob*)
then have *B M io* Ω = $\bigcup$ (*image* (λ *s* . *IO-set M s* Ω) {*q*})
by *simp*
then have *B M io* Ω = *IO-set M q* Ω
by *simp*
then have *RS* = *IO-set M q* Ω **using** ‹*RS* = *B M io* Ω›
by *simp*
moreover have *q* $\in$ *nodes M* **using** ‹{*q*} = *io-targets M* (*initial M*) *io*›
by (*metis FSM.nodes.initial insertI1 io-targets-nodes*)
ultimately show *?thesis*
using *that* **by** *simp*
qed

lemma *LB-count-helper-LB2* :
assumes *observable M1*
and $IO\text{-}set\ M1\ q\ \Omega \in (D\ M1\ T\ \Omega) - \{B\ M1\ xs'\ \Omega \mid xs'\ s'\ .\ s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\}$
shows $q \notin (\bigcup\ (image\ (\lambda\ s\ .\ \bigcup\ (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s\ vs\ xs\ V'')))\ S))$
proof
assume $q \in (\bigcup s{\in}S.\ UNION\ (RP\ M2\ s\ vs\ xs\ V'')\ (io\text{-}targets\ M1\ (initial\ M1)))$
then obtain s' **where** $s' \in S$ $q \in (\bigcup\ (image\ (io\text{-}targets\ M1\ (initial\ M1))\ (RP\ M2\ s'\ vs\ xs\ V'')))$
by *blast*
then obtain xs' **where** $q \in io\text{-}targets\ M1\ (initial\ M1)\ xs'$ $xs' \in RP\ M2\ s'\ vs\ xs\ V''$
by *blast*
then have $\{q\} = io\text{-}targets\ M1\ (initial\ M1)\ xs'$
by (*metis assms*(*1*) *observable-io-target-is-singleton*)
then have $B\ M1\ xs'\ \Omega = \bigcup\ (image\ (\lambda\ s\ .\ IO\text{-}set\ M1\ s\ \Omega)\ \{q\})$
by *simp*
then have $B\ M1\ xs'\ \Omega = IO\text{-}set\ M1\ q\ \Omega$
by *simp*
moreover have $B\ M1\ xs'\ \Omega \in \{B\ M1\ xs'\ \Omega \mid xs'\ s'\ .\ s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\}$
using ‹$s' \in S$› ‹$xs' \in RP\ M2\ s'\ vs\ xs\ V''$› **by** *blast*
ultimately have $IO\text{-}set\ M1\ q\ \Omega \in \{B\ M1\ xs'\ \Omega \mid xs'\ s'\ .\ s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\}$
by *blast*
moreover have $IO\text{-}set\ M1\ q\ \Omega \notin \{B\ M1\ xs'\ \Omega \mid xs'\ s'\ .\ s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\}$
using *assms*(*2*) **by** *blast*
ultimately show *False*
by *simp*
qed

4.7 Validity of the result of LB constituting a lower bound

lemma *LB-count* :
assumes $(vs\ @\ xs) \in L\ M1$
and *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and $test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
and $V'' \in Perm\ V\ M1$
and $Prereq\ M2\ M1\ vs\ xs\ T\ S\ \Omega\ V''$
and $\neg\ Rep\text{-}Pre\ M2\ M1\ vs\ xs$
and $\neg\ Rep\text{-}Cov\ M2\ M1\ V''\ vs\ xs$
shows $LB\ M2\ M1\ vs\ xs\ T\ S\ \Omega\ V'' \leq |M1|$
proof −

let $?D = D\ M1\ T\ \Omega$
let $?B = \{B\ M1\ xs'\ \Omega \mid xs'\ s'\ .\ s' \in S \wedge xs' \in RP\ M2\ s'\ vs\ xs\ V''\}$
let $?DB = ?D - ?B$
let $?RP = \bigcup s{\in}S.\ \bigcup a{\in}RP\ M2\ s\ vs\ xs\ V''.\ io\text{-}targets\ M1\ (initial\ M1)\ a$

have *finite* (*nodes M1*)
using *OFSM-props*[*OF assms*(*2*)] **unfolding** *well-formed.simps finite-FSM.simps* **by** *simp*
then have *finite ?D*
using *OFSM-props*[*OF assms*(*2*)] *assms*(*7*) $D\text{-}bound[of\ M1\ T\ \Omega]$ **unfolding** *Prereq.simps* **by** *linarith*
then have *finite ?DB*
by *simp*

— Proof sketch: Construct a function f (via induction) that maps each response set in ?DB to some state that produces that response set. This is then used to show that each response sets in ?DB indicates the existence of a distinct state in M1 not reached via the RP-sequences.

have *states-f* : $\bigwedge\ DB'\ .\ DB' \subseteq ?DB \Longrightarrow \exists\ f\ .\ inj\text{-}on\ f\ DB'$
$\wedge\ image\ f\ DB' \subseteq (nodes\ M1) - ?RP$
$\wedge\ (\forall\ RS \in DB'\ .\ IO\text{-}set\ M1\ (f\ RS)\ \Omega = RS)$
proof −
fix DB' **assume** $DB' \subseteq ?DB$
have *finite* DB'
proof (*rule ccontr*)
assume *infinite* DB'

```
    have infinite ?DB
      using infinite-super[OF ‹DB' ⊆ ?DB› ‹infinite DB'› ] by simp
    then show False
      using ‹finite ?DB› by simp
  qed
  then show ∃ f . inj-on f DB' ∧ image f DB' ⊆ (nodes M1) − ?RP
                          ∧ (∀ RS ∈ DB' . IO-set M1 (f RS) Ω = RS)
  using assms ‹DB' ⊆ ?DB› proof (induction DB')
    case empty
    show ?case by simp
  next
    case (insert RS DB')

    have DB' ⊆ ?DB
      using insert.prems(10) by blast
    obtain f' where inj-on f' DB'
                    image f' DB' ⊆ (nodes M1) − ?RP
                    ∀ RS ∈ DB' . IO-set M1 (f' RS) Ω = RS
      using insert.IH[OF insert.prems(1−9) ‹DB' ⊆ ?DB›]
      by blast

    have RS ∈ D M1 T Ω
      using insert.prems(10) by blast
    obtain q where q ∈ nodes M1 RS = IO-set M1 q Ω
      using insert.prems(2)  LB-count-helper-D-states[OF - ‹RS ∈ D M1 T Ω›]
      by blast
    then have IO-set M1 q Ω ∈ ?DB
      using insert.prems(10) by blast

    have q ∉ ?RP
      using insert.prems(2) LB-count-helper-LB2[OF - ‹IO-set M1 q Ω ∈ ?DB›]
      by blast

    let ?f = f'(RS := q)
    have inj-on ?f (insert RS DB')
    proof
      have ?f RS ∉ ?f ‘ (DB' − {RS})
      proof
        assume ?f RS ∈ ?f ‘ (DB' − {RS})
        then have q ∈ ?f ‘ (DB' − {RS}) by auto
        have RS ∈ DB'
        proof −
          have ∀ P c f. ∃ Pa. ((c::'c) ∉ f ‘ P ∨ (Pa::('a × 'b) list set) ∈ P)
                          ∧ (c ∉ f ‘ P ∨ f Pa = c)
            by auto
          moreover
          { assume q ∉ f' ‘ DB'
            moreover
            { assume q ∉ f'(RS := q) ‘ DB'
              then have ?thesis
                using ‹q ∈ f'(RS := q) ‘ (DB' − {RS})› by blast }
            ultimately have ?thesis
              by (metis fun-upd-image) }
          ultimately show ?thesis
            by (metis (no-types) ‹RS = IO-set M1 q Ω› ‹∀ RS∈DB'. IO-set M1 (f' RS) Ω = RS›)
        qed
        then show False using insert.hyps(2) by simp
      qed
      then show inj-on ?f DB' ∧ ?f RS ∉ ?f ‘ (DB' − {RS})
        using ‹inj-on f' DB'› inj-on-fun-updI by fastforce
    qed
    moreover have image ?f (insert RS DB') ⊆ (nodes M1) − ?RP
    proof −
      have image ?f {RS} = {q} by simp
      then have image ?f {RS} ⊆ (nodes M1) − ?RP
        using ‹q ∈ nodes M1› ‹q ∉ ?RP› by auto
```

moreover have *image ?f (insert RS DB′) = image ?f {RS} ∪ image ?f DB′*
by *auto*
ultimately show *?thesis*
by *(metis (no-types, lifting) ‹image f′ DB′ ⊆ (nodes M1) − ?RP› fun-upd-other image-cong image-insert insert.hyps(2) insert-subset)*
qed
moreover have ∀ *RS ∈ (insert RS DB′) . IO-set M1 (?f RS) Ω = RS*
using *‹RS = IO-set M1 q Ω› ‹∀ RS∈DB′. IO-set M1 (f′ RS) Ω = RS›* **by** *auto*

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

have *?DB ⊆ ?DB*
by *simp*
obtain *f* **where** *inj-on f ?DB image f ?DB ⊆ (nodes M1) − ?RP*
using *states-f[OF ‹?DB ⊆ ?DB›]* **by** *blast*
have *finite (nodes M1 − ?RP)*
using *‹finite (nodes M1)›* **by** *simp*
have *card ?DB ≤ card (nodes M1 − ?RP)*
using *card-inj-on-le[OF ‹inj-on f ?DB› ‹image f ?DB ⊆ (nodes M1) − ?RP› ‹finite (nodes M1 − ?RP)›]*
by *assumption*

have *?RP ⊆ nodes M1*
by *blast*
then have *card (nodes M1 − ?RP) = card (nodes M1) − card ?RP*
by *(meson ‹finite (nodes M1)› card-Diff-subset infinite-subset)*
then have *card ?DB ≤ card (nodes M1) − card ?RP*
using *‹card ?DB ≤ card (nodes M1 − ?RP)›* **by** *linarith*

have *vs @ xs ∈ L M2 ∩ L M1*
using *assms(7)* **by** *simp*
have *(sum (λ s . card (RP M2 s vs xs V′′)) S) = card ?RP*
using *LB-count-helper-RP-disjoint-M1-union[OF ‹vs @ xs ∈ L M2 ∩ L M1› assms(2−9)]* **by** *simp*
moreover have *card ?RP ≤ card (nodes M1)*
using *card-mono[OF ‹finite (nodes M1)› ‹?RP ⊆ nodes M1›]* **by** *assumption*
ultimately show *?thesis*
unfolding *LB.simps* **using** *‹card ?DB ≤ card (nodes M1) − card ?RP›*
by *linarith*
qed

lemma *contradiction-via-LB* :
assumes *(vs @ xs) ∈ L M1*
and *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and *test-tools M2 M1 FAIL PM V Ω*
and *V′′ ∈ Perm V M1*
and *Prereq M2 M1 vs xs T S Ω V′′*
and ¬ *Rep-Pre M2 M1 vs xs*
and ¬ *Rep-Cov M2 M1 V′′ vs xs*
and *LB M2 M1 vs xs T S Ω V′′ > m*
shows *False*
proof −
have *LB M2 M1 vs xs T S Ω V′′ ≤ card (nodes M1)*
using *LB-count[OF assms(1−9)]* **by** *assumption*
moreover have *card (nodes M1) ≤ m*
using *assms(4)* **by** *auto*
ultimately show *False*
using *assms(10)* **by** *linarith*
qed

end
theory *ASC-Suite*
imports *ASC-LB*
begin

5 Test suite generated by the Adaptive State Counting Algorithm

5.1 Maximum length contained prefix

```
fun mcp :: 'a list ⇒ 'a list set ⇒ 'a list ⇒ bool where
  mcp z W p = (prefix p z ∧ p ∈ W ∧
             (∀ p' . (prefix p' z ∧ p' ∈ W) ⟶ length p' ≤ length p))

lemma mcp-ex :
  assumes [] ∈ W
  and     finite W
obtains p
where mcp z W p
proof −
  let ?P = {p . prefix p z ∧ p ∈ W}
  let ?maxP = arg-max length (λ p . p ∈ ?P)

  have finite {p . prefix p z}
  proof −
    have {p . prefix p z} ⊆ image (λ i . take i z) (set [0 ..< Suc (length z)])
    proof
      fix p assume p ∈ {p . prefix p z}
      then obtain i where i ≤ length z ∧ p = take i z
        by (metis append-eq-conv-conj mem-Collect-eq prefix-def prefix-length-le)
      then have i < Suc (length z) ∧ p = take i z
        by simp
      then show p ∈ image (λ i . take i z) (set [0 ..< Suc (length z)])
        using atLeast-upt by blast
    qed
    then show ?thesis
      using finite-surj by blast
  qed
  then have finite ?P
    by simp

  have ?P ≠ {}
    using Nil-prefix assms(1) by blast

  have ∃ maxP ∈ ?P . ∀ p ∈ ?P . length p ≤ length maxP
  proof (rule ccontr)
    assume ¬(∃ maxP ∈ ?P . ∀ p ∈ ?P . length p ≤ length maxP)
    then have ∀ p ∈ ?P . ∃ p' ∈ ?P . length p < length p'
      by (meson not-less)
    then have ∀ l ∈ (image length ?P) . ∃ l' ∈ (image length ?P) . l < l'
      by auto

    then have infinite (image length ?P)
      by (metis (no-types, lifting) ‹?P ≠ {}› image-is-empty infinite-growing)
    then have infinite ?P
      by blast
    then show False
      using ‹finite ?P› by simp
  qed

  then obtain maxP where maxP ∈ ?P ∀ p ∈ ?P . length p ≤ length maxP
    by blast

  then have mcp z W maxP
    unfolding mcp.simps by blast
```

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


lemma mcp-unique :
  assumes mcp z W p
  and     mcp z W p'
shows p = p'
proof -
  have length p' ≤ length p
    using assms(1) assms(2) by auto
  moreover have length p ≤ length p'
    using assms(1) assms(2) by auto
  ultimately have length p' = length p
    by simp

  moreover have prefix p z
    using assms(1) by auto
  moreover have prefix p' z
    using assms(2) by auto
  ultimately show ?thesis
    by (metis append-eq-conv-conj prefixE)
qed

fun mcp' :: 'a list ⇒ 'a list set ⇒ 'a list where
  mcp' z W = (THE p . mcp z W p)

lemma mcp'-intro :
  assumes mcp z W p
shows mcp' z W = p
using assms mcp-unique by (metis mcp'.elims theI-unique)

lemma mcp-prefix-of-suffix :
  assumes mcp (vs@xs) V vs
  and     prefix xs' xs
shows mcp (vs@xs') V vs
proof (rule ccontr)
  assume ¬ mcp (vs @ xs') V vs
  then have ¬ (prefix vs (vs @ xs') ∧ vs ∈ V ∧
              (∀ p' . (prefix p' (vs @ xs') ∧ p' ∈ V) ⟶ length p' ≤ length vs))
    by auto
  then have ¬ (∀ p' . (prefix p' (vs @ xs') ∧ p' ∈ V) ⟶ length p' ≤ length vs)
    using assms(1) by auto
  then obtain vs' where vs' ∈ V ∧ prefix vs' (vs@xs) ∧ length vs < length vs'
    by (meson assms(2) leI prefix-append prefix-order.dual-order.trans)
  then have ¬ (mcp (vs@xs) V vs)
    by auto
  then show False
    using assms(1) by auto
qed

lemma minimal-sequence-to-failure-extending-mcp :
  assumes OFSM M1
  and     OFSM M2
  and     is-det-state-cover M2 V
  and     minimal-sequence-to-failure-extending V M1 M2 vs xs
shows mcp (map fst (vs@xs)) V (map fst vs)
proof (rule ccontr)
  assume ¬ mcp (map fst (vs @ xs)) V (map fst vs)
  moreover have prefix (map fst vs) (map fst (vs @ xs))
    by auto
  moreover have (map fst vs) ∈ V
    using mstfe-prefix-input-in-V assms(4) by auto
  ultimately obtain v' where prefix v' (map fst (vs @ xs))
                             v' ∈ V
```

```
                        length v' > length (map fst vs)
    using leI by auto

  then obtain x' where (map fst (vs@xs)) = v'@x'
    using prefixE by blast

  have vs@xs ∈ L M1 − L M2
    using assms(4) unfolding minimal-sequence-to-failure-extending.simps sequence-to-failure.simps
    by blast
  then have vs@xs ∈ L_in M1 {map fst (vs@xs)}
    by (meson DiffE insertI1 language-state-for-inputs-map-fst)
  have vs@xs ∈ L_in M1 {v'@x'}
    using ‹map fst (vs @ xs) = v' @ x'› ‹vs @ xs ∈ L_in M1 {map fst (vs @ xs)}›
    by presburger

  let ?vs' = take (length v') (vs@xs)
  let ?xs' = drop (length v') (vs@xs)

  have vs@xs = ?vs'@?xs'
    by (metis append-take-drop-id)

  have ?vs' ∈ L_in M1 V
    by (metis (no-types) DiffE ‹map fst (vs @ xs) = v' @ x'› ‹v' ∈ V› ‹vs @ xs ∈ L M1 − L M2›
        append-eq-conv-conj append-take-drop-id language-state-for-inputs-map-fst
        language-state-prefix take-map)

  have sequence-to-failure M1 M2 (?vs' @ ?xs')
    by (metis (full-types) ‹vs @ xs = take (length v') (vs @ xs) @ drop (length v') (vs @ xs)›
        assms(4) minimal-sequence-to-failure-extending.simps)

  have length ?xs' < length xs
    using ‹length (map fst vs) < length v'› ‹prefix v' (map fst (vs @ xs))›
          ‹vs @ xs = take (length v') (vs @ xs) @ drop (length v') (vs @ xs)› prefix-length-le
    by fastforce

  show False
    by (meson ‹length (drop (length v') (vs @ xs)) < length xs›
        ‹sequence-to-failure M1 M2 (take (length v') (vs @ xs) @ drop (length v') (vs @ xs))›
        ‹take (length v') (vs @ xs) ∈ L_in M1 V› assms(4)
        minimal-sequence-to-failure-extending.elims(2))

qed
```

5.2 Function N

Function `N` narrows the sets of reaction to the determinisitc state cover considered by the adaptive state counting algorithm to contain only relevant sequences. It is the main refinement of the original formulation of the algorithm as given in [2]. An example for the necessity for this refinement is given in [3].

```
fun N :: ('in × 'out) list ⇒ ('in, 'out, 'state) FSM ⇒ 'in list set ⇒ ('in × 'out) list set set
  where
  N io M V = { V'' ∈ Perm V M . (map fst (mcp' io V'')) = (mcp' (map fst io) V) }


lemma N-nonempty :
  assumes is-det-state-cover M2 V
  and     OFSM M1
  and     OFSM M2
  and     asc-fault-domain M2 M1 m
  and     io ∈ L M1
shows N io M1 V ≠ {}
proof −
  have [] ∈ V
    using assms(1) det-state-cover-empty by blast

  have inputs M1 = inputs M2
```

using *assms*(*4*) **by** *auto*

have *is-det-state-cover M2 V*
using *assms* **by** *auto*
moreover have *finite* (*nodes M2*)
using *assms*(*3*) **by** *auto*
moreover have *d-reachable M2* (*initial M2*) ⊆ *nodes M2*
by *auto*
ultimately have *finite V*
using *det-state-cover-card*[*of M2 V*]
by (*metis finite-if-finite-subsets-card-bdd infinite-subset is-det-state-cover.elims*(*2*) *surj-card-le*)

obtain *ioV* **where** *mcp* (*map fst io*) *V ioV*
using *mcp-ex*[*OF* ‹[] ∈ *V*› ‹*finite V*›] **by** *blast*
then have *ioV* ∈ *V*
by *auto*

— Proof sketch: - ioV uses only inputs of M2 - ioV uses only inputs of M1 - as M1 completely spec.: ex. reaction of M1 to ioV - this reaction is in some V”

obtain *q2* **where** *d-reaches M2* (*initial M2*) *ioV q2*
using *det-state-cover-d-reachable*[*OF assms*(*1*) ‹*ioV* ∈ *V*›] **by** *blast*
then obtain *ioV′ ioP* **where** *io-path* : *length ioV* = *length ioV′*
∧ *length ioV* = *length ioP*
∧ (*path M2* ((*ioV* || *ioV′*) || *ioP*) (*initial M2*))
∧ *target* ((*ioV* || *ioV′*) || *ioP*) (*initial M2*) = *q2*
by *auto*

have *well-formed M2*
using *assms* **by** *auto*

have *map fst* (*map fst* ((*ioV* || *ioV′*) || *ioP*)) = *ioV*
proof −
have *length* (*ioV* || *ioV′*) = *length ioP*
using *io-path* **by** *simp*
then show *?thesis*
using *io-path* **by** *auto*
qed
moreover have *set* (*map fst* (*map fst* ((*ioV* || *ioV′*) || *ioP*))) ⊆ *inputs M2*
using *path-input-containment*[*OF* ‹*well-formed M2*›, *of* (*ioV* || *ioV′*) || *ioP initial M2*] *io-path*
by *linarith*
ultimately have *set ioV* ⊆ *inputs M2*
by *presburger*

then have *set ioV* ⊆ *inputs M1*
using *assms* **by** *auto*

then have L_{in} *M1* {*ioV*} ≠ {}
using *assms*(*2*) *language-state-for-inputs-nonempty* **by** (*metis FSM.nodes.initial*)

have *prefix ioV* (*map fst io*)
using ‹*mcp* (*map fst io*) *V ioV*› *mcp.simps* **by** *blast*
then have *length ioV* ≤ *length* (*map fst io*)
using *prefix-length-le* **by** *blast*
then have *length ioV* ≤ *length io*
by *auto*

have (*map fst io* || *map snd io*) ∈ *L M1*
using *assms*(*5*) **by** *auto*
moreover have *length* (*map fst io*) = *length* (*map snd io*)
by *auto*
ultimately have (*map fst io* || *map snd io*)

∈ *language-state-for-input M1* (*initial M1*) (*map fst io*)
unfolding *language-state-def*
by (*metis* (*mono-tags*, *lifting*) ‹*map fst io* || *map snd io* ∈ *L M1*›
language-state-for-input.simps mem-Collect-eq)

have *ioV* = *take* (*length ioV*) (*map fst io*)
by (*metis* (*no-types*) ‹*prefix ioV* (*map fst io*)› *append-eq-conv-conj prefixE*)

then have *take* (*length ioV*) *io* ∈ *language-state-for-input M1* (*initial M1*) *ioV*
using *language-state-for-input-take*
by (*metis* ‹*map fst io* || *map snd io* ∈ *language-state-for-input M1* (*initial M1*) (*map fst io*)›
zip-map-fst-snd)

then obtain *V''* **where** *V''* ∈ *Perm V M1 take* (*length ioV*) *io* ∈ *V''*
using *perm-elem*[*OF assms*(*1*−*3*) ‹*inputs M1* = *inputs M2*› ‹*ioV* ∈ *V*›] **by** *blast*

have *ioV* = *mcp'* (*map fst io*) *V*
using ‹*mcp* (*map fst io*) *V ioV*› *mcp'-intro* **by** *blast*

have *map fst* (*take* (*length ioV*) *io*) = *ioV*
by (*metis* ‹*ioV* = *take* (*length ioV*) (*map fst io*)› *take-map*)

obtain *mcpV''* **where** *mcp io V'' mcpV''*
by (*meson* ‹*V''* ∈ *Perm V M1*› ‹*well-formed M2*› *assms*(*1*) *mcp-ex perm-elem-finite perm-empty*)

have *map fst mcpV''* ∈ *V* **using** *perm-inputs*
using ‹*V''* ∈ *Perm V M1*› ‹*mcp io V'' mcpV''*› *mcp.simps* **by** *blast*

have *map fst mcpV''* = *ioV*
by (*metis* (*no-types*) ‹*map fst* (*take* (*length ioV*) *io*) = *ioV*› ‹*map fst mcpV''* ∈ *V*›
‹*mcp* (*map fst io*) *V ioV*› ‹*mcp io V'' mcpV''*› ‹*take* (*length ioV*) *io* ∈ *V''*›
map-mono-prefix mcp.elims(*2*) *prefix-length-prefix prefix-order.dual-order.antisym*
take-is-prefix)

have *map fst* (*mcp' io V''*) = *mcp'* (*map fst io*) *V*
using ‹*ioV* = *mcp'* (*map fst io*) *V*› ‹*map fst mcpV''* = *ioV*› ‹*mcp io V'' mcpV''*› *mcp'-intro*
by *blast*

then show *?thesis*
using ‹*V''* ∈ *Perm V M1*› **by** *fastforce*
qed

lemma *N-mcp-prefix* :
assumes *map fst vs* = *mcp'* (*map fst* (*vs@xs*)) *V*
and *V''* ∈ *N* (*vs@xs*) *M1 V*
and *is-det-state-cover M2 V*
and *well-formed M2*
and *finite V*
shows *vs* ∈ *V'' vs* = *mcp'* (*vs@xs*) *V''*
proof −
have *map fst* (*mcp'* (*vs@xs*) *V''*) = *mcp'* (*map fst* (*vs@xs*)) *V*
using *assms*(*2*) **by** *auto*
then have *map fst* (*mcp'* (*vs@xs*) *V''*) = *map fst vs*
using *assms*(*1*) **by** *presburger*
then have *length* (*mcp'* (*vs@xs*) *V''*) = *length vs*
by (*metis length-map*)

have [] ∈ *V''*
using *perm-empty*[*OF assms*(*3*)] *N.simps assms*(*2*) **by** *blast*
moreover have *finite V''*
using *perm-elem-finite*[*OF assms*(*3*,*4*)] *N.simps assms*(*2*) **by** *blast*
ultimately obtain *p* **where** *mcp* (*vs@xs*) *V'' p*
using *mcp-ex* **by** *auto*
then have *mcp'* (*vs@xs*) *V''* = *p*

using *mcp'-intro* **by** *simp*

then have *prefix* (*mcp'* (*vs@xs*) *V''*) (*vs@xs*)
unfolding *mcp'.simps mcp.simps*
using ‹*mcp* (*vs* @ *xs*) *V'' p*› *mcp.elims*(*2*) **by** *blast*
then show *vs* = *mcp'* (*vs@xs*) *V''*
by (*metis* ‹*length* (*mcp'* (*vs* @ *xs*) *V''*) = *length vs*› *append-eq-append-conv prefix-def*)

show *vs* ∈ *V''*
using ‹*mcp* (*vs* @ *xs*) *V'' p*› ‹*mcp'* (*vs* @ *xs*) *V''* = *p*› ‹*vs* = *mcp'* (*vs* @ *xs*) *V''*›
by *auto*
qed

5.3 Functions TS, C, RM

Function TTS defines the calculation of the test suite used by the adaptive state counting algorithm in an iterative way. It is defined using the three functions `TS`, `C` and `RM` where `TS` represents the test suite calculated up to some iteration, `C` contains the sequences considered for extension in some iteration, and `RM` contains the sequences of the corresponding `C` result that are not to be extended, which we also call removed sequences.

