

# MUNTA: A Verified Model Checker for Timed Automata

Simon Wimmer

## Abstract

MUNTA is a verified model checker for timed automata. It has been described in detail in a PhD thesis [3] and conference publications [4, 2]. This entry supersedes earlier versions of MUNTA hosted on GitHub.

## Contents

|           |                                                                                    |            |
|-----------|------------------------------------------------------------------------------------|------------|
| <b>1</b>  | <b>Networks of Timed Automata</b>                                                  | <b>4</b>   |
| 1.1       | Syntax and Operational Semantics . . . . .                                         | 4          |
| 1.2       | Product Automaton . . . . .                                                        | 5          |
| <b>2</b>  | <b>Networks of Timed Automata With Discrete State</b>                              | <b>17</b>  |
| 2.1       | Syntax and Operational Semantics . . . . .                                         | 17         |
| 2.2       | Product Automaton . . . . .                                                        | 18         |
| <b>3</b>  | <b>Networks of Timed Automata – UPPAAL Style</b>                                   | <b>35</b>  |
| 3.1       | Syntax and Operational Semantics . . . . .                                         | 36         |
| 3.2       | Equivalent State Network Automaton . . . . .                                       | 37         |
| <b>4</b>  | <b>Implementation of UPPAAL Style Networks</b>                                     | <b>62</b>  |
| 4.1       | Pre-compiled Networks With States and Clocks as Natural Numbers . . . . .          | 66         |
| <b>5</b>  | <b>Imperative Implementation of UPPAAL Style Networks</b>                          | <b>97</b>  |
| 5.1       | Executable Successor Computation . . . . .                                         | 97         |
| 5.2       | Check Preconditions . . . . .                                                      | 147        |
| <b>6</b>  | <b>Simple Networks of Automata With Broadcast Channels and Committed Locations</b> | <b>177</b> |
| 6.1       | Product Construction . . . . .                                                     | 180        |
| <b>7</b>  | <b>Product Bisimulation</b>                                                        | <b>274</b> |
| <b>8</b>  | <b>The Final Semantics</b>                                                         | <b>276</b> |
| <b>9</b>  | <b>Instantiating the Model Checking Locale</b>                                     | <b>280</b> |
| 9.1       | Extracting an Efficient Implementation . . . . .                                   | 297        |
| <b>10</b> | <b>Deadlock Checking</b>                                                           | <b>302</b> |
| 10.1      | Notes . . . . .                                                                    | 303        |
| 10.2      | Abstract Reachability Checking . . . . .                                           | 303        |
| 10.3      | Operations . . . . .                                                               | 306        |
| 10.4      | Structural Properties . . . . .                                                    | 327        |
| 10.5      | Correctness of <i>dbm_subset_fed</i> . . . . .                                     | 334        |
| 10.6      | Refined DBM Operations . . . . .                                                   | 338        |
| 10.7      | Transferring Properties . . . . .                                                  | 339        |

|                                                            |            |
|------------------------------------------------------------|------------|
| 10.8 Functional Refinement . . . . .                       | 347        |
| 10.9 Imperative Refinement . . . . .                       | 352        |
| 10.10 Implementation for a Concrete Automaton . . . . .    | 356        |
| 10.11 Checking a Property on the Reachable Set . . . . .   | 360        |
| 10.12 Complete Deadlock Checker . . . . .                  | 363        |
| <b>11 Renaming Identifiers in Simple Networks</b>          | <b>376</b> |
| <b>12 Assembling the Model Checker and Generating Code</b> | <b>446</b> |
| <b>13 Build and Test Exported Program With MLton</b>       | <b>476</b> |

## Introduction

MUNTA is a verified model checker for timed automata. It has been described in detail in a PhD thesis [3] and presented in peer-reviewed conference publications [4, 2].

This AFP entry builds upon and extends formalizations from the following previous entries:

- `Timed_Automata` [1]
- `Worklist_Algorithms` [6]
- `Difference_Bound_Matrices` [5]

and supersedes an earlier version of MUNTA hosted on GitHub: <https://github.com/wimmers/munta>.

**Usage** Theory `Simple_Network_Language_Export_Code` (page 474) describes how to obtain executable code for and run MUNTA from within Isabelle.

This entry also provides a build setup for the MUNTA executable using the MLton SML compiler. Build and run instructions are given in theory `Munta_Compile_MLton` (on page 476). A number of example model files can be found in the `benchmarks` directory.

**Graphical Interface** A graphical user interface for MUNTA is also available: <https://munta.isabelle.systems>. The GUI frontend can communicate with a local instance of MUNTA behind a Python-based server or run MUNTA directly in the browser. Detailed run instructions are given on Github: <https://github.com/wimmers/munta/?tab=readme-ov-file#graphical-user-interface>.

**OCaml Code Generation** While the default backend targets SML/MLton, MUNTA in principle also supports code generation to OCaml (c.f. theory `Simple_Network_Language_Export_Code` on page 475). Among its use cases is the browser-based version of the GUI, which is compiled from the OCaml code. For instructions on compiling with the `dune` build system, see the `ocaml` branch: <https://github.com/wimmers/munta/tree/ocaml>.

```

theory Networks
imports Timed_Automata.Timed_Automata Munta_Base.Normalized_Zone_Semantics_Impl
          Munta_Base.Reordering_Quantifiers Munta_Base.TA_Syntax_Bundles
begin

unbundle no_library_syntax

```

# 1 Networks of Timed Automata

## 1.1 Syntax and Operational Semantics

Input, output and internal transitions

```
datatype 'a act = In 'a | Out 'a | Sil 'a
```

```
datatype 'b label = Del | Act 'b | Syn 'b 'b
```

```
type_synonym
```

```
('a, 'b, 'c, 't, 's) nta = ('a act * 'b, 'c, 't, 's) ta list
```

```
inductive step_n ::
```

```
('a, 'b, 'c, 't, 's) nta  $\Rightarrow$  's list  $\Rightarrow$  ('c, ('t::time)) cval  $\Rightarrow$  'b label
```

```
 $\Rightarrow$  's list  $\Rightarrow$  ('c, 't) cval  $\Rightarrow$  bool
```

```
( $\_ \vdash_N \langle \_, \_ \rangle \rightarrow \_ \langle \_, \_ \rangle$  [61,61,61,61,61,61] 61)
```

```
where
```

```
step_n_t:
```

```
 $\llbracket \forall p \in \{..<\text{length } N\}. N!p \vdash \langle L!p, u \rangle \rightarrow^d \langle L!p, u \oplus d \rangle; d \geq 0 \rrbracket$ 
```

```
 $\implies N \vdash_N \langle L, u \rangle \rightarrow_{Del} \langle L, u \oplus d \rangle \mid$ 
```

```
step_n_i:
```

```
 $\llbracket$ 
```

```
 $N!p \vdash l \longrightarrow^{g, (Sil\ a, b), r} l'; u \vdash g; \forall p \in \{..<\text{length } N\}. u' \vdash inv\_of\ (N!p)\ (L!p);$ 
```

```
 $L!p = l; p < \text{length } L; L' = L[p := l']; u' = [r \rightarrow 0]u$ 
```

```
 $\rrbracket$ 
```

```
 $\implies N \vdash_N \langle L, u \rangle \rightarrow_{Act\ b} \langle L', u' \rangle \mid$ 
```

```
step_n_s:
```

```
 $\llbracket N!p \vdash l1 \longrightarrow^{g1, (In\ a, b1), r1} l1'; N!q \vdash l2 \longrightarrow^{g2, (Out\ a, b2), r2} l2'; u \vdash g1; u \vdash g2;$ 
```

```
 $\forall p \in \{..<\text{length } N\}. u' \vdash inv\_of\ (N!p)\ (L!p);$ 
```

```
 $L!p = l1; L!q = l2; p < \text{length } L; q < \text{length } L; p \neq q;$ 
```

```
 $L' = L[p := l1', q := l2']; u' = [(r1 \ @ \ r2) \rightarrow 0]u$ 
```

```
 $\rrbracket \implies N \vdash_N \langle L, u \rangle \rightarrow_{Syn\ b1\ b2} \langle L', u' \rangle$ 
```

```
inductive_cases[elim!]:  $N \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$ 
```

```
inductive steps_n ::
```

```
('a, 'b, 'c, 't, 's) nta  $\Rightarrow$  's list  $\Rightarrow$  ('c, ('t::time)) cval
```

```
 $\Rightarrow$  's list  $\Rightarrow$  ('c, 't) cval  $\Rightarrow$  bool
```

```
( $\_ \vdash_N \langle \_, \_ \rangle \rightarrow^* \langle \_, \_ \rangle$  [61,61,61] 61)
```

```
where
```

```
refl:  $N \vdash_N \langle l, Z \rangle \rightarrow^* \langle l, Z \rangle \mid$ 
```

```
step:  $N \vdash_N \langle l, Z \rangle \rightarrow^* \langle l', Z' \rangle \implies N \vdash_N \langle l', Z' \rangle \rightarrow \_ \langle l'', Z'' \rangle$ 
```

```
 $\implies N \vdash_N \langle l, Z \rangle \rightarrow^* \langle l'', Z'' \rangle$ 
```

```
declare steps_n.intros[intro]
```

**lemma** *stepI2*:

$N \vdash_N \langle l, Z \rangle \rightarrow^* \langle l'', Z'' \rangle$  **if**  $N \vdash_N \langle l', Z' \rangle \rightarrow^* \langle l'', Z'' \rangle$   $N \vdash_N \langle l, Z \rangle \rightarrow_b \langle l', Z' \rangle$   
**using** *that*  
**apply** *induction*  
**apply** (*rule steps\_n.step*)  
**apply** (*rule refl*)  
**apply** *assumption*  
**apply** *simp*  
**by** (*rule steps\_n.step; assumption*)

**lemma** *step\_n\_t\_I*:

$\llbracket \forall p \in \{..<\text{length } N\}. u \vdash \text{inv\_of } (N!p) (L!p); \forall p \in \{..<\text{length } N\}. u \oplus d \vdash \text{inv\_of } (N!p) (L!p);$   
 $d \geq 0$   
 $\rrbracket \implies N \vdash_N \langle L, u \rangle \rightarrow_{Del} \langle L, u \oplus d \rangle$   
**by** (*auto intro: step\_n\_t*)

## 1.2 Product Automaton

**abbreviation** *state\_set* :: ('a, 'c, 'time, 's) *transition set*  $\Rightarrow$  's *set* **where**  
*state\_set* *T*  $\equiv \text{fst } 'T \cup (\text{snd } o \text{ snd } o \text{ snd } o \text{ snd}) 'T$

**lemma** *guard\_concat*:

**assumes**  $\forall g \in \text{set } xs. u \vdash g$   
**shows**  $u \vdash \text{concat } xs$   
**using** *assms* **by** (*induction xs*) (*auto simp: clock\_val\_def*)

**lemma** *guard\_append*:

**assumes**  $u \vdash g1 \ u \vdash g2$   
**shows**  $u \vdash g1 @ g2$   
**using** *assms* **by** (*auto simp: clock\_val\_def*)

**lemma** *concat\_guard*:

**assumes**  $u \vdash \text{concat } xs \ g \in \text{set } xs$   
**shows**  $u \vdash g$   
**using** *assms* **by** (*auto simp: list\_all\_iff clock\_val\_def*)

**lemma** *guard\_consE*:

**assumes**  $u \vdash ac \ \# \ cc$   
**obtains**  $u \vdash_a ac \ u \vdash cc$   
**using** *assms* **unfolding** *clock\_val\_def* **by** *auto*

**lemma** *guard\_consI*:

**assumes**  $u \vdash_a ac \ u \vdash cc$   
**shows**  $u \vdash ac \ \# \ cc$   
**using** *assms* **unfolding** *clock\_val\_def* **by** *auto*

**lemma** *collect\_clock\_pairs\_append\_cases*:

**assumes**  $x \in \text{collect\_clock\_pairs } (g1 @ g2)$   
**shows**  $x \in \text{collect\_clock\_pairs } g1 \ \vee \ x \in \text{collect\_clock\_pairs } g2$   
**using** *assms* **unfolding** *collect\_clock\_pairs\_def* **by** *auto*

**lemma** *list\_update\_nth\_split*:

**assumes**  $j < \text{length } xs$

**shows**  $P \ (xs[i := x] \ ! \ j) = ((i = j \longrightarrow P \ x) \wedge (i \neq j \longrightarrow P \ (xs \ ! \ j)))$   
**using** *assms* **by** (*cases*  $i = j$ ) *auto*

**locale** *Product\_TA\_Defs* =  
**fixes**  $N :: ('a, 'b, 'c, 't, 's) \text{ nta}$   
**begin**

**abbreviation**  $T \equiv \text{map trans\_of } N$

**abbreviation**  $I \equiv \text{map inv\_of } N$

**definition**  $\text{states} = \{L. \text{length } L = \text{length } T \wedge (\forall \ p \in \{..<\text{length } T\}. L!p \in \text{state\_set } (T!p))\}$

**definition**

$\text{product\_trans\_i} =$   
 $\{(L, g, (a, \text{Act } b), r, L[p := l']) \mid L \ p \ g \ a \ b \ r \ l'.$   
 $L \in \text{states} \wedge (L!p, g, (\text{Sil } a, b), r, l') \in T!p \wedge p < \text{length } L\}$

**definition**

$\text{product\_trans\_s} =$   
 $\{(L, g1 \ @ \ g2, (a, \text{Syn } b1 \ b2), r1 \ @ \ r2, L[p := l1', q := l2']) \mid L \ p \ q \ g1 \ g2 \ a \ b1 \ b2 \ r1 \ r2 \ l1' \ l2'.$   
 $L \in \text{states} \wedge (L!p, g1, (\text{In } a, b1), r1, l1') \in T!p \wedge (L!q, g2, (\text{Out } a, b2), r2, l2') \in T!q$   
 $\wedge p < \text{length } L \wedge q < \text{length } L \wedge p \neq q$   
 $\}$

**definition**

$\text{product\_trans} \equiv \text{product\_trans\_i} \cup \text{product\_trans\_s}$

**lemma** *product\_state\_set\_subs*:

**assumes**  $q < \text{length } N \ l \in \text{state\_set product\_trans}$

**shows**  $l \ ! \ q \in \text{state\_set } (T \ ! \ q)$

**using** *assms*

**unfolding** *product\_trans\_def product\_trans\_i\_def product\_trans\_s\_def states\_def*

**apply** *safe*

**apply** (*solves auto*)

**apply** (*solves auto*)

**apply** *simp*

**apply** (*subst list\_update\_nth\_split; force*)

**apply** *simp*

**apply** (*subst list\_update\_nth\_split*)

**apply** *simp*

**apply** *safe*

**apply** (*subst (asm) (2) list\_update\_nth\_split; force*)

**apply** (*subst list\_update\_nth\_split*)

**apply** *simp*

**by** (*subst (asm) (2) list\_update\_nth\_split; force*) — slow: 13s

**definition**

$\text{product\_invariant } L \equiv$   
 $\text{concat } (\text{map } (\lambda \ p. \text{if } p < \text{length } I \text{ then } (I!p) \ (L!p) \ \text{else } []) \ [0..<\text{length } L])$

**definition**  $\text{product\_ta} :: ('a \times 'b \text{ label}, 'c, 't, 's \text{ list}) \text{ ta}$  **where**

$\text{product\_ta} \equiv (\text{product\_trans}, \text{product\_invariant})$

**lemma** *collect\_clk\_i\_product\_invariant*:

```

Timed_Automata.collect_clki product invariant =  $\bigcup$  (Timed_Automata.collect_clki ' set I)
unfolding Timed_Automata.collect_clki_def product_invariant_def
apply (simp add: image_Union)
apply safe
subgoal
  unfolding collect_clock_pairs_def by fastforce
apply clarsimp
subgoal premises prems for a b aa ba x
proof -
  let ?L = map ( $\lambda \_.$  x) [0.. $\text{length } N$ ]
  let ?x = collect_clock_pairs
    (concat
      (map ( $\lambda p.$  if p < length N then (I ! p) (?L ! p) else []))
      [0.. $\text{length } ?L$ ]))
  show ?thesis
    apply (intro exI[where x = ?x] conjI)
    apply (rule exI; rule HOL.refl)
    apply simp
    using prems unfolding collect_clock_pairs_def by (fastforce dest: mem_nth)
qed
done

```

```

lemma states_length:
  assumes L  $\in$  states
  shows length L = length N
  using assms unfolding states_def by auto

```

```

lemma collect_clkt_product_trans_subs:
  Timed_Automata.collect_clkt product_trans  $\subseteq \bigcup$  (Timed_Automata.collect_clkt ' set T)
  unfolding
    Timed_Automata.collect_clkt_def product_trans_def product_trans_i_def product_trans_s_def
  by (fastforce dest: collect_clock_pairs_append_cases states_length)

```

```

lemma collect_clkvt_product_trans_subs:
  collect_clkvt product_trans  $\subseteq \bigcup$  (collect_clkvt ' set T)
  unfolding collect_clkvt_def product_trans_def product_trans_i_def product_trans_s_def
  by (fastforce dest: states_length)

```

```

lemma statesI_aux:
  fixes L
  assumes L: L  $\in$  states
  assumes
    p < length T
    (l, g, a, r, l')  $\in$  T ! p
  shows (L[p := l])  $\in$  states
  unfolding states_def apply clarsimp
  using L apply (simp add: states_def)
  apply safe
  apply (subst list_update_nth_split)
  using assms by (fastforce simp: states_def)+

```

```

lemma product_trans_s_alt_def:
  product_trans_s =
    {(L, g, (a, Syn b1 b2), r, L') | L g a b1 b2 r L'.  $\exists$  p q g1 g2 r1 r2 l1' l2'}.