```
abbreviation append-set :: 'a list set ⇒ 'a set ⇒ 'a list set where
  append-set T X ≡ {xs @ [x] | xs x . xs ∈ T ∧ x ∈ X}

abbreviation append-sets :: 'a list set ⇒ 'a list set ⇒ 'a list set where
  append-sets T X ≡ {xs @ xs' | xs xs' . xs ∈ T ∧ xs' ∈ X}

fun TS :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
          ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
          ⇒ 'in list set
and C  :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
          ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
          ⇒ 'in list set
and RM :: ('in, 'out, 'state1) FSM ⇒ ('in, 'out, 'state2) FSM
          ⇒ ('in, 'out) ATC set ⇒ 'in list set ⇒ nat ⇒ nat
          ⇒ 'in list set
  where
  RM M2 M1 Ω V m 0 = {} |
  TS M2 M1 Ω V m 0 = {} |
  TS M2 M1 Ω V m (Suc 0) = V |
  C M2 M1 Ω V m 0 = {} |
  C M2 M1 Ω V m (Suc 0) = V |
  RM M2 M1 Ω V m (Suc n) =
    {xs' ∈ C M2 M1 Ω V m (Suc n) .
      (¬ (L_in M1 {xs'} ⊆ L_in M2 {xs'}))
      ∨ (∀ io ∈ L_in M1 {xs'} .
          ∃ V'' ∈ N io M1 V .
            ∃ S1 .
              ∃ vs xs .
                io = (vs@xs)
                ∧ mcp (vs@xs) V'' vs
                ∧ S1 ⊆ nodes M2
                ∧ (∀ s1 ∈ S1 . ∀ s2 ∈ S1 .
                  s1 ≠ s2 ⟶
                    (∀ io1 ∈ RP M2 s1 vs xs V'' .
                       ∀ io2 ∈ RP M2 s2 vs xs V'' .
                         B M1 io1 Ω ≠ B M1 io2 Ω ))
                ∧ m < LB M2 M1 vs xs (TS M2 M1 Ω V m n ∪ V) S1 Ω V'')} |
  C M2 M1 Ω V m (Suc n) =
    (append-set ((C M2 M1 Ω V m n) − (RM M2 M1 Ω V m n)) (inputs M2))
    − (TS M2 M1 Ω V m n) |
  TS M2 M1 Ω V m (Suc n) =
    (TS M2 M1 Ω V m n) ∪ (C M2 M1 Ω V m (Suc n))
```

```
abbreviation lists-of-length :: 'a set ⇒ nat ⇒ 'a list set where
  lists-of-length X n ≡ {xs . length xs = n ∧ set xs ⊆ X}

lemma append-lists-of-length-alt-def :
  append-sets T (lists-of-length X (Suc n)) = append-set (append-sets T (lists-of-length X n)) X
proof
  show append-sets T (lists-of-length X (Suc n))
        ⊆ append-set (append-sets T (lists-of-length X n)) X
  proof
    fix tx assume tx ∈ append-sets T (lists-of-length X (Suc n))
    then obtain t x where t@x = tx t ∈ T length x = Suc n set x ⊆ X
      by blast
    then have x ≠ [] length (butlast x) = n
      by auto
    moreover have set (butlast x) ⊆ X
      using ‹set x ⊆ X› by (meson dual-order.trans prefixeq-butlast set-mono-prefix)
    ultimately have butlast x ∈ lists-of-length X n
      by auto
    then have t@(butlast x) ∈ append-sets T (lists-of-length X n)
      using ‹t ∈ T› by blast
    moreover have last x ∈ X
      using ‹set x ⊆ X› ‹x ≠ []› by auto
    ultimately have t@(butlast x)@[last x] ∈ append-set (append-sets T (lists-of-length X n)) X
      by auto
    then show tx ∈ append-set (append-sets T (lists-of-length X n)) X
      using ‹t@x = tx› by (simp add: ‹x ≠ []›)
  qed
  show append-set (append-sets T (lists-of-length X n)) X
        ⊆ append-sets T (lists-of-length X (Suc n))
  proof
    fix tx assume tx ∈ append-set (append-sets T (lists-of-length X n)) X
    then obtain tx' x where tx = tx' @ [x] tx' ∈ append-sets T (lists-of-length X n) x ∈ X
      by blast
    then obtain tx'' x' where tx''@x' = tx' tx'' ∈ T length x' = n set x' ⊆ X
      by blast
    then have tx''@x'@[x] = tx
      by (simp add: ‹tx = tx' @ [x]›)
    moreover have tx'' ∈ T
      by (meson ‹tx'' ∈ T›)
    moreover have length (x'@[x]) = Suc n
      by (simp add: ‹length x' = n›)
    moreover have set (x'@[x]) ⊆ X
      by (simp add: ‹set x' ⊆ X› ‹x ∈ X›)
    ultimately show tx ∈ append-sets T (lists-of-length X (Suc n))
      by blast
  qed
qed
```

5.4 Basic properties of the test suite calculation functions

```
lemma C-step :
  assumes n > 0
  shows C M2 M1 Ω V m (Suc n) ⊆ (append-set (C M2 M1 Ω V m n) (inputs M2)) − C M2 M1 Ω V m n
proof −
  let ?TS = λ n . TS M2 M1 Ω V m n
  let ?C = λ n . C M2 M1 Ω V m n
  let ?RM = λ n . RM M2 M1 Ω V m n

  obtain k where n-def[simp] : n = Suc k
    using assms not0-implies-Suc by blast

  have ?C (Suc n) = (append-set (?C n − ?RM n) (inputs M2)) − ?TS n
    using n-def C.simps(3) by blast
  moreover have ?C n ⊆ ?TS n
    using n-def by (metis C.simps(2) TS.elims UnCI assms neq0-conv subsetI)
  ultimately show ?C (Suc n) ⊆ append-set (?C n) (inputs M2) − ?C n
```

by *blast*
qed

lemma *C-extension* :
C M2 M1 Ω V m (*Suc n*) ⊆ *append-sets V* (*lists-of-length* (*inputs M2*) *n*)
proof (*induction n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc k*)

let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

have *0* < *Suc k* **by** *simp*
have *?C* (*Suc* (*Suc k*)) ⊆ (*append-set* (*?C* (*Suc k*)) (*inputs M2*)) − *?C* (*Suc k*)
using *C-step*[*OF* ‹*0* < *Suc k*›] **by** *blast*

then have *?C* (*Suc* (*Suc k*)) ⊆ *append-set* (*?C* (*Suc k*)) (*inputs M2*)
by *blast*
moreover have *append-set* (*?C* (*Suc k*)) (*inputs M2*)
⊆ *append-set* (*append-sets V* (*lists-of-length* (*inputs M2*) *k*)) (*inputs M2*)
using *Suc.IH* **by** *auto*
ultimately have *I-Step* :
?C (*Suc* (*Suc k*)) ⊆ *append-set* (*append-sets V* (*lists-of-length* (*inputs M2*) *k*)) (*inputs M2*)
by (*meson order-trans*)

show *?case*
using *append-lists-of-length-alt-def*[*symmetric*, *of V k inputs M2*] *I-Step*
by *presburger*
qed

lemma *TS-union* :
shows *TS M2 M1 Ω V m i* = (⋃ *j* ∈ (*set* [*0*..<*Suc i*]) . *C M2 M1 Ω V m j*)
proof (*induction i*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc i*)

let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

have *?TS* (*Suc i*) = *?TS i* ∪ *?C* (*Suc i*)
by (*metis* (*no-types*) *C.simps*(*2*) *TS.simps*(*1*) *TS.simps*(*2*) *TS.simps*(*3*) *not0-implies-Suc sup-bot.right-neutral sup-commute*)
then have *?TS* (*Suc i*) = (⋃ *j* ∈ (*set* [*0*..<*Suc i*]) . *?C j*) ∪ *?C* (*Suc i*)
using *Suc.IH* **by** *simp*
then show *?case*
by *auto*
qed

lemma *C-disj-le-gz* :
assumes *i* ≤ *j*
and *0* < *i*
shows *C M2 M1 Ω V m i* ∩ *C M2 M1 Ω V m* (*Suc j*) = {}
proof −
let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

have *Suc 0* < *Suc j*
  using *assms*(*1*−*2*) **by** *auto*
then obtain *k* **where** *Suc j* = *Suc* (*Suc k*)
  using *not0-implies-Suc* **by** *blast*
then have *?C* (*Suc j*) = (*append-set* (*?C j* − *?RM j*) (*inputs M2*)) − *?TS j*
  using *C.simps*(*3*) **by** *blast*
then have *?C* (*Suc j*) ∩ *?TS j* = {}
  by *blast*
moreover have *?C i* ⊆ *?TS j*
  using *assms*(*1*) *TS-union*[*of M2 M1* Ω *V m j*] **by** *fastforce*
ultimately show *?thesis*
  by *blast*
qed

lemma *C-disj-lt* :
  assumes *i* < *j*
shows *C M2 M1* Ω *V m i* ∩ *C M2 M1* Ω *V m j* = {}
proof (*cases i*)
  case *0*
  then show *?thesis* **by** *auto*
next
  case (*Suc k*)
  then show *?thesis*
    using *C-disj-le-gz*
    by (*metis assms gr-implies-not0 less-Suc-eq-le old.nat.exhaust zero-less-Suc*)
qed

lemma *C-disj* :
  assumes *i* ≠ *j*
shows *C M2 M1* Ω *V m i* ∩ *C M2 M1* Ω *V m j* = {}
  by (*metis C-disj-lt Int-commute antisym-conv3 assms*)

lemma *RM-subset* : *RM M2 M1* Ω *V m i* ⊆ *C M2 M1* Ω *V m i*
proof (*cases i*)
  case *0*
  then show *?thesis* **by** *auto*
next
  case (*Suc n*)
  then show *?thesis*
    using *RM.simps*(*2*) **by** *blast*
qed

lemma *RM-disj* :
  assumes *i* ≤ *j*
  and *0* < *i*
shows *RM M2 M1* Ω *V m i* ∩ *RM M2 M1* Ω *V m* (*Suc j*) = {}
proof −
  let *?TS* = λ *n* . *TS M2 M1* Ω *V m n*
  let *?C* = λ *n* . *C M2 M1* Ω *V m n*
  let *?RM* = λ *n* . *RM M2 M1* Ω *V m n*

  have *?RM i* ⊆ *?C i ?RM* (*Suc j*) ⊆ *?C* (*Suc j*)
    using *RM-subset* **by** *blast*+
  moreover have *?C i* ∩ *?C* (*Suc j*) = {}
    using *C-disj-le-gz*[*OF assms*] **by** *assumption*
  ultimately show *?thesis*
    by *blast*
qed

lemma *T-extension* :
  assumes $n > 0$
  shows *TS M2 M1* Ω *V m* (*Suc n*) − *TS M2 M1* Ω *V m n*
        ⊆ (*append-set* (*TS M2 M1* Ω *V m n*) (*inputs M2*)) − *TS M2 M1* Ω *V m n*
proof −
  let *?TS* = λ *n* . *TS M2 M1* Ω *V m n*
  let *?C* = λ *n* . *C M2 M1* Ω *V m n*
  let *?RM* = λ *n* . *RM M2 M1* Ω *V m n*

  obtain *k* **where** *n-def*[*simp*] : *n* = *Suc k*
    using *assms not0-implies-Suc*
    by *blast*

  have *?C* (*Suc n*) = (*append-set* (*?C n* − *?RM n*) (*inputs M2*)) − *?TS n*
    using *n-def* **using** *C.simps*(*3*) **by** *blast*
  then have *?C* (*Suc n*) ⊆ *append-set* (*?C n*) (*inputs M2*) − *?TS n*
    by *blast*
  moreover have *?C n* ⊆ *?TS n* **using** *TS-union*[*of M2 M1* Ω *V m n*]
    by *fastforce*
  ultimately have *?C* (*Suc n*) ⊆ *append-set* (*?TS n*) (*inputs M2*) − *?TS n*
    by *blast*
  moreover have *?TS* (*Suc n*) − *?TS n* ⊆ *?C* (*Suc n*)
    using *TS.simps*(*3*)[*of M2 M1* Ω *V m k*] **using** *n-def* **by** *blast*
  ultimately show *?thesis*
    by *blast*
qed

lemma *append-set-prefix* :
  assumes *xs* ∈ *append-set T X*
  shows *butlast xs* ∈ *T*
  using *assms* **by** *auto*

lemma *C-subset* : *C M2 M1* Ω *V m i* ⊆ *TS M2 M1* Ω *V m i*
  by (*simp add*: *TS-union*)

lemma *TS-subset* :
  assumes $i \leq j$
  shows *TS M2 M1* Ω *V m i* ⊆ *TS M2 M1* Ω *V m j*
proof −
  have *TS M2 M1* Ω *V m i* = (⋃ *k* ∈ (*set* [*0..<Suc i*]) . *C M2 M1* Ω *V m k*)
      TS M2 M1 Ω *V m j* = (⋃ *k* ∈ (*set* [*0..<Suc j*]) . *C M2 M1* Ω *V m k*)
    using *TS-union* **by** *assumption+*
  moreover have *set* [*0..<Suc i*] ⊆ *set* [*0..<Suc j*]
    using *assms* **by** *auto*
  ultimately show *?thesis*
    by *blast*
qed

lemma *C-immediate-prefix-containment* :
  assumes *vs*@*xs* ∈ *C M2 M1* Ω *V m* (*Suc* (*Suc i*))
  and   *xs* ≠ []
shows *vs*@(*butlast xs*) ∈ *C M2 M1* Ω *V m* (*Suc i*) − *RM M2 M1* Ω *V m* (*Suc i*)
proof (*rule ccontr*)
  let *?TS* = λ *n* . *TS M2 M1* Ω *V m n*
  let *?C* = λ *n* . *C M2 M1* Ω *V m n*
  let *?RM* = λ *n* . *RM M2 M1* Ω *V m n*

  assume *vs* @ *butlast xs* ∉ *C M2 M1* Ω *V m* (*Suc i*) − *RM M2 M1* Ω *V m* (*Suc i*)

  have *?C* (*Suc* (*Suc i*)) ⊆ *append-set* (*?C* (*Suc i*) − *?RM* (*Suc i*)) (*inputs M2*)
    using *C.simps*(*3*) **by** *blast*
  then have *?C* (*Suc* (*Suc i*)) ⊆ *append-set* (*?C* (*Suc i*) − *?RM* (*Suc i*)) *UNIV*

```
    by blast
  moreover have vs @ xs ∉ append-set (?C (Suc i) − ?RM (Suc i)) UNIV
  proof −
    have ∀ as a. vs @ xs ≠ as @ [a]
                ∨ as ∉ C M2 M1 Ω V m (Suc i) − RM M2 M1 Ω V m (Suc i)
                ∨ a ∉ UNIV
      by (metis ‹vs @ butlast xs ∉ C M2 M1 Ω V m (Suc i) − RM M2 M1 Ω V m (Suc i)›
          assms(2) butlast-append butlast-snoc)
    then show ?thesis
      by blast
  qed
  ultimately have vs @ xs ∉ ?C (Suc (Suc i))
    by blast
  then show False
    using assms(1) by blast
qed



lemma TS-immediate-prefix-containment :
  assumes vs@xs ∈ TS M2 M1 Ω V m i
  and     mcp (vs@xs) V vs
  and     0 < i
shows vs@(butlast xs) ∈ TS M2 M1 Ω V m i
proof −
  let ?TS = λ n . TS M2 M1 Ω V m n
  let ?C = λ n . C M2 M1 Ω V m n
  let ?RM = λ n . RM M2 M1 Ω V m n

  obtain j where j-def : j ≤ i ∧ vs@xs ∈ ?C j
    using assms(1)  TS-union[where i=i]
  proof −
    assume a1: ⋀j. j ≤ i ∧ vs @ xs ∈ C M2 M1 Ω V m j ⟹ thesis
    obtain nn :: nat set ⇒ (nat ⇒ 'a list set) ⇒ 'a list ⇒ nat where
     f2: ∀ x0 x1 x2. (∃ v3. v3 ∈ x0 ∧ x2 ∈ x1 v3) = (nn x0 x1 x2 ∈ x0 ∧ x2 ∈ x1 (nn x0 x1 x2))
      by moura
    have vs @ xs ∈ UNION (set [0..<Suc i]) (C M2 M1 Ω V m)
      by (metis ‹⋀Ω V T S M2 M1. TS M2 M1 Ω V m i = (⋃j∈set [0..<Suc i]. C M2 M1 Ω V m j)›
          ‹vs @ xs ∈ TS M2 M1 Ω V m i›)
    then have nn (set [0..<Suc i]) (C M2 M1 Ω V m) (vs @ xs) ∈ set [0..<Suc i]
                ∧ vs @ xs ∈ C M2 M1 Ω V m (nn (set [0..<Suc i]) (C M2 M1 Ω V m) (vs @ xs))
      using f2 by blast
    then show ?thesis
      using a1 by (metis (no-types) atLeastLessThan-iff leD not-less-eq-eq set-upt)
  qed

  show ?thesis
  proof (cases j)
    case 0
    then have ?C j = {}
      by auto
    moreover have vs@xs ∈ {}
      using j-def 0 by auto
    ultimately show ?thesis
      by auto
  next
    case (Suc k)
    then show ?thesis
    proof (cases k)
      case 0
      then have ?C j = V
        using Suc by auto
      then have vs@xs ∈ V
        using j-def by auto
      then have mcp (vs@xs) V (vs@xs)
        using assms(2) by auto
```

```
    then have vs@xs = vs
      using assms(2) mcp-unique by auto
    then have butlast xs = []
      by auto
    then show ?thesis
      using ‹vs @ xs = vs› assms(1) by auto
  next
    case (Suc n)
    assume j-assms : j = Suc k
                     k = Suc n
    then have ?C (Suc (Suc n)) = append-set (?C (Suc n) − ?RM (Suc n)) (inputs M2) − ?TS (Suc n)
      using C.simps(3) by blast
    then have ?C (Suc (Suc n)) ⊆ append-set (?C (Suc n)) (inputs M2)
      by blast

    have vs@xs ∈ ?C (Suc (Suc n))
      using j-assms j-def by blast

    have butlast (vs@xs) ∈ ?C (Suc n)
    proof −
      show ?thesis
        by (meson ‹?C (Suc (Suc n)) ⊆ append-set (?C (Suc n)) (inputs M2)›
            ‹vs @ xs ∈ ?C (Suc (Suc n))› append-set-prefix subsetCE)
    qed

    moreover have xs ≠ []
    proof −
      have 1 ≤ k
        using j-assms by auto
      then have ?C j ∩ ?C 1 = {}
        using C-disj-le-gz[of 1 k] j-assms(1) less-numeral-extra(1) by blast
      then have ?C j ∩ V = {}
        by auto
      then have vs@xs ∉ V
        using j-def by auto
      then show ?thesis
        using assms(2) by auto
    qed

    ultimately have vs@(butlast xs) ∈ ?C (Suc n)
      by (simp add: butlast-append)

    have Suc n < Suc j
      using j-assms by auto
    have ?C (Suc n) ⊆ ?TS j
      using TS-union[of M2 M1 Ω V m j] ‹Suc n < Suc j›
      by (metis UN-upper atLeast-upt lessThan-iff)


    have vs @ butlast xs ∈ TS M2 M1 Ω V m j
      using ‹vs@(butlast xs) ∈ ?C (Suc n)› ‹?C (Suc n) ⊆ ?TS j› j-def
      by auto
    then show ?thesis
      using j-def TS-subset[of j i]
      by blast
  qed
 qed
qed



lemma TS-prefix-containment :
 assumes vs@xs ∈ TS M2 M1 Ω V m i
 and     mcp (vs@xs) V vs
 and     prefix xs' xs
shows vs@xs' ∈ TS M2 M1 Ω V m i
```

— Proof sketch: Perform induction on length difference, as from each prefix it is possible to deduce the desired property for the prefix one element smaller than it via above results
using *assms* **proof** (*induction length xs − length xs′ arbitrary*: *xs′*)
  case *0*
  then have *xs* = *xs′*
    by (*metis append-Nil2 append-eq-conv-conj gr-implies-not0 length-drop length-greater-0-conv prefixE*)
  then show *?case*
    using *0* **by** *auto*
next
  case (*Suc k*)
  have *0* < *i*
    using *assms*(*1*) **using** *Suc.hyps*(*2*) *append-eq-append-conv assms*(*2*) **by** *auto*

  show *?case*
  proof (*cases xs′*)
    case *Nil*
    then show *?thesis*
      by (*metis* (*no-types*, *opaque-lifting*) ‹*0* < *i*› *TS.simps*(*2*) *TS-subset append-Nil2 assms*(*2*) *contra-subsetD leD mcp.elims*(*2*) *not-less-eq-eq*)
  next
    case (*Cons a list*)
    then show *?thesis*
    proof (*cases xs* = *xs′*)
      case *True*
      then show *?thesis*
        using *assms*(*1*) **by** *simp*
    next
      case *False*
      then obtain *xs′′* **where** *xs* = *xs′*@*xs′′*
        using *Suc.prems*(*3*) *prefixE* **by** *blast*
      then have *xs′′* ≠ []
        using *False* **by** *auto*
      then have *k* = *length xs* − *length* (*xs′* @ [*hd xs′′*])
        using ‹*xs* = *xs′*@*xs′′*› *Suc.hyps*(*2*) **by** *auto*
      moreover have *prefix* (*xs′* @ [*hd xs′′*]) *xs*
        using ‹*xs* = *xs′*@*xs′′*› ‹*xs′′* ≠ []›
        by (*metis Cons-prefix-Cons list.exhaust-sel prefix-code*(*1*) *same-prefix-prefix*)
      ultimately have *vs* @ (*xs′* @ [*hd xs′′*]) ∈ *TS M2 M1* Ω *V m i*
        using *Suc.hyps*(*1*)[*OF* - *Suc.prems*(*1*,*2*)] **by** *simp*

      have *mcp* (*vs* @ *xs′* @ [*hd xs′′*]) *V vs*
        using ‹*xs* = *xs′*@*xs′′*› ‹*xs′′* ≠ []› *assms*(*2*)
      proof −
        obtain *aas* :: *′a list* ⇒ *′a list set* ⇒ *′a list* ⇒ *′a list* **where**
          ∀ *x0 x1 x2*. (∃ *v3*. (*prefix v3 x2* ∧ *v3* ∈ *x1*) ∧ ¬ *length v3* ≤ *length x0*)
            = ((*prefix* (*aas x0 x1 x2*) *x2* ∧ *aas x0 x1 x2* ∈ *x1*)
              ∧ ¬ *length* (*aas x0 x1 x2*) ≤ *length x0*)
          by *moura*
        then have *f1*: ∀ *as A asa*. (¬ *mcp as A asa*
            ∨ *prefix asa as* ∧ *asa* ∈ *A* ∧ (∀ *asb*. (¬ *prefix asb as* ∨ *asb* ∉ *A*)
              ∨ *length asb* ≤ *length asa*))
          ∧ (*mcp as A asa*
            ∨ ¬ *prefix asa as*
            ∨ *asa* ∉ *A*
            ∨ (*prefix* (*aas asa A as*) *as* ∧ *aas asa A as* ∈ *A*)
              ∧ ¬ *length* (*aas asa A as*) ≤ *length asa*)
          by *auto*
        obtain *aasa* :: *′a list* ⇒ *′a list* ⇒ *′a list* **where**
          f2: ∀ *x0 x1*. (∃ *v2*. *x0* = *x1* @ *v2*) = (*x0* = *x1* @ *aasa x0 x1*)
          by *moura*
        then have *f3*: ([] @ [*hd xs′′*]) @ *aasa* (*xs′* @ *xs′′*) (*xs′* @ [*hd xs′′*])
            = ([] @ [*hd xs′′*]) @ *aasa* (([] @ [*hd xs′′*])
              @ *aasa* (*xs′* @ *xs′′*) (*xs′* @ [*hd xs′′*])) ([] @ [*hd xs′′*])
          by (*meson prefixE prefixI*)
        have *xs′* @ *xs′′* = (*xs′* @ [*hd xs′′*]) @ *aasa* (*xs′* @ *xs′′*) (*xs′* @ [*hd xs′′*])

```
        using f2 by (metis (no-types) ‹prefix (xs' @ [hd xs'']) xs› ‹xs = xs' @ xs''› prefixE)
      then have (vs @ (a # list) @ [hd xs'']) @ aasa (([] @ [hd xs''])
                    @ aasa (xs' @ xs'') (xs' @ [hd xs''])) ([] @ [hd xs''])
                 = vs @ xs
        using f3 by (simp add: ‹xs = xs' @ xs''› local.Cons)
      then have ¬ prefix (aas vs V (vs @ xs' @ [hd xs''])) (vs @ xs' @ [hd xs''])
                 ∨ aas vs V (vs @ xs' @ [hd xs'']) ∉ V
                 ∨ length (aas vs V (vs @ xs' @ [hd xs''])) ≤ length vs
        using f1 by (metis (no-types) ‹mcp (vs @ xs) V vs› local.Cons prefix-append)
      then show ?thesis
        using f1 by (meson ‹mcp (vs @ xs) V vs› prefixI)
    qed


    then have vs @ butlast (xs' @ [hd xs'']) ∈ TS M2 M1 Ω V m i
      using TS-immediate-prefix-containment
            [OF ‹vs @ (xs' @ [hd xs'']) ∈ TS M2 M1 Ω V m i› - ‹0 < i›]
      by simp

    moreover have xs' = butlast (xs' @ [hd xs''])
      using ‹xs'' ≠ []› by simp

    ultimately show ?thesis
      by simp
  qed
 qed
qed





lemma C-index :
  assumes vs @ xs ∈ C M2 M1 Ω V m i
  and     mcp (vs@xs) V vs
shows Suc (length xs) = i
using assms proof (induction xs arbitrary: i rule: rev-induct)
  case Nil
  then have vs @ [] ∈ C M2 M1 Ω V m 1
    by auto
  then have vs @ [] ∈ C M2 M1 Ω V m (Suc (length []))
    by simp

  show ?case
  proof (rule ccontr)
    assume Suc (length []) ≠ i
    moreover have vs @ [] ∈ C M2 M1 Ω V m i ∩ C M2 M1 Ω V m (Suc (length []))
      using Nil.prems(1) ‹vs @ [] ∈ C M2 M1 Ω V m (Suc (length []))› by auto
    ultimately show False
      using C-disj by blast
  qed
next
  case (snoc x xs')

  let ?TS = λ n . TS M2 M1 Ω V m n
  let ?C = λ n . C M2 M1 Ω V m n
  let ?RM = λ n . RM M2 M1 Ω V m n

  have vs @ xs' @ [x] ∉ V
    using snoc.prems(2) by auto
  then have vs @ xs' @ [x] ∉ ?C 1
    by auto
  moreover have vs @ xs' @ [x] ∉ ?C 0
    by auto
```