```

```


$$L \in \text{states} \wedge p < \text{length } L \wedge q < \text{length } L \wedge p \neq q \wedge$$


$$g = g1 \ @ \ g2 \wedge r = r1 \ @ \ r2 \wedge L' = L[p:=l1', q:=l2']$$


$$\wedge (L!p, g1, (In \ a, \ b1), r1, l1') \in T!p \wedge (L!q, g2, (Out \ a, \ b2), r2, l2') \in T!q$$


$$\}$$

unfolding product_trans_s_def by (safe; metis)

context
assumes states_not_empty: states  $\neq \{\}$ 
assumes trans_complete:
 $\forall p < \text{length } T. \forall t1 \in T!p. \text{case } t1 \text{ of } (l1, g1, (In \ a, \ b1), r1, l1')$ 
 $\Rightarrow \exists q < \text{length } T. p \neq q \wedge (\exists l2 \ g2 \ b2 \ r2 \ l2'. (l2, g2, (Out \ a, \ b2), r2, l2') \in T!q) \mid$ 
 $(l1, g1, (Out \ a, \ b1), r1, l1')$ 
 $\Rightarrow \exists q < \text{length } T. p \neq q \wedge (\exists l2 \ g2 \ b2 \ r2 \ l2'. (l2, g2, (In \ a, \ b2), r2, l2') \in T!q) \mid \_ \Rightarrow$ 
True
begin

lemma collect_clk_product_trans_sups:
 $\text{Timed\_Automata.collect\_clk\_product\_trans} \supseteq \bigcup (\text{Timed\_Automata.collect\_clk} \text{ 'set } T)$ 
proof
fix x assume  $x \in \bigcup (\text{Timed\_Automata.collect\_clk} \text{ 'set } T)$ 
then obtain trans l1 g1 a' b1 r1 l1' where x:
 $\text{trans} \in \text{set } T \ (l1, g1, (a', b1), r1, l1') \in \text{trans} \ x \in \text{collect\_clock\_pairs } g1$ 
unfolding Timed_Automata.collect_clk_def by force
then obtain p where p:
 $p < \text{length } T \ T!p = \text{trans}$ 
by (auto dest!: mem_nth)
from states_not_empty obtain L where  $L: L \in \text{states}$  by auto
have  $\text{length } T = \text{length } L$  by (auto simp: states_length[OF <L ∈ states>])
show  $x \in \text{Timed\_Automata.collect\_clk\_product\_trans}$ 
proof (cases a')
case a': (In a)
with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
 $q < \text{length } T \ (l2, g2, (Out \ a, \ b2), r2, l2') \in T!q \ p \neq q$ 
by atomize_elim fastforce
have  $L[p := l1, q := l2] \in \text{states}$  (is  $?L \in \_$ )
using  $L \ p(1) \ x(1,2) \ \text{trans2}$  unfolding  $p(2)[\text{symmetric}]$  by (auto intro!: statesI_aux)
moreover have  $?L!p = l1 \ ?L!q = l2$ 
using  $\langle p \neq q \rangle \langle p < \text{length } T \rangle \langle q < \text{length } T \rangle \langle L \in \text{states} \rangle$  by (auto dest: states_length)
moreover note t = calculation
have  $(?L, g1 \ @ \ g2, (a, \text{Syn } b1 \ b2), r1 \ @ \ r2, ?L[p := l1', q := l2']) \in \text{product\_trans\_s}$ 
unfolding product_trans_s_alt_def
apply clarsimp
using  $t \ p(1) \ x(1,2) \ \text{trans2}$  unfolding  $p(2)[\text{symmetric}]$   $a' \langle \text{length } T = \text{length } L \rangle$ 
by metis
moreover have  $x \in \text{collect\_clock\_pairs } (g1 \ @ \ g2)$ 
using  $x(3)$  by (auto simp: collect_clock_pairs_def)
ultimately show  $?thesis$  unfolding Timed_Automata.collect_clk_def product_trans_def
by force
next
case a': (Out a)
with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
 $q < \text{length } T \ (l2, g2, (In \ a, \ b2), r2, l2') \in T!q \ p \neq q$ 
by atomize_elim fastforce
have  $L[q := l2, p := l1] \in \text{states}$  (is  $?L \in \_$ )

```

```

    using L p(1) x(1,2) trans2 unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
  moreover have ?L ! p = l1 ?L ! q = l2
    using ⟨p ≠ q⟩ ⟨p < length T⟩ ⟨q < length T⟩ ⟨L ∈ states⟩ by (auto dest: states_length)
  moreover note t = calculation
  have (?L, g2 @ g1, (a, Syn b2 b1), r2 @ r1, ?L[q := l2', p := l1']) ∈ product_trans_s
    unfolding product_trans_s_alt_def
    apply clarsimp
    using t p(1) x(1,2) trans2 unfolding p(2)[symmetric] a' ⟨length T = length L⟩
    by metis
  moreover have x ∈ collect_clock_pairs (g2 @ g1)
    using x(3) by (auto simp: collect_clock_pairs_def)
  ultimately show ?thesis unfolding Timed_Automata.collect_clk_def product_trans_def
by force
next
  case a': (Sil a)
  have L[p := l1] ∈ states (is ?L ∈ _)
    using L p(1) x(1,2) unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
  moreover have ?L ! p = l1
    using ⟨p < length T⟩ ⟨L ∈ states⟩ by (auto dest: states_length)
  ultimately have (?L, g1, (a, Act b1), r1, ?L[p := l1']) ∈ product_trans_i
    using x p unfolding product_trans_i_def a' by (force simp: states_length[OF ⟨L ∈
states⟩])
  with x(3) show ?thesis unfolding Timed_Automata.collect_clk_def product_trans_def
by force
qed
qed

lemma collect_clkvt_product_trans_sup:
  collect_clkvt product_trans ⊇ ⋃ (collect_clkvt ' set T)
proof
  fix x assume x ∈ ⋃ (collect_clkvt ' set T)
  then obtain trans l1 g1 a' b1 r1 l1' where x:
    trans ∈ set T (l1, g1, (a', b1), r1, l1') ∈ trans x ∈ set r1
    unfolding collect_clkvt_def by force
  then obtain p where p:
    p < length T T ! p = trans
    by (auto dest!: mem_nth)
  from states_not_empty obtain L where L: L ∈ states by auto
  show x ∈ collect_clkvt product_trans
proof (cases a')
  case a': (In a)
  with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
    q < length T (l2, g2, (Out a, b2), r2, l2') ∈ T ! q p ≠ q
    by atomize_elim fastforce
  moreover have L[p := l1, q := l2] ∈ states (is ?L ∈ _)
    using L p(1) x(1,2) trans2 unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
  moreover have ?L ! p = l1 ?L ! q = l2
    using ⟨p ≠ q⟩ ⟨p < length T⟩ ⟨q < length T⟩ ⟨L ∈ states⟩ by (auto dest: states_length)
  ultimately have
    (?L, g1 @ g2, (a, Syn b1 b2), r1 @ r2, ?L[p := l1', q := l2']) ∈ product_trans_s
    using p(1) x trans2 unfolding p(2)[symmetric] a' product_trans_s_def
    by (fastforce simp: states_length[OF ⟨L ∈ states⟩])
  with x(3) show ?thesis unfolding collect_clkvt_def product_trans_def by force
next

```

```

case a': (Out a)
with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
  q < length T (l2, g2, (In a, b2), r2, l2') ∈ T ! q p ≠ q
  by atomize_elim fastforce
moreover have L[q := l2, p := l1] ∈ states (is ?L ∈ _)
  using L p(1) x(1,2) trans2 unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
moreover have ?L ! p = l1 ?L ! q = l2
  using ⟨p ≠ q⟩ ⟨p < length T⟩ ⟨q < length T⟩ ⟨L ∈ states⟩ by (auto dest: states_length)
  ultimately have (?L, g2 @ g1, (a, Syn b2 b1), r2 @ r1, ?L[q := l2', p := l1']) ∈
product_trans_s
  using p(1) x trans2 unfolding p(2)[symmetric] a' product_trans_s_def
  apply (simp add: states_length[OF ⟨L ∈ states⟩] del: ex_simps)
  apply (tactic ⟨rearrange_ex_tac @ {context} 1⟩)
  apply simp
  apply (rule exI, rule conjI, assumption)
  apply (simp only: ex_simps[symmetric])
  apply (tactic ⟨rearrange_ex_tac @ {context} 1⟩)
  apply simp
  by (fastforce simp: states_length[OF ⟨L ∈ states⟩])

with x(3) show ?thesis unfolding collect_clkvt_def product_trans_def by force
next
case a': (Sil a)
have L[p := l1] ∈ states (is ?L ∈ _)
  using L p(1) x(1,2) unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
moreover have ?L ! p = l1
  using ⟨p < length T⟩ ⟨L ∈ states⟩ by (auto dest: states_length)
ultimately have (?L, g1, (a, Act b1), r1, ?L[p := l1']) ∈ product_trans_i
  using x p unfolding product_trans_i_def a' by (force simp: states_length[OF ⟨L ∈
states⟩])
with x(3) show ?thesis unfolding collect_clkvt_def product_trans_def by force
qed
qed

lemma collect_clkvt_product_trans:
  Timed_Automata.collect_clkvt product_trans = ⋃ (Timed_Automata.collect_clkvt ' set T)
  using collect_clkvt_product_trans_sups collect_clkvt_product_trans_subs by simp

lemma collect_clkvt_product_trans:
  collect_clkvt product_trans = ⋃ (collect_clkvt ' set T)
  using collect_clkvt_product_trans_sups collect_clkvt_product_trans_subs by simp

end

context
  assumes finite_trans: ∀ A ∈ set N. finite (trans_of A)
begin

lemma finite_states:
  finite states
proof -
  have states ⊆ {L. set L ⊆
    (⋃ {fst ' trans_of (N ! p) | p. p < length N} ∪
    ⋃ {(snd ∘ snd ∘ snd ∘ snd) ' trans_of (N ! p) | p. p < length N})}

```

```

       $\wedge \text{length } L = \text{length } N\}$ 
unfolding states_def
apply clarsimp
apply (drule mem_nth)
apply safe
subgoal for  $\_\_ i$ 
  by (erule ballE[where  $x = i$ ]) auto
done
moreover have finite ... using finite_trans by  $-\text{(rule finite_lists_length_eq; auto)}$ 
ultimately show ?thesis by (rule finite_subset)
qed

```

```

lemma finite_product_trans_i:
  finite product_trans_i
proof  $-$ 
  let  $?N = \bigcup (\text{trans\_of } ' \text{ set } N)$ 
  let  $?S = \{(L, p, g, (a, \text{Act } b), r, l') \mid L \text{ p g a b r } l'. \\ L \in \text{states} \wedge (L ! p, g, (\text{Sil } a, b), r, l') \in \text{map trans\_of } N ! p \wedge p < \text{length } L\}$ 
  let  $?R = \{(L, p, g, (a, \text{Act } b), r, l') \mid L \text{ p g a b r } l'. \\ L \in \text{states} \wedge p < \text{length } L \wedge (g, (\text{Sil } a, b), r, l') \in \text{snd } ' ?N\}$ 
  have  $?S \subseteq ?R$  by (fastforce simp: states_length dest: nth_mem)
  moreover have finite ?R using finite_states [simproc add: finite_Collect]
  apply simp
  apply (rule finite_imageI)
  apply (rule finite_SigmaI)
  apply (rule finite_subset[where  $B = \{(p, L). p < \text{length } L \wedge L \in \text{states}\}$ ])
  apply force
  using states_length
  apply  $-$ 
  apply (rule finite_subset[where  $B = \{(p, L). p < \text{length } N \wedge L \in \text{states}\}$ ]; fastforce)
  using finite_trans
  apply (intro finite_vimageI)
  unfolding inj_on_def by auto
  ultimately have finite ?S by (rule finite_subset)
  moreover have product_trans_i =  $(\lambda (L, p, g, a, r, l'). (L, g, a, r, L[p := l'])) ' ?S$ 
  unfolding product_trans_i_def by (auto 4 3 simp: image_Collect)
  ultimately show ?thesis by auto
qed

```

```

lemma finite_Collect_bounded_ex_5' [simp]:
assumes finite  $\{(a, b, c, d, e) . P a b c d e\}$ 
shows
  finite  $\{x. \exists a b c d e. P a b c d e \wedge Q x a b c d e\}$ 
   $\longleftrightarrow (\forall a b c d e. P a b c d e \longrightarrow \text{finite } \{x. Q x a b c d e\})$ 
using assms finite_Collect_bounded_ex [OF assms, where  $Q = \lambda x. \lambda (a, b, c, d, e). Q x a b$ 
 $c d e$ ]
by clarsimp

```

```

lemma finite_product_trans_s:
  finite product_trans_s
proof  $-$ 
  let  $?N = \bigcup (\text{trans\_of } ' \text{ set } N)$ 
  have  $(g1, (\text{In } a, b), r1, l1') \in \text{snd } ' ?N$  if
     $(L ! p, g1, (\text{In } a, b), r1, l1') \in \text{map trans\_of } N ! p \wedge p < \text{length } L \wedge L \in \text{states}$ 

```

```

for  $L\ p\ g1\ a\ b\ r1\ l1'$ 
  using that by (auto simp: states_length dest: nth_mem)
let  $?S = \{(L, p, q, g1, g2, (a, \text{Syn } b1\ b2), r1, r2, l1', l2') \mid L\ p\ q\ g1\ g2\ a\ b1\ b2\ r1\ r2\ l1'\ l2'.$ 
   $L \in \text{states} \wedge (L \vdash p, g1, (\text{In } a, b1), r1, l1') \in \text{map trans\_of } N \vdash p \wedge$ 
   $(L \vdash q, g2, (\text{Out } a, b2), r2, l2') \in \text{map trans\_of } N \vdash q \wedge p < \text{length } L \wedge q < \text{length } L\}$ 
define  $F1$  where  $F1 \equiv \{(g1, a, b, r1, l1') \mid g1\ a\ b\ r1\ l1'. (g1, (\text{In } a, b), r1, l1') \in \text{snd } '?N\}$ 
have  $\text{fin1}$ : finite  $F1$  unfolding  $F1\_def$  using finite_trans
  using  $[[\text{simproc add: finite\_Collect}]]$  by (force simp: inj_on_def intro: finite_vimageI)
define  $F2$  where  $F2 \equiv \{(g1, a, b, r1, l1') \mid g1\ a\ b\ r1\ l1'. (g1, (\text{Out } a, b), r1, l1') \in \text{snd } '?N\}$ 
have  $\text{fin2}$ : finite  $F2$  unfolding  $F2\_def$  using finite_trans
  using  $[[\text{simproc add: finite\_Collect}]]$  by (force simp: inj_on_def intro: finite_vimageI)
define  $R$  where  $R \equiv \{(L, p, q, g1, g2, (a, \text{Syn } b1\ b2), r1, r2, l1', l2') \mid L\ p\ q\ g1\ g2\ a\ b1\ b2$ 
 $r1\ r2\ l1'\ l2'.$ 
 $L \in \text{states} \wedge p < \text{length } L \wedge q < \text{length } L \wedge (g1, a, b1, r1, l1') \in F1 \wedge (g2, a, b2, r2, l2') \in F2\}$ 

```

```

have  $R\_alt\_def$ :  $R = \{t. \exists\ L\ p\ q\ g1\ a\ b1\ r1\ l1'\ g2\ b2\ r2\ l2'.$ 
 $L \in \text{states} \wedge p < \text{length } L \wedge q < \text{length } L$ 
 $\wedge (g1, a, b1, r1, l1') \in F1 \wedge (g2, a, b2, r2, l2') \in F2$ 
 $\wedge t = (L, p, q, g1, g2, (a, \text{Syn } b1\ b2), r1, r2, l1', l2')\}$ 
unfolding  $R\_def$  by auto
have  $?S \subseteq R$  by (fastforce simp: R_alt_def F1_def F2_def states_length dest: nth_mem)
moreover have finite  $R$ 
  unfolding  $R\_alt\_def$ 
  using  $\text{fin1 fin2 finite\_states}$ 
  using  $[[\text{simproc add: finite\_Collect}]]$ 
  by (auto simp: inj_on_def intro: finite_vimageI intro!: finite_imageI)
ultimately have finite  $?S$  by (rule finite_subset)
moreover have
  product_trans_s
   $\subseteq (\lambda\ (L, p, q, g1, g2, a, r1, r2, l1', l2').$ 
   $(L, g1 @ g2, a, r1 @ r2, L[p := l1', q := l2']))\ '?S$ 
  unfolding product_trans_s_def by (simp add: image_Collect) blast
ultimately show  $?thesis$  by (auto intro: finite_subset)
qed

```

```

lemma finite_trans_of_product:
  shows finite (trans_of product_ta)
  using finite_product_trans_s finite_product_trans_i
  unfolding product_ta_def trans_of_def product_trans_def
  by auto

```

**end**

```

lemma inv_of_product[simp]:
  inv_of product_ta = product_invariant
  unfolding product_ta_def inv_of_def by simp

```

```

lemma concat_map_if_aux:
  assumes  $(m :: \text{nat}) \geq n$ 
  shows concat (map  $(\lambda\ i. \text{if } i < n \text{ then } f\ i \text{ else } [])\ [0..<m])$ 
   $= \text{concat} (\text{map} (\lambda\ i. \text{if } i < n \text{ then } f\ i \text{ else } [])\ [0..<n])$ 

```

```

using assms by (induction rule: dec_induct) auto

lemma finite_invariant_of_product[folded inv_of_product]:
  assumes  $\forall A \in \text{set } N. \text{finite } (\text{range } (\text{inv\_of } A))$ 
  shows  $\text{finite } (\text{range } \text{product\_invariant})$ 
proof -
  let ?N =  $\bigcup (\text{range } ' \text{inv\_of } ' \text{set } N)$ 
  let ?X =  $\{I. \text{set } I \subseteq ?N \wedge \text{length } I \leq \text{length } N\}$ 
  have []  $\in ?X$  by auto
  have finite ?N
    using assms
    by auto
  then have finite ?X by (rule finite_lists_length_le)
  moreover have  $\text{range } \text{product\_invariant} \subseteq \text{concat } ' ?X$ 
proof
  fix x assume  $x \in \text{range } \text{product\_invariant}$ 
  then obtain L where L:
     $x = \text{concat}$ 
      ( $\text{map } (\lambda p. \text{if } p < \text{length } (\text{map } \text{inv\_of } N)$ 
         $\text{then } (\text{map } \text{inv\_of } N ! p) (L ! p) \text{ else } [])$ 
      ( $[0..<\text{length } L]$ )
    unfolding product_invariant_def by auto
  show  $x \in \text{concat } ' ?X$ 
proof (cases  $\text{length } L \leq \text{length } N$ )
  case True
  then show ?thesis using L by (fastforce dest: nth_mem)
next
  case False
  then have  $x = \text{concat}$ 
    ( $\text{map } (\lambda p. \text{if } p < \text{length } (\text{map } \text{inv\_of } N)$ 
       $\text{then } (\text{map } \text{inv\_of } N ! p) (L ! p) \text{ else } [])$ 
    ( $[0..<\text{length } N]$ )
    using L by (auto intro: concat_map_if_aux)
  then show ?thesis by (fastforce dest: nth_mem)
qed
qed
ultimately show ?thesis by - (drule finite_subset; auto)
qed

end

locale Product_TA_Defs' =
  Product_TA_Defs N for N :: ('a, 'b, 'c, 't :: time, 's) nta
begin

lemma product_invariantD:
  assumes  $u \vdash \text{product\_invariant } L \ p < \text{length } N \ \text{length } L = \text{length } N$ 
  shows  $u \vdash \text{inv\_of } (N ! p) (L!p)$ 
using assms unfolding product_invariant_def by (force intro: concat_guard)

lemma states_step:
   $L' \in \text{states}$  if  $N \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle \ L \in \text{states}$ 
  using that unfolding states_def apply safe
  apply simp