```
  ultimately have 1 < i
    using snoc.prems(1) by (metis less-one linorder-neqE-nat)

  then have vs @ butlast (xs' @ [x]) ∈ C M2 M1 Ω V m (i−1)
  proof −
    have Suc 0 < i
      using ‹1 < i› by auto
    then have f1: Suc (i − Suc (Suc 0)) = i − Suc 0
      using Suc-diff-Suc by presburger
    have 0 < i
      by (metis (no-types) One-nat-def Suc-lessD ‹1 < i›)
    then show ?thesis
      using f1 by (metis C-immediate-prefix-containment DiffD1 One-nat-def Suc-pred' snoc.prems(1)
                snoc-eq-iff-butlast)
  qed

  moreover have mcp (vs @ butlast (xs' @ [x])) V vs
    by (meson mcp-prefix-of-suffix prefixeq-butlast snoc.prems(2))

  ultimately have Suc (length xs') = i−1
    using snoc.IH by simp

  then show ?case
    by auto
qed

lemma TS-index :
  assumes vs @ xs ∈ TS M2 M1 Ω V m i
  and     mcp (vs@xs) V vs
shows Suc (length xs) ≤ i vs@xs ∈ C M2 M1 Ω V m (Suc (length xs))
proof −
  let ?TS = λ n . TS M2 M1 Ω V m n
  let ?C = λ n . C M2 M1 Ω V m n
  let ?RM = λ n . RM M2 M1 Ω V m n

  obtain j where j < Suc i vs@xs ∈ ?C j
    using TS-union[of M2 M1 Ω V m i]
    by (metis (full-types) UN-iff assms(1) atLeastLessThan-iff set-upt)
  then have Suc (length xs) = j
    using C-index assms(2) by blast
  then show Suc (length xs) ≤ i
    using ‹j < Suc i› by auto
  show vs@xs ∈ C M2 M1 Ω V m (Suc (length xs))
    using ‹vs@xs ∈ ?C j› ‹Suc (length xs) = j› by auto
qed


lemma C-extension-options :
  assumes vs @ xs ∈ C M2 M1 Ω V m i
  and     mcp (vs @ xs @ [x]) V vs
  and     x ∈ inputs M2
  and     0 < i
shows vs@xs@[x] ∈ C M2 M1 Ω V m (Suc i) ∨ vs@xs ∈ RM M2 M1 Ω V m i
proof (cases vs@xs ∈ RM M2 M1 Ω V m i)
  case True
  then show ?thesis by auto
next
  case False

  let ?TS = λ n . TS M2 M1 Ω V m n
  let ?C = λ n . C M2 M1 Ω V m n
  let ?RM = λ n . RM M2 M1 Ω V m n

  obtain k where i = Suc k
    using assms(4) gr0-implies-Suc by blast
  then have ?C (Suc i) = append-set (?C i − ?RM i) (inputs M2) − ?TS i
```

using *C.simps(3)* **by** *blast*

moreover have *vs@xs* ∈ *?C i* − *?RM i*
using *assms(1) False* **by** *blast*

ultimately have *vs@xs@[x]* ∈ *append-set* (*?C i* − *?RM i*) (*inputs M2*)
by (*simp add*: *assms(3)*)

moreover have *vs@xs@[x]* ∉ *?TS i*
proof (*rule ccontr*)
assume ¬ *vs @ xs @ [x]* ∉ *?TS i*
then obtain *j* **where** *j* < *Suc i vs@xs@[x]* ∈ *?C j*
using *TS-union[of M2 M1 Ω V m i]* **by** *fastforce*
then have *Suc* (*length* (*xs@[x]*)) = *j*
using *C-index assms(2)* **by** *blast*

then have *Suc* (*length* (*xs@[x]*)) < *Suc i*
using ‹*j* < *Suc i*› **by** *auto*
moreover have *Suc* (*length xs*) = *i*
using *C-index*
by (*metis assms(1) assms(2) mcp-prefix-of-suffix prefixI*)
ultimately have *Suc* (*length* (*xs@[x]*)) < *Suc* (*Suc* (*length xs*))
by *auto*
then show *False*
by *auto*
qed

ultimately show *?thesis*
by (*simp add*: ‹*?C* (*Suc i*) = *append-set* (*?C i* − *?RM i*) (*inputs M2*) − *?TS i*›)
qed

lemma *TS-non-containment-causes* :
assumes *vs@xs* ∉ *TS M2 M1 Ω V m i*
and *mcp* (*vs@xs*) *V vs*
and *set xs* ⊆ *inputs M2*
and *0* < *i*
shows (∃ *xr j* . *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs@xr* ∈ *RM M2 M1 Ω V m j*)
∨ (∃ *xc* . *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs@xc* ∈ (*C M2 M1 Ω V m i*) − (*RM M2 M1 Ω V m i*))
(**is** *?PrefPreviouslyRemoved* ∨ *?PrefJustContained*)
¬ ((∃ *xr j* . *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs@xr* ∈ *RM M2 M1 Ω V m j*)
∧ (∃ *xc* . *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs@xc* ∈ (*C M2 M1 Ω V m i*) − (*RM M2 M1 Ω V m i*)))
— If a sequence is not contained in TS up to (incl.) iteration i, then either a prefix of it has been removed or a prefix of it is contained in the C set for iteration i
proof −

let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

show *?PrefPreviouslyRemoved* ∨ *?PrefJustContained*
proof (*rule ccontr*)
assume ¬ (*?PrefPreviouslyRemoved* ∨ *?PrefJustContained*)
then have ¬ *?PrefPreviouslyRemoved* ¬ *?PrefJustContained* **by** *auto*

have ¬ (∃ *xr j*. *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs @ xr* ∈ *?RM j*)
proof
assume ∃ *xr j*. *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs @ xr* ∈ *RM M2 M1 Ω V m j*
then obtain *xr j* **where** *prefix xr xs j* ≤ *i vs @ xr* ∈ *?RM j*
by *blast*
then show *False*
proof (*cases xr* = *xs*)

```
      case True
      then have vs @ xs ∈ ?RM j using ‹vs @ xr ∈ ?RM j› by auto
      then have vs @ xs ∈ ?TS j
        using C-subset RM-subset ‹vs @ xr ∈ ?RM j› by blast
      then have vs @ xs ∈ ?TS i
        using TS-subset ‹j ≤ i› by blast
      then show ?thesis using assms(1) by blast
    next
      case False
      then show ?thesis
        using ‹¬ ?PrefPreviouslyRemoved› ‹prefix xr xs› ‹j ≤ i› ‹vs @ xr ∈ ?RM j›
        by blast
    qed
  qed


  have vs ∈ V using assms(2) by auto
  then have vs ∈ ?C 1 by auto

  have ⋀ k . (1 ≤ Suc k ∧ Suc k ≤ i) ⟶ vs @ (take k xs) ∈ ?C (Suc k) − ?RM (Suc k)
  proof
    fix k assume 1 ≤ Suc k ∧ Suc k ≤ i
    then show vs @ (take k xs) ∈ ?C (Suc k) − ?RM (Suc k)
    proof (induction k)
      case 0
      show ?case using ‹vs ∈ ?C 1›
        by (metis 0.prems DiffI One-nat-def
              ‹¬ (∃ xr j. prefix xr xs ∧ j ≤ i ∧ vs @ xr ∈ RM M2 M1 Ω V m j)›
              append-Nil2 take-0 take-is-prefix)
    next
      case (Suc k)

      have 1 ≤ Suc k ∧ Suc k ≤ i
        using Suc.prems by auto
      then have vs @ take k xs ∈ ?C (Suc k)
        using Suc.IH by simp

      moreover have vs @ take k xs ∉ ?RM (Suc k)
        using ‹1 ≤ Suc k ∧ Suc k ≤ i› ‹¬ ?PrefPreviouslyRemoved› take-is-prefix Suc.IH
        by blast

      ultimately have vs @ take k xs ∈ (?C (Suc k)) − (?RM (Suc k))
        by blast

      have k < length xs
      proof (rule ccontr)
        assume ¬ k < length xs
        then have vs @ xs ∈ ?C (Suc k) using ‹vs @ take k xs ∈ ?C (Suc k)›
          by simp
        have vs @ xs ∈ ?TS i
          by (metis C-subset TS-subset ‹1 ≤ Suc k ∧ Suc k ≤ i› ‹vs @ xs ∈ ?C (Suc k)›
              contra-subsetD)
        then show False
          using assms(1) by simp
      qed
      moreover have set xs ⊆ inputs M2
        using assms(3) by auto
      ultimately have last (take (Suc k) xs) ∈ inputs M2
        by (simp add: subset-eq take-Suc-conv-app-nth)

      have vs @ take (Suc k) xs ∈ append-set ((?C (Suc k)) − (?RM (Suc k))) (inputs M2)
      proof −
        have f1: xs ! k ∈ inputs M2
          by (meson ‹k < length xs› ‹set xs ⊆ inputs M2› nth-mem subset-iff)
        have vs @ take (Suc k) xs = (vs @ take k xs) @ [xs ! k]
          by (simp add: ‹k < length xs› take-Suc-conv-app-nth)
```

```
      then show ?thesis
        using f1 ‹vs @ take k xs ∈ C M2 M1 Ω V m (Suc k) − RM M2 M1 Ω V m (Suc k)› by blast
    qed

    moreover have vs @ take (Suc k) xs ∉ ?TS (Suc k)
    proof
      assume vs @ take (Suc k) xs ∈ ?TS (Suc k)
      then have Suc (length (take (Suc k) xs)) ≤ Suc k
        using TS-index(1) assms(2) mcp-prefix-of-suffix take-is-prefix by blast
      moreover have Suc (length (take k xs)) = Suc k using C-index ‹vs @ take k xs ∈ ?C (Suc k)›
        by (metis assms(2) mcp-prefix-of-suffix take-is-prefix)
      ultimately show False using ‹k < length xs›
        by simp
    qed


    show vs @ take (Suc k) xs ∈ ?C (Suc (Suc k)) − ?RM (Suc (Suc k))
      using C.simps(3)[of M2 M1 Ω V m k]
      by (metis (no-types, lifting) DiffI Suc.prems
          ‹¬ (∃ xr j. prefix xr xs ∧ j ≤ i ∧ vs @ xr ∈ RM M2 M1 Ω V m j)›
          ‹vs @ take (Suc k) xs ∉ TS M2 M1 Ω V m (Suc k)› calculation take-is-prefix)
  qed
qed

then have vs @ take (i−1) xs ∈ C M2 M1 Ω V m i − RM M2 M1 Ω V m i
  using assms(4)
  by (metis One-nat-def Suc-diff-1 Suc-leI le-less)
then have ?PrefJustContained
  by (metis C-subset DiffD1 assms(1) subsetCE take-is-prefix)
then show False
  using ‹¬ ?PrefJustContained› by simp
qed



show ¬ (?PrefPreviouslyRemoved ∧ ?PrefJustContained)
proof
  assume ?PrefPreviouslyRemoved ∧ ?PrefJustContained
  then have ?PrefPreviouslyRemoved
          ?PrefJustContained
    by auto

  obtain xr j where prefix xr xs j ≤ i vs@xr ∈ ?RM j
    using ‹?PrefPreviouslyRemoved› by blast
  obtain xc where prefix xc xs vs@xc ∈ ?C i − ?RM i
    using ‹?PrefJustContained› by blast

  then have Suc (length xc) = i
    using C-index
    by (metis Diff-iff assms(2) mcp-prefix-of-suffix)
  moreover have length xc ≤ length xs
    using ‹prefix xc xs› by (simp add: prefix-length-le)
  moreover have xc ≠ xs
  proof
    assume xc = xs
    then have vs@xs ∈ ?C i
      using ‹vs@xc ∈ ?C i − ?RM i› by auto
    then have vs@xs ∈ ?TS i
      using C-subset by blast
    then show False
      using assms(1) by blast
  qed
  ultimately have i ≤ length xs
    using ‹prefix xc xs› not-less-eq-eq prefix-length-prefix prefix-order.antisym
    by blast
```

```
have ⋀ n . (n < i) ⟹ vs@(take n xs) ∈ ?C (Suc n)
proof -
  fix n assume n < i
  show vs @ take n xs ∈ C M2 M1 Ω V m (Suc n)
  proof -
    have n ≤ length xc
      using ‹n < i› ‹Suc (length xc) = i› less-Suc-eq-le
      by blast
    then have prefix (vs @ (take n xs)) (vs @ xc)
    proof -
      have n ≤ length xs
        using ‹length xc ≤ length xs› ‹n ≤ length xc› order-trans
        by blast
      then have prefix (take n xs) xc
        by (metis (no-types) ‹n ≤ length xc› ‹prefix xc xs› length-take min.absorb2
            prefix-length-prefix take-is-prefix)
      then show ?thesis
        by simp
    qed
    then have vs @ take n xs ∈ ?TS i
      by (meson C-subset DiffD1 TS-prefix-containment ‹prefix xc xs›
          ‹vs @ xc ∈ C M2 M1 Ω V m i − RM M2 M1 Ω V m i› assms(2) contra-subsetD
          mcp-prefix-of-suffix same-prefix-prefix)
    then obtain jn where jn < Suc i vs@(take n xs) ∈ ?C jn
      using TS-union[of M2 M1 Ω V m i]
      by (metis UN-iff atLeast-upt lessThan-iff)
    moreover have mcp (vs @ take n xs) V vs
      by (meson assms(2) mcp-prefix-of-suffix take-is-prefix)
    ultimately have jn = Suc (length (take n xs))
      using C-index[of vs take n xs M2 M1 Ω V m jn] by auto
    then have jn = Suc n
      using ‹length xc ≤ length xs› ‹n ≤ length xc› by auto
    then show vs@(take n xs) ∈ ?C (Suc n)
      using ‹vs@(take n xs) ∈ ?C jn› by auto
  qed
qed


have ⋀ n . (n < i) ⟹ vs@(take n xs) ∉ ?RM (Suc n)
proof -
  fix n assume n < i
  show vs @ take n xs ∉ RM M2 M1 Ω V m (Suc n)
  proof (cases n = length xc)
    case True
    then show ?thesis
      using ‹vs@xc ∈ ?C i − ?RM i›
      by (metis DiffD2 ‹Suc (length xc) = i› ‹prefix xc xs› append-eq-conv-conj prefixE)
  next
    case False
    then have n < length xc
      using ‹n < i› ‹Suc (length xc) = i› by linarith

    show ?thesis
    proof (cases Suc n < length xc)
      case True
      then have Suc n < i
        using ‹Suc (length xc) = i› ‹n < length xc› by blast
      then have vs @ (take (Suc n) xs) ∈ ?C (Suc (Suc n))
        using ‹⋀ n . (n < i) ⟹ vs@(take n xs) ∈ ?C (Suc n)› by blast
      then have vs @ butlast (take (Suc n) xs) ∈ ?C (Suc n) − ?RM (Suc n)
        using True C-immediate-prefix-containment[of vs take (Suc n) xs M2 M1 Ω V m n]
        by (metis Suc-neq-Zero ‹prefix xc xs› ‹xc ≠ xs› prefix-Nil take-eq-Nil)
      then show ?thesis
        by (metis DiffD2 Suc-lessD True ‹length xc ≤ length xs› butlast-snoc less-le-trans
```

```
          take-Suc-conv-app-nth)
      next
        case False
        then have Suc n = length xc
          using Suc-lessI ‹n < length xc› by blast
        then have vs @ (take (Suc n) xs) ∈ ?C (Suc (Suc n))
          using ‹Suc (length xc) = i› ‹⋀n. n < i ⟹ vs @ take n xs ∈ C M2 M1 Ω V m (Suc n)›
          by auto
        then have vs @ butlast (take (Suc n) xs) ∈ ?C (Suc n) − ?RM (Suc n)
          using False C-immediate-prefix-containment[of vs take (Suc n) xs M2 M1 Ω V m n]
          by (metis Suc-neq-Zero ‹prefix xc xs› ‹xc ≠ xs› prefix-Nil take-eq-Nil)
        then show ?thesis
          by (metis Diff-iff ‹Suc n = length xc› ‹length xc ≤ length xs› butlast-take diff-Suc-1)
      qed
    qed
  qed


  have xr = take j xs
  proof −
    have vs@xr ∈ ?C j
      using ‹vs@xr ∈ ?RM j› RM-subset by blast
    then show ?thesis
      using C-index
      by (metis Suc-le-lessD ‹⋀n. n < i ⟹ vs @ take n xs ∉ RM M2 M1 Ω V m (Suc n)› ‹j ≤ i›
          ‹prefix xr xs› ‹vs @ xr ∈ RM M2 M1 Ω V m j› append-eq-conv-conj assms(2)
          mcp-prefix-of-suffix prefix-def)
  qed

  have vs@xr ∉ ?RM j
    by (metis (no-types) C-index RM-subset ‹i ≤ length xs› ‹j ≤ i› ‹prefix xr xs›
        ‹xr = take j xs› assms(2) contra-subsetD dual-order.trans length-take lessI less-irrefl
        mcp-prefix-of-suffix min.absorb2)

  then show False
    using ‹vs@xr ∈ ?RM j› by simp
 qed
qed




lemma TS-non-containment-causes-rev :
 assumes mcp (vs@xs) V vs
 and (∃ xr j . xr ≠ xs ∧ prefix xr xs ∧ j ≤ i ∧ vs@xr ∈ RM M2 M1 Ω V m j)
     ∨ (∃ xc . xc ≠ xs ∧ prefix xc xs ∧ vs@xc ∈ (C M2 M1 Ω V m i) − (RM M2 M1 Ω V m i))
   (is ?PrefPreviouslyRemoved ∨ ?PrefJustContained)
shows vs@xs ∉ TS M2 M1 Ω V m i
proof
 let ?TS = λ n . TS M2 M1 Ω V m n
 let ?C = λ n . C M2 M1 Ω V m n
 let ?RM = λ n . RM M2 M1 Ω V m n

 assume vs @ xs ∈ TS M2 M1 Ω V m i

 have ?PrefPreviouslyRemoved ⟹ False
 proof −
  assume ?PrefPreviouslyRemoved
  then obtain xr j where xr ≠ xs prefix xr xs j ≤ i vs@xr ∈ ?RM j
   by blast
  then have vs@xr ∉ ?C j − ?RM j
   by blast



  have vs@(take (Suc (length xr)) xs) ∉ ?C (Suc j)
```

```
  proof −
    have vs@(take (length xr) xs) ∉ ?C j − ?RM j
      by (metis ‹prefix xr xs› ‹vs @ xr ∉ C M2 M1 Ω V m j − RM M2 M1 Ω V m j›
          append-eq-conv-conj prefix-def)
    show ?thesis
    proof (cases j)
      case 0
      then show ?thesis
        using RM.simps(1) ‹vs @ xr ∈ RM M2 M1 Ω V m j› by blast
    next
      case (Suc j')
      then have ?C (Suc j) ⊆ append-set (?C j − ?RM j) (inputs M2)
        using C.simps(3) Suc by blast
      obtain x where vs@(take (Suc (length xr)) xs) = vs@(take (length xr) xs) @ [x]
        by (metis ‹prefix xr xs› ‹xr ≠ xs› append-eq-conv-conj not-le prefix-def
            take-Suc-conv-app-nth take-all)
      have vs@(take (length xr) xs) @ [x] ∉ append-set (?C j − ?RM j) (inputs M2)
        using ‹vs@(take (length xr) xs) ∉ ?C j − ?RM j› by simp
      then have vs@(take (length xr) xs) @ [x] ∉ ?C (Suc j)
        using ‹?C (Suc j) ⊆ append-set (?C j − ?RM j) (inputs M2)› by blast
      then show ?thesis
        using ‹vs@(take (Suc (length xr)) xs) = vs@(take (length xr) xs) @ [x]› by auto
    qed
  qed

  have prefix (take (Suc (length xr)) xs) xs
    by (simp add: take-is-prefix)
  then have vs@(take (Suc (length xr)) xs) ∈ ?TS i
    using TS-prefix-containment[OF ‹vs @ xs ∈ TS M2 M1 Ω V m i› assms(1)] by simp
  then obtain j' where j' < Suc i ∧ vs@(take (Suc (length xr)) xs) ∈ ?C j'
    using TS-union[of M2 M1 Ω V m i] by fastforce
  then have Suc (Suc (length xr)) = j'
    using C-index[of vs take (Suc (length xr)) xs]
  proof −
    have ¬ length xs ≤ length xr
      by (metis (no-types) ‹prefix xr xs› ‹xr ≠ xs› append-Nil2 append-eq-conv-conj leD
          nat-less-le prefix-def prefix-length-le)
    then show ?thesis
      by (metis (no-types) ‹⋀i Ω V T S M2 M1. ⟦vs @ take (Suc (length xr)) xs ∈ C M2 M1 Ω V m i;
                                          mcp (vs @ take (Suc (length xr)) xs) V vs⟧
                                         ⟹ Suc (length (take (Suc (length xr)) xs)) = i›
          ‹j' < Suc i ∧ vs @ take (Suc (length xr)) xs ∈ C M2 M1 Ω V m j'›
          append-eq-conv-conj assms(1) length-take mcp-prefix-of-suffix min.absorb2
          not-less-eq-eq prefix-def)
  qed
  moreover have Suc (length xr) = j
    using ‹vs@xr ∈ ?RM j› RM-subset C-index
    by (metis ‹prefix xr xs› assms(1) mcp-prefix-of-suffix subsetCE)
  ultimately have j' = Suc j
    by auto

  then have vs@(take (Suc (length xr)) xs) ∈ ?C (Suc j)
    using ‹j' < Suc i ∧ vs@(take (Suc (length xr)) xs) ∈ ?C j'› by auto
  then show False
    using ‹vs@(take (Suc (length xr)) xs) ∉ ?C (Suc j)› by blast
qed


moreover have ?PrefJustContained ⟹ False
proof −
  assume ?PrefJustContained
  then obtain xc where xc ≠ xs
                       prefix xc xs
                       vs @ xc ∈ ?C i − ?RM i
    by blast
```

— only possible if xc = xs
then show *False*
by (*metis C-index DiffD1 Suc-less-eq TS-index*(*1*) ‹*vs* @ *xs* ∈ *?TS i*› *assms*(*1*) *leD le-neq-trans mcp-prefix-of-suffix prefix-length-le prefix-length-prefix prefix-order.dual-order.antisym prefix-order.order-refl*)
qed

ultimately show *False*
using *assms*(*2*) by *auto*
qed

lemma *TS-finite* :
assumes *finite V*
and *finite* (*inputs M2*)
shows *finite* (*TS M2 M1* Ω *V m n*)
using *assms* **proof** (*induction n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)

let *?TS* = λ *n* . *TS M2 M1* Ω *V m n*
let *?C* = λ *n* . *C M2 M1* Ω *V m n*
let *?RM* = λ *n* . *RM M2 M1* Ω *V m n*

show *?case*
proof (*cases n=0*)
case *True*
then have *?TS* (*Suc n*) = *V*
by *auto*
then show *?thesis*
using ‹*finite V*› **by** *auto*
next
case *False*
then have *?TS* (*Suc n*) = *?TS n* ∪ *?C* (*Suc n*)
by (*metis TS.simps*(*3*) *gr0-implies-Suc neq0-conv*)
moreover have *finite* (*?TS n*)
using *Suc.IH*[*OF Suc.prems*] **by** *assumption*
moreover have *finite* (*?C* (*Suc n*))
proof −
have *?C* (*Suc n*) ⊆ *append-set* (*?C n*) (*inputs M2*)
using *C-step False* **by** *blast*
moreover have *?C n* ⊆ *?TS n*
by (*simp add*: *C-subset*)
ultimately have *?C* (*Suc n*) ⊆ *append-set* (*?TS n*) (*inputs M2*)
by *blast*
moreover have *finite* (*append-set* (*?TS n*) (*inputs M2*))
by (*simp add*: ‹*finite* (*TS M2 M1* Ω *V m n*)› *assms*(*2*) *finite-image-set2*)
ultimately show *?thesis*
using *infinite-subset* **by** *auto*
qed
ultimately show *?thesis*
by *auto*
qed
qed

lemma *C-finite* :
assumes *finite V*
and *finite* (*inputs M2*)
shows *finite* (*C M2 M1* Ω *V m n*)
proof −
have *C M2 M1* Ω *V m n* ⊆ *TS M2 M1* Ω *V m n*
by (*simp add*: *C-subset*)

then show *?thesis* **using** *TS-finite*[*OF assms*]
using *Finite-Set.finite-subset* **by** *blast*
qed

5.5 Final iteration

The result of calculating TS for some iteration is final if the result does not change for the next iteration.

Such a final iteration exists and is at most equal to the number of states of FSM M2 multiplied by an upper bound on the number of states of FSM M1.

Furthermore, for any sequence not contained in the final iteration of the test suite, a prefix of this sequence must be contained in the latter.

abbreviation *final-iteration M2 M1* Ω *V m i* $\equiv$ *TS M2 M1* Ω *V m i* = *TS M2 M1* Ω *V m* (*Suc i*)

lemma *final-iteration-ex* :
assumes *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and *test-tools M2 M1 FAIL PM V* Ω
shows *final-iteration M2 M1* Ω *V m* (*Suc* (|*M2*| ∗ *m*))
proof −
let *?i* = *Suc* (|*M2*| ∗ *m*)

let *?TS* = λ *n* . *TS M2 M1* Ω *V m n*
let *?C* = λ *n* . *C M2 M1* Ω *V m n*
let *?RM* = λ *n* . *RM M2 M1* Ω *V m n*

have *is-det-state-cover M2 V*
using *assms* **by** *auto*
moreover have *finite* (*nodes M2*)
using *assms*(*2*) **by** *auto*
moreover have *d-reachable M2* (*initial M2*) ⊆ *nodes M2*
by *auto*
ultimately have *finite V*
using *det-state-cover-card*[*of M2 V*]
by (*metis finite-if-finite-subsets-card-bdd infinite-subset is-det-state-cover.elims*(*2*)
surj-card-le)

have ∀ *seq* ∈ *?C ?i* . *seq* ∈ *?RM ?i*
proof
fix *seq* **assume** *seq* ∈ *?C ?i*
show *seq* ∈ *?RM ?i*
proof −

have [] ∈ *V*
using ‹*is-det-state-cover M2 V*› *det-state-cover-empty*
by *blast*
then obtain *vs* **where** *mcp seq V vs*
using *mcp-ex*[*OF -* ‹*finite V*›]
by *blast*
then obtain *xs* **where** *seq* = *vs*@*xs*
using *prefixE* **by** *auto*

then have *Suc* (*length xs*) = *?i* **using** *C-index*
using ‹*mcp seq V vs*› ‹*seq* ∈ *C M2 M1* Ω *V m* (*Suc* (|*M2*| ∗ *m*))› **by** *blast*
then have *length xs* = (|*M2*| ∗ *m*) **by** *auto*

have *RM-def* : *?RM ?i* = {*xs'* ∈ *C M2 M1* Ω *V m ?i* .
(¬ (L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*}))
∨ (∀ *io* ∈ L_{in} *M1* {*xs'*} .
(∃ *V''* ∈ *N io M1 V* .
(∃ *S1* .
(∃ *vs xs* .

```
                        io = (vs@xs)
                        ∧ mcp (vs@xs) V'' vs
                        ∧ S1 ⊆ nodes M2
                        ∧ (∀ s1 ∈ S1 . ∀ s2 ∈ S1 .
                          s1 ≠ s2 ⟶
                            (∀ io1 ∈ RP M2 s1 vs xs V'' .
                              ∀ io2 ∈ RP M2 s2 vs xs V'' .
                                B M1 io1 Ω ≠ B M1 io2 Ω ))
          ∧ m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) S1 Ω V'' )))))}
  using RM.simps(2)[of M2 M1 Ω V m ((card (nodes M2))∗m)] by assumption

have (¬ (L_in M1 {seq} ⊆ L_in M2 {seq}))
    ∨ (∀ io ∈ L_in M1 {seq} .
        (∃ V'' ∈ N io M1 V .
          (∃ S1 .
            (∃ vs xs .
              io = (vs@xs)
              ∧ mcp (vs@xs) V'' vs
              ∧ S1 ⊆ nodes M2
              ∧ (∀ s1 ∈ S1 . ∀ s2 ∈ S1 .
                s1 ≠ s2 ⟶
                  (∀ io1 ∈ RP M2 s1 vs xs V'' .
                    ∀ io2 ∈ RP M2 s2 vs xs V'' .
                      B M1 io1 Ω ≠ B M1 io2 Ω ))
              ∧ m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) S1 Ω V'' ))))
  proof (cases (¬ (L_in M1 {seq} ⊆ L_in M2 {seq})))
    case True
    then show ?thesis
      using RM-def by blast
  next
    case False
    have (∀ io ∈ L_in M1 {seq} .
        (∃ V'' ∈ N io M1 V .
          (∃ S1 .
            (∃ vs xs .
              io = (vs@xs)
              ∧ mcp (vs@xs) V'' vs
              ∧ S1 ⊆ nodes M2
              ∧ (∀ s1 ∈ S1 . ∀ s2 ∈ S1 .
                s1 ≠ s2 ⟶
                  (∀ io1 ∈ RP M2 s1 vs xs V'' .
                    ∀ io2 ∈ RP M2 s2 vs xs V'' .
                      B M1 io1 Ω ≠ B M1 io2 Ω ))
              ∧ m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) S1 Ω V'' ))))
    proof
      fix io assume io∈L_in M1 {seq}
      then have io ∈ L M1
        by auto
      moreover have is-det-state-cover M2 V
        using assms(4) by auto
      ultimately obtain V'' where V'' ∈ N io M1 V
        using N-nonempty[OF - assms(1−3), of V io] by blast

      have io ∈ L M2
        using ‹io∈L_in M1 {seq}› False by auto



      have V'' ∈ Perm V M1
        using ‹V'' ∈ N io M1 V› by auto

      have [] ∈ V''
        using ‹V'' ∈ Perm V M1› assms(4) perm-empty by blast
      have finite V''
        using ‹V'' ∈ Perm V M1› assms(2) assms(4) perm-elem-finite by blast
      obtain vs where mcp io V'' vs
```

using *mcp-ex*[*OF* ‹[] ∈ *V''*› ‹*finite V''*›] **by** *blast*

obtain *xs* **where** *io* = (*vs@xs*)
using ‹*mcp io V'' vs*› *prefixE* **by** *auto*

then have *vs@xs* ∈ *L M1 vs@xs* ∈ *L M2*
using ‹*io* ∈ *L M1*› ‹*io* ∈ *L M2*› **by** *auto*

have *io* ∈ *L M1 map fst io* ∈ {*seq*}
using ‹$io \in L_{in}$ *M1* {*seq*}› **by** *auto*
then have *map fst io* = *seq*
by *auto*
then have *map fst io* ∈ *?C ?i*
using ‹*seq* ∈ *?C ?i*› **by** *blast*
then have (*map fst vs*) @ (*map fst xs*) ∈ *?C ?i*
using ‹*io* = (*vs@xs*)› **by** (*metis map-append*)

have *mcp' io V''* = *vs*
using ‹*mcp io V'' vs*› *mcp'-intro* **by** *blast*

have *mcp'* (*map fst io*) *V* = (*map fst vs*)
using ‹*V''* ∈ *N io M1 V*› ‹*mcp' io V''* = *vs*› **by** *auto*

then have *mcp* (*map fst io*) *V* (*map fst vs*)
by (*metis* ‹⋀*thesis.* (⋀*vs. mcp seq V vs* ⟹ *thesis*) ⟹ *thesis*›
‹*map fst io* = *seq*› *mcp'-intro*)

then have *mcp* (*map fst vs* @ *map fst xs*) *V* (*map fst vs*)
by (*simp add*: ‹*io* = *vs* @ *xs*›)

then have *Suc* (*length xs*) = *?i* **using** *C-index*[*OF* ‹(*map fst vs*) @ (*map fst xs*) ∈ *?C ?i*›]
by *simp*

then have (|*M2*| ∗ *m*) ≤ *length xs*
by *simp*

have |*M1*| ≤ *m*
using *assms*(*3*) **by** *auto*
have *vs* @ *xs* ∈ *L M2* ∩ *L M1*
using ‹*vs* @ *xs* ∈ *L M1*› ‹*vs* @ *xs* ∈ *L M2*› **by** *blast*
obtain *q* **where** *q* ∈ *nodes M2 m* < *card* (*RP M2 q vs xs V''*)
using *RP-state-repetition-distribution-productF*
[*OF assms*(*2*,*1*) ‹(|*M2*| ∗ *m*) ≤ *length xs*› ‹|*M1*| ≤ *m*› ‹*vs* @ *xs* ∈ *L M2* ∩ *L M1*›
‹*is-det-state-cover M2 V*› ‹*V''* ∈ *Perm V M1*›]
by *blast*

have *m* < *LB M2 M1 vs xs* (*?TS* ((|*M2*| ∗ *m*)) ∪ *V*) {*q*} Ω *V''*
proof −
have *m* < (*sum* (λ *s* . *card* (*RP M2 s vs xs V''*)) {*q*})
using ‹*m* < *card* (*RP M2 q vs xs V''*)›
by *auto*
moreover have (*sum* (λ *s* . *card* (*RP M2 s vs xs V''*)) {*q*})
≤ *LB M2 M1 vs xs* (*?TS* ((|*M2*| ∗ *m*)) ∪ *V*) {*q*} Ω *V''*
by *auto*
ultimately show *?thesis*
by *linarith*
qed

show ∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧

```
          S1 ⊆ nodes M2 ∧
          (∀ s1∈S1.
             ∀ s2∈S1.
                s1 ≠ s2 ⟶
                (∀ io1∈RP M2 s1 vs xs V''. ∀ io2∈RP M2 s2 vs xs V''.
                                             B M1 io1 Ω ≠ B M1 io2 Ω)) ∧
          m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) S1 Ω V''
  proof −

    have io = vs@xs
      using ‹io = vs@xs› by assumption
    moreover have mcp (vs@xs) V'' vs
      using ‹io = vs @ xs› ‹mcp io V'' vs› by presburger
    moreover have {q} ⊆ nodes M2
      using ‹q ∈ nodes M2› by auto
    moreover have (∀ s1 ∈ {q} . ∀ s2 ∈ {q} .
              s1 ≠ s2 ⟶
                (∀ io1 ∈ RP M2 s1 vs xs V'' .
                   ∀ io2 ∈ RP M2 s2 vs xs V'' .
                     B M1 io1 Ω ≠ B M1 io2 Ω ))
    proof −
      have ∀ s1 ∈ {q} . ∀ s2 ∈ {q} . s1 = s2
        by blast
      then show ?thesis
        by blast
    qed


    ultimately have RM-body : io = (vs@xs)
            ∧ mcp (vs@xs) V'' vs
            ∧ {q} ⊆ nodes M2
            ∧ (∀ s1 ∈ {q} . ∀ s2 ∈ {q} .
              s1 ≠ s2 ⟶
                (∀ io1 ∈ RP M2 s1 vs xs V'' .
                   ∀ io2 ∈ RP M2 s2 vs xs V'' .
                     B M1 io1 Ω ≠ B M1 io2 Ω ))
            ∧ m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) {q} Ω V''
      using ‹m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) {q} Ω V''›
      by linarith

    show ?thesis
      using ‹V''∈N io M1 V› RM-body
      by metis
  qed
 qed

 then show ?thesis
   by metis
qed

then have seq ∈ {xs' ∈ C M2 M1 Ω V m ((Suc ( |M2| ∗ m))).
                 ¬ L_in M1 {xs'} ⊆ L_in M2 {xs'} ∨
                 (∀ io∈L_in M1 {xs'}.
                     ∃ V''∈N io M1 V.
                        ∃ S1 vs xs.
                           io = vs @ xs ∧
                           mcp (vs @ xs) V'' vs ∧
                           S1 ⊆ nodes M2 ∧
                           (∀ s1∈S1.
                              ∀ s2∈S1.
                                 s1 ≠ s2 ⟶
                                 (∀ io1∈RP M2 s1 vs xs V''. ∀ io2∈RP M2 s2 vs xs V''.
                                                              B M1 io1 Ω ≠ B M1 io2 Ω)) ∧
                           m < LB M2 M1 vs xs (?TS (( |M2| ∗ m)) ∪ V) S1 Ω V'')}
  using ‹seq ∈ ?C ?i› by blast
```

then show *?thesis*
using *RM-def* **by** *blast*
qed
qed

then have *?C ?i − ?RM ?i* = {}
by *blast*

have *?C* (*Suc ?i*) = *append-set* (*?C ?i − ?RM ?i*) (*inputs M2*) − *?TS ?i*
using *C.simps*(*3*) **by** *blast*

then have *?C* (*Suc ?i*) = {} **using** ‹*?C ?i − ?RM ?i* = {}›
by *blast*
then have *?TS* (*Suc ?i*) = *?TS ?i*
using *TS.simps*(*3*) **by** *blast*
then show *final-iteration M2 M1 Ω V m ?i*
by *blast*
qed

lemma *TS-non-containment-causes-final* :
assumes *vs@xs* ∉ *TS M2 M1 Ω V m i*
and *mcp* (*vs@xs*) *V vs*
and *set xs* ⊆ *inputs M2*
and *final-iteration M2 M1 Ω V m i*
and *OFSM M2*
shows (∃ *xr j* . *xr* ≠ *xs*
∧ *prefix xr xs*
∧ *j* ≤ *i*
∧ *vs@xr* ∈ *RM M2 M1 Ω V m j*)
proof −
let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

have {} ≠ *V*
using *assms*(*2*) **by** *fastforce*
then have *?TS 0* ≠ *?TS* (*Suc 0*)
by *simp*
then have *0* < *i*
using *assms*(*4*) **by** *auto*

have *ncc1* : (∃ *xr j*. *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs* @ *xr* ∈ *RM M2 M1 Ω V m j*) ∨
(∃ *xc*. *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs* @ *xc* ∈ *C M2 M1 Ω V m i* − *RM M2 M1 Ω V m i*)
using *TS-non-containment-causes*(*1*)[*OF assms*(*1*−*3*) ‹*0* < *i*›] **by** *assumption*
have *ncc2* : ¬ ((∃ *xr j*. *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs* @ *xr* ∈ *RM M2 M1 Ω V m j*) ∧
(∃ *xc*. *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs* @ *xc* ∈ *C M2 M1 Ω V m i* − *RM M2 M1 Ω V m i*))
using *TS-non-containment-causes*(*2*)[*OF assms*(*1*−*3*) ‹*0* < *i*›] **by** *assumption*

from *ncc1* **show** *?thesis*
proof
show ∃ *xr j*. *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs* @ *xr* ∈ *RM M2 M1 Ω V m j* ⟹
∃ *xr j*. *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs* @ *xr* ∈ *RM M2 M1 Ω V m j*
by *simp*

show ∃ *xc*. *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs* @ *xc* ∈ *C M2 M1 Ω V m i* − *RM M2 M1 Ω V m i* ⟹
∃ *xr j*. *xr* ≠ *xs* ∧ *prefix xr xs* ∧ *j* ≤ *i* ∧ *vs* @ *xr* ∈ *RM M2 M1 Ω V m j*
proof −
assume ∃ *xc*. *xc* ≠ *xs* ∧ *prefix xc xs* ∧ *vs* @ *xc* ∈ *C M2 M1 Ω V m i* − *RM M2 M1 Ω V m i*
then obtain *xc* **where** *xc* ≠ *xs prefix xc xs vs* @ *xc* ∈ *?C i* − *?RM i*
by *blast*

then have *vs* @ *xc* ∈ *?C i*
  by *blast*
have *mcp* (*vs* @ *xc*) *V vs*
  using ‹*prefix xc xs*› *assms*(*2*) *mcp-prefix-of-suffix* **by** *blast*
then have *Suc* (*length xc*) = *i* **using** *C-index*[*OF* ‹*vs* @ *xc* ∈ *?C i*›]
  by *simp*

have *length xc* < *length xs*
  by (*metis* ‹*prefix xc xs*› ‹*xc* ≠ *xs*› *append-eq-conv-conj nat-less-le prefix-def prefix-length-le take-all*)
then obtain *x* **where** *prefix* (*vs*@*xc*@[*x*]) (*vs*@*xs*)
  using ‹*prefix xc xs*› *append-one-prefix same-prefix-prefix* **by** *blast*

— Proof sketch: vs-xs-x must not be in TS (i+1), else not final iteration vs-xs-x can not be in TS i due to its length vs-xs-x must therefore not be contained in (append-set (C i - R i) (inputs M2)) vs-xs must therefore not be contained in (C i - R i) contradiction

have *?TS* (*Suc i*) = *?TS i*
  using *assms*(*4*) **by** *auto*

have *vs*@*xc*@[*x*] ∉ *?C* (*Suc i*)
proof
  assume *vs* @ *xc* @ [*x*] ∈ *?C* (*Suc i*)
  then have *vs* @ *xc* @ [*x*] ∉ *?TS i*
    by (*metis* (*no-types*, *lifting*) *C*.*simps*(*3*) *DiffE* ‹*Suc* (*length xc*) = *i*›)
  then have *?TS i* ≠ *?TS* (*Suc i*)
    using *C-subset* ‹*vs* @ *xc* @ [*x*] ∈ *C M2 M1* Ω *V m* (*Suc i*)› **by** *blast*
  then show *False* **using** *assms*(*4*)
    by *auto*
qed
moreover have *?C* (*Suc i*) = *append-set* (*?C i* − *?RM i*) (*inputs M2*) − *?TS i*
  using *C*.*simps*(*3*) ‹*Suc* (*length xc*) = *i*› **by** *blast*
ultimately have *vs* @ *xc* @ [*x*] ∉ *append-set* (*?C i* − *?RM i*) (*inputs M2*) − *?TS i*
  by *blast*

have *vs* @ *xc* @ [*x*] ∉ *?TS* (*Suc i*)
  by (*metis Suc-n-not-le-n TS-index*(*1*) ‹*Suc* (*length xc*) = *i*›
    ‹*prefix* (*vs* @ *xc* @ [*x*]) (*vs* @ *xs*)› *assms*(*2*) *assms*(*4*) *length-append-singleton*
    mcp-prefix-of-suffix same-prefix-prefix)
then have *vs* @ *xc* @ [*x*] ∉ *?TS i*
  by (*simp add*: *assms*(*4*))

have *vs* @ *xc* @ [*x*] ∉ *append-set* (*?C i* − *?RM i*) (*inputs M2*)
  using ‹*vs* @ *xc* @ [*x*] ∉ *TS M2 M1* Ω *V m i*›
    ‹*vs* @ *xc* @ [*x*] ∉ *append-set* (*C M2 M1* Ω *V m i* − *RM M2 M1* Ω *V m i*) (*inputs M2*)
      − *TS M2 M1* Ω *V m i*›
  by *blast*

then have *vs* @ *xc* ∉ (*?C i* − *?RM i*)
proof −
  have *f1*: ∀ *a A Aa*. (*a*::′*a*) ∉ *A* ∧ *a* ∉ *Aa* ∨ *a* ∈ *Aa* ∪ *A*
    by (*meson UnCI*)
  obtain *aas* :: ′*a list* ⇒ ′*a list* ⇒ ′*a list* **where**
    ∀ *x0 x1*. (∃ *v2*. *x0* = *x1* @ *v2*) = (*x0* = *x1* @ *aas x0 x1*)
    by *moura*
  then have *vs* @ *xs* = (*vs* @ *xc* @ [*x*]) @ *aas* (*vs* @ *xs*) (*vs* @ *xc* @ [*x*])
    by (*meson* ‹*prefix* (*vs* @ *xc* @ [*x*]) (*vs* @ *xs*)› *prefixE*)
  then have *xs* = (*xc* @ [*x*]) @ *aas* (*vs* @ *xs*) (*vs* @ *xc* @ [*x*])
    by *simp*
  then have *x* ∈ *inputs M2*
    using *f1* **by** (*metis* (*no-types*) *assms*(*3*) *contra-subsetD insert-iff list*.*set*(*2*) *set-append*)
  then show *?thesis*
    using ‹*vs* @ *xc* @ [*x*] ∉ *append-set* (*C M2 M1* Ω *V m i* − *RM M2 M1* Ω *V m i*) (*inputs M2*)›
    by *force*
qed

```
      then have False
        using ‹vs @ xc ∈ ?C i − ?RM i› by blast
      then show ?thesis by simp
    qed
  qed
qed
```

```
lemma TS-non-containment-causes-final-suc :
  assumes vs@xs ∉ TS M2 M1 Ω V m i
  and     mcp (vs@xs) V vs
  and     set xs ⊆ inputs M2
  and     final-iteration M2 M1 Ω V m i
  and     OFSM M2
obtains xr j
where xr ≠ xs prefix xr xs Suc j ≤ i vs@xr ∈ RM M2 M1 Ω V m (Suc j)
proof −
  obtain xr j where xr ≠ xs ∧ prefix xr xs ∧ j ≤ i ∧ vs@xr ∈ RM M2 M1 Ω V m j
    using TS-non-containment-causes-final[OF assms] by blast
  moreover have RM M2 M1 Ω V m 0 = {}
    by auto
  ultimately have j ≠ 0
    by (metis empty-iff)
  then obtain jp where j = Suc jp
    using not0-implies-Suc by blast
  then have xr ≠ xs ∧ prefix xr xs ∧ Suc jp ≤ i ∧ vs@xr ∈ RM M2 M1 Ω V m (Suc jp)
    using ‹xr ≠ xs ∧ prefix xr xs ∧ j ≤ i ∧ vs@xr ∈ RM M2 M1 Ω V m j›
    by blast
  then show ?thesis
    using that by blast
qed
```

```
end
theory ASC-Sufficiency
  imports ASC-Suite
begin
```

6 Sufficiency of the test suite to test for reduction

This section provides a proof that the test suite generated by the adaptive state counting algorithm is sufficient to test for reduction.

6.1 Properties of minimal sequences to failures extending the deterministic state cover

The following two lemmata show that minimal sequences to failures extending the deterministic state cover do not with their extending suffix visit any state twice or visit a state also reached by a sequence in the chosen permutation of reactions to the deterministic state cover.