```

```

    apply simp
    apply force
    apply (subst list_update_nth_split)
    apply simp
    apply (rule conjI)
    apply clarsimp
    subgoal premises prems for p g a r l'
      using prems(3-6) by force
    apply clarsimp
    apply (solves auto)
    apply (subst list_update_nth_split)
    apply (simp; fail)
    apply safe
    apply simp
    subgoal premises prems
      using prems(3-7) by force
    apply (subst list_update_nth_split)
    apply simp
    apply safe
    apply simp
    subgoal premises prems
      using prems(3-6) by force
    by auto

lemma states_steps:
   $L' \in \text{states}$  if  $N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$   $L \in \text{states}$ 
  using that apply (induction  $N \equiv N \_ \_ \_ \_ \text{rule: steps\_n.induct}$ )
  apply assumption
  apply simp
  by (rule states_step; assumption)

end

lemma steps_altI:
   $A \vdash \langle l, Z \rangle \rightarrow^* \langle l'', Z'' \rangle$  if
   $A \vdash \langle l, Z \rangle \rightarrow^* \langle l', Z' \rangle$   $A \vdash \langle l', Z' \rangle \rightarrow \langle l'', Z'' \rangle$ 
  using that by (induction; blast)

locale Product_TA =
  Product_TA_Defs' N for N :: ('a, 'b, 'c, 't :: time, 's) nta +
  fixes L :: 's list
  assumes states[intro]:  $L \in \text{states}$ 
begin

  lemma
    len[simp]:  $\text{length } L = \text{length } N$ 
  using states unfolding states_def by auto

  lemma product_delay_complete:
    assumes step:  $N \vdash_N \langle L, u \rangle \rightarrow_{\text{Del}} \langle L', u' \rangle$ 
    obtains d where product_ta  $\vdash \langle L, u \rangle \rightarrow^d \langle L', u' \rangle$ 
  using step proof cases
    case prems: (step_n_t d)

```

```

from prems have *:
   $\forall p \in \{..<length\ N\}. u \oplus d \vdash inv\_of\ (N\ !\ p)\ (L\ !\ p)$ 
by (auto elim!: step_t.cases)
from prems * show ?thesis
by (fastforce simp add: product_ta_def inv_of_def product_invariant_def[abs_def]
      intro!: guard_concat that
    )
qed

```

```

lemma product_int_complete:
  assumes step:  $N \vdash_N \langle L, u \rangle \rightarrow_{Act\ b} \langle L', u' \rangle$ 
  obtains a where product_ta  $\vdash \langle L, u \rangle \rightarrow_{(a, Act\ b)} \langle L', u' \rangle$ 
  using step proof cases
  case prems: (step_n_i p l g a r l')
  from prems show ?thesis
    apply -
    apply (rule that)
    apply (rule step_a.intros[where g = g and r = r])
    by (fastforce
        simp: product_trans_def product_trans_i_def product_invariant_def product_ta_def
        trans_of_def inv_of_def
        intro: guard_concat
      )+
qed

```

```

lemma product_sync_complete:
  assumes step:  $N \vdash_N \langle L, u \rangle \rightarrow_{Syn\ b1\ b2} \langle L', u' \rangle$ 
  obtains a where product_ta  $\vdash \langle L, u \rangle \rightarrow_{(a, Syn\ b1\ b2)} \langle L', u' \rangle$ 
  using step proof cases
  case prems: (step_n_s p l1 g1 a r1 l1' q l2 g2 r2 l2')
  from prems show ?thesis
    apply -
    apply (rule that)
    apply (rule step_a.intros[where g = g1 @ g2 and a = (a, Syn b1 b2) and r = r1 @ r2])
    subgoal premises prems
    apply
      (clarsimp simp add:
        product_trans_def product_trans_s_def product_invariant_def product_ta_def
        trans_of_def inv_of_def
      )
    apply (erule allE[where x = p])
    using  $\langle p < \_ \rangle$  apply simp
    apply (erule allE[where x = q])
    using prems by (fastforce simp: trans_of_def)
    by (fastforce
        simp: product_invariant_def product_ta_def inv_of_def
        intro: guard_concat guard_append
      )+
qed

```

```

lemma product_complete:
  assumes step:  $N \vdash_N \langle L, u \rangle \rightarrow_b \langle L', u' \rangle$ 
  shows product_ta  $\vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$ 
  using step

```

by (cases b) (rule product\_delay\_complete product\_int\_complete product\_sync\_complete, simp, blast)+

lemma product\_ta\_cases:  
 assumes product\_ta  $\vdash L \xrightarrow{g,a,r} L'$   
 shows  $(L, g, a, r, L') \in \text{product\_trans\_i} \vee (L, g, a, r, L') \in \text{product\_trans\_s}$   
 using assms unfolding product\_ta\_def trans\_of\_def product\_trans\_def by auto

lemma product\_delay\_sound:  
 assumes step: product\_ta  $\vdash \langle L, u \rangle \rightarrow^d \langle L', u' \rangle$   
 shows  $N \vdash_N \langle L, u \rangle \rightarrow_{Del} \langle L', u' \rangle$   
 using assms by cases (force intro!: step\_n\_t product\_invariantD len)

lemma product\_action\_sound:  
 assumes step: product\_ta  $\vdash \langle L, u \rangle \rightarrow_{(a,b)} \langle L', u' \rangle$   
 shows  $N \vdash_N \langle L, u \rangle \rightarrow_b \langle L', u' \rangle$   
 using assms  
 apply cases  
 apply simp  
 apply (drule product\_ta\_cases)  
 apply (erule disjE)  
 unfolding product\_trans\_i\_def product\_trans\_s\_def  
 apply (solves (auto intro!: product\_invariantD step\_n\_i))  
 by (auto intro!: product\_invariantD step\_n\_s simp: clock\_val\_def)

lemma product\_sound:  
 assumes step: product\_ta  $\vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$   
 obtains a where  $N \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$   
 using step by cases (force dest!: product\_action\_sound product\_delay\_sound)+

lemma states\_product\_step[intro]:  
 $L' \in \text{states}$  if product\_ta  $\vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$   
 by (auto intro: product\_sound that elim!: states\_step)

lemma product\_steps\_sound:  
 $N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$  if product\_ta  $\vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$   
 using that states proof (induction A  $\equiv$  product\_ta \_ \_ \_ rule: steps.induct)  
 case (refl l u)  
 then show ?case by blast  
 next  
 case prems: (step l u l' u' l'' u'')  
 interpret interp: Product\_TA N l by standard (rule prems)  
 from prems have \*:  $N \vdash_N \langle l', u' \rangle \rightarrow^* \langle l'', u'' \rangle$  by auto  
 from prems show ?case by - (erule interp.product\_sound, rule stepI2, rule \*)  
 qed

lemma product\_steps\_complete:  
 product\_ta  $\vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$  if  $N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$   
 using that states proof (induction N  $\equiv$  N \_ \_ \_ rule: steps\_n.induct)  
 case (refl l Z)  
 then show ?case by blast  
 next  
 case prems: (step l Z l' Z' \_ l'' Z'')

```

interpret interp: Product_TA N l' by standard (blast intro: prems states_steps)
from prems show ?case by – (assumption | rule steps_altI interp.product_complete) +
qed

lemma product_correct:
  product_ta  $\vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle \longleftrightarrow N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$ 
by (metis product_steps_complete product_steps_sound)

end

end
theory State_Networks
imports Networks Munta_Base.Normalized_Zone_Semantics_Impl
         Munta_Base.More_Methods
begin

unbundle no_library_syntax

```

## 2 Networks of Timed Automata With Discrete State

### 2.1 Syntax and Operational Semantics

We extend Networks of Timed Automata with arbitrary shared (global) state. Syntactically, this extension is very simple. We can just use the free action label slot to annotate edges with a guard and an update function on discrete states. The slightly more clumsy part is adding invariants for discrete states by directly specifying an invariant annotating function.

```

type_synonym
  ('a, 'c, 'time, 's, 'st) transition =
    's * ('st  $\Rightarrow$  ('c, 'time) cconstraint) * 'a * ('st  $\Rightarrow$  'c list) * 's

type_synonym
  ('a, 'c, 'time, 's, 'st) sta = ('a, 'c, 'time, 's, 'st) transition set * ('c, 'time, 's) invassn

type_synonym
  ('a, 'c, 't, 's, 'st) snta =
    ('a act  $\times$  ('st  $\Rightarrow$  bool)  $\times$  ('st  $\Rightarrow$  'st option), 'c, 't, 's, 'st) sta list  $\times$  ('s  $\Rightarrow$  'st  $\Rightarrow$  bool) list

Semantic states now consist of three things: a list of process locations, the shared state, and a clock valuation. The semantic extension then is also obvious: we can take the same transitions as in the network without shared state, however we have to add state updates and checks for guards on the shared state. The updates on discrete state for synchronizing transitions are in the same order as in UPPAAL (output before input).

datatype 'b label = Del | Act 'b | Syn 'b

inductive step_sn ::
  ('a, 'c, 't, 's, 'st) snta  $\Rightarrow$  's list  $\Rightarrow$  'st  $\Rightarrow$  ('c, ('t::time)) cval  $\Rightarrow$  'a label
   $\Rightarrow$  's list  $\Rightarrow$  'st  $\Rightarrow$  ('c, 't) cval  $\Rightarrow$  bool
  ( $\_ \vdash \_ \rightarrow \_$ )  $\rightarrow \_$  [61,61,61,61,61] 61)
where
  step_sn_t:
    (N, I)  $\vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L, s, u \oplus d \rangle$ 
    if  $\forall p \in \{..<length\ N\}. u \oplus d \vdash snd\ (N\ !\ p)\ (L\ !\ p)$ 
        $d \geq 0\ length\ N = length\ I$  |

```

*step\_sn\_i*:  
 $(N, I) \vdash \langle L, s, u \rangle \rightarrow_{Act\ a} \langle L', s', u' \rangle$   
**if**  $(l, g, (Sil\ a, c, m), f, l') \in fst\ (N!p)$   
 $u \vdash g\ s\ \forall\ p \in \{..<length\ N\}. u' \vdash snd\ (N!p)\ (L'!p)$   
 $r = f\ s$   
 $L!p = l\ p < length\ L\ L' = L[p := l']\ u' = [r \rightarrow 0]u$   
 $c\ s\ \forall\ p < length\ I. (I!p)\ (L'!p)\ s'\ Some\ s' = m\ s$   
 $length\ N = length\ I\ |$   
*step\_sn\_s*:  
 $(N, I) \vdash \langle L, s, u \rangle \rightarrow_{Syn\ a} \langle L', s', u' \rangle$   
**if**  $(l1, g1, (In\ a, ci, mi), f1, l1') \in fst\ (N!p)$   
 $(l2, g2, (Out\ a, co, mo), f2, l2') \in fst\ (N!q)\ u \vdash g1\ s\ u \vdash g2\ s$   
 $\forall\ p \in \{..<length\ N\}. u' \vdash snd\ (N!p)\ (L'!p)$   
 $r1 = f1\ s\ r2 = f2\ s$   
 $L!p = l1\ L!q = l2\ p < length\ L\ q < length\ L\ p \neq q$   
 $L' = L[p := l1', q := l2']\ u' = [(r1\ @\ r2) \rightarrow 0]u$   
 $ci\ s\ co\ s\ \forall\ p < length\ I. (I!p)\ (L'!p)\ s'$   
 $Some\ so = mo\ s\ Some\ s' = mi\ so\ length\ N = length\ I$

**inductive\_cases**[elim!]:  $N \vdash \langle L, s, u \rangle \rightarrow_{Syn\ a} \langle L', s', u' \rangle$

**inductive steps\_sn** ::  
 $( 'a, 'c, 't, 's, 'st)\ snta \Rightarrow 's\ list \Rightarrow 'st \Rightarrow ('c, ('t::time))\ cval$   
 $\Rightarrow 's\ list \Rightarrow 'st \Rightarrow ('c, 't)\ cval \Rightarrow bool$   
 $(\_ \vdash \_, \_, \_) \rightarrow^* \_, \_, \_ [61, 61, 61, 61, 61]\ 61)$   
**where**  
 $refl: N \vdash \langle L, s, u \rangle \rightarrow^* \langle L, s, u \rangle\ |$   
 $step: N \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \Longrightarrow N \vdash \langle L', s', u' \rangle \rightarrow_l \langle L'', s'', u'' \rangle$   
 $\Longrightarrow N \vdash \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$

**declare steps\_sn.intros**[intro]

**lemma stepI2**:  
 $N \vdash \langle l, s, u \rangle \rightarrow^* \langle l'', s'', u'' \rangle$  **if**  
 $N \vdash \langle l', s', u' \rangle \rightarrow^* \langle l'', s'', u'' \rangle\ N \vdash \langle l, s, u \rangle \rightarrow_a \langle l', s', u' \rangle$   
**using that**  
**apply induction**  
**apply rule**  
**apply (rule refl)**  
**apply assumption**  
**apply simp**  
**by (rule; assumption)**

**abbreviation state\_set** ::  $( 'a, 'c, 't, 's, 'st)\ transition\ set \Rightarrow 's\ set$  **where**  
 $state\_set\ T \equiv fst\ 'T \cup (snd\ o\ snd\ o\ snd\ o\ snd)\ 'T$

## 2.2 Product Automaton

**locale Prod\_TA\_Defs** =  
**fixes**  $A :: ('a, 'c, 't, 's, 'st)\ snta$   
**begin**

**definition**  
 $T\_s\ p\ s = \{(l, g\ s, a, f\ s, l') \mid l\ g\ a\ f\ l'. (l, g, a, f, l') \in fst\ (fst\ A!p)\}$

**definition**

$$N\_s\ s = \text{map } (\lambda\ p.\ (T\_s\ p\ s,\ \text{snd}\ (\text{fst}\ A\ !\ p)))\ [0..\text{length}\ (\text{fst}\ A)]$$

**abbreviation**  $P \equiv \text{snd}\ A$

**definition**  $p \equiv \text{length}\ (\text{fst}\ A)$

**abbreviation**  $\text{product}\ s \equiv \text{Product\_TA\_Defs.product\_ta}\ (N\_s\ s)$

**abbreviation**  $T'\ s \equiv \text{trans\_of}\ (\text{product}\ s)$

**abbreviation**  $I'\ s \equiv \text{inv\_of}\ (\text{product}\ s)$

**definition**

$$\begin{aligned} \text{prod\_trans\_i} = & \\ & \{((L, s), g, a, r, (L', s')) \mid L\ s\ g\ c\ a\ r\ m\ L'\ s'. \\ & (\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s') \\ & \wedge (L, g, (a, \text{Networks.label.Act}\ (c, m)), r, L') \in T'\ s \wedge c\ s \wedge \text{Some}\ s' = m\ s\} \end{aligned}$$
**definition**

$$\begin{aligned} \text{prod\_trans\_s} = & \\ & \{((L, s), g, a, r, (L', s')) \mid L\ s\ g\ ci\ co\ a\ r\ mi\ mo\ L'\ s'\ so. \\ & ci\ s \wedge co\ s \\ & \wedge (\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s') \\ & \wedge (L, g, (a, \text{Networks.label.Syn}\ (ci, mi)\ (co, mo)), r, L') \in T'\ s \\ & \wedge \text{Some}\ so = mo\ s \\ & \wedge \text{Some}\ s' = mi\ so \\ & \} \end{aligned}$$
**definition**

$$\text{prod\_trans} \equiv \text{prod\_trans\_i} \cup \text{prod\_trans\_s}$$
**definition**

$$\text{prod\_invariant} \equiv \lambda\ (L, s).\ I'\ s\ L$$

**definition**  $\text{prod\_ta} :: ('a, 'c, 't, 's\ \text{list} \times 'st)\ \text{ta} \rightarrow \text{ta}$  **where**

$$\text{prod\_ta} \equiv (\text{prod\_trans}, \text{prod\_invariant})$$

**lemma**  $\text{prod\_ta\_cases}$ :

**assumes**  $\text{prod\_ta} \vdash L \xrightarrow{g,a,r} L'$

**shows**  $(L, g, a, r, L') \in \text{prod\_trans\_i} \vee (L, g, a, r, L') \in \text{prod\_trans\_s}$

**using** *assms* **unfolding**  $\text{prod\_ta\_def}\ \text{trans\_of\_def}\ \text{prod\_trans\_def}$  **by** *auto*

**lemma**  $\text{inv\_of\_simp}$ :

$$\text{inv\_of}\ \text{prod\_ta}\ (L, s) = I'\ s\ L$$

**unfolding**  $\text{inv\_of\_def}\ \text{prod\_ta\_def}\ \text{prod\_invariant\_def}$  **by** *simp*

**lemma**  $I'\_simp$ :

$$I'\ s\ L = I'\ s'\ L$$

**unfolding**  $\text{Product\_TA\_Defs.product\_ta\_def}\ \text{inv\_of\_def}\ \text{Product\_TA\_Defs.product\_invariant\_def}$   
 $N\_s\_def$

**apply** *simp*

**apply**  $(\text{rule}\ \text{arg\_cong}[\text{where}\ f = \text{concat}])$

**by** *simp*

**lemma** *collect\_clk\_i\_prod\_invariant*:

*Timed\_Automata.collect\_clk\_i\_prod\_invariant* = *Timed\_Automata.collect\_clk\_i* (*I'* *s*)  
**unfolding** *prod\_invariant\_def* *Timed\_Automata.collect\_clk\_i\_def*  
**apply** (*simp split: prod.split*)  
**apply** *safe*  
**apply** (*subst (asm) I'\_simp[where s' = s]*)  
**by** *auto*