```
lemma minimal-sequence-to-failure-extending-implies-Rep-Pre :
  assumes minimal-sequence-to-failure-extending V M1 M2 vs xs
  and     OFSM M1
  and     OFSM M2
  and     test-tools M2 M1 FAIL PM V Ω
  and     V'' ∈ N (vs@xs') M1 V
  and     prefix xs' xs
  shows ¬ Rep-Pre M2 M1 vs xs'
proof
  assume Rep-Pre M2 M1 vs xs'
  then obtain xs1 xs2 s1 s2 where prefix xs1 xs2
                                  prefix xs2 xs'
                                  xs1 ≠ xs2
```

io-targets M2 (*initial M2*) (*vs* @ *xs1*) = {*s2*}
io-targets M2 (*initial M2*) (*vs* @ *xs2*) = {*s2*}
io-targets M1 (*initial M1*) (*vs* @ *xs1*) = {*s1*}
io-targets M1 (*initial M1*) (*vs* @ *xs2*) = {*s1*}
by *auto*
then have *s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs1*)
s2 ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs2*)
s1 ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs1*)
s1 ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs2*)
by *auto*

have *vs@xs1* ∈ *L M1*
using *io-target-implies-L*[*OF* ‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs1*)›] **by** *assumption*
have *vs@xs2* ∈ *L M1*
using *io-target-implies-L*[*OF* ‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs2*)›] **by** *assumption*
have *vs@xs1* ∈ *L M2*
using *io-target-implies-L*[*OF* ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs1*)›] **by** *assumption*
have *vs@xs2* ∈ *L M2*
using *io-target-implies-L*[*OF* ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs2*)›] **by** *assumption*

obtain *tr1-1* **where** *path M1* (*vs@xs1* || *tr1-1*) (*initial M1*)
length tr1-1 = *length* (*vs@xs1*)
target (*vs@xs1* || *tr1-1*) (*initial M1*) = *s1*
using ‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs1*)› **by** *auto*
obtain *tr1-2* **where** *path M1* (*vs@xs2* || *tr1-2*) (*initial M1*)
length tr1-2 = *length* (*vs@xs2*)
target (*vs@xs2* || *tr1-2*) (*initial M1*) = *s1*
using ‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs2*)› **by** *auto*
obtain *tr2-1* **where** *path M2* (*vs@xs1* || *tr2-1*) (*initial M2*)
length tr2-1 = *length* (*vs@xs1*)
target (*vs@xs1* || *tr2-1*) (*initial M2*) = *s2*
using ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs1*)› **by** *auto*
obtain *tr2-2* **where** *path M2* (*vs@xs2* || *tr2-2*) (*initial M2*)
length tr2-2 = *length* (*vs@xs2*)
target (*vs@xs2* || *tr2-2*) (*initial M2*) = *s2*
using ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs2*)› **by** *auto*

have *productF M2 M1 FAIL PM*
using *assms*(*4*) **by** *auto*
have *well-formed M1*
using *assms*(*2*) **by** *auto*
have *well-formed M2*
using *assms*(*3*) **by** *auto*
have *observable PM*
by (*meson assms*(*2*) *assms*(*3*) *assms*(*4*) *observable-productF*)

have *length* (*vs@xs1*) = *length tr2-1*
using ‹*length tr2-1* = *length* (*vs* @ *xs1*)› **by** *presburger*
then have *length tr2-1* = *length tr1-1*
using ‹*length tr1-1* = *length* (*vs@xs1*)› **by** *presburger*

have *vs@xs1* ∈ *L PM*
using *productF-path-inclusion*[*OF* ‹*length* (*vs@xs1*) = *length tr2-1*› ‹*length tr2-1* = *length tr1-1*›
‹*productF M2 M1 FAIL PM*› ‹*well-formed M2*› ‹*well-formed M1*›]
by (*meson Int-iff* ‹*productF M2 M1 FAIL PM*› ‹*vs* @ *xs1* ∈ *L M1*› ‹*vs* @ *xs1* ∈ *L M2*› ‹*well-formed M1*›
‹*well-formed M2*› *productF-language*)

have *length* (*vs@xs2*) = *length tr2-2*
using ‹*length tr2-2* = *length* (*vs* @ *xs2*)› **by** *presburger*
then have *length tr2-2* = *length tr1-2*
using ‹*length tr1-2* = *length* (*vs@xs2*)› **by** *presburger*

have *vs@xs2* ∈ *L PM*
using *productF-path-inclusion*[*OF* ‹*length* (*vs@xs2*) = *length tr2-2*› ‹*length tr2-2* = *length tr1-2*›

⟨productF M2 M1 FAIL PM⟩ ⟨well-formed M2⟩ ⟨well-formed M1⟩]
by (*meson Int-iff* ⟨*productF M2 M1 FAIL PM*⟩ ⟨*vs @ xs2* ∈ *L M1*⟩ ⟨*vs @ xs2* ∈ *L M2*⟩ ⟨*well-formed M1*⟩ ⟨*well-formed M2*⟩ *productF-language*)

have *io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs1*) = {(*s2*, *s1*)}
using *productF-path-io-targets-reverse*
[*OF* ⟨*productF M2 M1 FAIL PM*⟩ ⟨*s2* ∈ *io-targets M2* (*initial M2*) (*vs @ xs1*)⟩
⟨*s1* ∈ *io-targets M1* (*initial M1*) (*vs @ xs1*)⟩ ⟨*vs @ xs1* ∈ *L M2*⟩ ⟨*vs @ xs1* ∈ *L M1*⟩]
proof −
have ∀ *c f*. *c* ≠ *initial* (*f*::(′*a*, ′*b*, ′*c*) *FSM*) ∨ *c* ∈ *nodes f*
by *blast*
then show *?thesis*
by (*metis* (*no-types*) ⟨⟦*observable M2*; *observable M1*; *well-formed M2*; *well-formed M1*;
initial M2 ∈ *nodes M2*; *initial M1* ∈ *nodes M1*⟧
⟹ *io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs1*) = {(*s2*, *s1*)}⟩
assms(*2*) *assms*(*3*))
qed

have *io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs2*) = {(*s2*, *s1*)}
using *productF-path-io-targets-reverse*
[*OF* ⟨*productF M2 M1 FAIL PM*⟩ ⟨*s2* ∈ *io-targets M2* (*initial M2*) (*vs @ xs2*)⟩
⟨*s1* ∈ *io-targets M1* (*initial M1*) (*vs @ xs2*)⟩ ⟨*vs @ xs2* ∈ *L M2*⟩ ⟨*vs @ xs2* ∈ *L M1*⟩]
proof −
have ∀ *c f*. *c* ≠ *initial* (*f*::(′*a*, ′*b*, ′*c*) *FSM*) ∨ *c* ∈ *nodes f*
by *blast*
then show *?thesis*
by (*metis* (*no-types*) ⟨⟦*observable M2*; *observable M1*; *well-formed M2*; *well-formed M1*;
initial M2 ∈ *nodes M2*; *initial M1* ∈ *nodes M1*⟧
⟹ *io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs2*) = {(*s2*, *s1*)}⟩
assms(*2*) *assms*(*3*))
qed

have *prefix* (*vs @ xs1*) (*vs @ xs2*)
using ⟨*prefix xs1 xs2*⟩ **by** *auto*

have *sequence-to-failure M1 M2* (*vs@xs*)
using *assms*(*1*) **by** *auto*

have *prefix* (*vs@xs1*) (*vs@xs*′)
using ⟨*prefix xs1 xs2*⟩ ⟨*prefix xs2 xs*′⟩ *prefix-order.dual-order.trans same-prefix-prefix*
by *blast*
have *prefix* (*vs@xs2*) (*vs@xs*′)
using ⟨*prefix xs2 xs*′⟩ *prefix-order.dual-order.trans same-prefix-prefix* **by** *blast*

have *io-targets PM* (*initial PM*) (*vs @ xs1*) = {(*s2*,*s1*)}
using ⟨*io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs1*) = {(*s2*, *s1*)}⟩ *assms*(*4*) **by** *auto*
have *io-targets PM* (*initial PM*) (*vs @ xs2*) = {(*s2*,*s1*)}
using ⟨*io-targets PM* (*initial M2*, *initial M1*) (*vs @ xs2*) = {(*s2*, *s1*)}⟩ *assms*(*4*) **by** *auto*

have (*vs @ xs2*) @ (*drop* (*length xs2*) *xs*) = *vs@xs*
by (*metis* ⟨*prefix xs2 xs*′⟩ *append-eq-appendI append-eq-conv-conj assms*(*6*) *prefixE*)
moreover have *io-targets PM* (*initial PM*) (*vs@xs*) = {*FAIL*}
using *sequence-to-failure-reaches-FAIL-ob*[*OF* ⟨*sequence-to-failure M1 M2* (*vs@xs*)⟩ *assms*(*2*,*3*)
⟨*productF M2 M1 FAIL PM*⟩]
by *assumption*
ultimately have *io-targets PM* (*initial PM*) ((*vs @ xs2*) @ (*drop* (*length xs2*) *xs*)) = {*FAIL*}
by *auto*

have *io-targets PM* (*s2*,*s1*) (*drop* (*length xs2*) *xs*) = {*FAIL*}
  using *observable-io-targets-split*
    [*OF* ‹*observable PM*›
      ‹*io-targets PM* (*initial PM*) ((*vs* @ *xs2*) @ (*drop* (*length xs2*) *xs*)) = {*FAIL*}›
      ‹*io-targets PM* (*initial PM*) (*vs* @ *xs2*) = {(*s2*, *s1*)}›]
  by *assumption*

have *io-targets PM* (*initial PM*) (*vs*@*xs1*@(*drop* (*length xs2*) *xs*)) = {*FAIL*}
  using *observable-io-targets-append*
    [*OF* ‹*observable PM*› ‹*io-targets PM* (*initial PM*) (*vs* @ *xs1*) = {(*s2*,*s1*)}›
      ‹*io-targets PM* (*s2*,*s1*) (*drop* (*length xs2*) *xs*) = {*FAIL*}›]
  by *simp*
have *sequence-to-failure M1 M2* (*vs*@*xs1*@(*drop* (*length xs2*) *xs*))
  using *sequence-to-failure-alt-def*
    [*OF* ‹*io-targets PM* (*initial PM*) (*vs*@*xs1*@(*drop* (*length xs2*) *xs*)) = {*FAIL*}› *assms*(*2*,*3*)]
    assms(*4*)
  by *blast*

have *length xs1* < *length xs2*
  using ‹*prefix xs1 xs2*› ‹*xs1* ≠ *xs2*› *prefix-length-prefix* **by** *fastforce*

have *prefix-drop*: *ys* = *ys1* @ (*drop* (*length ys1*)) *ys* **if** *prefix ys1 ys*
  for *ys ys1* :: ('*a* × '*b*) *list*
  using *that* **by** (*induction ys1*) (*auto elim*: *prefixE*)
then have *xs* = (*xs1* @ (*drop* (*length xs1*) *xs*))
  using ‹*prefix xs1 xs2*› ‹*prefix xs2 xs*'› ‹*prefix xs*' *xs*› **by** *simp*
then have *length xs1* < *length xs*
  using *prefix-drop*[*OF* ‹*prefix xs2 xs*'›] ‹*prefix xs2 xs*'› ‹*prefix xs*' *xs*›
  using ‹*length xs1* < *length xs2*›
  by (*auto dest*!: *prefix-length-le*)
have *length* (*xs1*@(*drop* (*length xs2*) *xs*)) < *length xs*
  using ‹*length xs1* < *length xs2*› ‹*length xs1* < *length xs*› **by** *auto*

have *vs* ∈ L_{in} *M1 V*
    ∧ *sequence-to-failure M1 M2* (*vs* @ *xs1*@(*drop* (*length xs2*) *xs*))
    ∧ *length* (*xs1*@(*drop* (*length xs2*) *xs*)) < *length xs*
  using ‹*length* (*xs1* @ *drop* (*length xs2*) *xs*) < *length xs*›
    ‹*sequence-to-failure M1 M2* (*vs* @ *xs1* @ *drop* (*length xs2*) *xs*)›
    assms(*1*) *minimal-sequence-to-failure-extending.simps*
  by *blast*

then have ¬ *minimal-sequence-to-failure-extending V M1 M2 vs xs*
  by (*meson minimal-sequence-to-failure-extending.elims*(*2*))

then show *False*
  using *assms*(*1*) **by** *linarith*
qed

lemma *minimal-sequence-to-failure-extending-implies-Rep-Cov* :
  assumes *minimal-sequence-to-failure-extending V M1 M2 vs xs*
  and *OFSM M1*
  and *OFSM M2*
  and *test-tools M2 M1 FAIL PM V* Ω
  and *V*'' ∈ *N* (*vs*@*xsR*) *M1 V*
  and *prefix xsR xs*
shows ¬ *Rep-Cov M2 M1 V*'' *vs xsR*
proof
  assume *Rep-Cov M2 M1 V*'' *vs xsR*
  then obtain *xs*' *vs*' *s2 s1* **where** *xs*' ≠ []
      prefix xs' *xsR*

$vs' \in V''$
$io\text{-}targets\ M2\ (initial\ M2)\ (vs\ @\ xs') = \{s2\}$
$io\text{-}targets\ M2\ (initial\ M2)\ (vs') = \{s2\}$
$io\text{-}targets\ M1\ (initial\ M1)\ (vs\ @\ xs') = \{s1\}$
$io\text{-}targets\ M1\ (initial\ M1)\ (vs') = \{s1\}$
by *auto*

then have $s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs\ @\ xs')$
$s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs')$
$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs\ @\ xs')$
$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs')$
by *auto*

have $vs@xs' \in L\ M1$
using *io-target-implies-L*[*OF* ‹$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs\ @\ xs')$›] **by** *assumption*
have $vs' \in L\ M1$
using *io-target-implies-L*[*OF* ‹$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs')$›] **by** *assumption*
have $vs@xs' \in L\ M2$
using *io-target-implies-L*[*OF* ‹$s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs\ @\ xs')$›] **by** *assumption*
have $vs' \in L\ M2$
using *io-target-implies-L*[*OF* ‹$s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs')$›] **by** *assumption*

obtain *tr1-1* **where** $path\ M1\ (vs@xs' \mid\mid tr1\text{-}1)\ (initial\ M1)$
$length\ tr1\text{-}1 = length\ (vs@xs')$
$target\ (vs@xs' \mid\mid tr1\text{-}1)\ (initial\ M1) = s1$
using ‹$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs\ @\ xs')$› **by** *auto*
obtain *tr1-2* **where** $path\ M1\ (vs' \mid\mid tr1\text{-}2)\ (initial\ M1)$
$length\ tr1\text{-}2 = length\ (vs')$
$target\ (vs' \mid\mid tr1\text{-}2)\ (initial\ M1) = s1$
using ‹$s1 \in io\text{-}targets\ M1\ (initial\ M1)\ (vs')$› **by** *auto*
obtain *tr2-1* **where** $path\ M2\ (vs@xs' \mid\mid tr2\text{-}1)\ (initial\ M2)$
$length\ tr2\text{-}1 = length\ (vs@xs')$
$target\ (vs@xs' \mid\mid tr2\text{-}1)\ (initial\ M2) = s2$
using ‹$s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs\ @\ xs')$› **by** *auto*
obtain *tr2-2* **where** $path\ M2\ (vs' \mid\mid tr2\text{-}2)\ (initial\ M2)$
$length\ tr2\text{-}2 = length\ (vs')$
$target\ (vs' \mid\mid tr2\text{-}2)\ (initial\ M2) = s2$
using ‹$s2 \in io\text{-}targets\ M2\ (initial\ M2)\ (vs')$› **by** *auto*

have *productF M2 M1 FAIL PM*
using *assms*(*4*) **by** *auto*
have *well-formed M1*
using *assms*(*2*) **by** *auto*
have *well-formed M2*
using *assms*(*3*) **by** *auto*
have *observable PM*
by (*meson assms*(*2*) *assms*(*3*) *assms*(*4*) *observable-productF*)

have $length\ (vs@xs') = length\ tr2\text{-}1$
using ‹$length\ tr2\text{-}1 = length\ (vs\ @\ xs')$› **by** *presburger*
then have $length\ tr2\text{-}1 = length\ tr1\text{-}1$
using ‹$length\ tr1\text{-}1 = length\ (vs@xs')$› **by** *presburger*

have $vs@xs' \in L\ PM$
using *productF-path-inclusion*[*OF* ‹$length\ (vs@xs') = length\ tr2\text{-}1$› ‹$length\ tr2\text{-}1 = length\ tr1\text{-}1$›
‹*productF M2 M1 FAIL PM*› ‹*well-formed M2*› ‹*well-formed M1*›]
by (*meson Int-iff* ‹*productF M2 M1 FAIL PM*› ‹$vs\ @\ xs' \in L\ M1$› ‹$vs\ @\ xs' \in L\ M2$› ‹*well-formed M1*›
‹*well-formed M2*› *productF-language*)

have $length\ (vs') = length\ tr2\text{-}2$
using ‹$length\ tr2\text{-}2 = length\ (vs')$› **by** *presburger*
then have $length\ tr2\text{-}2 = length\ tr1\text{-}2$
using ‹$length\ tr1\text{-}2 = length\ (vs')$› **by** *presburger*

have *vs′* ∈ *L PM*
using *productF-path-inclusion*[*OF* ‹*length* (*vs′*) = *length tr2-2*› ‹*length tr2-2* = *length tr1-2*›
‹*productF M2 M1 FAIL PM*› ‹*well-formed M2*› ‹*well-formed M1*›]
by (*meson Int-iff* ‹*productF M2 M1 FAIL PM*› ‹*vs′* ∈ *L M1*› ‹*vs′* ∈ *L M2*› ‹*well-formed M1*›
‹*well-formed M2*› *productF-language*)

have *io-targets PM* (*initial M2*, *initial M1*) (*vs* @ *xs′*) = {(*s2*, *s1*)}
using *productF-path-io-targets-reverse*
[*OF* ‹*productF M2 M1 FAIL PM*› ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs* @ *xs′*)›
‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs* @ *xs′*)› ‹*vs* @ *xs′* ∈ *L M2*› ‹*vs* @ *xs′* ∈ *L M1*›]
proof −
have ∀ *c f*. *c* ≠ *initial* (*f*::(′*a*, ′*b*, ′*c*) *FSM*) ∨ *c* ∈ *nodes f*
by *blast*
then show *?thesis*
by (*metis* (*no-types*) ‹⟦*observable M2*; *observable M1*; *well-formed M2*; *well-formed M1*;
initial M2 ∈ *nodes M2*; *initial M1* ∈ *nodes M1*⟧
⟹ *io-targets PM* (*initial M2*, *initial M1*) (*vs* @ *xs′*) = {(*s2*, *s1*)}›
assms(*2*) *assms*(*3*))
qed

have *io-targets PM* (*initial M2*, *initial M1*) (*vs′*) = {(*s2*, *s1*)}
using *productF-path-io-targets-reverse*
[*OF* ‹*productF M2 M1 FAIL PM*› ‹*s2* ∈ *io-targets M2* (*initial M2*) (*vs′*)›
‹*s1* ∈ *io-targets M1* (*initial M1*) (*vs′*)› ‹*vs′* ∈ *L M2*› ‹*vs′* ∈ *L M1*›]
proof −
have ∀ *c f*. *c* ≠ *initial* (*f*::(′*a*, ′*b*, ′*c*) *FSM*) ∨ *c* ∈ *nodes f*
by *blast*
then show *?thesis*
by (*metis* (*no-types*) ‹⟦*observable M2*; *observable M1*; *well-formed M2*; *well-formed M1*;
initial M2 ∈ *nodes M2*; *initial M1* ∈ *nodes M1*⟧
⟹ *io-targets PM* (*initial M2*, *initial M1*) (*vs′*) = {(*s2*, *s1*)}›
assms(*2*) *assms*(*3*))
qed
have *io-targets PM* (*initial PM*) (*vs′*) = {(*s2*, *s1*)}
by (*metis* (*no-types*) ‹*io-targets PM* (*initial M2*, *initial M1*) *vs′* = {(*s2*, *s1*)}›
‹*productF M2 M1 FAIL PM*› *productF-simps*(*4*))

have *sequence-to-failure M1 M2* (*vs*@*xs*)
using *assms*(*1*) **by** *auto*

have *xs* = *xs′* @ (*drop* (*length xs′*) *xs*)
by (*metis* ‹*prefix xs′ xsR*› *append-assoc append-eq-conv-conj assms*(*6*) *prefixE*)
then have *io-targets PM* (*initial M2*, *initial M1*) (*vs* @ *xs′* @ (*drop* (*length xs′*) *xs*)) = {*FAIL*}
by (*metis* ‹*productF M2 M1 FAIL PM*› ‹*sequence-to-failure M1 M2* (*vs* @ *xs*)› *assms*(*2*) *assms*(*3*)
productF-simps(*4*) *sequence-to-failure-reaches-FAIL-ob*)
then have *io-targets PM* (*initial M2*, *initial M1*) ((*vs* @ *xs′*) @ (*drop* (*length xs′*) *xs*)) = {*FAIL*}
by *auto*
have *io-targets PM* (*s2*, *s1*) (*drop* (*length xs′*) *xs*) = {*FAIL*}
using *observable-io-targets-split*
[*OF* ‹*observable PM*›
‹*io-targets PM* (*initial M2*,*initial M1*) ((*vs* @ *xs′*) @ (*drop* (*length xs′*) *xs*)) = {*FAIL*}›
‹*io-targets PM* (*initial M2*, *initial M1*) (*vs* @ *xs′*) = {(*s2*, *s1*)}›]
by *assumption*

have *io-targets PM* (*initial PM*) (*vs′* @ (*drop* (*length xs′*) *xs*)) = {*FAIL*}
using *observable-io-targets-append*
[*OF* ‹*observable PM*› ‹*io-targets PM* (*initial PM*) (*vs′*) = {(*s2*, *s1*)}›
‹*io-targets PM* (*s2*, *s1*) (*drop* (*length xs′*) *xs*) = {*FAIL*}›]
by *assumption*

have *sequence-to-failure M1 M2* (*vs′* @ (*drop* (*length xs′*) *xs*))
using *sequence-to-failure-alt-def*

```
      [OF ‹io-targets PM (initial PM) (vs' @ (drop (length xs') xs)) = {FAIL}› assms(2,3)]
      assms(4)
    by blast

  have length (drop (length xs') xs) < length xs
    by (metis (no-types) ‹xs = xs' @ drop (length xs') xs› ‹xs' ≠ []› length-append
        length-greater-0-conv less-add-same-cancel2)

  have vs' ∈ L_in M1 V
  proof −
    have V'' ∈ Perm V M1
      using assms(5) unfolding N.simps by blast

    then obtain f where f-def : V'' = image f V
                                ∧ (∀ v ∈ V . f v ∈ language-state-for-input M1 (initial M1) v)
      unfolding Perm.simps by blast
    then obtain v where v ∈ V vs' = f v
      using ‹vs' ∈ V''› by auto
    then have vs' ∈ language-state-for-input M1 (initial M1) v
      using f-def by auto

    have language-state-for-input M1 (initial M1) v = L_in M1 {v}
      by auto
    moreover have {v} ⊆ V
      using ‹v ∈ V› by blast
    ultimately have language-state-for-input M1 (initial M1) v ⊆ L_in M1 V
      unfolding language-state-for-inputs.simps language-state-for-input.simps by blast
    then show ?thesis
      using‹vs' ∈ language-state-for-input M1 (initial M1) v› by blast
  qed

  have ¬ minimal-sequence-to-failure-extending V M1 M2 vs xs
    using ‹vs' ∈ L_in M1 V›
          ‹sequence-to-failure M1 M2 (vs' @ (drop (length xs') xs))›
          ‹length (drop (length xs') xs) < length xs›
    using minimal-sequence-to-failure-extending.elims(2) by blast
  then show False
    using assms(1) by linarith
qed