**lemma** *collect\_clk\_i\_prod\_invariant'*:

*Timed\_Automata.collect\_clk\_i\_prod\_invariant*  
 $\subseteq \bigcup \{ \text{Timed\_Automata.collect\_clk\_i (snd (fst A ! p))} \mid p. p < \text{length (fst A)} \}$   
**unfolding** *collect\_clk\_i\_prod\_invariant[of s]*  
**unfolding** *inv\_of\_def Product\_TA\_Defs.product\_ta\_def*  
**unfolding** *Product\_TA\_Defs.product\_invariant\_def*  
**unfolding** *inv\_of\_def N\_s\_def*  
**unfolding** *Timed\_Automata.collect\_clk\_i\_def*  
**unfolding** *collect\_clock\_pairs\_def*  
**by** *auto*

**lemma** *collect\_clkt\_prod\_trans\_subs*:

*Timed\_Automata.collect\_clkt\_prod\_trans*  $\subseteq$  *Timed\_Automata.collect\_clkt* ( $\bigcup (T' \text{ ' UNIV})$ )  
**unfolding** *Timed\_Automata.collect\_clkt\_def prod\_trans\_def prod\_trans\_i\_def prod\_trans\_s\_def*  
**by** *fastforce*

**lemma** *collect\_clkvt\_prod\_trans\_subs*:

*collect\_clkvt\_prod\_trans*  $\subseteq$  *collect\_clkvt* ( $\bigcup (T' \text{ ' UNIV})$ )  
**unfolding** *collect\_clkvt\_def prod\_trans\_def prod\_trans\_i\_def prod\_trans\_s\_def* **by** *fastforce*

**lemma** *T\_simp*:

*T ! q = trans\_of (N ! q)* **if** *q < length N*  
**using** *that oops*

**abbreviation** *N*  $\equiv$  *fst A*

**context**

**fixes** *Q*

**assumes** *finite\_state*:

$\forall l. \forall q < p. (P ! q) \ l \ s \longrightarrow Q \ s$

*finite*  $\{s. Q \ s\}$

**and** *finite\_trans*:  $\forall A \in \text{set } N. \text{finite (fst A)}$

**and** *p\_gt\_0*: *p > 0*

**begin**

**lemma** *finite\_state'*:

*finite*  $\{s. \forall q < p. (P ! q) \ (L ! q) \ s\}$  **(is finite ?S)**

**proof** –

**from** *p\_gt\_0* **obtain** *q* **where** *q < p* **by** *blast*

**then have**  $?S \subseteq \{s. Q \ s\}$  **using** *finite\_state(1)* **by** *auto*

**moreover have** *finite ...* **by** (*rule finite\_state(2)*)

**ultimately show** *?thesis* **by** (*rule finite\_subset*)

**qed**

```

lemma finite_trans':
   $\forall A \in \text{set } (N\_s\ s). \text{finite } (\text{trans\_of } A)$ 
unfolding N_s_def apply clarsimp
  unfolding trans_of_def T_s_def
  apply simp
  apply (drule nth_mem)
  using finite_trans
  using [[simproc add: finite_Collect]]
  apply simp
  apply (rule finite_imageI)
  apply (rule finite_vimageI)
  apply simp
  unfolding inj_on_def by auto

lemma finite_states:
  finite (Product_TA_Defs.states (N_s s))
  using finite_trans' by (rule Product_TA_Defs.finite_states)

lemma
  finite (T' s)
  using finite_trans' by (rule Product_TA_Defs.finite_trans_of_product)

lemma finite_product_1:
  finite (T' s)
  unfolding product_def
  unfolding trans_of_def Product_TA_Defs.product_ta_def
  apply simp
  unfolding Product_TA_Defs.product_trans_def
proof safe
  have Product_TA_Defs.product_trans_i (N_s s)
     $\subseteq \{(L, g, (a, \text{Networks.label.Act } (aa, b)), r, L[p := l']) \mid L\ p\ g\ a\ aa\ b\ r\ l'. \\ L \in \text{Product\_TA\_Defs.states } (N\_s\ s) \wedge p < \text{length } (N\_s\ s) \wedge \\ (L \ !\ p, g, (\text{Sil } a, aa, b), r, l') \in \bigcup (\text{trans\_of } ' \text{set } (N\_s\ s))\}$ 
  unfolding Product_TA_Defs.product_trans_i_def
  by (fastforce simp: Product_TA_Defs.states_length)
moreover have finite ...
  apply defer_ex
  using finite_states[of s] apply clarsimp
  apply (subst finite_Collect_bounded_ex_6)
  subgoal premises prems for y y'
  proof -
    have
       $\{(a, b, c, d, e, f). \exists x \in \text{set } (N\_s\ s). x \vdash y \ !\ y' \longrightarrow^{a, (\text{Sil } b, c, d), e} f\}$ 
       $= \{xx. \exists x\ a\ b\ c\ d\ e\ f. x \in \text{set } (N\_s\ s) \wedge x \vdash y \ !\ y' \longrightarrow^{a, (\text{Sil } b, c, d), e} f \\ \wedge xx = (a, b, c, d, e, f)\}$ 
    by force
  moreover have finite ...
    using finite_trans'[of s]
    using [[simproc add: finite_Collect]]
    by (auto simp: inj_on_def intro: finite_vimageI simp del: finite_Collect_bounded_ex)
  ultimately show ?thesis by simp

```

```

qed
by auto
ultimately show finite (Product_TA_Defs.product_trans_i (N_s s)) by (rule finite_subset)
next
have Product_TA_Defs.product_trans_s (N_s s)
  ⊆ {(L, g1 @ g2, (a, Networks.label.Syn b1 b2), r1 @ r2, L[p := l1', q := l2']) |
    L p q g1 g2 a b1 b2 r1 r2 l1' l2'.
    L ∈ Product_TA_Defs.states (N_s s) ∧
    p < length (N_s s) ∧ q < length (N_s s) ∧
    (L ! p, g1, (In a, b1), r1, l1') ∈ map trans_of (N_s s) ! p ∧
    (L ! q, g2, (Out a, b2), r2, l2') ∈ map trans_of (N_s s) ! q ∧ p ≠ q}
  unfolding Product_TA_Defs.product_trans_s_def
  by (fastforce simp: Product_TA_Defs.states_length)
moreover have finite ...
  apply defer_ex
  using finite_states[of s]
  apply clarsimp
  subgoal
    apply (mini_existential, simp)
    apply (mini_existential, simp)
    apply (mini_existential, simp)
    apply (mini_existential, simp)
    apply (subst finite_Collect_bounded_ex_6)
  subgoal
    using [[simproc add: finite_Collect]] finite_trans'[of s]
    by (auto simp: inj_on_def intro: finite_vimageI)
  apply safe
  apply (subst finite_Collect_bounded_ex_5)
  subgoal
    using [[simproc add: finite_Collect]] finite_trans'[of s]
    by (auto 4 3 simp: simp: inj_on_def intro: finite_vimageI)
  by auto
done
ultimately show finite (Product_TA_Defs.product_trans_s (N_s s)) by (rule finite_subset)
qed

```

```

lemma prod_trans_i_alt_def:
  prod_trans_i =
    {((L, s), g, a, r, (L', s')) | L s g c a r m L' s'.
      (L, g, (a, Networks.label.Act (c, m)), r, L') ∈ T' s ∧
      (∀ q < p. (P ! q) (L ! q) s) ∧ (∀ q < p. (P ! q) (L' ! q) s')
      ∧ c s ∧ Some s' = m s}
  unfolding prod_trans_i_def by (safe; metis)

```

```

lemma Some_finite:
  finite {x. Some x = y}
  using not_finite_existsD by fastforce

```

```

lemma finite_prod_trans:
  finite prod_trans if p > 0
  unfolding prod_trans_def
proof safe
  have prod_trans_i ⊆
    {((L, s), g, a, r, (L', s')) | L s g c a r m L' s'.

```

```

     $Q\ s \wedge$ 
     $(L, g, (a, \text{Networks.label.Act } (c, m)), r, L') \in T'\ s \wedge$ 
     $(\forall q < p. (P ! q) (L ! q) s) \wedge (\forall q < p. (P ! q) (L' ! q) s')$ 
     $\wedge c\ s \wedge \text{Some } s' = m\ s\}$ 

    unfolding prod_trans_i_alt_def
    using finite_state(1) p_gt_0 by force
moreover have
  finite ...
  apply defer_ex
  apply (mini_existential, simp only: ex_simps)
  using finite_state(2) apply clarsimp
  apply (subst finite_Collect_bounded_ex_7)
  using [simproc add: finite_Collect] finite_state' finite_product_1
  by (auto 4 3 simp: inj_on_def intro: finite_vimageI)
ultimately show finite prod_trans_i by (rule finite_subset)
next
have prod_trans_s  $\subseteq$ 
   $\{((L, s), g, a, r, (L', s')) \mid L\ s\ g\ ci\ co\ a\ r\ mi\ mo\ L'\ s'\ so.$ 
   $Q\ s \wedge$ 
   $product\ s \vdash L \longrightarrow^{g, (a, \text{Networks.label.Syn } (ci, mi) (co, mo)), r} L' \wedge$ 
   $(\forall q < p. (P ! q) (L ! q) s) \wedge (\forall q < p. (P ! q) (L' ! q) s') \wedge$ 
   $ci\ s \wedge co\ s \wedge \text{Some } so = mo\ s \wedge \text{Some } s' = mi\ so\}$ 

  unfolding prod_trans_s_def
  using finite_state(1) p_gt_0 by fastforce
moreover have
  finite ...
  apply defer_ex
  apply (mini_existential, simp only: ex_simps)
  using finite_state(2) apply clarsimp
  apply (subst finite_Collect_bounded_ex_9)

  subgoal
    using [simproc add: finite_Collect] finite_state' finite_product_1
    by (auto 4 3 simp: inj_on_def intro: finite_vimageI)[]

  apply safe
  subgoal for s a b c d e f g h i
    apply (rule finite_subset [where  $B =$ 
       $(\lambda s'. ((a, s), b, e, f, i, s'))\ ' \{ s'. \exists so. \text{Some } so = h\ s \wedge \text{Some } s' = g\ so\}$ 
       $\}$ ])
    apply force
    apply (rule finite_imageI)
    apply (subst finite_Collect_bounded_ex)
    by (force intro: Some_finite) +
  done
ultimately show finite prod_trans_s by (rule finite_subset)
qed

end

abbreviation states' s  $\equiv$  Product_TA_Defs.states (N_s s)

```

```

lemma N_s_length:
  length (N_s s) = p
  unfolding N_s_def p_def by simp

end

locale Prod_TA_Defs' =
  Prod_TA_Defs A for A :: ('a, 'c, 't :: time, 's, 'st) snta
begin

lemma A_unfold:
  A  $\equiv$  (N, P)
  by auto

lemma network_step_delay:
  assumes step: (N, P)  $\vdash$   $\langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$  and len: length L = p
  shows N_s s  $\vdash_N \langle L, u \rangle \rightarrow_{Networks.label.Del} \langle L', u' \rangle$ 
  subgoal
    using step
    apply cases
    subgoal
      apply simp
      apply (rule step_n_t)
      subgoal
        unfolding N_s_def by (auto simp: inv_of_def)
      apply assumption
    done
  done
done

lemma network_step_silent:
  assumes step: (N, P)  $\vdash$   $\langle L, s, u \rangle \rightarrow_{Act\ a} \langle L', s', u' \rangle$  and len: length L = p
  obtains a where N_s s  $\vdash_N \langle L, u \rangle \rightarrow_{Networks.label.Act\ a} \langle L', u' \rangle$ 
  subgoal premises prems
    using step
    apply cases
    subgoal
      apply (rule prems)
      apply (rule step_n_i)
      unfolding N_s_def T_s_def by (auto  $\&$  0 simp: trans_of_def inv_of_def len p_def)
    done
  done

lemma network_step_sync:
  assumes step: (N, P)  $\vdash$   $\langle L, s, u \rangle \rightarrow_{Syn\ a} \langle L', s', u' \rangle$  and len: length L = p
  obtains a b where N_s s  $\vdash_N \langle L, u \rangle \rightarrow_{Networks.label.Syn\ a\ b} \langle L', u' \rangle$ 
  subgoal premises prems
    using step
    apply cases
    subgoal
      subgoal premises A
        apply (rule prems)
        apply (rule step_n_s)
      defer

```

```

      defer
      apply (rule A; fail)
      apply (rule A(4); fail)
    subgoal
      using A unfolding N_s_def by (auto simp: inv_of_def len)
      defer
      defer
      apply (rule A; fail)
      apply (rule A(11); fail)
      using A unfolding N_s_def T_s_def by (auto 4 0 simp: trans_of_def len p_def)
    done
  done
done

```

**lemma** *network\_step*:

```

  assumes step:  $(N, P) \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$  and len:  $\text{length } L = p$ 
  obtains a where  $N_s s \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$ 
  subgoal
    using step
    apply (cases a; simp)
    apply (rule that, erule network_step_delay[OF _ len, simplified])
    apply (erule network_step_silent[OF _ len, simplified], erule that)
    apply (erule network_step_sync[OF _ len, simplified], erule that)
  done
done

```

**lemma** *trans\_of\_N\_s\_1*:

```

  (fst ' trans_of (N_s s ! q)) = fst ' fst (N ! q) if  $q < p$ 
  using that unfolding trans_of_def N_s_def p_def T_s_def by (auto 0 7 simp: image_iff)

```

**lemma** *trans\_of\_N\_s\_2*:

```

  ((snd o snd o snd o snd) ' trans_of (N_s s ! q)) = (snd o snd o snd o snd) ' fst (N ! q) if  $q < p$ 
  using that unfolding trans_of_def N_s_def p_def T_s_def by force

```

**lemma**

```

  fst ' trans_of (N_s s ! q) = fst ' trans_of (N_s s' ! q) if  $q < p$ 
  using that by (simp add: trans_of_N_s_1)

```

**lemma** *states'\_simp*:

```

  states' s = states' s'
  unfolding Product_TA_Defs.states_def using trans_of_N_s_1 trans_of_N_s_2 by (simp
  add: N_s_length)

```

**lemma** *states\_step*:

```

   $L' \in \text{states}' s$  if  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$   $L \in \text{states}' s$ 

```

**proof** –

```

  interpret Product_TA_Defs' N_s s .
  from  $\langle L \in \_ \rangle$  have  $L \in \text{states}$  .
  from  $\langle L \in \_ \rangle$  have  $\text{length } L = p$  by (simp add: N_s_length states_length)
  with network_step[folded A_unfold, OF that(1)] obtain a where
     $N_s s \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$ 
  by auto
  then show ?thesis using that(2) by (rule states_step)

```



$u \vdash \text{inv\_of\_prod\_ta } (l, s') \text{ if } u \vdash \text{inv\_of\_product\_ta } l \text{ length } l = p$   
**using that by** (*simp add: product\_inv\_prod\_simp*)

**lemma** *A\_simp*[*simp*]:  
 $N' = N \ P' = P \text{ if } A = (N', P')$   
**using that by** *auto*

**lemma** *length\_L*[*intro*]:  
 $\text{length } L = p$   
**by** (*simp add: N\_s\_length*)

**lemma** *prod\_complete\_delay*:  
**assumes** *step*:  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$   
**obtains** *d* **where**  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$   
**using step proof cases**  
**case** *prems*: (*step\_sn\_t* *N d P*)  
**note** [*simp*] = *A\_simp*[*OF prems(1)*]  
**from prems have**  $N\_s \ s \vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.Del}} \langle L', u' \rangle$   
**unfolding** *N\_s\_def* **by** (*auto 4 3 simp: inv\_of\_def intro: step\_n\_t*)  
**with prems show** *?thesis*  
**by** (*auto 4 4*  
*intro: that*  
*simp: product\_inv\_prod\_simp[OF length\_L]*  
*elim!: product\_delay\_complete step\_t.cases*)

**qed**

**lemma** *prod\_complete\_silent*:  
**assumes** *step*:  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Act } a} \langle L', s', u' \rangle$   
**obtains** *a* **where**  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$   
**using step proof cases**  
**case** *prems*: (*step\_sn\_i l g c m f l' N q r I*)  
**note** [*simp*] = *A\_simp*[*OF prems(1)*]  
**from prems(13) have** [*simp*]:  $\text{length } P = p$  **by** (*simp add: p\_def*)  
**have**  $N\_s \ s \vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.label.Act } (c, m)} \langle L', u' \rangle$   
**apply** (*rule step\_n\_i*)  
**using prems unfolding** *N\_s\_def T\_s\_def* **by** (*auto 3 0 simp: trans\_of\_def inv\_of\_def*  
 $N\_s\_length$ )  
**with**  $\langle \text{length } P = p \rangle$  **obtain** *b* **where**  
 $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow_{(b, \text{Networks.label.Act } (c, m))} \langle L', u' \rangle$   
**by** (*clarsimp elim!: product\_int\_complete*)  
**with prems inv obtain** *g r* **where** *step*:  
 $((L, s), g, b, r, (L', s')) \in \text{prod\_trans\_i}$   
 $u \vdash g \ [r \rightarrow 0] u = u' \ u' \vdash \text{inv\_of\_product\_ta } L'$   
**apply** *atomize\_elim*  
**unfolding** *prod\_trans\_i\_def* **by** - (*erule step\_a.cases; auto*)  
**then have**  $((L, s), g, b, r, (L', s')) \in \text{trans\_of\_prod\_ta}$   
**by** (*simp add: prod\_trans\_def trans\_of\_def prod\_ta\_def*)  
**moreover have**  $\text{length } L' = p$   
**using** *length\_L prems(8)* **by** *auto*  
**ultimately show** *?thesis*  
**apply** -  
**apply** (*rule that*)  
**apply** *rule*

```

    using step(2-) by force+
qed

lemma prod_complete_sync:
  assumes step:  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Syn } a} \langle L', s', u' \rangle$ 
  obtains a where prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ 
using step proof cases
case prems: (step_sn_s l1 g1 ci mi f1 l1' N q1 l2 g2 co mo f2 l2' q2 r1 r2 I so)
note [simp] = A_simp[OF prems(1)]
from prems(21) have [simp]: length P = p by (simp add: p_def)

have N_s s  $\vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.label.Syn } (ci, mi) (co, mo)} \langle L', u' \rangle$ 
  apply (rule step_n_s)
    defer
    defer
    apply (rule prems; fail)
    apply (rule prems(5); fail)
  subgoal
    using prems unfolding N_s_def by (auto simp: inv_of_def)
    defer
    defer
    apply (rule prems; fail)
    apply (rule prems(12); fail)
    using prems unfolding N_s_def T_s_def by (auto 3 0 simp: trans_of_def p_def
N_s_length)
  with  $\langle \text{length } P = p \rangle$  obtain a where
    product_ta  $\vdash \langle L, u \rangle \rightarrow_{(a, \text{Networks.label.Syn } (ci, mi) (co, mo))} \langle L', u' \rangle$ 
    by (auto elim!: product_sync_complete)
  with prems inv obtain g r where step:
     $((L, s), g, a, r, (L', s')) \in \text{prod\_trans\_s}$ 
     $u \vdash g [r \rightarrow 0] u = u' u' \vdash \text{inv\_of product\_ta } L'$ 
    apply atomize_elim
    unfolding prod_trans_s_def by - (erule step_a.cases; auto; blast)
  then have  $((L, s), g, a, r, (L', s')) \in \text{trans\_of prod\_ta}$ 
    by (simp add: prod_trans_def trans_of_def prod_ta_def)
  moreover have length L' = p
    using length_L  $\langle L' = \_ \rangle$  by auto
  ultimately show ?thesis
    apply -
    apply (rule that)
    apply rule
    using step(2-) by force+
qed

lemma prod_complete:
  assumes step:  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
  shows prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  using step
  by (cases a; simp; blast elim!: prod_complete_delay prod_complete_silent prod_complete_sync)

lemma A_unfold:
  A = (N, P)
  by simp

```

lemma *prod\_sound'\_delay*:  
 assumes *step*:  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$   
 shows  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle \wedge \text{product\_ta} \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$   
 $\wedge (\forall p < \text{length } P. (P ! p) (L' ! p) s')$   
 using *assms* **proof** *cases*  
 case *prems*: 1  
 then have  $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow^d \langle L', u' \rangle$  **unfolding** *inv\_of\_simp* **by** *fast*  
 moreover from *product\_delay\_sound*[*OF this*] *prems*(1-3) **have**  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$   
 apply *simp*  
 apply (*subst A\_unfold*)  
 apply (*rule step\_sn\_t*)  
 by (*auto simp: N\_s\_def inv\_of\_def step\_t.simps N\_s\_length p\_def Len intro: <0 ≤ d>*)  
 ultimately show *?thesis* **using** *prems inv* **by** *fast*  
**qed**

lemma *prod\_sound'\_action*:  
 assumes *step*:  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$   
 obtains *a* **where**  $(A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq \text{Del}) \wedge \text{product\_ta} \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$   
 $\wedge (\forall p < \text{length } P. (P ! p) (L' ! p) s')$   
 using *assms*  
**proof** *cases*  
 case *prems*: (1 *g r*)  
 from *Len* **have** [*simp*]:  $\text{length } P = p$  **by** (*simp add: p\_def*)  
 from *prems*(1)[*THEN prod\_ta\_cases*] **show** *?thesis*  
**proof** (*rule disjE, goal\_cases*)  
 case 1  
 then obtain *c m* **where** \*:  
 $\text{Some } s' = m \ s \ \forall q < p. (P ! q) (L ! q) \ s \ \forall q < p. (P ! q) (L' ! q) \ s'$   
 $\text{product\_ta} \vdash L \xrightarrow{g(a, \text{Networks.label.Act } (c, m)), r} L' \ c \ s$   
 unfolding *prod\_trans\_i\_def* **by** *auto*  
 with *prems* **have**  $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow_{(a, \text{Networks.label.Act } (c, m))} \langle L', u' \rangle$   
 unfolding *inv\_of\_simp* **by** (*metis I'\_simp step\_a.intros*)  
 moreover from *product\_action\_sound*[*OF this*] *prems*(3-4) **obtain** *a* **where**  
 $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Act } a} \langle L', s', u' \rangle$   
 apply *safe*  
 apply (*simp add: N\_s\_def trans\_of\_def N\_s\_length T\_s\_def*)  
 apply (*simp only: ex\_simps[symmetric]*)  
 apply (*erule exE, erule exE*)  
 apply (*rule that*)  
 apply (*subst A\_unfold*)  
 apply (*rule step\_sn\_i*)  
 apply *fast*  
 using \*(3) **by** (*auto simp: N\_s\_def inv\_of\_def p\_def <Some s' = m s> intro: <c s>*)  
 ultimately show *?thesis* **using** \* **by** (*auto intro: that*)  
**next**  
 case 2  
 then obtain *ci co mi mo si* **where** \*:  
 $\text{Some } s' = mi \ si \ \text{Some } si = mo \ s \ \forall q < p. (P ! q) (L ! q) \ s \ \forall q < p. (P ! q) (L' ! q) \ s'$   
 $\text{product\_ta} \vdash L \xrightarrow{g(a, \text{Networks.label.Syn } (ci, mi) (co, mo)), r} L'$   
 $ci \ s \ co \ s$   
 unfolding *prod\_trans\_s\_def* **by** *auto*

```

with prems have product_ta ⊢ ⟨L, u⟩ →(a, Networks.label.Syn (ci, mi) (co, mo)) ⟨L', u'⟩
  unfolding inv_of_simp by (metis I'_simp step_a.intros)
moreover from product_action_sound[OF this] prems(3-4) obtain a where
  A ⊢ ⟨L, s, u⟩ →Syn a ⟨L', s', u'⟩
  apply safe
  apply (simp add: N_s_def trans_of_def N_s_length T_s_def)
  apply (simp only: ex_simps[symmetric])
  apply (erule exE, erule exE, erule exE, erule exE)
  apply (rule that)
  apply (subst A_unfold)
  apply (rule step_sn_s)
    apply fast
    apply blast
  using *(4) by (auto simp: N_s_def inv_of_def p_def ⟨Some s' = _⟩ ⟨Some si = _⟩
intro: *(6-))
  ultimately show ?thesis using * by (intro that conjI) auto
qed
qed

lemma prod_sound':
  assumes step: prod_ta ⊢ ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩
  obtains a where A ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩ ∧ product_ta ⊢ ⟨L, u⟩ → ⟨L', u'⟩
    ∧ (∀ p < length P. (P ! p) (L' ! p) s')
  using assms apply cases
  apply (erule prod_sound'_action; blast intro: that)
  apply (rule that, erule prod_sound'_delay)
  done

lemmas prod_sound = prod_sound'[THEN conjunct1]
lemmas prod_inv_1 = prod_sound'[THEN conjunct2, THEN conjunct1]
lemmas prod_inv_2 = prod_sound'[THEN conjunct2, THEN conjunct2]

lemma states_prod_step[intro]:
  L' ∈ states if prod_ta ⊢ ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩
  by (blast intro: prod_inv_1[OF that])

lemma inv_prod_step[intro]:
  ∀ p < length P. (P ! p) (L' ! p) s' if prod_ta ⊢ ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩
  using that by (blast intro: prod_inv_2)

lemma prod_steps_sound:
  assumes step: prod_ta ⊢ ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩
  shows A ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩
  using step states inv
proof (induction A ≡ prod_ta l ≡ (L, s) _ l' ≡ (L', s') _ arbitrary: L s rule: steps.induct)
  case (refl u)
  then show ?case by blast
next
  case prems: (step u l' u' u'' L s)
  obtain L'' s'' where l' = (L'', s'') by force
  interpret inter: Prod_TA A L s by (standard; rule prems Len)
  from prems(3)[OF ⟨l' = _⟩] prems(1,2,4-) have *: A ⊢ ⟨L'', s'', u'⟩ →* ⟨L', s', u'⟩
  unfolding ⟨l' = _⟩

```

```

    by (metis Prod_TA_Defs'.states'_simp interp.states_prod_step interp.inv_prod_step)
  show ?case
    using * prems by (auto simp: <l' = _> intro: interp.prod_sound stepI2)
qed

lemma prod_steps_complete:
  prod_ta ⊢ ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩ if A ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩
  using that states inv proof (induction A ≡ A L _ _ _ rule: steps_sn.induct)
  case (refl L s u)
  then show ?case by blast
next
  case prems: (step L s u L' s' u' L'' s'' u'')
  interpret interp: Prod_TA A L' s' apply standard
  using prems by - (assumption | rule Prod_TA_Defs'.states_steps Len Prod_TA_Defs'.inv_steps)+
  from prems show ?case by - (rule steps_altI, auto intro!: interp.prod_complete)
qed

lemma prod_correct:
  prod_ta ⊢ ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩ ⟷ A ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩
  by (metis prod_steps_complete prod_steps_sound)

end

end
theory UPPAAL_Asm
  imports Main
begin

type_synonym addr = nat
type_synonym val = int
type_synonym reg = nat

datatype instr =
  JMPZ addr |
  ADD |
  NOT |
  AND |
  LT |
  LE |
  EQ |
  PUSH int — Push value on stack |
  POP |
  LID reg — Push register value on stack |
  STORE — Store stack value in register |
  STOREI reg val — Store value in register |
  COPY |
  CALL |
  RETURN |
  HALT |
  STOREC nat int — Special instruction, signals a clock reset |
  SETF bool — Meta instruction

type_synonym stack = int list
type_synonym flag = bool

```

**type\_synonym** *rstate* = *int list* — Partial map from registers to values  
**type\_synonym** *state* = *addr \* stack \* rstate \* flag \* nat list*  
 — Instruction pointer, stack, register state, comparison flag, reset clocks

**definition** *int\_of* :: *bool*  $\Rightarrow$  *int* **where**  
*int\_of* *x*  $\equiv$  *if* *x* *then* 1 *else* 0

**fun** *step* :: *instr*  $\Rightarrow$  *state*  $\Rightarrow$  *state option* **where**  
*step* (JMPZ *q*) (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*if* *f* *then* (*pc* + 1) *else* *q*, *st*, *m*, *f*, *rs*) |  
*step* ADD (*pc*, *a* # *b* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, (*a* + *b*) # *st*, *m*, *f*, *rs*) |  
*step* NOT (*pc*, *b* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*,  $\neg$  *f*, *rs*) |  
*step* AND (*pc*, *b* # *st*, *m*, *f*, *rs*) =  
 (*if* *b* = 0  $\vee$  *b* = 1  
   *then* *Some* (*pc* + 1, *st*, *m*, *b* = 1  $\wedge$  *f*, *rs*)  
   *else* *None*) |  
*step* LT (*pc*, *a* # *b* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*, *a* < *b*, *rs*) |  
*step* LE (*pc*, *a* # *b* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*, *a*  $\leq$  *b*, *rs*) |  
*step* EQ (*pc*, *a* # *b* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*, *a* = *b*, *rs*) |  
*step* (PUSH *v*) (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *v* # *st*, *m*, *f*, *rs*) |  
*step* POP (*pc*, *v* # *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*, *f*, *rs*) |  
*step* (LID *r*) (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *m* ! *r* # *st*, *m*, *f*, *rs*) |  
*step* STORE (*pc*, *v* # *r* # *st*, *m*, *f*, *rs*) =  
 (*if* *r*  $\geq$  0 *then* *Some* (*pc* + 1, *st*, *m*[*nat r* := *v*], *f*, *rs*) *else* *None*) |  
*step* (STOREI *r v*) (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*[*r* := *v*], *f*, *rs*) |  
*step* COPY (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *int\_of* *f* # *st*, *m*, *f*, *rs*) |  
*step* CALL (*pc*, *q* # *st*, *m*, *f*, *rs*) =  
 (*if* *q*  $\geq$  0 *then* *Some* (*nat q*, *int pc* # *st*, *m*, *f*, *rs*) *else* *None*) |  
*step* RETURN (*pc*, *q* # *st*, *m*, *f*, *rs*) =  
 (*if* *q*  $\geq$  0 *then* *Some* (*nat q* + 1, *st*, *m*, *f*, *rs*) *else* *None*) |  
  
*step* (STOREC *c d*) (*pc*, *st*, *m*, *f*, *rs*) =  
 (*if* *d* = 0 *then* *Some* (*pc* + 1, *st*, *m*, *f*, *c* # *rs*) *else* *None*) |  
*step* (SETF *b*) (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc* + 1, *st*, *m*, *b*, *rs*) |  
*step* \_\_ = *None*

**type\_synonym** *program* = *addr*  $\Rightarrow$  *instr option*  
**type\_synonym** *fuel* = *nat*

**fun** *exec* :: *program*  $\Rightarrow$  *fuel*  $\Rightarrow$  *state*  $\Rightarrow$  *addr list*  $\Rightarrow$  (*state* \* *addr list*) *option* **where**  
*exec* \_ 0 \_ = *None* |  
*exec* prog (*Suc n*) (*pc*, *st*, *m*, *f*, *rs*) *pcs* =  
 (*case* prog *pc* of  
   *Some instr*  $\Rightarrow$  (  
   *if* *instr* = HALT  
   *then* *Some* ((*pc*, *st*, *m*, *f*, *rs*), *pc* # *pcs*)  
   *else*  
     *case* *step instr* (*pc*, *st*, *m*, *f*, *rs*) of  
       *Some s*  $\Rightarrow$  *exec* prog *n s* (*pc* # *pcs*)  
       | *None*  $\Rightarrow$  *None*)  
   | *None*  $\Rightarrow$  *None*)

**inductive** *steps* :: *program*  $\Rightarrow$  *fuel*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  $\Rightarrow$  *bool* **where**  
*steps* prog (*Suc n*) *start* *start* |  
*steps* prog (*Suc n*) (*pc*, *st*, *m*, *f*, *rs*) *s'* **if**

```

step cmd (pc, st, m, f, rs) = Some s
prog pc = Some cmd
steps prog n s s'

```

**inductive** *visited* :: *program*  $\Rightarrow$  *fuel*  $\Rightarrow$  *state*  $\Rightarrow$  *state*  $\Rightarrow$  *addr list*  $\Rightarrow$  *bool* **where**  
*visited* prog (Suc n) s s [] |  
*visited* prog (Suc n) (pc, st, m, f, rs) s' (pcs @ [pc]) **if**  
step cmd (pc, st, m, f, rs) = Some s  
prog pc = Some cmd  
*visited* prog n s s' pcs

**lemmas** [intro] = steps.intros

**lemma** *exec\_steps*:  
**assumes** *exec* prog n s pcs = Some ((pc, st, m, f, rs), pcs')  
**shows** steps prog n s (pc, st, m, f, rs)  $\wedge$  prog pc = Some *HALT*  
**using** *assms* **proof** (induction P  $\equiv$  prog n s pcs arbitrary: pc st m f rs rule: *exec.induct*)  
case 1  
**then show** ?case **by** simp  
**next**  
case (2 n pc' st' m' f' rs' pcs')  
**then obtain** instr **where** prog pc' = Some instr **by** (cases prog pc') auto  
**show** ?case  
**proof** (cases instr = *HALT*)  
case True  
**with** 2.premis <prog pc' = \_\_> **show** ?thesis **by** auto  
**next**  
case False  
**with** 2 <prog pc' = \_\_> **show** ?thesis **by** (auto split: option.split\_asm)  
**qed**  
**qed**

**lemma** *steps\_halt*:  
**assumes** steps prog n (pc, st, m, f, rs) s prog pc = Some *HALT*  
**shows** s = (pc, st, m, f, rs)  
**using** *assms* **by** (induction prog n (pc, st, m, f, rs) s) auto

**lemma** *steps\_exec*:  
**assumes** steps prog n s (pc, st, m, f, rs) prog pc = Some *HALT*  
**shows**  $\exists$  pcs'. *exec* prog n s pcs = Some ((pc, st, m, f, rs), pcs')  
**using** *assms* **proof** (induction P  $\equiv$  prog n s (pc, st, m, f, rs) arbitrary: pcs rule: *steps.induct*)  
case (1 n)  
**then show** ?case **by** simp  
**next**  
case (2 cmd pc' st' m' f' rs' s n)  
**then obtain** pcs' **where** *exec* prog n s (pc' # pcs) = Some ((pc, st, m, f, rs), pcs') **by** auto  
**with** 2(1-3) **show** ?case **by** (auto dest!: steps\_halt)  
**qed**

**lemma** *visited\_halt*:  
**assumes** *visited* prog n (pc, st, m, f, rs) s pcs prog pc = Some *HALT*  
**shows** s = (pc, st, m, f, rs)  
**using** *assms* **by** (induction prog n (pc, st, m, f, rs) s pcs) auto

```

lemma exec_length_mono:
  assumes exec prog n s pcs = Some (s', pcs')
  shows length pcs' > length pcs
  using assms
  by (induction P ≡ prog n s pcs arbitrary: s' pcs' rule: exec.induct)
    (force split: option.splits if_splits)+

inductive_cases visitedE1: visited prog n s (pc, st, m, f, rs) []

lemma visited_exec:
  assumes visited prog n s (pc, st, m, f, rs) pcs prog pc = Some HALT
  shows exec prog n s pcs' = Some ((pc, st, m, f, rs), pc # pcs @ pcs')
using assms proof (induction P ≡ prog n s (pc, st, m, f, rs) pcs arbitrary: pcs' rule: visited.induct)
  case (1 n)
  then show ?case by simp
next
  case (2 cmd pc' st' m' f' rs' s n pcs pcs')
  with 2 have exec prog n s (pc' # pcs') = Some ((pc, st, m, f, rs), pc # pcs @ pc' # pcs')
    by auto
  with 2(1-3) show ?case by (auto dest!: steps_halt)
qed

lemma visited_exec':
  assumes visited prog n s (pc, st, m, f, rs) pcs prog pc = Some HALT
  shows exec prog n s [] = Some ((pc, st, m, f, rs), pc # pcs)
using visited_exec assms by auto

end
theory UPPAAL_Asm_Clocks
  imports Timed_Automata.Timed_Automata Timed_Automata.Normalized_Zone_Semantics
    UPPAAL_Asm Complex_Main
begin

datatype 't instrc =
  INSTR instr |
  CEXP (nat, 't) acconstraint

fun stepc :: 't instrc ⇒ (nat, 't :: time) cval ⇒ state ⇒ state option where
  stepc (INSTR instr) u s =
    (case step instr s of
      Some s' ⇒ Some s'
      | None ⇒ None) |
  stepc (CEXP cc) u (pc, st, m, f, rs) = Some (pc + 1, st, m, u ⊢a cc, rs)

type_synonym 't programc = addr ⇒ 't instrc option

inductive stepsc :: 't programc ⇒ fuel ⇒ (nat, 't :: time) cval ⇒ state ⇒ state ⇒ bool where
  stepsc prog (Suc n) u start start |
  stepsc prog (Suc n) u (pc, st, m, f, rs) s' if