lemma mstfe-no-repetition :
  assumes minimal-sequence-to-failure-extending V M1 M2 vs xs
  and     OFSM M1
  and     OFSM M2
  and     test-tools M2 M1 FAIL PM V Ω
  and     V'' ∈ N (vs@xs') M1 V
  and     prefix xs' xs
shows ¬ Rep-Pre M2 M1 vs xs'
  and ¬ Rep-Cov M2 M1 V'' vs xs'
  using minimal-sequence-to-failure-extending-implies-Rep-Pre[OF assms]
        minimal-sequence-to-failure-extending-implies-Rep-Cov[OF assms]
  by linarith+
```

6.2 Sufficiency of the test suite to test for reduction

The following lemma proves that set of input sequences generated in the final iteration of the `TS` function constitutes a test suite sufficient to test for reduction the FSMs it has been generated for.

This proof is performed by contradiction: If the test suite is not sufficient, then some minimal sequence to a failure extending the deterministic state cover must exist. Due to the test suite being assumed insufficient, this sequence cannot be contained in it and hence a prefix of it must have been contained in one of the sets calculated by the `R` function. This is only possible if the prefix is not a minimal sequence to a failure extending the deterministic state cover or if the test suite observes a failure, both of which violates the assumptions.

lemma *asc-sufficiency* :
assumes *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and *test-tools M2 M1 FAIL PM V Ω*
and *final-iteration M2 M1 Ω V m i*
shows *M1* $\preceq$⟦(*TS M2 M1 Ω V m i*) . Ω⟧ *M2* ⟶ *M1* $\preceq$ *M2*
proof
assume *atc-io-reduction-on-sets M1* (*TS M2 M1 Ω V m i*) *Ω M2*
show *M1* $\preceq$ *M2*
proof (*rule ccontr*)

let *?TS* = λ *n* . *TS M2 M1 Ω V m n*
let *?C* = λ *n* . *C M2 M1 Ω V m n*
let *?RM* = λ *n* . *RM M2 M1 Ω V m n*

assume ¬ *M1* $\preceq$ *M2*
obtain *vs xs* **where** *minimal-sequence-to-failure-extending V M1 M2 vs xs*
using *assms*(*1*) *assms*(*2*) *assms*(*4*)
minimal-sequence-to-failure-extending-det-state-cover-ob[*OF* - - - - ‹¬ *M1* $\preceq$ *M2*›, *of V*]
by *blast*

then have *vs* ∈ L_{in} *M1 V*
sequence-to-failure M1 M2 (*vs* @ *xs*)
¬ (∃ *io′* . ∃ *w′* ∈ L_{in} *M1 V* . *sequence-to-failure M1 M2* (*w′* @ *io′*)
∧ *length io′* < *length xs*)
by *auto*

then have *vs*@*xs* ∈ *L M1* − *L M2*
by *auto*

have *vs*@*xs* ∈ L_{in} *M1* {*map fst* (*vs*@*xs*)}
by (*metis* (*full-types*) *Diff-iff* ‹*vs* @ *xs* ∈ *L M1* − *L M2*› *insertI1*
language-state-for-inputs-map-fst)

have *vs*@*xs* ∉ L_{in} *M2* {*map fst* (*vs*@*xs*)}
by (*meson Diff-iff* ‹*vs* @ *xs* ∈ *L M1* − *L M2*› *language-state-for-inputs-in-language-state*
subsetCE)

have *finite V*
using *det-state-cover-finite assms*(*4*,*2*) **by** *auto*
then have *finite* (*?TS i*)
using *TS-finite*[*of V M2*] *assms*(*2*) **by** *auto*
then have *io-reduction-on M1* (*?TS i*) *M2*
using *io-reduction-from-atc-io-reduction*
[*OF* ‹*atc-io-reduction-on-sets M1* (*TS M2 M1 Ω V m i*) *Ω M2*›]
by *auto*

have *map fst* (*vs*@*xs*) ∉ *?TS i*
proof −
have *f1*: ∀ *ps P Pa*. (*ps*::(′*a* × ′*b*) *list*) ∉ *P* − *Pa* ∨ *ps* ∈ *P* ∧ *ps* ∉ *Pa*
by *blast*
have ∀ *P Pa ps*. ¬ *P* ⊆ *Pa* ∨ (*ps*::(′*a* × ′*b*) *list*) ∈ *Pa* ∨ *ps* ∉ *P*
by *blast*
then show *?thesis*
using *f1* **by** (*metis* (*no-types*) ‹*vs* @ *xs* ∈ *L M1* − *L M2*› ‹*io-reduction-on M1* (*?TS i*) *M2*›
language-state-for-inputs-in-language-state language-state-for-inputs-map-fst)
qed

have *map fst vs* ∈ *V*
using ‹*vs* ∈ L_{in} *M1 V*› **by** *auto*

let *?stf* = *map fst* (*vs*@*xs*)
let *?stfV* = *map fst vs*
let *?stfX* = *map fst xs*

have *?stf* = *?stfV* @ *?stfX*
  by *simp*

then have *?stfV* @ *?stfX* ∉ *?TS i*
  using ‹*?stf* ∉ *?TS i*› **by** *auto*

have *mcp* (*?stfV* @ *?stfX*) *V* *?stfV*
  by (*metis* ‹*map fst* (*vs* @ *xs*) = *map fst vs* @ *map fst xs*›
    ‹*minimal-sequence-to-failure-extending V M1 M2 vs xs*› *assms*(*1*) *assms*(*2*) *assms*(*4*)
    minimal-sequence-to-failure-extending-mcp)

have *set ?stf* ⊆ *inputs M1*
  by (*meson DiffD1* ‹*vs* @ *xs* ∈ *L M1* − *L M2*› *assms*(*1*) *language-state-inputs*)
then have *set ?stf* ⊆ *inputs M2*
  using *assms*(*3*) **by** *blast*
moreover have *set ?stf* = *set ?stfV* ∪ *set ?stfX*
  by *simp*
ultimately have *set ?stfX* ⊆ *inputs M2*
  by *blast*

obtain *xr j* **where** *xr* ≠ *?stfX*
    prefix xr ?stfX
    Suc j ≤ *i*
    ?stfV@*xr* ∈ *RM M2 M1* Ω *V m* (*Suc j*)
  using *TS-non-containment-causes-final-suc*[*OF* ‹*?stfV* @ *?stfX* ∉ *?TS i*›
    ‹*mcp* (*?stfV* @ *?stfX*) *V* *?stfV*› ‹*set ?stfX* ⊆ *inputs M2*› *assms*(*5*,*2*)]
  by *blast*

let *?yr* = *take* (*length xr*) (*map snd xs*)
have *length ?yr* = *length xr*
  using ‹*prefix xr* (*map fst xs*)› *prefix-length-le* **by** *fastforce*
have (*xr* || *?yr*) = *take* (*length xr*) *xs*
  by (*metis* (*no-types*, *opaque-lifting*) ‹*prefix xr* (*map fst xs*)› *append-eq-conv-conj prefixE take-zip*
    zip-map-fst-snd)

have *prefix* (*vs*@(*xr* || *?yr*)) (*vs*@*xs*)
  by (*simp add*: ‹*xr* || *take* (*length xr*) (*map snd xs*) = *take* (*length xr*) *xs*› *take-is-prefix*)

have *xr* = *take* (*length xr*) (*map fst xs*)
  by (*metis* ‹*length* (*take* (*length xr*) (*map snd xs*)) = *length xr*›
    ‹*xr* || *take* (*length xr*) (*map snd xs*) = *take* (*length xr*) *xs*› *map-fst-zip take-map*)

have *vs*@(*xr* || *?yr*) ∈ *L M1*
  by (*metis DiffD1* ‹*prefix* (*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*))) (*vs* @ *xs*)›
    ‹*vs* @ *xs* ∈ *L M1* − *L M2*› *language-state-prefix prefixE*)

then have *vs*@(*xr* || *?yr*) ∈ L_{in} *M1* {*?stfV* @ *xr*}
  by (*metis* ‹*length* (*take* (*length xr*) (*map snd xs*)) = *length xr*› *insertI1*
    language-state-for-inputs-map-fst map-append map-fst-zip)

have *length xr* < *length xs*
  by (*metis* ‹*xr* = *take* (*length xr*) (*map fst xs*)› ‹*xr* ≠ *map fst xs*› *not-le-imp-less take-all*
    take-map)

from ‹*?stfV*@*xr* ∈ *RM M2 M1* Ω *V m* (*Suc j*)› **have** *?stfV*@*xr* ∈ {*xs'* ∈ *C M2 M1* Ω *V m* (*Suc j*) .
  (¬ (L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*}))
  ∨ (∀ *io* ∈ L_{in} *M1* {*xs'*} .
    (∃ *V''* ∈ *N io M1 V* .
      (∃ *S1* .
        (∃ *vs xs* .
          io = (*vs*@*xs*)
          ∧ *mcp* (*vs*@*xs*) *V''* *vs*

$\wedge$ *S1* $\subseteq$ *nodes M2*
$\wedge$ ($\forall$ *s1* $\in$ *S1* . $\forall$ *s2* $\in$ *S1* .
s1 $\neq$ *s2* $\longrightarrow$
($\forall$ *io1* $\in$ *RP M2 s1 vs xs V''* .
$\forall$ *io2* $\in$ *RP M2 s2 vs xs V''* .
B M1 io1 Ω $\neq$ *B M1 io2* Ω))
$\wedge$ *m* $<$ *LB M2 M1 vs xs* (*TS M2 M1* Ω *V m j* $\cup$ *V*) *S1* Ω *V''*)))))}
unfolding *RM.simps* **by** *blast*

moreover have $\forall$ *xs'* $\in$ *?C* (*Suc j*) . L_{in} *M1* {*xs'*} $\subseteq$ L_{in} *M2* {*xs'*}
proof
fix *xs'* **assume** *xs'* $\in$ *?C* (*Suc j*)
from ‹*Suc j* $\leq$ *i*› **have** *?C* (*Suc j*) $\subseteq$ *?TS i*
using *C-subset TS-subset* **by** *blast*
then have {*xs'*} $\subseteq$ *?TS i*
using ‹*xs'* $\in$ *?C* (*Suc j*)› **by** *blast*
show L_{in} *M1* {*xs'*} $\subseteq$ L_{in} *M2* {*xs'*}
using *io-reduction-on-subset*[*OF* ‹*io-reduction-on M1* (*?TS i*) *M2*› ‹{*xs'*} $\subseteq$ *?TS i*›]
by *assumption*
qed

ultimately have ($\forall$ *io* $\in$ L_{in} *M1* {*?stfV@xr*} .
($\exists$ *V''* $\in$ *N io M1 V* .
($\exists$ *S1* .
($\exists$ *vs xs* .
io = (*vs@xs*)
$\wedge$ *mcp* (*vs@xs*) *V'' vs*
$\wedge$ *S1* $\subseteq$ *nodes M2*
$\wedge$ ($\forall$ *s1* $\in$ *S1* . $\forall$ *s2* $\in$ *S1* .
s1 $\neq$ *s2* $\longrightarrow$
($\forall$ *io1* $\in$ *RP M2 s1 vs xs V''* .
$\forall$ *io2* $\in$ *RP M2 s2 vs xs V''* .
B M1 io1 Ω $\neq$ *B M1 io2* Ω))
$\wedge$ *m* $<$ *LB M2 M1 vs xs* (*TS M2 M1* Ω *V m j* $\cup$ *V*) *S1* Ω *V''*))))
by *blast*

then have
($\exists$ *V''* $\in$ *N* (*vs@*(*xr* || *?yr*)) *M1 V* .
($\exists$ *S1* .
($\exists$ *vs' xs'* .
vs@(*xr* || *?yr*) = (*vs'@xs'*)
$\wedge$ *mcp* (*vs'@xs'*) *V'' vs'*
$\wedge$ *S1* $\subseteq$ *nodes M2*
$\wedge$ ($\forall$ *s1* $\in$ *S1* . $\forall$ *s2* $\in$ *S1* .
s1 $\neq$ *s2* $\longrightarrow$
($\forall$ *io1* $\in$ *RP M2 s1 vs' xs' V''* .
$\forall$ *io2* $\in$ *RP M2 s2 vs' xs' V''* .
B M1 io1 Ω $\neq$ *B M1 io2* Ω))
$\wedge$ *m* $<$ *LB M2 M1 vs' xs'* (*TS M2 M1* Ω *V m j* $\cup$ *V*) *S1* Ω *V''*)))
using ‹*vs@*(*xr* || *?yr*) $\in$ L_{in} *M1* {*?stfV @ xr*}›
by *blast*

then obtain *V'' S1 vs' xs'* **where** *RM-impl* :
V'' $\in$ *N* (*vs@*(*xr* || *?yr*)) *M1 V*
vs@(*xr* || *?yr*) = (*vs'@xs'*)
mcp (*vs'@xs'*) *V'' vs'*
S1 $\subseteq$ *nodes M2*
($\forall$ *s1* $\in$ *S1* . $\forall$ *s2* $\in$ *S1* .
s1 $\neq$ *s2* $\longrightarrow$
($\forall$ *io1* $\in$ *RP M2 s1 vs' xs' V''* .
$\forall$ *io2* $\in$ *RP M2 s2 vs' xs' V''* .
B M1 io1 Ω $\neq$ *B M1 io2* Ω))
m $<$ *LB M2 M1 vs' xs'* (*TS M2 M1* Ω *V m j* $\cup$ *V*) *S1* Ω *V''*
by *blast*

have *?stfV* = *mcp′* (*map fst* (*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)))) *V*
  by (*metis* (*full-types*) ‹*length* (*take* (*length xr*) (*map snd xs*)) = *length xr*›
    ‹*mcp* (*map fst vs* @ *map fst xs*) *V* (*map fst vs*)› ‹*prefix xr* (*map fst xs*)› *map-append*
    map-fst-zip mcp′-intro mcp-prefix-of-suffix)

have *is-det-state-cover M2 V*
  using *assms*(*4*) **by** *blast*
moreover have *well-formed M2*
  using *assms*(*2*) **by** *auto*
moreover have *finite V*
  using *det-state-cover-finite assms*(*4*,*2*) **by** *auto*
ultimately have *vs* ∈ *V′′*
    vs = *mcp′* (*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*))) *V′′*
  using *N-mcp-prefix*[*OF* ‹*?stfV* = *mcp′* (*map fst* (*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)))) *V*›
    ‹*V′′* ∈ *N* (*vs*@(*xr* || *?yr*)) *M1 V*›, *of M2*]
  by *simp*+

have *vs′* = *vs*
  by (*metis* (*no-types*) ‹*mcp* (*vs′* @ *xs′*) *V′′ vs′*›
    ‹*vs* = *mcp′* (*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*))) *V′′*›
    ‹*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)) = *vs′* @ *xs′*› *mcp′-intro*)

then have *xs′* = (*xr* || *?yr*)
  using ‹*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)) = *vs′* @ *xs′*› **by** *blast*

have *V* ⊆ *?TS i*
proof −
  have *1* ≤ *i*
    using ‹*Suc j* ≤ *i*› **by** *linarith*
  then have *?TS 1* ⊆ *?TS i*
    using *TS-subset* **by** *blast*
  then show *?thesis*
    by *auto*
qed

have *?stfV*@*xr* ∈ *?C* (*Suc j*)
  using ‹*?stfV*@*xr* ∈ *RM M2 M1* Ω *V m* (*Suc j*)› **unfolding** *RM.simps* **by** *blast*

— show that the prerequisites (`Prereq`) for `LB` are met by construction

have (∀ *vs′a*∈*V′′*. *prefix vs′a* (*vs′* @ *xs′*) ⟶ *length vs′a* ≤ *length vs′*)
  using ‹*mcp* (*vs′* @ *xs′*) *V′′ vs′*› **by** *auto*

moreover have *atc-io-reduction-on-sets M1* (*?TS j* ∪ *V*) Ω *M2*
proof −
  have *j* < *i*
    using ‹*Suc j* ≤ *i*› **by** *auto*
  then have *?TS j* ⊆ *?TS i*
    by (*simp add*: *TS-subset*)
  then show *?thesis*
    using *atc-io-reduction-on-subset*
      [*OF* ‹*atc-io-reduction-on-sets M1* (*TS M2 M1* Ω *V m i*) Ω *M2*›, *of ?TS j*]
    by (*meson Un-subset-iff* ‹*V* ⊆ *?TS i*› ‹*atc-io-reduction-on-sets M1* (*TS M2 M1* Ω *V m i*) Ω *M2*›
      atc-io-reduction-on-subset)
qed

moreover have *finite* (*?TS j* ∪ *V*)
proof −
  have *finite* (*?TS j*)
    using *TS-finite*[*OF* ‹*finite V*›, *of M2 M1* Ω *m j*] *assms*(*2*) **by** *auto*
  then show *?thesis*
    using ‹*finite V*› **by** *blast*
qed

moreover have *V* ⊆ *?TS j* ∪ *V*
  by *blast*

moreover have (∀ *p* . (*prefix p xs′* ∧ *p* ≠ *xs′*) ⟶ *map fst* (*vs′* @ *p*) ∈ *?TS j* ∪ *V*)
proof
  fix *p*
  show *prefix p xs′* ∧ *p* ≠ *xs′* ⟶ *map fst* (*vs′* @ *p*) ∈ *TS M2 M1* Ω *V m j* ∪ *V*
  proof
    assume *prefix p xs′* ∧ *p* ≠ *xs′*

    have *prefix* (*map fst* (*vs′* @ *p*)) (*map fst* (*vs′* @ *xs′*))
      by (*simp add*: ‹*prefix p xs′* ∧ *p* ≠ *xs′*› *map-mono-prefix*)
    have *prefix* (*map fst* (*vs′* @ *p*)) (*?stfV* @ *xr*)
      using ‹*length* (*take* (*length xr*) (*map snd xs*)) = *length xr*›
            ‹*prefix* (*map fst* (*vs′* @ *p*)) (*map fst* (*vs′* @ *xs′*))›
            ‹*vs′* = *vs*› ‹*xs′* = *xr* || *take* (*length xr*) (*map snd xs*)›
      by *auto*
    then have *prefix* (*map fst vs′* @ *map fst p*) (*?stfV* @ *xr*)
      by *simp*
    then have *prefix* (*map fst p*) *xr*
      by (*simp add*: ‹*vs′* = *vs*›)

    have *?stfV* @ *xr* ∈ *?TS* (*Suc j*)
    proof (*cases j*)
      case *0*
      then show *?thesis*
        using ‹*map fst vs* @ *xr* ∈ *C M2 M1* Ω *V m* (*Suc j*)› **by** *auto*
    next
      case (*Suc nat*)
      then show *?thesis*
        using *TS.simps*(*3*) ‹*map fst vs* @ *xr* ∈ *C M2 M1* Ω *V m* (*Suc j*)› **by** *blast*
    qed

    have *mcp* (*map fst vs* @ *xr*) *V* (*map fst vs*)
      using ‹*mcp* (*map fst vs* @ *map fst xs*) *V* (*map fst vs*)› ‹*prefix xr* (*map fst xs*)›
            mcp-prefix-of-suffix
      by *blast*

    have *map fst vs* @ *map fst p* ∈ *TS M2 M1* Ω *V m* (*Suc j*)
      using *TS-prefix-containment*[*OF* ‹*?stfV* @ *xr* ∈ *?TS* (*Suc j*)›
                                    ‹*mcp* (*map fst vs* @ *xr*) *V* (*map fst vs*)›
                                    ‹*prefix* (*map fst p*) *xr*›]
      by *assumption*

    have *Suc* (*length xr*) = (*Suc j*)
      using *C-index*[*OF* ‹*?stfV*@*xr* ∈ *?C* (*Suc j*)› ‹*mcp* (*map fst vs* @ *xr*) *V* (*map fst vs*)›]
      by *assumption*

    have*Suc* (*length p*) < (*Suc j*)
    proof −
      have *map fst xs′* = *xr*
        by (*metis* ‹*xr* = *take* (*length xr*) (*map fst xs*)›
            ‹*xr* || *take* (*length xr*) (*map snd xs*) = *take* (*length xr*) *xs*›
            ‹*xs′* = *xr* || *take* (*length xr*) (*map snd xs*)› *take-map*)
      then show *?thesis*
        by (*metis* (*no-types*) *Suc-less-eq* ‹*Suc* (*length xr*) = *Suc j*› ‹*prefix p xs′* ∧ *p* ≠ *xs′*›
            append-eq-conv-conj length-map nat-less-le prefixE prefix-length-le take-all)
    qed

    have *mcp* (*map fst vs* @ *map fst p*) *V* (*map fst vs*)
      using ‹*mcp* (*map fst vs* @ *xr*) *V* (*map fst vs*)› ‹*prefix* (*map fst p*) *xr*› *mcp-prefix-of-suffix*
      by *blast*

    then have *map fst vs* @ *map fst p* ∈ *?C* (*Suc* (*length* (*map fst p*)))

using *TS-index*(*2*)[*OF* ‹*map fst vs* @ *map fst p* ∈ *TS M2 M1* Ω *V m* (*Suc j*)›] **by** *auto*

have *map fst vs* @ *map fst p* ∈ *?TS j*
using *TS-union*[*of M2 M1* Ω *V m j*]
proof −
have *Suc* (*length p*) ∈ {*0*..<*Suc j*}
using ‹*Suc* (*length p*) < *Suc j*› **by** *force*
then show *?thesis*
by (*metis UN-I* ‹*TS M2 M1* Ω *V m j* = (⋃*j*∈*set* [*0*..<*Suc j*]. *C M2 M1* Ω *V m j*)›
‹*map fst vs* @ *map fst p* ∈ *C M2 M1* Ω *V m* (*Suc* (*length* (*map fst p*)))›
length-map set-upt)
qed

then show *map fst* (*vs′* @ *p*) ∈ *TS M2 M1* Ω *V m j* ∪ *V*
by (*simp add*: ‹*vs′* = *vs*›)
qed
qed

moreover have *vs′* @ *xs′* ∈ *L M2* ∩ *L M1*
by (*metis* (*no-types*, *lifting*) *IntI RM-impl*(*2*)
‹∀ *xs′*∈*C M2 M1* Ω *V m* (*Suc j*). L_{in} *M1* {*xs′*} ⊆ L_{in} *M2* {*xs′*}›
‹*map fst vs* @ *xr* ∈ *C M2 M1* Ω *V m* (*Suc j*)›
‹*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)) ∈ L_{in} *M1* {*map fst vs* @ *xr*}›
language-state-for-inputs-in-language-state subsetCE)

ultimately have *Prereq M2 M1 vs′ xs′* (*?TS j* ∪ *V*) *S1* Ω *V′′*
using *RM-impl*(*4*,*5*) **unfolding** *Prereq.simps* **by** *blast*

have *V′′* ∈ *Perm V M1*
using ‹*V′′* ∈ *N* (*vs*@(*xr* || *?yr*)) *M1 V*› **unfolding** *N.simps* **by** *blast*

have ‹*prefix* (*xr* || *?yr*) *xs*›
by (*simp add*: ‹*xr* || *take* (*length xr*) (*map snd xs*) = *take* (*length xr*) *xs*› *take-is-prefix*)

— show that furthermore neither `Rep_Pre` nor `Rep_Cov` holds

have ¬ *Rep-Pre M2 M1 vs* (*xr* || *?yr*)
using *minimal-sequence-to-failure-extending-implies-Rep-Pre*
[*OF* ‹*minimal-sequence-to-failure-extending V M1 M2 vs xs*› *assms*(*1*,*2*)
‹*test-tools M2 M1 FAIL PM V* Ω› *RM-impl*(*1*)
‹*prefix* (*xr* || *take* (*length xr*) (*map snd xs*)) *xs*›]
by *assumption*
then have ¬ *Rep-Pre M2 M1 vs′ xs′*
using ‹*vs′* = *vs*› ‹*xs′* = *xr* || *?yr*› **by** *blast*

have ¬ *Rep-Cov M2 M1 V′′ vs* (*xr* || *?yr*)
using *minimal-sequence-to-failure-extending-implies-Rep-Cov*
[*OF* ‹*minimal-sequence-to-failure-extending V M1 M2 vs xs*› *assms*(*1*,*2*)
‹*test-tools M2 M1 FAIL PM V* Ω› *RM-impl*(*1*)
‹*prefix* (*xr* || *take* (*length xr*) (*map snd xs*)) *xs*›]
by *assumption*
then have ¬ *Rep-Cov M2 M1 V′′ vs′ xs′*
using ‹*vs′* = *vs*› ‹*xs′* = *xr* || *?yr*› **by** *blast*

have *vs′*@*xs′* ∈ *L M1*
using ‹*vs* @ (*xr* || *take* (*length xr*) (*map snd xs*)) ∈ *L M1*›
‹*vs′* = *vs*› ‹*xs′* = *xr* || *take* (*length xr*) (*map snd xs*)›
by *blast*

— therefore it is impossible to remove the prefix of the minimal sequence to a failure, as this would require `M1` to have more than m states

have *LB M2 M1 vs′ xs′ (?TS j ∪ V) S1 Ω V′′ ≤ card (nodes M1)*
using *LB-count[OF ‹vs′@xs′ ∈ L M1› assms(1,2,3) ‹test-tools M2 M1 FAIL PM V Ω›
‹V′′ ∈ Perm V M1› ‹Prereq M2 M1 vs′ xs′ (?TS j ∪ V) S1 Ω V′′›
‹¬ Rep-Pre M2 M1 vs′ xs′› ‹ ¬ Rep-Cov M2 M1 V′′ vs′ xs′›]*
by *assumption*
then have *LB M2 M1 vs′ xs′ (?TS j ∪ V) S1 Ω V′′ ≤ m*
using *assms(3)* **by** *linarith*

then show *False*
using *‹m < LB M2 M1 vs′ xs′ (?TS j ∪ V) S1 Ω V′′›* **by** *linarith*
qed
qed

6.3 Main result

The following lemmata add to the previous result to show that some FSM `M1` is a reduction of FSM `M2` if and only if it is a reduction on the test suite generated by the adaptive state counting algorithm for these FSMs.

lemma *asc-soundness* :
assumes *OFSM M1*
and *OFSM M2*
shows *M1 ≼ M2 ⟶ atc-io-reduction-on-sets M1 T Ω M2*
using *atc-io-reduction-on-sets-reduction assms* **by** *blast*

lemma *asc-main-theorem* :
assumes *OFSM M1*
and *OFSM M2*
and *asc-fault-domain M2 M1 m*
and *test-tools M2 M1 FAIL PM V Ω*
and *final-iteration M2 M1 Ω V m i*
shows *M1 ≼ M2 ⟷ atc-io-reduction-on-sets M1 (TS M2 M1 Ω V m i) Ω M2*
by *(metis asc-sufficiency assms(1−5) atc-io-reduction-on-sets-reduction)*

end
theory *ASC-Hoare*
imports *ASC-Sufficiency HOL−Hoare.Hoare-Logic*
begin

7 Correctness of the Adaptive State Counting Algorithm in Hoare-Logic

In this section we give an example implementation of the adaptive state counting algorithm in a simple WHILE-language and prove that this implementation produces a certain output if and only if input FSM `M1` is a reduction of input FSM `M2`.

lemma *atc-io-reduction-on-sets-from-obs* :
assumes L_{in} *M1 T* ⊆ L_{in} *M2 T*
and (⋃ *io*∈L_{in} *M1 T*. {*io*} × *B M1 io* Ω) ⊆ (⋃ *io*∈L_{in} *M2 T*. {*io*} × *B M2 io* Ω)
shows *atc-io-reduction-on-sets M1 T Ω M2*
unfolding *atc-io-reduction-on-sets.simps atc-io-reduction-on.simps*
proof
fix *iseq* **assume** *iseq ∈ T*
have L_{in} *M1* {*iseq*} ⊆ L_{in} *M2* {*iseq*}
by *(metis ‹iseq ∈ T› assms(1) bot.extremum insert-mono io-reduction-on-subset
mk-disjoint-insert)*
moreover have ∀ *io*∈L_{in} *M1* {*iseq*}. *B M1 io Ω ⊆ B M2 io Ω*
proof
fix *io* **assume** *io* ∈ L_{in} *M1* {*iseq*}
then have *io* ∈ L_{in} *M2* {*iseq*}

using *calculation* **by** *blast*
show $B\ M1\ io\ \Omega \subseteq B\ M2\ io\ \Omega$
proof
fix x **assume** $x \in B\ M1\ io\ \Omega$

have $io \in L_{in}\ M1\ T$
using ‹$io \in L_{in}\ M1\ \{iseq\}$› ‹$iseq \in T$› **by** *auto*
moreover have $(io,x) \in \{io\} \times B\ M1\ io\ \Omega$
using ‹$x \in B\ M1\ io\ \Omega$› **by** *blast*
ultimately have $(io,x) \in (\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega)$
by *blast*

then have $(io,x) \in (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega)$
using *assms(2)* **by** *blast*
then have $(io,x) \in \{io\} \times B\ M2\ io\ \Omega$
by *blast*
then show $x \in B\ M2\ io\ \Omega$
by *blast*
qed
qed
ultimately show $L_{in}\ M1\ \{iseq\} \subseteq L_{in}\ M2\ \{iseq\}$
$\wedge (\forall io{\in}L_{in}\ M1\ \{iseq\}.\ B\ M1\ io\ \Omega \subseteq B\ M2\ io\ \Omega)$
by *linarith*
qed

lemma *atc-io-reduction-on-sets-to-obs* :
assumes *atc-io-reduction-on-sets* $M1\ T\ \Omega\ M2$
shows $L_{in}\ M1\ T \subseteq L_{in}\ M2\ T$
and $(\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega) \subseteq (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega)$
proof
fix x **assume** $x \in L_{in}\ M1\ T$
show $x \in L_{in}\ M2\ T$
using *assms* **unfolding** *atc-io-reduction-on-sets.simps atc-io-reduction-on.simps*
proof −
assume *a1*: $\forall iseq{\in}T.\ L_{in}\ M1\ \{iseq\} \subseteq L_{in}\ M2\ \{iseq\}$
$\wedge (\forall io{\in}L_{in}\ M1\ \{iseq\}.\ B\ M1\ io\ \Omega \subseteq B\ M2\ io\ \Omega)$
have *f2*: $x \in UNION\ T\ (language\text{-}state\text{-}for\text{-}input\ M1\ (initial\ M1))$
by (*metis* (*no-types*) ‹$x \in L_{in}\ M1\ T$› *language-state-for-inputs-alt-def*)
obtain *aas* :: $'a\ list\ set \Rightarrow ('a\ list \Rightarrow ('a \times 'b)\ list\ set) \Rightarrow ('a \times 'b)\ list \Rightarrow 'a\ list$
where
$\forall x0\ x1\ x2.\ (\exists v3.\ v3 \in x0 \wedge x2 \in x1\ v3) = (aas\ x0\ x1\ x2 \in x0 \wedge x2 \in x1\ (aas\ x0\ x1\ x2))$
by *moura*
then have $\forall ps\ f\ A.\ (ps \notin UNION\ A\ f \vee aas\ A\ f\ ps \in A \wedge ps \in f\ (aas\ A\ f\ ps))$
$\wedge (ps \in UNION\ A\ f \vee (\forall as.\ as \notin A \vee ps \notin f\ as))$
by *blast*
then show *?thesis*
using *f2 a1* **by** (*metis* (*no-types*) *contra-subsetD language-state-for-input-alt-def*
language-state-for-inputs-alt-def)
qed
next
show $(\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega) \subseteq (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega)$
proof
fix *iox* **assume** $iox \in (\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega)$
then obtain *io x* **where** $iox = (io,x)$
by *blast*

have $io \in L_{in}\ M1\ T$
using ‹$iox = (io, x)$› ‹$iox \in (\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega)$› **by** *blast*
have $(io,x) \in \{io\} \times B\ M1\ io\ \Omega$
using ‹$iox = (io, x)$› ‹$iox \in (\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega)$› **by** *blast*
then have $x \in B\ M1\ io\ \Omega$
by *blast*

then have $x \in B\ M2\ io\ \Omega$
using *assms* **unfolding** *atc-io-reduction-on-sets.simps atc-io-reduction-on.simps*

by (*metis* (*no-types*, *lifting*) *UN-E* ‹$io \in L_{in}$ *M1 T*› *language-state-for-input-alt-def language-state-for-inputs-alt-def subsetCE*)
then have $(io,x) \in \{io\} \times B\ M2\ io\ \Omega$
by *blast*
then have $(io,x) \in (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega)$
using ‹$io \in L_{in}\ M1\ T$› **by** *auto*
then show $iox \in (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega)$
using ‹$iox = (io, x)$› **by** *auto*
qed
qed

lemma *atc-io-reduction-on-sets-alt-def* :
shows *atc-io-reduction-on-sets M1 T Ω M2* =
$(L_{in}\ M1\ T \subseteq L_{in}\ M2\ T$
$\wedge (\bigcup io{\in}L_{in}\ M1\ T.\ \{io\} \times B\ M1\ io\ \Omega)$
$\subseteq (\bigcup io{\in}L_{in}\ M2\ T.\ \{io\} \times B\ M2\ io\ \Omega))$
using *atc-io-reduction-on-sets-to-obs*[*of M1 T Ω M2*]
and *atc-io-reduction-on-sets-from-obs*[*of M1 T M2 Ω*]
by *blast*

lemma *asc-algorithm-correctness*:
VARS tsN cN rmN obs obsI obs_Ω $obsI_\Omega$ *iter isReduction*
{
OFSM M1 ∧ *OFSM M2* ∧ *asc-fault-domain M2 M1 m* ∧ *test-tools M2 M1 FAIL PM V Ω*
}
tsN := {};
cN := *V*;
rmN := {};
obs := L_{in} *M2 cN*;
obsI := L_{in} *M1 cN*;
$obs_\Omega := (\bigcup io{\in}L_{in}\ M2\ cN.\ \{io\} \times B\ M2\ io\ \Omega)$;
$obsI_\Omega := (\bigcup io{\in}L_{in}\ M1\ cN.\ \{io\} \times B\ M1\ io\ \Omega)$;
iter := *1*;
WHILE $(cN \neq \{\} \wedge obsI \subseteq obs \wedge obsI_\Omega \subseteq obs_\Omega)$
INV {
$0 < iter$
$\wedge\ tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
$\wedge\ cN = C\ M2\ M1\ \Omega\ V\ m\ iter$
$\wedge\ rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
$\wedge\ obs = L_{in}\ M2\ (tsN \cup cN)$
$\wedge\ obsI = L_{in}\ M1\ (tsN \cup cN)$
$\wedge\ obs_\Omega = (\bigcup io{\in}L_{in}\ M2\ (tsN \cup cN).\ \{io\} \times B\ M2\ io\ \Omega)$
$\wedge\ obsI_\Omega = (\bigcup io{\in}L_{in}\ M1\ (tsN \cup cN).\ \{io\} \times B\ M1\ io\ \Omega)$
∧ *OFSM M1* ∧ *OFSM M2* ∧ *asc-fault-domain M2 M1 m* ∧ *test-tools M2 M1 FAIL PM V Ω*
}
DO
iter := *iter* + *1*;
$rmN := \{xs' \in cN$.
$(\neg (L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\}))$
$\vee (\forall\ io \in L_{in}\ M1\ \{xs'\}$.
$(\exists\ V'' \in N\ io\ M1\ V$.
$(\exists\ S1$.
$(\exists\ vs\ xs$.
$io = (vs@xs)$
$\wedge\ mcp\ (vs@xs)\ V''\ vs$
$\wedge\ S1 \subseteq nodes\ M2$
$\wedge (\forall\ s1 \in S1 . \forall\ s2 \in S1$.
$s1 \neq s2 \longrightarrow$
$(\forall\ io1 \in RP\ M2\ s1\ vs\ xs\ V''$.
$\forall\ io2 \in RP\ M2\ s2\ vs\ xs\ V''$.
$B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega\ ))$
$\wedge\ m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V''\ ))))\}$;

$tsN := tsN \cup cN$;
$cN := append\text{-}set\ (cN - rmN)\ (inputs\ M2) - tsN$;
$obs := obs \cup L_{in}\ M2\ cN$;
$obsI := obsI \cup L_{in}\ M1\ cN$;
$obs_{\Omega} := obs_{\Omega} \cup (\bigcup io{\in}L_{in}\ M2\ cN.\ \{io\} \times B\ M2\ io\ \Omega)$;
$obsI_{\Omega} := obsI_{\Omega} \cup (\bigcup io{\in}L_{in}\ M1\ cN.\ \{io\} \times B\ M1\ io\ \Omega)$
OD;
$isReduction := ((obsI \subseteq obs) \wedge (obsI_{\Omega} \subseteq obs_{\Omega}))$
{
$isReduction = M1 \preceq M2$ — variable isReduction is used only as a return value, it is true if and only if M1 is a reduction of M2
}
proof (*vcg*)
assume *precond* : $OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
have $\{\} = TS\ M2\ M1\ \Omega\ V\ m\ (1{-}1)$
$V = C\ M2\ M1\ \Omega\ V\ m\ 1$
$\{\} = RM\ M2\ M1\ \Omega\ V\ m\ (1{-}1)$
$L_{in}\ M2\ V = L_{in}\ M2\ (\{\} \cup V)$
$L_{in}\ M1\ V = L_{in}\ M1\ (\{\} \cup V)$
$(\bigcup io{\in}L_{in}\ M2\ V.\ \{io\} \times B\ M2\ io\ \Omega)$
$= (\bigcup io{\in}L_{in}\ M2\ (\{\} \cup V).\ \{io\} \times B\ M2\ io\ \Omega)$
$(\bigcup io{\in}L_{in}\ M1\ V.\ \{io\} \times B\ M1\ io\ \Omega)$
$= (\bigcup io{\in}L_{in}\ M1\ (\{\} \cup V).\ \{io\} \times B\ M1\ io\ \Omega)$
using *precond* **by** *auto*
moreover have $OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
using *precond* **by** *assumption*
ultimately show $0 < (1{::}nat) \wedge$
$\{\} = TS\ M2\ M1\ \Omega\ V\ m\ (1 - 1) \wedge$
$V = C\ M2\ M1\ \Omega\ V\ m\ 1 \wedge$
$\{\} = RM\ M2\ M1\ \Omega\ V\ m\ (1 - 1) \wedge$
$L_{in}\ M2\ V = L_{in}\ M2\ (\{\} \cup V) \wedge$
$L_{in}\ M1\ V = L_{in}\ M1\ (\{\} \cup V) \wedge$
$(\bigcup io{\in}L_{in}\ M2\ V.\ \{io\} \times B\ M2\ io\ \Omega)$
$= (\bigcup io{\in}L_{in}\ M2\ (\{\} \cup V).\ \{io\} \times B\ M2\ io\ \Omega) \wedge$
$(\bigcup io{\in}L_{in}\ M1\ V.\ \{io\} \times B\ M1\ io\ \Omega)$
$= (\bigcup io{\in}L_{in}\ M1\ (\{\} \cup V).\ \{io\} \times B\ M1\ io\ \Omega) \wedge$
$OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
by *linarith+*
next
fix $tsN\ cN\ rmN\ obs\ obsI\ obs_{\Omega}\ obsI_{\Omega}\ iter\ isReduction$
assume *precond* : $(0 < iter \wedge$
$tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter - 1) \wedge$
$cN = C\ M2\ M1\ \Omega\ V\ m\ iter \wedge$
$rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter - 1) \wedge$
$obs = L_{in}\ M2\ (tsN \cup cN) \wedge$
$obsI = L_{in}\ M1\ (tsN \cup cN) \wedge$
$obs_{\Omega} = (\bigcup io{\in}L_{in}\ M2\ (tsN \cup cN).\ \{io\} \times B\ M2\ io\ \Omega) \wedge$
$obsI_{\Omega} = (\bigcup io{\in}L_{in}\ M1\ (tsN \cup cN).\ \{io\} \times B\ M1\ io\ \Omega) \wedge$
$OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega)$
$\wedge\ cN \neq \{\} \wedge obsI \subseteq obs \wedge obsI_{\Omega} \subseteq obs_{\Omega}$
then have $0 < iter$
$OFSM\ M1$
$OFSM\ M2$
$asc\text{-}fault\text{-}domain\ M2\ M1\ m$
$test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
$cN \neq \{\}$
$obsI \subseteq obs$
$tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter{-}1)$
$cN = C\ M2\ M1\ \Omega\ V\ m\ iter$
$rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter{-}1)$
$obs = L_{in}\ M2\ (tsN \cup cN)$
$obsI = L_{in}\ M1\ (tsN \cup cN)$
$obs_{\Omega} = (\bigcup io{\in}L_{in}\ M2\ (tsN \cup cN).\ \{io\} \times B\ M2\ io\ \Omega)$
$obsI_{\Omega} = (\bigcup io{\in}L_{in}\ M1\ (tsN \cup cN).\ \{io\} \times B\ M1\ io\ \Omega)$
by *linarith+*

obtain *k* **where** *iter* = *Suc k*
  using *gr0-implies-Suc*[*OF* ‹*0* < *iter*›] **by** *blast*
then have *cN* = *C M2 M1 Ω V m* (*Suc k*)
    tsN = *TS M2 M1 Ω V m k*
  using ‹*cN* = *C M2 M1 Ω V m iter*› ‹*tsN* = *TS M2 M1 Ω V m* (*iter*−*1*)› **by** *auto*
have *TS M2 M1 Ω V m iter* = *TS M2 M1 Ω V m* (*Suc k*)
    C M2 M1 Ω V m iter = *C M2 M1 Ω V m* (*Suc k*)
    RM M2 M1 Ω V m iter = *RM M2 M1 Ω V m* (*Suc k*)
  using ‹*iter* = *Suc k*› **by** *presburger*+

have *rmN-calc*[*simp*] : {*xs*′ ∈ *cN*.
    ¬ *io-reduction-on M1* {*xs*′} *M2* ∨
    (∀ *io*∈L_{in} *M1* {*xs*′}.
      ∃ *V*′′∈*N io M1 V*.
        ∃ *S1 vs xs*.
          io = *vs* @ *xs* ∧
          mcp (*vs* @ *xs*) *V*′′ *vs* ∧
          S1 ⊆ *nodes M2* ∧
          (∀ *s1*∈*S1*.
            ∀ *s2*∈*S1*.
              s1 ≠ *s2* ⟶
              (∀ *io1*∈*RP M2 s1 vs xs V*′′. ∀ *io2*∈*RP M2 s2 vs xs V*′′.
                B M1 io1 Ω ≠ *B M1 io2 Ω*)) ∧
          m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1 Ω V*′′)} =
   RM M2 M1 Ω V m iter
proof −

  have {*xs*′ ∈ *cN*.
      ¬ *io-reduction-on M1* {*xs*′} *M2* ∨
      (∀ *io*∈L_{in} *M1* {*xs*′}.
        ∃ *V*′′∈*N io M1 V*.
          ∃ *S1 vs xs*.
            io = *vs* @ *xs* ∧
            mcp (*vs* @ *xs*) *V*′′ *vs* ∧
            S1 ⊆ *nodes M2* ∧
            (∀ *s1*∈*S1*.
              ∀ *s2*∈*S1*.
                s1 ≠ *s2* ⟶
                (∀ *io1*∈*RP M2 s1 vs xs V*′′. ∀ *io2*∈*RP M2 s2 vs xs V*′′.
                  B M1 io1 Ω ≠ *B M1 io2 Ω*)) ∧
            m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1 Ω V*′′)} =
      {*xs*′ ∈ *C M2 M1 Ω V m* (*Suc k*).
      ¬ *io-reduction-on M1* {*xs*′} *M2* ∨
      (∀ *io*∈L_{in} *M1* {*xs*′}.
        ∃ *V*′′∈*N io M1 V*.
          ∃ *S1 vs xs*.
            io = *vs* @ *xs* ∧
            mcp (*vs* @ *xs*) *V*′′ *vs* ∧
            S1 ⊆ *nodes M2* ∧
            (∀ *s1*∈*S1*.
              ∀ *s2*∈*S1*.
                s1 ≠ *s2* ⟶
                (∀ *io1*∈*RP M2 s1 vs xs V*′′. ∀ *io2*∈*RP M2 s2 vs xs V*′′.
                  B M1 io1 Ω ≠ *B M1 io2 Ω*)) ∧
            m < *LB M2 M1 vs xs* ((*TS M2 M1 Ω V m k*) ∪ *V*) *S1 Ω V*′′)}
    using ‹*cN* = *C M2 M1 Ω V m* (*Suc k*)› ‹*tsN* = *TS M2 M1 Ω V m k*› **by** *blast*

  moreover have {*xs*′ ∈ *C M2 M1 Ω V m* (*Suc k*).
          ¬ *io-reduction-on M1* {*xs*′} *M2* ∨
          (∀ *io*∈L_{in} *M1* {*xs*′}.
            ∃ *V*′′∈*N io M1 V*.
              ∃ *S1 vs xs*.
                io = *vs* @ *xs* ∧