```

```

stepc cmd u (pc, st, m, f, rs) = Some s
prog pc = Some cmd
stepsc prog n u s s'

```

```

inductive visitedc :: 't programc ⇒ fuel ⇒ (nat, 't :: time) cval ⇒ state ⇒ state ⇒ addr list ⇒
bool where
  visitedc prog (Suc n) u start start [] |
  visitedc prog (Suc n) u (pc, st, m, f, rs) s' (pcs @ [pc]) if
    stepc cmd u (pc, st, m, f, rs) = Some s
  prog pc = Some cmd
  visitedc prog n u s s' pcs

```

**definition** *stepst* prog n u start  $\equiv \lambda (pc, st, m, f, rs).$   
*stepsc* prog n u start (pc, st, m, f, rs)  $\wedge$  prog pc = Some (INSTR HALT)

```

fun strip :: 't instrc ⇒ instr where
  strip (INSTR instr) = instr |
  strip _ = HALT

```

```

fun stripf :: 't instrc ⇒ instr where
  stripf (INSTR instr) = instr |
  stripf _ = SETF False

```

```

fun stript :: 't instrc ⇒ instr where
  stript (INSTR instr) = instr |
  stript _ = SETF True

```

**end**

**theory** UPPAAL\_State\_Networks

```

imports Munta_Model_Checker.State_Networks Timed_Automata.Normalized_Zone_Semantics
UPPAAL_Asm_Clocks
AutoCorres2.Subgoals

```

**begin**

### 3 Networks of Timed Automata – UPPAAL Style

Networks of Timed Automata with Shared State and UPPAAL-style Assembler guards and updates.

```

no_notation Ref.update (_ := _ 62)
no_notation fun_rel_syn (infixr → 60)

```

**lemma** *finite\_lists\_boundedI*:

```

assumes  $\forall i < r. \text{finite } (S\ i)$ 
shows  $\text{finite } \{s. \text{length } s = r \wedge (\forall i < r. s\ !\ i \in S\ i)\}$  (is finite ?R)

```

**proof** –

```

let ?S =  $\bigcup \{S\ i \mid i. i < r\}$ 
have ?R  $\subseteq \{s. \text{set } s \subseteq ?S \wedge \text{length } s = r\}$ 
by (auto dest!: mem_nth)

```

moreover have *finite ...* by (rule *finite\_lists\_length\_eq*) (use *assms* in *auto*)  
 ultimately show *?thesis* by (rule *finite\_subset*)  
 qed

### 3.1 Syntax and Operational Semantics

We formalize Networks of Timed Automata with integer variable state using UPPAAL-style guards and updates. The specification language for guards and updates is our formalization of the UPPAAL like Assembler language. We extend Networks of Timed Automata with arbitrary shared (global) state. Syntactically, this extension is very simple. We can just use the free action label slot to annotate edges with a guard and an update function on discrete states. The slightly more clumsy part is adding invariants for discrete states by directly specifying an invariant annotating function.

**type\_synonym**

$('c, 'time, 's) \text{ invassn} = 's \Rightarrow ('c, 'time) \text{ cconstraint}$

**type\_synonym**

$('a, 's) \text{ transition} = 's * \text{addr} * 'a * \text{addr} * 's$

**type\_synonym**

$('a, 'c, 'time, 's) \text{ uta} = ('a, 's) \text{ transition set} * ('c, 'time, 's) \text{ invassn}$

**type\_synonym**

$('a, 'time, 's) \text{ unta} =$   
 $'time \text{ programc} \times ('a \text{ act, nat, 'time, 's) uta list} \times ('s \Rightarrow \text{addr}) \text{ list} \times (\text{int} * \text{int}) \text{ list}$

**definition**

$\text{bounded bounds } s \equiv$   
 $\text{length } s = \text{length bounds} \wedge (\forall i < \text{length } s. \text{fst}(\text{bounds } ! i) < s ! i \wedge s ! i < \text{snd}(\text{bounds } ! i))$

**inductive** *step\_u* ::

$('a, 't :: \text{time}, 's) \text{ unta} \Rightarrow \text{nat} \Rightarrow 's \text{ list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, 't) \text{ cval} \Rightarrow 'a \text{ label}$   
 $\Rightarrow 's \text{ list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, 't) \text{ cval} \Rightarrow \text{bool}$   
 $(\_ \vdash \_ \langle \_, \_, \_ \rangle \rightarrow \_ \langle \_, \_, \_ \rangle) [61, 61, 61, 61, 61, 61] \ 61)$

**where**

*step\_u t*:

$\llbracket$   
 $\forall p < \text{length } N. \exists pc \ st \ s' \ rs.$   
 $\text{stepst } P \ n \ (u \oplus d) ((I ! p) (L ! p), [], s, \text{True}, []) (pc, st, s', \text{True}, rs);$   
 $\forall p < \text{length } N. u \oplus d \vdash \text{snd} (N ! p) (L ! p);$   
 $d \geq 0;$   
 $\text{bounded } B \ s$   
 $\rrbracket$   
 $\Longrightarrow (P, N, I, B) \vdash_n \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L, s, u \oplus d \rangle \mid$

*step\_u i*:

$\llbracket$   
 $\text{stepst } P \ n \ u (pc\_g, [], s, \text{True}, []) (\_, \_, \_, \text{True}, \_);$   
 $\text{stepst } P \ n \ u (pc\_u, [], s, \text{True}, []) (\_, \_, s', \_, r);$   
 $\forall p < \text{length } N. \exists pc \ st \ s \ rs.$   
 $\text{stepst } P \ n \ u' ((I ! p) (L' ! p), [], s', \text{True}, []) (pc, st, s, \text{True}, rs);$   
 $(l, pc\_g, \text{Sil } a, pc\_u, l') \in \text{fst} (N ! p);$   
 $\forall p < \text{length } N. u' \vdash \text{snd} (N ! p) (L' ! p);$   
 $L.p = l; p < \text{length } L; L' = L[p := l']; u' = [r \rightarrow 0]u;$

$\text{bounded } B \ s'$   
 $\mathbb{I} \implies (P, N, I, B) \vdash_n \langle L, s, u \rangle \rightarrow_{Act \ a} \langle L', s', u' \rangle \mid$   
 $\text{step\_u\_s:}$   
 $\mathbb{I}$   
 $\text{stepst } P \ n \ u \ (pc\_g1, [], s, True, []) \ (\_, \_, \_, True, \_);$   
 $\text{stepst } P \ n \ u \ (pc\_g2, [], s, True, []) \ (\_, \_, \_, True, \_);$   
 $\text{stepst } P \ n \ u \ (pc\_u2, [], s, True, []) \ (\_, \_, s1, \_, r2);$   
 $\text{--- UPPAAL semantics quirk}$   
 $((\exists \ pc \ st \ s' \ f. \text{stepst } P \ n \ u \ (pc\_u1, [], s, True, []) \ (pc, st, s', f, r1))$   
 $\vee (\neg (\exists \ pc \ st \ s' \ f \ r'. \text{stepst } P \ n \ u \ (pc\_u1, [], s, True, []) \ (pc, st, s', f, r') \wedge r1 = []));$   
 $\text{stepst } P \ n \ u \ (pc\_u1, [], s1, True, []) \ (\_, \_, s', \_, \_);$   
 ~~$\text{stepst } P \ n \ u \ (pc\_u1, [], s1, True, []) \ (\_, \_, s', \_, \_);$~~   
 $\forall \ p < \text{length } N. \exists \ pc \ st \ s \ rs.$   
 $\text{stepst } P \ n \ u' \ ((I ! p) \ (L' ! p), [], s', True, []) \ (pc, st, s, True, rs);$   
 $(l1, pc\_g1, In \ a, pc\_u1, l1') \in \text{fst } (N ! p);$   
 $(l2, pc\_g2, Out \ a, pc\_u2, l2') \in \text{fst } (N ! q);$   
 $\forall \ p < \text{length } N. u' \vdash \text{snd } (N ! p) \ (L' ! p);$   
 $L!p = l1; L!q = l2; p < \text{length } L; q < \text{length } L; p \neq q;$   
 $L' = L[p := l1', q := l2']; u' = [(r1 @ r2) \rightarrow 0]u;$   
 $\text{bounded } B \ s'$   
 $\mathbb{I} \implies (P, N, I, B) \vdash_n \langle L, s, u \rangle \rightarrow_{Syn \ a} \langle L', s', u' \rangle$

**inductive\_cases**[elim!]:  $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$

**inductive** *steps\_un* ::

$(a, t :: \text{time}, s) \text{ unta} \Rightarrow \text{nat} \Rightarrow s \text{ list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, t) \text{ cval}$   
 $\Rightarrow s \text{ list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, t) \text{ cval} \Rightarrow \text{bool}$   
 $(\_ \vdash \_ \langle \_, \_, \_ \rangle \rightarrow^* \langle \_, \_, \_ \rangle [61, 61, 61, 61, 61, 61] \ 61)$

**where**

$\text{refl: } A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L, s, u \rangle \mid$   
 $\text{step: } A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \implies A \vdash_n \langle L', s', u' \rangle \rightarrow_a \langle L'', s'', u'' \rangle$   
 $\implies A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$

**declare** *steps\_un.intros*[intro]

**lemma** *stepI2*:

$A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$  **if**  
 $A \vdash_n \langle L', s', u' \rangle \rightarrow^* \langle L'', s'', u'' \rangle \ A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$   
**using** *that*  
**apply** *induction*  
**apply** (*rule steps\_un.step*)  
**apply** (*rule refl*)  
**apply** *assumption*  
**apply** *simp*  
**by** (*rule steps\_un.step; assumption*)

### 3.2 Equivalent State Network Automaton

**definition** *stripp*  $p \equiv \text{map\_option strip } o \ p$

**definition** *stripfp*  $p \equiv \text{map\_option stripf } o \ p$

**definition** *striptp*  $p \equiv \text{map\_option stript } o \ p$

**locale** *Equiv\_TA\_Defs* =

**fixes**  $A :: ('a, 't, 's) \text{ unta}$   
**and**  $n :: \text{nat} \text{ --- Fuel}$   
**begin**

**abbreviation**  $N \equiv \text{fst } (\text{snd } A)$   
**abbreviation**  $P \equiv \text{fst } A$   
**abbreviation**  $I \equiv \text{fst } (\text{snd } (\text{snd } A))$   
**abbreviation**  $B \equiv \text{snd } (\text{snd } (\text{snd } A))$   
**abbreviation**  $P' \equiv \text{stripfp } P$   
**abbreviation**  $PF \equiv \text{stripfp } P$   
**abbreviation**  $PT \equiv \text{striptp } P$   
**definition**  $p \equiv \text{length } N$

**definition**  $\text{make\_f } pc\_u \equiv \lambda s.$   
 $\text{case } (\text{exec } P' n (pc\_u, [], s, \text{True}, []) []) \text{ of}$   
 $\text{None} \Rightarrow []$   
 $| \text{Some } ((\_, \_, \_, \_, r), \_) \Rightarrow r$

**definition**  $\text{make\_mt } pc\_u \equiv \lambda s.$   
 $\text{case } (\text{exec } PT n (pc\_u, [], s, \text{True}, []) []) \text{ of}$   
 $\text{None} \Rightarrow \text{None}$   
 $| \text{Some } ((\_, \_, s', \_, r), \_) \Rightarrow \text{Some } s'$

**definition**  $\text{make\_mf } pc\_u \equiv \lambda s.$   
 $\text{case } (\text{exec } PF n (pc\_u, [], s, \text{True}, []) []) \text{ of}$   
 $\text{None} \Rightarrow \text{None}$   
 $| \text{Some } ((\_, \_, s', \_, r), \_) \Rightarrow \text{Some } s'$

**definition**  $\text{make\_c } pc\_g \equiv \lambda s.$   
 $\text{case } (\text{exec } PT n (pc\_g, [], s, \text{True}, []) []) \text{ of}$   
 $\text{None} \Rightarrow \text{False}$   
 $| \text{Some } ((\_, \_, \_, f, \_), \_) \Rightarrow f$

**definition**  $\text{make\_g } pc\_g \equiv \lambda s.$   
 $\text{case } (\text{exec } PT n (pc\_g, [], s, \text{True}, []) []) \text{ of}$   
 $\text{None} \Rightarrow []$   
 $| \text{Some } ((\_, \_, \_, \_, \_), pcs) \Rightarrow$   
 $\text{List.map\_filter } (\lambda pc.$   
 $\text{case } P pc \text{ of}$   
 $\text{Some } (\text{CEXP } ac) \Rightarrow \text{Some } ac$   
 $| \_ \Rightarrow \text{None}$   
 $)$   
 $pcs$

**definition**  
 $\text{state\_trans\_t } i \equiv$   
 $\{(l, \text{make\_g } pc\_g, (a, \text{make\_c } pc\_g, \text{make\_mf } pc\_u), \text{make\_f } pc\_u, l') \mid l a l' pc\_g pc\_u.$   
 $(l, pc\_g, a, pc\_u, l') \in \text{fst } (N ! i)$   
 $\}$

**abbreviation**  $state\_trans \equiv state\_trans\_t$

**definition**

$state\_pred\ i \equiv \lambda\ l\ s.$   
 $case\ (exec\ P'\ n\ ((I\ !\ i)\ l,\ [],\ s,\ True,\ [])\ [])\ of$   
 $None \Rightarrow False$   
 $| Some\ ((_,\ _,\ _,\ f,\ _),\ _) \Rightarrow f \wedge bounded\ B\ s$

**definition**

$state\_inv\ i \equiv snd\ (N\ !\ i)$

**definition**

$state\_ta \equiv (map\ (\lambda\ p.\ (state\_trans\ p,\ state\_inv\ p))\ [0..<p],\ map\ state\_pred\ [0..<p])$

**sublocale**  $defs: Prod\_TA\_Defs\ state\_ta\ .$

**lemma**  $bounded\_finite:$

$finite\ \{s.\ bounded\ B\ s\}\ (is\ finite\ ?S)$

**proof**  $-$

**have**

$?S \subseteq \{s.\ length\ s = length\ B \wedge (\forall\ i < length\ B.\ fst\ (B\ !\ i) < s\ !\ i \wedge s\ !\ i < snd\ (B\ !\ i))\}$

**unfolding**  $bounded\_def$  **by**  $auto$

**moreover have**  $finite\ \dots$  **unfolding**  $bounded\_def$  **using**  $finite\_lists\_boundedI$  **by**  $force$

**ultimately show**  $finite\ ?S$  **by**  $(rule\ finite\_subset)$

**qed**

**lemma**  $finite\_state:$

$\forall\ q < p.\ \forall\ l.\ finite\ \{s.\ (defs.P\ !\ q)\ l\ s\}$

**proof**  $safe$

**fix**  $q\ l$  **assume**  $\langle q < p \rangle$

**let**  $?S = \{s.\ (defs.P\ !\ q)\ l\ s\}$

**from**  $\langle q < p \rangle$  **have**  $?S \subseteq \{s.\ bounded\ B\ s\}$

**unfolding**  $state\_ta\_def\ state\_pred\_def$  **by**  $(auto\ split: option.splits)$

**moreover have**  $finite\ \dots$  **by**  $(rule\ bounded\_finite)$

**ultimately show**  $finite\ ?S$  **by**  $(rule\ finite\_subset)$

**qed**

**end**

**fun**  $is\_instr :: 't\ instrc \Rightarrow bool$  **where**

$is\_instr\ (INSTR\ \_) = True\ |$

$is\_instr\ \_ = False$

**lemma**  $step\_stripf:$

**assumes**

$is\_instr\ cmd$

**shows**

$stepc\ cmd\ u\ (pc,\ st,\ s,\ f,\ rs) = step\ (stripf\ cmd)\ (pc,\ st,\ s,\ f,\ rs)$

**proof**  $(cases\ cmd)$

```

    case (INSTR instr)
    with assms(1) show ?thesis
      by (cases instr) (auto split: option.split)
next
    case (CEXP x2)
    with assms show ?thesis by auto
qed

```

```

lemma step_strip:
  assumes
    is_instr cmd
  shows
    stepc cmd u (pc, st, s, f, rs) = step (strip cmd) (pc, st, s, f, rs)
proof (cases cmd)
  case (INSTR instr)
  with assms(1) show ?thesis
    by (cases instr) (auto split: option.split)
next
  case (CEXP x2)
  with assms show ?thesis by auto
qed

```

lemmas [intro] = stepsc.intros

```

lemma stepsc_f_complete:
  assumes
    stepsc P n' u start end
     $\bigwedge pc' st s' f' rs \text{ cmd.}$ 
    stepsc P n' u start (pc', st, s', f', rs)  $\implies P pc' = \text{Some cmd}$ 
 $\implies is\_instr \text{ cmd}$ 
  shows
    steps (stripfp P) n' start end
  using assms proof (induction P  $\equiv P n' u \equiv u x4 \equiv start \text{ end arbitrary: start rule: stepsc.induct}$ )
    case 1
    then show ?case unfolding stripfp_def by auto
  next
    case (2 cmd pc st m f rs s n' s')
    have is_instr cmd if
      stepsc P n' u s (pc', st, s', f', rs)  $P pc' = \text{Some cmd}$  for  $pc' st s' f' rs \text{ cmd}$ 
    using 2(1,2) that by (auto intro: 2(5))
    with 2(4) have *: steps (stripfp P) n' s s' by auto
    show ?case
  proof (cases cmd)
    case (INSTR instr)
    with 2(1) step_stripf have
      step (stripf cmd) (pc, st, m, f, rs) = Some s
    by (auto split: option.split_asm)
    with 2(1-3) 2(5-) * show ?thesis unfolding stripfp_def by auto
  next
    case (CEXP ac)
    with 2 show ?thesis by fastforce
  qed
qed