```
                    mcp (vs @ xs) V'' vs ∧
                    S1 ⊆ nodes M2 ∧
                    (∀ s1∈S1.
                        ∀ s2∈S1.
                            s1 ≠ s2 ⟶
                            (∀ io1∈RP M2 s1 vs xs V''. ∀ io2∈RP M2 s2 vs xs V''.
                                B M1 io1 Ω ≠ B M1 io2 Ω)) ∧
                    m < LB M2 M1 vs xs ((TS M2 M1 Ω V m k) ∪ V) S1 Ω V'')} =
            RM M2 M1 Ω V m (Suc k)
    using RM.simps(2)[of M2 M1 Ω V m k] by blast
  ultimately have {xs' ∈ cN.
            ¬ io-reduction-on M1 {xs'} M2 ∨
            (∀ io∈L_in M1 {xs'}.
                ∃ V''∈N io M1 V.
                    ∃ S1 vs xs.
                        io = vs @ xs ∧
                        mcp (vs @ xs) V'' vs ∧
                        S1 ⊆ nodes M2 ∧
                        (∀ s1∈S1.
                            ∀ s2∈S1.
                                s1 ≠ s2 ⟶
                                (∀ io1∈RP M2 s1 vs xs V''. ∀ io2∈RP M2 s2 vs xs V''.
                                    B M1 io1 Ω ≠ B M1 io2 Ω)) ∧
                        m < LB M2 M1 vs xs (tsN ∪ V) S1 Ω V'')} =
            RM M2 M1 Ω V m (Suc k)
    by presburger
  then show ?thesis
    using ‹iter = Suc k› by presburger
qed
moreover have RM M2 M1 Ω V m iter = RM M2 M1 Ω V m (iter + 1 − 1) by simp
ultimately have rmN-calc' : {xs' ∈ cN.
      ¬ io-reduction-on M1 {xs'} M2 ∨
      (∀ io∈L_in M1 {xs'}.
          ∃ V''∈N io M1 V.
              ∃ S1 vs xs.
                  io = vs @ xs ∧
                  mcp (vs @ xs) V'' vs ∧
                  S1 ⊆ nodes M2 ∧
                  (∀ s1∈S1.
                      ∀ s2∈S1.
                          s1 ≠ s2 ⟶
                          (∀ io1∈RP M2 s1 vs xs V''. ∀ io2∈RP M2 s2 vs xs V''.
                              B M1 io1 Ω ≠ B M1 io2 Ω)) ∧
                  m < LB M2 M1 vs xs (tsN ∪ V) S1 Ω V'')} =
    RM M2 M1 Ω V m (iter + 1 − 1) by presburger

have tsN ∪ cN = TS M2 M1 Ω V m (Suc k)
proof (cases k)
  case 0
  then show ?thesis
    using ‹tsN = TS M2 M1 Ω V m k› ‹cN = C M2 M1 Ω V m (Suc k)› by auto
next
  case (Suc nat)
  then have TS M2 M1 Ω V m (Suc k) = TS M2 M1 Ω V m k ∪ C M2 M1 Ω V m (Suc k)
    using TS.simps(3) by blast
  moreover have tsN ∪ cN = TS M2 M1 Ω V m k ∪ C M2 M1 Ω V m (Suc k)
    using ‹tsN = TS M2 M1 Ω V m k› ‹cN = C M2 M1 Ω V m (Suc k)› by auto
  ultimately show ?thesis
    by auto
qed
then have tsN-calc : tsN ∪ cN = TS M2 M1 Ω V m iter
  using ‹iter = Suc k› by presburger


have cN-calc : append-set
      (cN −
```

$\{xs' \in cN.$
$\neg$ *io-reduction-on M1* $\{xs'\}$ *M2* $\vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
$S1 \subseteq nodes\ M2 \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN) =$
$C\ M2\ M1\ \Omega\ V\ m\ (iter + 1)$
proof –
have *append-set*
$(cN -$
$\{xs' \in cN.$
$\neg$ *io-reduction-on M1* $\{xs'\}$ *M2* $\vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
$S1 \subseteq nodes\ M2 \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN) =$
append-set
$((C\ M2\ M1\ \Omega\ V\ m\ iter) -$
$(RM\ M2\ M1\ \Omega\ V\ m\ iter))$
$(inputs\ M2) -$
$(TS\ M2\ M1\ \Omega\ V\ m\ iter)$
using ‹$cN = C\ M2\ M1\ \Omega\ V\ m\ iter$› ‹$tsN \cup cN = TS\ M2\ M1\ \Omega\ V\ m\ iter$› *rmN-calc* **by** *presburger*
moreover have *append-set*
$((C\ M2\ M1\ \Omega\ V\ m\ iter) -$
$(RM\ M2\ M1\ \Omega\ V\ m\ iter))$
$(inputs\ M2) -$
$(TS\ M2\ M1\ \Omega\ V\ m\ iter) = C\ M2\ M1\ \Omega\ V\ m\ (iter + 1)$
proof –
have $C\ M2\ M1\ \Omega\ V\ m\ (iter + 1) = C\ M2\ M1\ \Omega\ V\ m\ ((Suc\ k) + 1)$
using ‹$iter = Suc\ k$› **by** *presburger+*
moreover have $(Suc\ k) + 1 = Suc\ (Suc\ k)$
by *simp*
ultimately have $C\ M2\ M1\ \Omega\ V\ m\ (iter + 1) = C\ M2\ M1\ \Omega\ V\ m\ (Suc\ (Suc\ k))$
by *presburger*

have $C\ M2\ M1\ \Omega\ V\ m\ (Suc\ (Suc\ k))$
$= append\text{-}set\ (C\ M2\ M1\ \Omega\ V\ m\ (Suc\ k) - RM\ M2\ M1\ \Omega\ V\ m\ (Suc\ k))\ (inputs\ M2)$
$- TS\ M2\ M1\ \Omega\ V\ m\ (Suc\ k)$
using *C.simps(3)[of M2 M1 Ω V m k]* **by** *linarith*
show *?thesis*
using *Suc-eq-plus1*
‹$C\ M2\ M1\ \Omega\ V\ m\ (Suc\ (Suc\ k))$
$= append\text{-}set\ (C\ M2\ M1\ \Omega\ V\ m\ (Suc\ k) - RM\ M2\ M1\ \Omega\ V\ m\ (Suc\ k))\ (inputs\ M2)$
$- TS\ M2\ M1\ \Omega\ V\ m\ (Suc\ k)$›
‹$iter = Suc\ k$›
by *presburger*

qed

ultimately show *?thesis*
by *presburger*
qed

have *obs-calc* : *obs* ∪
L_{in} *M2*
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs V''*.
∀ *io2*∈*RP M2 s2 vs xs V''*. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω *V''*)})
(*inputs M2*) −
(*tsN* ∪ *cN*)) =
L_{in} *M2*
(*tsN* ∪ *cN* ∪
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs V''*.
∀ *io2*∈*RP M2 s2 vs xs V''*. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω *V''*)})
(*inputs M2*) −
(*tsN* ∪ *cN*)))
proof −
have ⋀*A*. L_{in} *M2* (*tsN* ∪ *cN* ∪ *A*) = *obs* ∪ L_{in} *M2 A*
by (*metis* (*no-types*) *language-state-for-inputs-union precond*)
then show *?thesis*
by *blast*
qed

have *obsI-calc* : *obsI* ∪
L_{in} *M1*
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧

S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs V''*.
∀ *io2*∈*RP M2 s2 vs xs V''*. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω *V''*)})
(*inputs M2*) −
(*tsN* ∪ *cN*)) =
L_{in} *M1*
(*tsN* ∪ *cN* ∪
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs V''*.
∀ *io2*∈*RP M2 s2 vs xs V''*. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω *V''*)})
(*inputs M2*) −
(*tsN* ∪ *cN*)))
proof −
have ⋀*A*. L_{in} *M1* (*tsN* ∪ *cN* ∪ *A*) = *obsI* ∪ L_{in} *M1 A*
by (*metis* (*no-types*) *language-state-for-inputs-union precond*)
then show *?thesis*
by *blast*
qed

have obs_{Ω}*-calc* : obs_{Ω} ∪
(⋃ *io*∈L_{in} *M2*
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) *V'' vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs V''*.
∀ *io2*∈*RP M2 s2 vs xs V''*. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω *V''*)})
(*inputs M2*) −
(*tsN* ∪ *cN*)).
{*io*} × *B M2 io* Ω) =
(⋃ *io*∈L_{in} *M2*
(*tsN* ∪ *cN* ∪
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ *V''*∈*N io M1 V*.
∃ *S1 vs xs*.

io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) V'' *vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs* V''.
∀ *io2*∈*RP M2 s2 vs xs* V''. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω V'')})
(*inputs M2*) −
(*tsN* ∪ *cN*))).
{*io*} × *B M2 io* Ω)
using ‹*obs* = L_{in} *M2* (*tsN* ∪ *cN*)›
‹obs_Ω = (⋃ *io*∈L_{in} *M2* (*tsN* ∪ *cN*). {*io*} × *B M2 io* Ω)›
obs-calc
by *blast*

have $obsI_\Omega$*-calc* : $obsI_\Omega$ ∪
(⋃ *io*∈L_{in} *M1*
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ V''∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) V'' *vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs* V''.
∀ *io2*∈*RP M2 s2 vs xs* V''. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω V'')})
(*inputs M2*) −
(*tsN* ∪ *cN*)).
{*io*} × *B M1 io* Ω) =
(⋃ *io*∈L_{in} *M1*
(*tsN* ∪ *cN* ∪
(*append-set*
(*cN* −
{*xs'* ∈ *cN*.
¬ L_{in} *M1* {*xs'*} ⊆ L_{in} *M2* {*xs'*} ∨
(∀ *io*∈L_{in} *M1* {*xs'*}.
∃ V''∈*N io M1 V*.
∃ *S1 vs xs*.
io = *vs* @ *xs* ∧
mcp (*vs* @ *xs*) V'' *vs* ∧
S1 ⊆ *nodes M2* ∧
(∀ *s1*∈*S1*.
∀ *s2*∈*S1*.
s1 ≠ *s2* ⟶
(∀ *io1*∈*RP M2 s1 vs xs* V''.
∀ *io2*∈*RP M2 s2 vs xs* V''. *B M1 io1* Ω ≠ *B M1 io2* Ω)) ∧
m < *LB M2 M1 vs xs* (*tsN* ∪ *V*) *S1* Ω V'')})
(*inputs M2*) −
(*tsN* ∪ *cN*))).
{*io*} × *B M1 io* Ω)
using ‹*obsI* = L_{in} *M1* (*tsN* ∪ *cN*)›
‹$obsI_\Omega$ = (⋃ *io*∈L_{in} *M1* (*tsN* ∪ *cN*). {*io*} × *B M1 io* Ω)›
obsI-calc
by *blast*

have $0 < iter + 1$
  using ‹$0 < iter$› **by** *simp*
have $tsN \cup cN = TS\ M2\ M1\ \Omega\ V\ m\ (iter + 1 - 1)$
  using *tsN-calc* **by** *simp*

from ‹$0 < iter + 1$›
  ‹$tsN \cup cN = TS\ M2\ M1\ \Omega\ V\ m\ (iter + 1 - 1)$›
  cN-calc
  rmN-calc'
  obs-calc
  obsI-calc
  obs_{Ω}*-calc*
  $obsI_{\Omega}$*-calc*
  ‹*OFSM M1*›
  ‹*OFSM M2*›
  ‹*asc-fault-domain M2 M1 m*›
  ‹*test-tools M2 M1 FAIL PM V* Ω›
show $0 < iter + 1 \wedge$
  $tsN \cup cN = TS\ M2\ M1\ \Omega\ V\ m\ (iter + 1 - 1) \wedge$
  append-set
   $(cN -$
    $\{xs' \in cN.$
     $\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
     $(\forall io \in L_{in}\ M1\ \{xs'\}.$
         $\exists V'' \in N\ io\ M1\ V.$
            $\exists S1\ vs\ xs.$
               $io = vs\ @\ xs \wedge$
               $mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
               $S1 \subseteq nodes\ M2 \wedge$
               $(\forall s1 \in S1.$
                   $\forall s2 \in S1.$
                      $s1 \neq s2 \longrightarrow$
                      $(\forall io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
                          $\forall io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
               $m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
   $(inputs\ M2) -$
  $(tsN \cup cN) =$
  $C\ M2\ M1\ \Omega\ V\ m\ (iter + 1) \wedge$
  $\{xs' \in cN.$
   $\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
   $(\forall io \in L_{in}\ M1\ \{xs'\}.$
       $\exists V'' \in N\ io\ M1\ V.$
          $\exists S1\ vs\ xs.$
             $io = vs\ @\ xs \wedge$
             $mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
             $S1 \subseteq nodes\ M2 \wedge$
             $(\forall s1 \in S1.$
                 $\forall s2 \in S1.$
                    $s1 \neq s2 \longrightarrow$
                    $(\forall io1 \in RP\ M2\ s1\ vs\ xs\ V''.\ \forall io2 \in RP\ M2\ s2\ vs\ xs\ V''.$
                       $B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
             $m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\} =$
  $RM\ M2\ M1\ \Omega\ V\ m\ (iter + 1 - 1) \wedge$
  $obs \cup$
  $L_{in}\ M2$
   $(append\text{-}set$
     $(cN -$
      $\{xs' \in cN.$
       $\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
       $(\forall io \in L_{in}\ M1\ \{xs'\}.$
           $\exists V'' \in N\ io\ M1\ V.$
              $\exists S1\ vs\ xs.$
                 $io = vs\ @\ xs \wedge$
                 $mcp\ (vs\ @\ xs)\ V''\ vs \wedge$

$S1 \subseteq nodes\ M2\ \wedge$
$(\forall\, s1{\in}S1.$
$\forall\, s2{\in}S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1{\in}RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2{\in}RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega))\ \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN)) =$
$L_{in}\ M2$
$(tsN \cup cN \cup$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io{\in}L_{in}\ M1\ \{xs'\}.$
$\exists\, V''{\in}N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs\ \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs\ \wedge$
$S1 \subseteq nodes\ M2\ \wedge$
$(\forall\, s1{\in}S1.$
$\forall\, s2{\in}S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1{\in}RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2{\in}RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega))\ \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN)))\ \wedge$
$obsI\ \cup$
$L_{in}\ M1$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io{\in}L_{in}\ M1\ \{xs'\}.$
$\exists\, V''{\in}N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs\ \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs\ \wedge$
$S1 \subseteq nodes\ M2\ \wedge$
$(\forall\, s1{\in}S1.$
$\forall\, s2{\in}S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1{\in}RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2{\in}RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega))\ \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN)) =$
$L_{in}\ M1$
$(tsN \cup cN \cup$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io{\in}L_{in}\ M1\ \{xs'\}.$
$\exists\, V''{\in}N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs\ \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs\ \wedge$
$S1 \subseteq nodes\ M2\ \wedge$
$(\forall\, s1{\in}S1.$
$\forall\, s2{\in}S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1{\in}RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2{\in}RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega))\ \wedge$

$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN))) \wedge$
$obs_\Omega \cup$
$(\bigcup io \in L_{in}\ M2$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
$S1 \subseteq nodes\ M2 \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN)).$
$\{io\} \times B\ M2\ io\ \Omega) =$
$(\bigcup io \in L_{in}\ M2$
$(tsN \cup cN \cup$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
$S1 \subseteq nodes\ M2 \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN))).$
$\{io\} \times B\ M2\ io\ \Omega) \wedge$
$obsI_\Omega \cup$
$(\bigcup io \in L_{in}\ M1$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs \wedge$
$S1 \subseteq nodes\ M2 \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN)).$
$\{io\} \times B\ M1\ io\ \Omega) =$

$(\bigcup io \in L_{in}\ M1$
$(tsN \cup cN \cup$
$(append\text{-}set$
$(cN -$
$\{xs' \in cN.$
$\neg\ L_{in}\ M1\ \{xs'\} \subseteq L_{in}\ M2\ \{xs'\} \vee$
$(\forall\, io \in L_{in}\ M1\ \{xs'\}.$
$\exists\, V'' \in N\ io\ M1\ V.$
$\exists\, S1\ vs\ xs.$
$io = vs\ @\ xs\ \wedge$
$mcp\ (vs\ @\ xs)\ V''\ vs\ \wedge$
$S1 \subseteq nodes\ M2\ \wedge$
$(\forall\, s1 \in S1.$
$\forall\, s2 \in S1.$
$s1 \neq s2 \longrightarrow$
$(\forall\, io1 \in RP\ M2\ s1\ vs\ xs\ V''.$
$\forall\, io2 \in RP\ M2\ s2\ vs\ xs\ V''.\ B\ M1\ io1\ \Omega \neq B\ M1\ io2\ \Omega)) \wedge$
$m < LB\ M2\ M1\ vs\ xs\ (tsN \cup V)\ S1\ \Omega\ V'')\})$
$(inputs\ M2) -$
$(tsN \cup cN))).$
$\{io\} \times B\ M1\ io\ \Omega) \wedge$
$OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
by *linarith*
next
fix $tsN\ cN\ rmN\ obs\ obsI\ obs_{\Omega}\ obsI_{\Omega}\ iter\ isReduction$
assume $precond : (0 < iter \wedge$
$tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter - 1) \wedge$
$cN = C\ M2\ M1\ \Omega\ V\ m\ iter \wedge$
$rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter - 1) \wedge$
$obs = L_{in}\ M2\ (tsN \cup cN) \wedge$
$obsI = L_{in}\ M1\ (tsN \cup cN) \wedge$
$obs_{\Omega} = (\bigcup io \in L_{in}\ M2\ (tsN \cup cN).\ \{io\} \times B\ M2\ io\ \Omega) \wedge$
$obsI_{\Omega} = (\bigcup io \in L_{in}\ M1\ (tsN \cup cN).\ \{io\} \times B\ M1\ io\ \Omega) \wedge$
$OFSM\ M1 \wedge OFSM\ M2 \wedge asc\text{-}fault\text{-}domain\ M2\ M1\ m \wedge test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega) \wedge$
$\neg\ (cN \neq \{\} \wedge obsI \subseteq obs \wedge obsI_{\Omega} \subseteq obs_{\Omega})$
then have $0 < iter$
$OFSM\ M1$
$OFSM\ M2$
$asc\text{-}fault\text{-}domain\ M2\ M1\ m$
$test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$
$cN = \{\} \vee \neg\ obsI \subseteq obs \vee \neg\ obsI_{\Omega} \subseteq obs_{\Omega}$
$tsN = TS\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
$cN = C\ M2\ M1\ \Omega\ V\ m\ iter$
$rmN = RM\ M2\ M1\ \Omega\ V\ m\ (iter-1)$
$obs = L_{in}\ M2\ (tsN \cup cN)$
$obsI = L_{in}\ M1\ (tsN \cup cN)$
$obs_{\Omega} = (\bigcup io \in L_{in}\ M2\ (tsN \cup cN).\ \{io\} \times B\ M2\ io\ \Omega)$
$obsI_{\Omega} = (\bigcup io \in L_{in}\ M1\ (tsN \cup cN).\ \{io\} \times B\ M1\ io\ \Omega)$
by *linarith+*

show $(obsI \subseteq obs \wedge obsI_{\Omega} \subseteq obs_{\Omega}) = M1 \preceq M2$
proof $(cases\ cN = \{\})$
case *True*
then have $C\ M2\ M1\ \Omega\ V\ m\ iter = \{\}$
using ‹$cN = C\ M2\ M1\ \Omega\ V\ m\ iter$› **by** *auto*

have $is\text{-}det\text{-}state\text{-}cover\ M2\ V$
using ‹$test\text{-}tools\ M2\ M1\ FAIL\ PM\ V\ \Omega$› **by** *auto*
then have $[] \in V$
using $det\text{-}state\text{-}cover\text{-}initial[of\ M2\ V]$ **by** *simp*
then have $V \neq \{\}$
by *blast*
have $Suc\ 0 < iter$
proof $(rule\ ccontr)$

  assume ¬ *Suc 0 < iter*
  then have *iter = Suc 0*
    using ‹*0 < iter*› **by** *auto*
  then have *C M2 M1 Ω V m (Suc 0) = {}*
    using ‹*C M2 M1 Ω V m iter = {}*› **by** *auto*
  moreover have *C M2 M1 Ω V m (Suc 0) = V*
    by *auto*
  ultimately show*False*
    using ‹*V ≠ {}*› **by** *blast*
qed

obtain *k* **where** *iter = Suc k*
  using *gr0-implies-Suc*[*OF* ‹*0 < iter*›] **by** *blast*
then have *Suc 0 < Suc k*
  using ‹*Suc 0 < iter*› **by** *auto*
then have *0 < k*
  by *simp*
then obtain *k′* **where** *k = Suc k′*
  using *gr0-implies-Suc* **by** *blast*
have *iter = Suc (Suc k′)*
  using ‹*iter = Suc k*› ‹*k = Suc k′*› **by** *simp*

have *TS M2 M1 Ω V m (Suc (Suc k′)) = TS M2 M1 Ω V m (Suc k′) ∪ C M2 M1 Ω V m (Suc (Suc k′))*
  using *TS.simps(3)*[*of M2 M1 Ω V m k′*] **by** *blast*
then have *TS M2 M1 Ω V m iter = TS M2 M1 Ω V m (Suc k′)*
  using *True* ‹*cN = C M2 M1 Ω V m iter*› ‹*iter = Suc (Suc k′)*› **by** *blast*
moreover have *Suc k′ = iter − 1*
  using ‹*iter = Suc (Suc k′)*› **by** *presburger*
ultimately have *TS M2 M1 Ω V m iter = TS M2 M1 Ω V m (iter − 1)*
  by *auto*
then have *tsN = TS M2 M1 Ω V m iter*
  using ‹*tsN = TS M2 M1 Ω V m (iter−1)*› **by** *simp*

then have *TS M2 M1 Ω V m iter = TS M2 M1 Ω V m (iter − 1)*
  using ‹*tsN = TS M2 M1 Ω V m (iter − 1)*› **by** *auto*
then have *final-iteration M2 M1 Ω V m (iter−1)*
  using ‹*0 < iter*› **by** *auto*

have *M1 ⪯ M2 = atc-io-reduction-on-sets M1 tsN Ω M2*
  using *asc-main-theorem*[*OF* ‹*OFSM M1*› ‹*OFSM M2*›
                          ‹*asc-fault-domain M2 M1 m*›
                          ‹*test-tools M2 M1 FAIL PM V Ω*›
                          ‹*final-iteration M2 M1 Ω V m (iter−1)*›]
  using ‹*tsN = TS M2 M1 Ω V m (iter − 1)*›
  by *blast*
moreover have *tsN ∪ cN = tsN*
  using ‹*cN = {}*› **by** *blast*
ultimately have *M1 ⪯ M2 = atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2*
  by *presburger*

have *obsI ⊆ obs ≡* L_{in} *M1 (tsN ∪ cN) ⊆* L_{in} *M2 (tsN ∪ cN)*
  by (*simp add*: ‹*obs =* L_{in} *M2 (tsN ∪ cN)*› ‹*obsI =* L_{in} *M1 (tsN ∪ cN)*›)

have $obsI_\Omega \subseteq obs_\Omega$ ≡ (⋃*io*∈L_{in} *M1 (tsN ∪ cN). {io} × B M1 io Ω)*
                    ⊆ (⋃*io*∈L_{in} *M2 (tsN ∪ cN). {io} × B M2 io Ω)*
  by (*simp add*: ‹$obsI_\Omega$ = (⋃*io*∈L_{in} *M1 (tsN ∪ cN). {io} × B M1 io Ω)*›
              ‹obs_Ω = (⋃*io*∈L_{in} *M2 (tsN ∪ cN). {io} × B M2 io Ω)*›)

have (*obsI ⊆ obs ∧* $obsI_\Omega \subseteq obs_\Omega$) *= atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2*
proof
  assume *obsI ⊆ obs ∧* $obsI_\Omega \subseteq obs_\Omega$
  show *atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2*
    using *atc-io-reduction-on-sets-from-obs*[*of M1 tsN ∪ cN M2 Ω*]
    using ‹*obsI ⊆ obs ∧* $obsI_\Omega \subseteq obs_\Omega$› ‹*obsI ⊆ obs ≡* L_{in} *M1 (tsN ∪ cN) ⊆* L_{in} *M2 (tsN ∪ cN)*›
        ‹$obsI_\Omega \subseteq obs_\Omega$ ≡ (⋃*io*∈L_{in} *M1 (tsN ∪ cN). {io} × B M1 io Ω)*

```
                          ⊆ (⋃io∈L_in M2 (tsN ∪ cN). {io} × B M2 io Ω)›
      by linarith
    next
      assume atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2
      show obsI ⊆ obs ∧ obsI_Ω ⊆ obs_Ω
        using atc-io-reduction-on-sets-to-obs[of M1 ‹tsN ∪ cN› Ω M2]
              ‹atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2›
              ‹obsI ⊆ obs ≡ L_in M1 (tsN ∪ cN) ⊆ L_in M2 (tsN ∪ cN)›
              ‹obsI_Ω ⊆ obs_Ω ≡ (⋃io∈L_in M1 (tsN ∪ cN). {io} × B M1 io Ω)
                          ⊆ (⋃io∈L_in M2 (tsN ∪ cN). {io} × B M2 io Ω)›
        by blast
    qed
    then show ?thesis
      using ‹M1 ≼ M2 = atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2› by linarith
next
    case False


    then have ¬ obsI ⊆ obs ∨ ¬ obsI_Ω ⊆ obs_Ω
      using ‹cN = {} ∨ ¬ obsI ⊆ obs ∨ ¬ obsI_Ω ⊆ obs_Ω› by auto

    have ¬ atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2
      using atc-io-reduction-on-sets-to-obs[of M1 tsN ∪ cN Ω M2]
            ‹¬ obsI ⊆ obs ∨ ¬ obsI_Ω ⊆ obs_Ω› precond
      by fastforce

    have ¬ M1 ≼ M2
    proof
      assume M1 ≼ M2
      have atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2
        using asc-soundness[OF ‹OFSM M1› ‹OFSM M2›] ‹M1 ≼ M2› by blast
      then show False
        using ‹¬ atc-io-reduction-on-sets M1 (tsN ∪ cN) Ω M2› by blast
    qed

    then show ?thesis
      using ‹¬ obsI ⊆ obs ∨ ¬ obsI_Ω ⊆ obs_Ω› by blast

  qed
qed


end
theory ASC-Example
  imports ASC-Hoare
begin
```

8 Example product machines and properties

This section provides example FSMs and shows that the assumptions on the inputs of the adaptive state counting algorithm are not vacuous.

8.1 Constructing FSMs from transition relations

This subsection provides a function to more easily create FSMs, only requiring a set of transition-tuples and an initial state.

```
fun from-rel :: ('state × ('in × 'out) × 'state) set ⇒ 'state ⇒ ('in, 'out, 'state) FSM where
from-rel rel q0 = ⦇ succ = λ io p . { q . (p,io,q) ∈ rel },
                    inputs = image (fst ∘ fst ∘ snd) rel,
                    outputs = image (snd ∘ fst ∘ snd) rel,
                    initial = q0 ⦈
```

lemma *nodes-from-rel* : *nodes* (*from-rel rel q0*) ⊆ *insert q0* (*image* (*snd* ∘ *snd*) *rel*)
  (**is** *nodes ?M* ⊆ *insert q0* (*image* (*snd* ∘ *snd*) *rel*))
proof −
  have ⋀ *q io p* . *q* ∈ *succ ?M io p* ⟹ *q* ∈ *image* (*snd* ∘ *snd*) *rel*
    by *force*
  have ⋀ *q* . *q* ∈ *nodes ?M* ⟹ *q* = *q0* ∨ *q* ∈ *image* (*snd* ∘ *snd*) *rel*
  proof −
    fix *q* **assume** *q* ∈ *nodes ?M*
    then show *q* = *q0* ∨ *q* ∈ *image* (*snd* ∘ *snd*) *rel*
    proof (*cases rule*: *FSM.nodes.cases*)
      case *initial*
      then show *?thesis* **by** *auto*
    next
      case (*execute p a*)
      then show *?thesis*
        using ‹⋀ *q io p* . *q* ∈ *succ ?M io p* ⟹ *q* ∈ *image* (*snd* ∘ *snd*) *rel*› **by** *blast*
    qed
  qed
  then show *nodes ?M* ⊆ *insert q0* (*image* (*snd* ∘ *snd*) *rel*)
    by *blast*
qed

fun *well-formed-rel* :: (′*state* × (′*in* × ′*out*) × ′*state*) *set* ⇒ *bool* **where**
  well-formed-rel rel = (*finite rel*
      ∧ (∀ *s1 x y* . (*x* ∉ *image* (*fst* ∘ *fst* ∘ *snd*) *rel*
          ∨ *y* ∉ *image* (*snd* ∘ *fst* ∘ *snd*) *rel*)
        ⟶ ¬(∃ *s2* . (*s1*,(*x*,*y*),*s2*) ∈ *rel*))
      ∧ *rel* ≠ {})

lemma *well-formed-from-rel* :
  assumes *well-formed-rel rel*
  shows *well-formed* (*from-rel rel q0*) (**is** *well-formed ?M*)
proof −
  have *nodes ?M* ⊆ *insert q0* (*image* (*snd* ∘ *snd*) *rel*)
    using *nodes-from-rel*[*of rel q0*] **by** *auto*
  moreover have *finite* (*insert q0* (*image* (*snd* ∘ *snd*) *rel*))
    using *assms* **by** *auto*
  ultimately have *finite* (*nodes ?M*)
    by (*simp add*: *Finite-Set.finite-subset*)
  moreover have *finite* (*inputs ?M*) *finite* (*outputs ?M*)
    using *assms* **by** *auto*
  ultimately have *finite-FSM ?M*
    by *auto*

  moreover have *inputs ?M* ≠ {}
    using *assms* **by** *auto*
  moreover have *outputs ?M* ≠ {}
    using *assms* **by** *auto*
  moreover have ⋀ *s1 x y* . (*x* ∉ *inputs ?M* ∨ *y* ∉ *outputs ?M*) ⟶ *succ ?M* (*x*,*y*) *s1* = {}
    using *assms* **by** *auto*

  ultimately show *?thesis*
    by *auto*
qed

fun *completely-specified-rel-over* :: (′*state* × (′*in* × ′*out*) × ′*state*) *set* ⇒ ′*state set* ⇒ *bool*
  where
  completely-specified-rel-over rel nods = (∀ *s1* ∈ *nods* .
      ∀ *x* ∈ *image* (*fst* ∘ *fst* ∘ *snd*) *rel* .
        ∃ *y* ∈ *image* (*snd* ∘ *fst* ∘ *snd*) *rel* .
          ∃ *s2* . (*s1*,(*x*,*y*),*s2*) ∈ *rel*)

lemma *completely-specified-from-rel* :
assumes *completely-specified-rel-over rel* (*nodes* ((*from-rel rel q0*)))
shows *completely-specified* (*from-rel rel q0*) (**is** *completely-specified ?M*)
unfolding *completely-specified.simps*
proof
fix *s1* **assume** *s1* ∈ *nodes* (*from-rel rel q0*)
show ∀ *x*∈*inputs ?M*. ∃ *y*∈*outputs ?M*. ∃ *s2*. *s2* ∈ *succ ?M* (*x*, *y*) *s1*
proof
fix *x* **assume** *x* ∈ *inputs* (*from-rel rel q0*)
then have *x* ∈ *image* (*fst* ∘ *fst* ∘ *snd*) *rel*
using *assms* **by** *auto*

obtain *y s2* **where** *y* ∈ *image* (*snd* ∘ *fst* ∘ *snd*) *rel* (*s1*,(*x*,*y*),*s2*) ∈ *rel*
using *assms* ‹*s1* ∈ *nodes* (*from-rel rel q0*)› ‹*x* ∈ *image* (*fst* ∘ *fst* ∘ *snd*) *rel*›
by (*meson completely-specified-rel-over.elims*(*2*))

then have *y* ∈ *outputs* (*from-rel rel q0*) *s2* ∈ *succ* (*from-rel rel q0*) (*x*, *y*) *s1*
by *auto*

then show ∃ *y*∈*outputs* (*from-rel rel q0*). ∃ *s2*. *s2* ∈ *succ* (*from-rel rel q0*) (*x*, *y*) *s1*
by *blast*
qed
qed

fun *observable-rel* :: ('*state* × ('*in* × '*out*) × '*state*) *set* ⇒ *bool* **where**
observable-rel rel = (∀ *io s1* . { *s2* . (*s1*,*io*,*s2*) ∈ *rel* } = {}
∨ (∃ *s2* . { *s2'* . (*s1*,*io*,*s2'*) ∈ *rel* } = {*s2*}))

lemma *observable-from-rel* :
assumes *observable-rel rel*
shows *observable* (*from-rel rel q0*) (**is** *observable ?M*)
proof −
have ⋀ *io s1* . { *s2* . (*s1*,*io*,*s2*) ∈ *rel* } = *succ ?M io s1*
by *auto*
then show *?thesis* **using** *assms* **by** *auto*
qed

abbreviation *OFSM-rel rel q0* ≡ *well-formed-rel rel*
∧ *completely-specified-rel-over rel* (*nodes* (*from-rel rel q0*))
∧ *observable-rel rel*

lemma *OFMS-from-rel* :
assumes *OFSM-rel rel q0*
shows *OFSM* (*from-rel rel q0*)
by (*metis assms completely-specified-from-rel observable-from-rel well-formed-from-rel*)

8.2 Example FSMs and properties

abbreviation M_S*-rel* :: (*nat*×(*nat*×*nat*)×*nat*) *set* ≡ {(*0*,(*0*,*0*),*1*), (*0*,(*0*,*1*),*1*), (*1*,(*0*,*2*),*1*)}
abbreviation M_S :: (*nat*,*nat*,*nat*) *FSM* ≡ *from-rel* M_S*-rel 0*

abbreviation M_I*-rel* :: (*nat*×(*nat*×*nat*)×*nat*) *set* ≡ {(*0*,(*0*,*0*),*1*), (*0*,(*0*,*1*),*1*), (*1*,(*0*,*2*),*0*)}
abbreviation M_I :: (*nat*,*nat*,*nat*) *FSM* ≡ *from-rel* M_I*-rel 0*

lemma *example-nodes* :
nodes M_S = {*0*,*1*} *nodes* M_I = {*0*,*1*}
proof −

have *0* ∈ *nodes* M_S **by** *auto*
have *1* ∈ *succ* M_S (*0*,*0*) *0* **by** *auto*
have *1* ∈ *nodes* M_S
by (*meson* ‹*0* ∈ *nodes* M_S› ‹*1* ∈ *succ* M_S (*0*, *0*) *0*› *succ-nodes*)

have {*0*,*1*} ⊆ *nodes* M_S
using ‹*0* ∈ *nodes* M_S› ‹*1* ∈ *nodes* M_S› **by** *auto*
moreover have *nodes* M_S ⊆ {*0*,*1*}
using *nodes-from-rel*[*of* M_S*-rel 0*] **by** *auto*
ultimately show *nodes* M_S = {*0*,*1*}
by *blast*
next
have *0* ∈ *nodes* M_I **by** *auto*
have *1* ∈ *succ* M_I (*0*,*0*) *0* **by** *auto*
have *1* ∈ *nodes* M_I
by (*meson* ‹*0* ∈ *nodes* M_I› ‹*1* ∈ *succ* M_I (*0*, *0*) *0*› *succ-nodes*)

have {*0*,*1*} ⊆ *nodes* M_I
using ‹*0* ∈ *nodes* M_I› ‹*1* ∈ *nodes* M_I› **by** *auto*
moreover have *nodes* M_I ⊆ {*0*,*1*}
using *nodes-from-rel*[*of* M_I*-rel 0*] **by** *auto*
ultimately show *nodes* M_I = {*0*,*1*}
by *blast*
qed

lemma *example-OFSM* :
OFSM M_S *OFSM* M_I
proof −
have *well-formed-rel* M_S*-rel*
unfolding *well-formed-rel.simps* **by** *auto*

moreover have *completely-specified-rel-over* M_S*-rel* (*nodes* (*from-rel* M_S*-rel 0*))
unfolding *completely-specified-rel-over.simps*
proof
fix *s1* **assume** (*s1*::*nat*) ∈ *nodes* (*from-rel* M_S*-rel 0*)
then have *s1* ∈ (*insert 0* (*image* (*snd* ∘ *snd*) M_S*-rel*))
using *nodes-from-rel*[*of* M_S*-rel 0*] **by** *blast*
moreover have *completely-specified-rel-over* M_S*-rel* (*insert 0* (*image* (*snd* ∘ *snd*) M_S*-rel*))
unfolding *completely-specified-rel-over.simps* **by** *auto*
ultimately show ∀ *x*∈(*fst* ∘ *fst* ∘ *snd*) ‘ M_S*-rel*.
∃ *y*∈(*snd* ∘ *fst* ∘ *snd*) ‘ M_S*-rel*. ∃ *s2*. (*s1*, (*x*, *y*), *s2*) ∈ M_S*-rel*
by *simp*
qed

moreover have *observable-rel* M_S*-rel*
by *auto*

ultimately have *OFSM-rel* M_S*-rel 0*
by *auto*

then show *OFSM* M_S
using *OFMS-from-rel*[*of* M_S*-rel 0*] **by** *linarith*
next
have *well-formed-rel* M_I*-rel*
unfolding *well-formed-rel.simps* **by** *auto*

moreover have *completely-specified-rel-over* M_I*-rel* (*nodes* (*from-rel* M_I*-rel 0*))
unfolding *completely-specified-rel-over.simps*
proof
fix *s1* **assume** (*s1*::*nat*) ∈ *nodes* (*from-rel* M_I*-rel 0*)
then have *s1* ∈ (*insert 0* (*image* (*snd* ∘ *snd*) M_I*-rel*))
using *nodes-from-rel*[*of* M_I*-rel 0*] **by** *blast*
have *completely-specified-rel-over* M_I*-rel* (*insert 0* (*image* (*snd* ∘ *snd*) M_I*-rel*))
unfolding *completely-specified-rel-over.simps* **by** *auto*

show $\forall x \in (fst \circ fst \circ snd)$ ‘ M_I-*rel*.
$\exists y \in (snd \circ fst \circ snd)$ ‘ M_I-*rel*. $\exists s2. (s1, (x, y), s2) \in M_I$-*rel*
by (*meson* ‹*completely-specified-rel-over* M_I-*rel* (*insert 0* ((*snd* ∘ *snd*) ‘ M_I-*rel*))›
‹*s1* ∈ *insert 0* ((*snd* ∘ *snd*) ‘ M_I-*rel*)› *completely-specified-rel-over.elims*(*2*))
qed

moreover have *observable-rel* M_I-*rel*
by *auto*

ultimately have *OFSM-rel* M_I-*rel 0*
by *auto*

then show *OFSM* M_I
using *OFMS-from-rel*[*of* M_I-*rel 0*] **by** *linarith*
qed

lemma *example-fault-domain* : *asc-fault-domain* M_S M_I *2*
proof −
have *inputs* M_S = *inputs* M_I
by *auto*
moreover have *card* (*nodes* M_I) ≤ *2*
using *example-nodes*(*2*) **by** *auto*
ultimately show *asc-fault-domain* M_S M_I *2*
by *auto*
qed

abbreviation $FAIL_I$ :: (*nat*×*nat*) ≡ (*3*,*3*)
abbreviation PM_I :: (*nat*, *nat*, *nat*×*nat*) *FSM* ≡ ⦇
succ = (λ *a* (*p1*,*p2*) . (*if* (*p1* ∈ *nodes* M_S ∧ *p2* ∈ *nodes* M_I ∧ (*fst a* ∈ *inputs* M_S)
∧ (*snd a* ∈ *outputs* M_S ∪ *outputs* M_I))
then (*if* (*succ* M_S *a p1* = {} ∧ *succ* M_I *a p2* ≠ {})
then {$FAIL_I$}
else (*succ* M_S *a p1* × *succ* M_I *a p2*))
else {})),
inputs = *inputs* M_S,
outputs = *outputs* M_S ∪ *outputs* M_I,
initial = (*initial* M_S, *initial* M_I)
⦈

lemma *example-productF* : *productF* M_S M_I $FAIL_I$ PM_I
proof −
have *inputs* M_S = *inputs* M_I
by *auto*
moreover have *fst* $FAIL_I$ ∉ *nodes* M_S
using *example-nodes*(*1*) **by** *auto*
moreover have *snd* $FAIL_I$ ∉ *nodes* M_I
using *example-nodes*(*2*) **by** *auto*
ultimately show *?thesis*
unfolding *productF.simps* **by** *blast*
qed

abbreviation V_I :: *nat list set* ≡ {[],[*0*]}

lemma *example-det-state-cover* : *is-det-state-cover* M_S V_I
proof −
have *d-reaches* M_S (*initial* M_S) [] (*initial* M_S)
by *auto*
then have *initial* M_S ∈ *d-reachable* M_S (*initial* M_S)
unfolding *d-reachable.simps* **by** *blast*

have *d-reached-by* M_S (*initial* M_S) [*0*] *1* [*1*] [*0*]
proof

show *length* [*0*] = *length* [*0*] ∧
length [*0*] = *length* [*1*] ∧ *path* M_S (([*0*] || [*0*]) || [*1*]) (*initial* M_S)
∧ *target* (([*0*] || [*0*]) || [*1*]) (*initial* M_S) = *1*
by *auto*

have ⋀*ys2 tr2*.
length [*0*] = *length ys2*
∧ *length* [*0*] = *length tr2*
∧ *path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)
⟶ *target* (([*0*] || *ys2*) || *tr2*) (*initial* M_S) = *1*
proof
fix *ys2 tr2* **assume** *length* [*0*] = *length ys2* ∧ *length* [*0*] = *length tr2*
∧ *path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)
then have *length ys2* = *1 length tr2* = *1 path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)
by *auto*
moreover obtain *y2* **where** *ys2* = [*y2*]
using ‹*length ys2* = *1*›
by (*metis One-nat-def* ‹*length* [*0*] = *length ys2* ∧ *length* [*0*] = *length tr2*
∧ *path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)› *append.simps*(*1*) *append-butlast-last-id*
butlast-snoc length-butlast length-greater-0-conv list.size(*3*) *nat.simps*(*3*))
moreover obtain *t2* **where** *tr2* = [*t2*]
using ‹*length tr2* = *1*›
by (*metis One-nat-def* ‹*length* [*0*] = *length ys2* ∧ *length* [*0*] = *length tr2*
∧ *path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)› *append.simps*(*1*) *append-butlast-last-id*
butlast-snoc length-butlast length-greater-0-conv list.size(*3*) *nat.simps*(*3*))
ultimately have *path* M_S [((*0*,*y2*),*t2*)] (*initial* M_S)
by *auto*
then have *t2* ∈ *succ* M_S (*0*,*y2*) (*initial* M_S)
by *auto*
moreover have ⋀ *y* . *succ* M_S (*0*,*y*) (*initial* M_S) ⊆ {*1*}
by *auto*
ultimately have *t2* = *1*
by *blast*

show *target* (([*0*] || *ys2*) || *tr2*) (*initial* M_S) = *1*
using ‹*ys2* = [*y2*]› ‹*tr2* = [*t2*]› ‹*t2* = *1*› **by** *auto*
qed
then show ∀ *ys2 tr2*.
length [*0*] = *length ys2* ∧ *length* [*0*] = *length tr2*
∧ *path* M_S (([*0*] || *ys2*) || *tr2*) (*initial* M_S)
⟶ *target* (([*0*] || *ys2*) || *tr2*) (*initial* M_S) = *1*
by *auto*
qed

then have *d-reaches* M_S (*initial* M_S) [*0*] *1*
unfolding *d-reaches.simps* **by** *blast*
then have *1* ∈ *d-reachable* M_S (*initial* M_S)
unfolding *d-reachable.simps* **by** *blast*

then have {*0*,*1*} ⊆ *d-reachable* M_S (*initial* M_S)
using ‹*initial* M_S ∈ *d-reachable* M_S (*initial* M_S)› **by** *auto*
moreover have *d-reachable* M_S (*initial* M_S) ⊆ *nodes* M_S
proof
fix *s* **assume** *s*∈*d-reachable* M_S (*initial* M_S)
then have *s* ∈ *reachable* M_S (*initial* M_S)
using *d-reachable-reachable* **by** *auto*
then show *s* ∈ *nodes* M_S
by *blast*
qed
ultimately have *d-reachable* M_S (*initial* M_S) = {*0*,*1*}
using *example-nodes*(*1*) **by** *blast*

fix *f'* :: *nat* ⇒ *nat list*
let *?f* = *f'*(*0* := [], *1* := [*0*])

have *is-det-state-cover-ass* M_S *?f*
unfolding *is-det-state-cover-ass.simps*
proof
show *?f* (*initial* M_S) = [] **by** *auto*
show ∀ *s*∈*d-reachable* M_S (*initial* M_S). *d-reaches* M_S (*initial* M_S) (*?f s*) *s*
proof
fix *s* **assume** *s*∈*d-reachable* M_S (*initial* M_S)
then have *s* ∈ *reachable* M_S (*initial* M_S)
using *d-reachable-reachable* **by** *auto*
then have *s* ∈ *nodes* M_S
by *blast*
then have *s* = *0* ∨ *s* = *1*
using *example-nodes*(*1*) **by** *blast*
then show *d-reaches* M_S (*initial* M_S) (*?f s*) *s*
proof
assume *s* = *0*
then show *d-reaches* M_S (*initial* M_S) (*?f s*) *s*
using ‹*d-reaches* M_S (*initial* M_S) [] (*initial* M_S)› **by** *auto*
next
assume *s* = *1*
then show *d-reaches* M_S (*initial* M_S) (*?f s*) *s*
using ‹*d-reaches* M_S (*initial* M_S) [*0*] *1*› **by** *auto*
qed
qed
qed

moreover have V_I = *image ?f* (*d-reachable* M_S (*initial* M_S))
using ‹*d-reachable* M_S (*initial* M_S) = {*0*,*1*}› **by** *auto*

ultimately show *?thesis*
unfolding *is-det-state-cover.simps* **by** *blast*
qed

abbreviation Ω_I::(*nat*,*nat*) *ATC set* ≡ { *Node 0* (λ *y* . *Leaf*) }

lemma *applicable-set* M_S Ω_I
by *auto*

lemma *example-test-tools* : *test-tools* M_S M_I $FAIL_I$ PM_I V_I Ω_I
using *example-productF example-det-state-cover* **by** *auto*

lemma *OFSM-not-vacuous* :
∃ *M* :: (*nat*,*nat*,*nat*) *FSM* . *OFSM M*
using *example-OFSM*(*1*) **by** *blast*

lemma *fault-domain-not-vacuous* :
∃ (*M2*::(*nat*,*nat*,*nat*) *FSM*) (*M1*::(*nat*,*nat*,*nat*) *FSM*) *m* . *asc-fault-domain M2 M1 m*
using *example-fault-domain* **by** *blast*

lemma *test-tools-not-vacuous* :
∃ (*M2*::(*nat*,*nat*,*nat*) *FSM*)
(*M1*::(*nat*,*nat*,*nat*) *FSM*)
(*FAIL*::(*nat*×*nat*))
(*PM*::(*nat*,*nat*,*nat*×*nat*) *FSM*)
(*V*::(*nat list set*))
(Ω::(*nat*,*nat*) *ATC set*) . *test-tools M2 M1 FAIL PM V* Ω
proof (*rule exI*, *rule exI*)

show ∃ *FAIL PM V* Ω. *test-tools* M_S M_I *FAIL PM V* Ω
using *example-test-tools* **by** *blast*
qed

lemma *precondition-not-vacuous* :
shows ∃ (*M2*::(*nat*,*nat*,*nat*) *FSM*)
(*M1*::(*nat*,*nat*,*nat*) *FSM*)
(*FAIL*::(*nat*×*nat*))
(*PM*::(*nat*,*nat*,*nat*×*nat*) *FSM*)
(*V*::(*nat list set*))
(Ω::(*nat*,*nat*) *ATC set*)
(*m* :: *nat*) .
OFSM M1 ∧ *OFSM M2* ∧ *asc-fault-domain M2 M1 m* ∧ *test-tools M2 M1 FAIL PM V* Ω
proof (*intro exI*)
show *OFSM* M_I ∧ *OFSM* M_S ∧ *asc-fault-domain* M_S M_I *2* ∧ *test-tools* M_S M_I $FAIL_I$ PM_I V_I Ω_I
using *example-OFSM*(*2*,*1*) *example-fault-domain example-test-tools* **by** *linarith*
qed

end

References

[1] J. Brunner. Transition systems and automata. *Archive of Formal Proofs*, Oct. 2017. http://isa-afp.org/entries/Transition_Systems_and_Automata.html, Formal proof development.

[2] R. M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, 53(10):1330–1342, 2004.

[3] R. Sachtleben, J. Peleska, R. Hierons, and W.-L. Huang. A mechanised proof of an adaptive state counting algorithm. In *IFIP International Conference on Testing Software and Systems*. Springer, 2019. to appear.