```

```

lemma stepsc_f_sound:
  assumes
    steps (stripfp P) n' start end
     $\bigwedge pc' st s' f' rs \text{ cmd.}$ 
     $\text{stepsc } P \text{ } n' \text{ } u \text{ } \text{start } (pc', st, s', f', rs) \implies P \text{ } pc' = \text{Some cmd}$ 
     $\implies \text{is\_instr cmd}$ 
  shows
     $\text{stepsc } P \text{ } n' \text{ } u \text{ } \text{start end}$ 
using assms proof (induction stripfp P n' start end)
  case (1 n start)
  then show ?case by auto
next
  case (2 instr pc st m f rs s n s')
  from 2(2) obtain cmd where  $P \text{ } pc = \text{Some cmd}$  unfolding stripfp_def by auto
  show ?case
  proof (cases cmd)
    case prems: (INSTR instr)
    with  $\langle P \text{ } pc = \_ \rangle$  2(2) step_stripfp[of cmd, symmetric] 2(1) have step:
       $\text{stepc cmd } u \text{ } (pc, st, m, f, rs) = \text{Some } s$ 
      unfolding stripfp_def by auto
    with  $\langle P \text{ } pc = \_ \rangle$  have  $\text{is\_instr cmd}$  if
       $\text{stepsc } P \text{ } n \text{ } u \text{ } s \text{ } (pc', st, s', f', rs) \text{ } P \text{ } pc' = \text{Some cmd}$  for  $pc' st s' f' rs \text{ cmd}$ 
      using that unfolding stripfp_def by (force intro: 2(5))
    with 2(4) have  $\text{stepsc } P \text{ } n \text{ } u \text{ } s \text{ } s'$  by auto
    with step  $\langle P \text{ } pc = \_ \rangle$  show ?thesis unfolding stripfp_def by auto
  next
    case prems: (CEXP ac)
    from  $\langle P \text{ } pc = \_ \rangle$  2(5) have  $\text{is\_instr cmd}$  by blast
    with prems show ?thesis by auto
  qed
qed

```

**definition**

```

time_indep P n start  $\equiv$ 
   $\forall pc' st s' f' rs \text{ cmd } u.$ 
     $\text{stepsc } P \text{ } n \text{ } u \text{ } \text{start } (pc', st, s', f', rs) \wedge P \text{ } pc' = \text{Some cmd}$ 
     $\longrightarrow \text{is\_instr cmd}$ 

```

**lemma** *stepsc\_t\_complete*:

```

assumes
   $\text{stepsc } P \text{ } n' \text{ } u \text{ } \text{start end}$ 
   $\bigwedge pc' st s' f' rs \text{ ac.}$ 
   $\text{stepsc } P \text{ } n' \text{ } u \text{ } \text{start } (pc', st, s', f', rs) \implies P \text{ } pc' = \text{Some } (\text{CEXP } ac) \implies u \vdash_a ac$ 
shows
   $\text{steps (striptp P) } n' \text{ start end}$ 
using assms proof (induction P  $\equiv$  P n' u  $\equiv$  u x4  $\equiv$  start end arbitrary: start rule: stepsc.induct)
  case 1
  then show ?case unfolding stripfp_def by auto
next
  case (2 cmd pc st m f rs s n' s')
  have  $u \vdash_a ac$  if
     $\text{stepsc } P \text{ } n' \text{ } u \text{ } s \text{ } (pc', st, s', f', rs) \text{ } P \text{ } pc' = \text{Some } (\text{CEXP } ac)$  for  $pc' st s' f' rs \text{ ac}$ 
    using 2(1,2) that by (auto intro: 2(5))

```

```

with 2(4) have *: steps (striptp P) n' s s' by auto
show ?case
proof (cases cmd)
  case (INSTR instr)
  with 2(1) step_stript have
    step (stript cmd) (pc, st, m, f, rs) = Some s
    by (auto split: option.split_asm)
  with 2(1-3) 2(5-) * show ?thesis unfolding striptp_def by auto
next
case (CEXP ac)
with 2(1-3) have u ⊢a ac by (auto intro: 2(5))
with 2(1) have
  step (stript cmd) (pc, st, m, f, rs) = Some s
  using ⟨cmd = _⟩ by auto
with 2(1-3) 2(5-) * show ?thesis unfolding striptp_def by auto
qed
qed

lemma stepsc_t_complete2:
  assumes
    stepsc P n' u start (pc', st', s', f', rs')
    ∧ pc' st s' f' rs ac.
  shows
    stepsc P n' u start (pc', st, s', f', rs) ⇒ P pc' = Some (CEXP ac) ⇒ u ⊢a ac
  using assms
  proof (induction P ≡ P n' u ≡ u x4 ≡ start (pc', st', s', f', rs') arbitrary: start rule:
    stepsc.induct)
    case 1
    then show ?case unfolding stripfp_def by blast
  next
  case (2 cmd pc st m f rs s n')
  have u ⊢a ac if
    stepsc P n' u s (pc', st, s', f', rs) P pc' = Some (CEXP ac) for pc' st s' f' rs ac
    using 2(1,2) that by (auto intro: 2(5))
  with 2(4) have *:
    steps (striptp P) n' s (pc', st', s', f', rs') ∀ ac. P pc' = Some (CEXP ac) ⇒ u ⊢a ac
    by auto
  show ?case
  proof (cases cmd)
    case (INSTR instr)
    with 2(1) step_stript have
      step (stript cmd) (pc, st, m, f, rs) = Some s
      by (auto split: option.split_asm)
    with 2(1-3) 2(5-) * show ?thesis unfolding striptp_def by auto
  next
  case (CEXP ac)
  with 2(1-3) have u ⊢a ac by (auto intro: 2(5))
  with 2(1) have
    step (stript cmd) (pc, st, m, f, rs) = Some s
    using ⟨cmd = _⟩ by auto
  with 2(1-3) 2(5-) * show ?thesis unfolding striptp_def by auto
  qed
qed

```

**lemma** *stepsc\_t\_visitedc*:

**assumes**

*stepsc P n' u start end*

$\bigwedge pc' st s' f' rs.$

*stepsc P n' u start (pc', st, s', f', rs)  $\implies$  Q pc'*

**shows**  $\exists pcs. visitedc P n' u start end pcs$

$\wedge (\forall pc \in set pcs. Q pc)$

**using** *assms* **by** (*induction*) (*fastforce intro!*: *visitedc.intros*)**+**

**lemma** *visitedc\_t\_visited*:

**assumes**

*visitedc P n' u start end pcs*

$\bigwedge pc' ac. pc' \in set pcs \implies P pc' = Some (CEXP ac) \implies u \vdash_a ac$

**shows**

*visited (striptp P) n' start end pcs*

$\wedge (\forall pc ac. pc' \in set pcs \wedge P pc' = Some (CEXP ac) \longrightarrow u \vdash_a ac)$

**using** *assms*

**proof** (*induction P  $\equiv$  P n' u  $\equiv$  u x4  $\equiv$  start end pcs arbitrary: start rule: visitedc.induct*)

**case** 1

**then show** ?case **by** (*auto intro: visited.intros*)

**next**

**case** (2 *cmd pc st m f rs s n' s' pcs*)

**have**  $u \vdash_a ac$  **if**

$pc' \in set pcs \wedge P pc' = Some (CEXP ac)$  **for**  $pc' ac$

**using** 2(1,2) **that** **by** (*auto intro: 2(5)*)

**with** 2(4) **have** \*:

*visited (striptp P) n' s s' pcs  $\forall pc ac. pc' \in set pcs \wedge P pc' = Some (CEXP ac) \longrightarrow u \vdash_a ac$*

**by** *auto*

**show** ?case

**proof** (*cases cmd*)

**case** (*INSTR instr*)

**with** 2(1) *step\_stript* **have**

*step (stript cmd) (pc, st, m, f, rs) = Some s*

**by** (*auto split: option.split\_asm*)

**with** 2(1-3) 2(5-) \* **show** ?thesis **unfolding** *striptp\_def* **by** (*auto intro: visited.intros*)

**next**

**case** (*CEXP ac*)

**with** 2(1-3) **have**  $u \vdash_a ac$  **by** (*auto intro: 2(5)*)

**with** 2(1) **have**

*step (stript cmd) (pc, st, m, f, rs) = Some s*

**using**  $\langle cmd = \_ \rangle$  **by** *auto*

**with** 2(1-3) 2(5-) \* **show** ?thesis **unfolding** *striptp\_def* **by** (*auto intro: visited.intros*)

**qed**

**qed**

**lemma** *stepsc\_t\_sound*:

**assumes**

*steps (striptp P) n' start end*

$\bigwedge pc' st s' f' rs ac.$

*stepsc P n' u start (pc', st, s', f', rs)  $\implies$  P pc' = Some (CEXP ac)  $\implies$  u  $\vdash_a$  ac*

**shows**

*stepsc P n' u start end*

**using** *assms* **proof** (*induction striptp P n' start end*)

```

case (1 n start)
then show ?case by auto
next
case (2 instr pc st m f rs s n s')
from 2(2) obtain cmd where P pc = Some cmd unfolding striptp_def by auto
show ?case
proof (cases cmd)
case prems: (INSTR instr)
with ⟨P pc = _⟩ 2(2) step_stript[of cmd, symmetric] 2(1) have step:
  stepc cmd u (pc, st, m, f, rs) = Some s
  unfolding striptp_def by auto
with ⟨P pc = _⟩ have u ⊢a ac if
  stepsc P n u s (pc', st, s', f', rs) P pc' = Some (CEXP ac) for pc' st s' f' rs ac
  using that unfolding striptp_def by (force intro: 2(5))
with 2(4) have stepsc P n u s s' by auto
with step ⟨P pc = _⟩ show ?thesis unfolding striptp_def by auto
next
case prems: (CEXP ac)
with ⟨P pc = _⟩ 2(2) have [simp]: P pc = Some (CEXP ac) by auto
then have u ⊢a ac by (auto intro: 2(5))
with 2(2,1) ⟨cmd = _⟩ have step:
  stepc cmd u (pc, st, m, f, rs) = Some s
  unfolding striptp_def by auto
with ⟨P pc = Some cmd⟩ have u ⊢a ac if
  stepsc P n u s (pc', st, s', f', rs) P pc' = Some (CEXP ac) for pc' st s' f' rs ac
  using that unfolding striptp_def by (force intro: 2(5))
with 2(4) have stepsc P n u s s' by auto
with step ⟨P pc = Some cmd⟩ show ?thesis unfolding striptp_def by auto
qed
qed

```

**lemma** *stepsc\_visitedc*:  
 $\exists$  cc. *visitedc* P n u start end cc **if** *stepsc* P n u start end  
**using** that **by** *induction* (auto intro: *visitedc.intros*)

**lemma** *visitedc\_stepsc*:  
*stepsc* P n u start end **if** *visitedc* P n u start end cc  
**using** that **by** (*induction*; *blast*)

**lemma** *steps\_visited*:  
 $\exists$  cc. *visited* P n start end cc **if** *steps* P n start end  
**using** that **by** *induction* (auto intro: *visited.intros*)

**lemma** *visited\_steps*:  
*steps* P n start end **if** *visited* P n start end cc  
**using** that **by** (*induction*; *blast*)

**context**  
**fixes** P n u start  
**assumes** *constraints\_conj*:  $\forall$  pc' st s' f' rs ac.  
*stepsc* P n u start (pc', st, s', f', rs)  $\wedge$  P pc' = Some (CEXP ac)  $\longrightarrow$  u ⊢<sub>a</sub> ac  
**begin**

**lemma** *stepsc\_t\_sound'*:

```

assumes
  steps (striptp P) n start end
shows
  stepsc P n u start end
using assms constraints_conj by (auto intro: stepsc_t_sound)

lemma stepsc_t_complete':
assumes
  stepsc P n u start end
shows
  steps (striptp P) n start end
using assms constraints_conj by (auto intro: stepsc_t_complete)

lemma stepsc_t_complete'':
assumes
  stepsc P n u start end
shows
   $\exists pcs. \text{visitedc } P \ n \ u \ \text{start end } pcs \wedge (\forall pc \in \text{set } pcs. \forall ac. P \ pc = \text{Some } (CEXP \ ac) \longrightarrow u \vdash_a ac)$ 
using assms constraints_conj by (auto intro: stepsc_t_complete stepsc_t_visitedc)

lemma stepsc_t_visited:
assumes
  stepsc P n u start end
shows
   $\exists pcs. \text{visited } (striptp \ P) \ n \ \text{start end } pcs \wedge (\forall pc \in \text{set } pcs. \forall ac. P \ pc = \text{Some } (CEXP \ ac) \longrightarrow u \vdash_a ac)$ 
using stepsc_t_complete'' [OF assms] visitedc_t_visited by blast

lemma stepst_t_complete:
assumes
  stepst P n u start end
shows
   $\exists pcs. \text{exec } (striptp \ P) \ n \ \text{start } [] = \text{Some } (end, pcs) \wedge (\forall pc \in \text{set } pcs. \forall ac. P \ pc = \text{Some } (CEXP \ ac) \longrightarrow u \vdash_a ac)$ 
using assms by (auto dest!: stepsc_t_visited visited_exec' simp: striptp_def stepst_def)

lemma stepst_t_equiv:
   $(\exists pcs'. \text{exec } (striptp \ P) \ n \ \text{start } pcs = \text{Some } ((pc, st, m, f, rs), pcs'))$ 
 $\longleftrightarrow \text{stepst } P \ n \ u \ \text{start } (pc, st, m, f, rs)$ 
apply safe
apply (drule exec_steps)
unfolding stepst_def
apply safe
apply (rule stepsc_t_sound'; assumption)
apply (simp add: striptp_def)
apply safe
subgoal for z
by (cases z) auto
by (auto dest!: stepsc_t_complete' steps_exec simp: striptp_def)

end

context

```

```

fixes P n start
assumes time_indep: time_indep P n start
begin

lemma time_indep':
   $\bigwedge pc' st s' f' rs cmd.$ 
   $stepsc P n u start (pc', st, s', f', rs) \implies P pc' = Some cmd$ 
 $\implies is\_instr cmd$ 
  using time_indep unfolding time_indep_def by blast

lemma stepsc_f_complete':
  assumes
    stepsc P n u start end
  shows
    steps (stripfp P) n start end
  using assms time_indep' by (auto intro: stepsc_f_complete[where P = P])

lemma stepsc_f_sound':
  assumes
    steps (stripfp P) n start end
  shows
    stepsc P n u start end
  using assms time_indep' by (auto intro: stepsc_f_sound[where P = P])

lemma stepsc_f_equiv:
  steps (stripfp P) n start end  $\longleftrightarrow$  stepsc P n u start end
  using stepsc_f_sound' stepsc_f_complete' by fast

lemma stepst_f_equiv:
   $(\exists pcs'. exec (stripfp P) n start pcs = Some ((pc, st, m, f, rs), pcs'))$ 
 $\longleftrightarrow$  stepst P n u start (pc, st, m, f, rs)
  apply safe
  apply (drule exec_steps)
  unfolding stepst_def
  apply safe
  apply (rule stepsc_f_sound'; assumption)
  apply (simp add: stripfp_def)
  apply safe
  subgoal for z
    by (cases z) auto
  by (auto dest!: stepsc_f_complete' steps_exec simp: stripfp_def)

end

lemma exec_acc:
  assumes exec P n s pcs = Some (s', pcs')
  shows  $\exists pcs''. pcs' = pcs'' @ pcs$ 
  using assms by (induction P n s pcs rule: exec.induct; force split: option.split_asm if_split_asm)

lemma exec_acc':
  assumes Some (s', pcs') = exec P n s pcs
  shows  $\exists pcs''. pcs' = pcs'' @ pcs$ 
  using assms
  using assms exec_acc by metis

```

```

lemma exec_min_steps:
  assumes exec P n s pcs = Some (s', pcs' @ pcs)
  shows exec P (length pcs') s pcs = Some (s', pcs' @ pcs)
using assms proof (induction n arbitrary: s pcs s' pcs')
  case 0
  then show ?case by auto
next
  case (Suc n)
  obtain pc st m f rs pc' st' m' f' rs' where [simp]:
    s = (pc, st, m, f, rs) s' = (pc', st', m', f', rs')
  using prod.exhaust by metis
  from Suc obtain instr where P pc = Some instr by (auto split: option.splits if_splits)
  show ?case
  proof (cases instr = HALT)
    case True
    with ⟨P pc = ⟩ Suc show ?thesis by auto
  next
    case False
    with Suc.prem1 ⟨P pc = ⟩ obtain s'' where s'':
      step instr s = Some s'' exec P n s'' (pc # pcs) = Some (s', pcs' @ pcs)
    by (auto split: option.splits)
    with exec_acc[OF this(2)] obtain pcs'' where pcs' = pcs'' @ [pc] by auto
    with Suc.IH[of s'' pc # pcs s' pcs'] ⟨P pc = ⟩ False s'' show ?thesis by auto
  qed
qed

lemma exec_steps_visited:
  assumes
    exec P (length pcs') s pcs = Some (s', pcs' @ pcs)
    steps P (length pcs') s (pc, st, m, f, rs)
  shows pc ∈ set pcs'
  using assms proof (induction P ≡ P length pcs' s pcs arbitrary: pc st m f rs pcs' rule:
    exec.induct)
    case 1
    then show ?case by simp
  next
    case (2 n pc' st' m' f' rs' pcs pcs')
    from this(2)[symmetric] this(3) obtain instr where P pc' = Some instr by (cases P pc') auto
    show ?case
    proof (cases instr = HALT)
      case True
      with 2.prem1 ⟨P pc' = ⟩ ⟨Suc n = ⟩[symmetric] show ?thesis by (force elim: steps.cases)
    next
      case False
      with 2 obtain pcs'' where pcs' = pcs'' @ [pc]
      apply atomize_elim
      by (erule exec.elims) (auto dest: exec_acc' split: option.split_asm if_split_asm)
      with False 2 ⟨P pc' = ⟩ ⟨Suc n = ⟩[symmetric] show ?thesis
      by (auto split: option.split_asm elim: steps.cases)
    qed
  qed
qed

lemma stepsc_mono:

```

```

assumes stepsc P n u start end n' ≥ n
shows stepsc P n' u start end
using assms proof (induction arbitrary: n')
case (1 prog n u start)
then show ?case by (cases n') auto
next
case (2 cmd u pc st m f rs s prog n s')
then show ?case by (cases n') auto
qed

```

```

lemma stepst_mono:
assumes stepst P n u start end n' ≥ n
shows stepst P n' u start end
using assms stepsc_mono unfolding stepst_def by blast

```

**definition**

```

state_indep P n ≡
  ∀ pc f pc' st f' rs u s1 s1' s2 s2' rs1 rs2.
    stepsc P n u (pc, st, s1, f, rs) (pc', st, s1', f', rs1) ∧
    stepsc P n u (pc, st, s2, f, rs) (pc', st, s2', f', rs2)
  → rs1 = rs2

```

```

lemma exec_len:
n ≥ (length pcs' - length pcs) if exec P n s pcs = Some (s', pcs')
using that
by (induction P n s pcs arbitrary: rule: exec.induct)
(force split: option.split_asm if_split_asm)+

```

**lemma steps\_stripty\_steps:**

```

assumes
  ∧ pc st m f rs ac.
  steps (stripty P) n s' (pc, st, m, f, rs) ⇒ P pc = Some (CEXP ac)
  ⇒ u' ⊢a ac

  and steps (stripty P) n s' s''
shows stepsc P n u' s' s''
using assms(2,1)
proof (induction stripty P n s' s'')
case (1 n start)
show ?case by (rule stepsc.intros)
next
case (2 cmd pc st m f rs s n s')
have u' ⊢a ac
if P pc = Some (CEXP ac) UPPAAL_Asm.steps (stripty P) n s (pc, st, m, f, rs)
for pc st m f rs ac
using that 2(1-3) by - (rule 2(5); force)
with 2(4) have stepsc P n u' s s' by auto
with 2(1-3) show ?case
apply (cases P pc)
apply (simp add: stripty_def)
subgoal for cmd'
apply (cases cmd')
subgoal
by (force split: option.split simp: stripty_def)

```

```

    subgoal
      by (rule stepsc.intros) (auto intro!: 2(5) simp: striptp_def)
    done
  done
qed

locale Equiv_TA =
  Equiv_TA_Defs A n for A :: ('a, 't :: time, 's) unta and n :: nat +
  fixes L :: 's list and s :: int list
  assumes states[intro]: L ∈ defs.states' s

  and pred_time_indep:
    ∀ s. ∀ L ∈ defs.states' s. ∀ q < p. time_indep P n ((I ! q) (L ! q), [], s, True, [])
  and upd_time_indep:
    ∀ l pc_g a l' pc_u s. ∀ q < p. (l, pc_g, a, pc_u, l') ∈ fst (N ! q)
    → time_indep P n (pc_u, [], s, True, [])
  and clock_conj:
    ∀ l pc_g a l' pc_u s u. ∀ q < p. (l, pc_g, a, pc_u, l') ∈ fst (N ! q) ∧
    (∃ pc' st s' rs. stepst P n u (pc_g, [], s, True, []) (pc', st, s', True, rs)) →
    (∀ pc' st s' f' rs ac.
      stepsc P n u (pc_g, [], s, True, []) (pc', st, s', f', rs) ∧ P pc' = Some (CEXP ac) →
      u ⊢a ac)

  assumes Len: length N = length I
  and inv: ∀ q < p. ∃ pc st s' rs pcs.
    exec P' n ((I ! q) (L ! q), [], s, True, []) [] = Some ((pc, st, s', True, rs), pcs)
  and bounded: bounded B s
begin

lemma length_defs_N[simp]:
  length defs.N = p
  unfolding p_def state_ta_def by simp

lemma length_defs_P[simp]:
  length defs.P = p
  unfolding p_def state_ta_def by simp

lemma inv':
  ∀ p < length defs.P. (defs.P ! p) (L ! p) s
  using inv bounded unfolding state_ta_def state_pred_def by (force simp: p_def)

lemma inv'':
  ∃ pc st s' rs. stepst P n u' ((I ! q) (L ! q), [], s, True, []) (pc, st, s', True, rs)
  if q < p for u'
proof –
  from pred_time_indep that have time_indep P n ((I ! q) (L ! q), [], s, True, []) by blast
  with inv that stepst_f_equiv[symmetric] show ?thesis by blast
qed

lemma A_simp[simp]:
  PP = P N' = N I' = I B' = B if A = (PP, N', I', B')
using that by auto

```

**lemma** *A\_unfold*:

$A = (P, N, I, B)$

**by** *simp*

**sublocale** *prod*: *Prod\_TA state\_ta* **by** *standard* (*auto simp: inv'*)

**lemma** *inv\_simp*:

$\text{snd } (defs.N ! q) (L' ! q) = \text{snd } (N ! q) (L' ! q)$  **if**  $q < p$  **for**  $L'$

**using** *that unfolding state\_ta\_def state\_inv\_def* **by** *simp*

**lemma** *defs\_p\_eq[simp]*:

$defs.p = p$

**by** (*simp add: defs.p\_def p\_def*)

**lemma** *ball\_lessThan[simp]*:

$(\forall x \in \{.. $m\}. Q x) \longleftrightarrow (\forall x < m. Q x)$$

**by** *auto*

**lemma** *trans\_state\_tD*:

**assumes**  $(l, g, (a, c, m), f, l') \in \text{fst } (defs.N ! q) \ q < p$

**shows**

$(l, g, (a, c, m), f, l') \in \text{state\_trans\_t } q$

**using** *assms unfolding state\_ta\_def* **by** *simp*

**lemma** *N\_transD*:

**assumes**  $(l, pc\_g, a, pc\_u, l') \in \text{fst } (N ! q) \ q < p$

**shows**  $(l, \text{make\_g } pc\_g, (a, \text{make\_c } pc\_g, \text{make\_mf } pc\_u), \text{make\_f } pc\_u, l') \in \text{fst } (defs.N ! q)$

**using** *assms unfolding state\_ta\_def state\_trans\_t\_def* **by** *auto*

**lemma** *pred\_time\_indep'*:

$\forall L' s' u'. \forall p' < p. A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$

$\rightarrow \text{time\_indep } P \ n \ ((I ! p') (L' ! p'), [], s', \text{True}, [])$

**using** *pred\_time\_indep oops*

**lemma** *P\_steps\_upd*:

**assumes**

$\text{Some } s'' = \text{make\_mf } pc\_u \ s' (l, pc\_g, a, pc\_u, l') \in \text{fst } (N ! q) \ q < p$

**shows**

$\exists pc \ st \ f. \text{stepst } P \ n \ u' (pc\_u, [], s', \text{True}, []) (pc, st, s'', f, \text{make\_f } pc\_u \ s')$

**proof** –

**from** *assms upd\_time\_indep* **have**  $*$ :  $\text{time\_indep } P \ n \ (pc\_u, [], s', \text{True}, [])$  **by** *auto*

**from** *assms(1) <math>q < p</math>* **obtain**  $pc \ st \ f \ rs \ pcs$  **where**

$\text{exec } PF \ n \ (pc\_u, [], s', \text{True}, []) [] = \text{Some } ((pc, st, s'', f, rs), pcs)$

**unfolding** *make\_mf\_def state\_ta\_def* **by** (*fastforce split: option.splits*)

**with** *stepst\_f\_equiv[OF \*]* **show** *?thesis* **unfolding** *make\_f\_def* **by** *fastforce*

**qed**

**lemma** *P\_steps\_reset*:

**assumes**

$q < p \ (l, pc\_g, a, pc\_u, l') \in \text{fst } (N ! q)$

**shows**

$(\exists pc \ st \ s'' \ f. \text{stepst } P \ n \ u (pc\_u, [], s', \text{True}, []) (pc, st, s'', f, \text{make\_f } pc\_u \ s')) \vee$

$(\nexists pc \ st \ s'' \ f \ r'. \text{stepst } P \ n \ u (pc\_u, [], s', \text{True}, []) (pc, st, s'', f, r')) \wedge$

```

      make_f pc_u s' = []
proof (cases  $\exists pc\ st\ s''\ f\ r'.\ stepst\ P\ n\ u\ (pc\_u, [], s', True, [])\ (pc, st, s'', f, r')$ )
  case True
  then obtain  $pc\ st\ s''\ f\ r'$  where *:
     $stepst\ P\ n\ u\ (pc\_u, [], s', True, [])\ (pc, st, s'', f, r')$ 
  by blast
  from assms upd_time_indep have  $time\_indep\ P\ n\ (pc\_u, [], s', True, [])$  by auto
  from *  $stepst\_f\_equiv[OF\ this, symmetric, \text{where } pcs = []]$  show ?thesis
  unfolding make_f_def by (force split: option.split)
next
  case False
  from assms upd_time_indep have  $time\_indep\ P\ n\ (pc\_u, [], s', True, [])$  by auto
  from False  $stepst\_f\_equiv[OF\ this, \text{where } pcs = []]$  show ?thesis
  unfolding make_f_def by (auto split: option.split)
qed

lemma steps_P_reset:
  assumes
     $(\exists pc\ st\ s''\ f.\ stepst\ P\ n\ u\ (pc\_u, [], s', True, [])\ (pc, st, s'', f, r)) \vee$ 
     $(\nexists pc\ st\ s''\ f\ r'.\ stepst\ P\ n\ u\ (pc\_u, [], s', True, [])\ (pc, st, s'', f, r')) \wedge r = []$ 
     $q < p\ (l, pc\_g, a, pc\_u, l') \in fst\ (N\ !\ q)$ 
  shows  $make\_f\ pc\_u\ s' = r$ 
  using assms(1)
proof (safe, goal_cases)
  case prems: (1 pc st s'' f)
  from assms upd_time_indep have  $time\_indep\ P\ n\ (pc\_u, [], s', True, [])$  by auto
  from  $stepst\_f\_equiv[OF\ this, symmetric]$  prems  $\langle q < p \rangle$  obtain  $pc\ st\ f\ pcs$  where
     $exec\ PF\ n\ (pc\_u, [], s', True, [])\ [] = Some\ ((pc, st, s'', f, r), pcs)$ 
  unfolding make_f_def state_ta_def by (fastforce split: option.splits)
  then show ?case unfolding make_f_def by (auto split: option.split)
next
  case prems: 2
  have  $exec\ PF\ n\ (pc\_u, [], s', True, [])\ [] = None$ 
  proof (cases exec PF n (pc_u, [], s', True, []) [])
  case None
  then show ?thesis .
next
  case (Some a)
  obtain  $pc''\ st''\ s''\ f''\ rs''\ pcs''$  where  $a = ((pc'', st'', s'', f'', rs''), pcs'')$ 
  by (cases a) auto
  from assms upd_time_indep have  $time\_indep\ P\ n\ (pc\_u, [], s', True, [])$  by auto
  from  $stepst\_f\_equiv[OF\ this]$   $\langle \_ = Some\ a \rangle$  prems  $\langle a = \_ \rangle$  show ?thesis by auto metis
qed
  then show ?case unfolding make_f_def by simp
qed

```

```

lemma steps_P_upd:
  assumes
     $stepst\ P\ n\ u'\ (pc\_u, [], s', True, [])\ (pc, st, s'', f, r)$ 
     $q < p\ (l, pc\_g, a, pc\_u, l') \in fst\ (N\ !\ q)$ 
  shows
     $Some\ s'' = make\_mf\ pc\_u\ s'\ (is\ ?A)\ r = make\_f\ pc\_u\ s'\ (is\ ?B)$ 
proof –

```

**from** *assms upd time\_indep* **have** *time\_indep P n (pc\_u, [], s', True, [])* **by** *auto*  
**from** *stepst\_f\_equiv[OF this, symmetric]* *assms(1) ⟨q < p⟩* **obtain** *pc st f pcs* **where**  
*exec PF n (pc\_u, [], s', True, []) [] = Some ((pc, st, s'', f, r), pcs)*  
**unfolding** *make\_mf\_def state\_ta\_def* **by** (*fastforce split: option.splits*)  
**then show** *?A ?B* **unfolding** *make\_f\_def make\_mf\_def* **by** (*auto split: option.split*)  
**qed**

**lemma** *steps\_P\_guard*:

**assumes**

*stepst P n u' (pc\_g, [], s', True, []) (pc, st, s'', True, rs)*

*q < p (l, pc\_g, a, pc\_u, l') ∈ fst (N ! q)*

**shows**

*make\_c pc\_g s' (is ?A) u' ⊢ make\_g pc\_g s' (is ?B)*

**proof** –

**from** *stepst\_t\_complete[OF \_\_ assms(1)] clock\_conj assms* **obtain** *pcs* **where**

*exec PT n (pc\_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs)*

*∀ pc ∈ set pcs. ∀ ac. P pc = Some (CEXP ac) ⟶ u' ⊢<sub>a</sub> ac*

**by** *fastforce*

**then show** *?A ?B* **unfolding** *make\_c\_def make\_g\_def*

**by** (*auto split: option.split instrc.split\_asm simp: list\_all\_iff set\_map\_filter clock\_val\_def*)

**qed**

**lemma** *P\_steps\_guard*:

**assumes**

*make\_c pc\_g s' u' ⊢ make\_g pc\_g s'*

*q < p (l, pc\_g, a, pc\_u, l') ∈ fst (N ! q)*

**shows**

*∃ pc s'' st rs. stepst P n u' (pc\_g, [], s', True, []) (pc, st, s'', True, rs)*

**proof** –

**from** *assms(1) ⟨q < p⟩* **obtain** *pc st s'' rs pcs* **where** \*:

*exec PT n (pc\_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs)*

**unfolding** *make\_c\_def state\_ta\_def* **by** (*fastforce split: option.splits*)

**with** *exec\_min\_steps[of PT n \_ [] \_ pcs]* **have** \*\*:

*exec PT (length pcs) (pc\_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs @ [])*

**by** *auto*

**from** \* *assms(2)* **have**

*u' ⊢ List.map\_filter (λpc. case P pc of*

*None ⇒ None*

*| Some (INSTR xa) ⇒ Map.empty xa*

*| Some (CEXP xa) ⇒ Some xa) pcs* **unfolding** *make\_g\_def*

**by** *auto*

**moreover from** \*\* *exec\_steps\_visited[of PT pcs \_ []]* **have** *pc ∈ set pcs*

**if** *steps PT (length pcs) (pc\_g, [], s', True, []) (pc, st, m, f, rs)* **for** *pc st m f rs*

**using** *that* **by** *fastforce*

**ultimately have** *u' ⊢<sub>a</sub> ac*

**if** *steps PT (length pcs) (pc\_g, [], s', True, []) (pc, st, m, f, rs)* *P pc = Some (CEXP ac)*

**for** *pc st m f rs ac*

**using** *that* **by** (*auto 4 3 split: option.splits simp: list\_all\_iff set\_map\_filter clock\_val\_def*)

**moreover from** \*\* **have**

*steps PT (length pcs) (pc\_g, [], s', True, []) (pc, st, s'', True, rs) PT pc = Some HALT*

**by** (*auto dest: exec\_steps*)

**ultimately have** *stepst P (length pcs) u' (pc\_g, [], s', True, []) (pc, st, s'', True, rs)*

**by** (*auto intro: steps\_stripty\_stepsc simp: stepst\_def stripty\_def elim: stripty.elims*)

**moreover from** *exec\_len[OF \*]* **have** *n ≥ length pcs* **by** *simp*

ultimately show *?thesis* by (blast intro: stepst\_mono)  
qed

lemma *P\_bounded*:  
assumes  
   $(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s' \ q < p$   
shows *bounded B s'*  
using *assms unfolding state\_pred\_def state\_ta\_def* by (auto split: option.splits)

lemma *P\_steps*:  
assumes  
   $(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s'$   
   $q < p \ L' \in \text{defs.states}' \ s'$   
shows  
   $\exists \ pc \ st \ s'' \ rs. \text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs)$   
proof –  
  from *assms pred\_time\_indep* have \*: *time\_indep P n ((I ! q) (L' ! q), [], s', True, [])* by auto  
  from *assms(1) <q < p>* obtain *pc st s'' rs pcs* where  
     $\text{exec } PF \ n \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ [] = \text{Some} \ ((pc, \ st, \ s'', \ \text{True}, \ rs), \ pcs)$   
    *unfolding state\_pred\_def state\_ta\_def* by (auto split: option.splits)  
  with *stepst\_f\_equiv[OF \*]* show *?thesis* by blast  
qed

lemma *steps\_P*:  
assumes  
   $\text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs)$   
   $q < p \ L' \in \text{defs.states}' \ s'$   
  *bounded B s'*  
shows  
   $(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s'$   
proof –  
  from *assms pred\_time\_indep* have *time\_indep P n ((I ! q) (L' ! q), [], s', True, [])* by auto  
  from *stepst\_f\_equiv[OF this] assms(1)* obtain *pcs'* where  
     $\text{exec } PF \ n \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ [] = \text{Some} \ ((pc, \ st, \ s'', \ \text{True}, \ rs), \ pcs')$   
    by blast  
  with  $\langle q < p \rangle \langle \text{bounded } B \ s' \rangle$  show *?thesis* *unfolding state\_pred\_def state\_ta\_def* by simp  
qed

lemma *P\_iff*:  
   $(\exists \ pc \ st \ rs \ s''. \text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs) \wedge \text{bounded } B \ s')$   
   $\longleftrightarrow (\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s' \text{ if } q < p \ L' \in \text{defs.states}' \ s'$   
  using *that* by (metis *steps\_P P\_steps P\_bounded*)

lemma *states'\_updI'*:  
assumes  $(L' \ ! \ q, \ g, \ (a, \ c, \ m), \ f, \ l') \in \text{fst} \ (\text{defs}.N \ ! \ q) \ L' \in \text{defs.states}' \ s''$   
shows  $L'[q := l'] \in \text{defs.states}' \ s'$   
using *assms*  
  *unfolding prod.states'\_simp[of s' s'']*  
  *unfolding Product\_TA\_Defs.states\_def*  
  *apply clarsimp*  
  subgoal for *p*  
    by (cases  $p = q$ ; force simp: *prod.trans\_of\_N\_s\_2[simplified] Prod\_TA\_Defs.N\_s\_length*)  
done

```

lemma states'_updI:
  assumes  $(L \mid q, g, (a, c, m), f, l') \in \text{fst } (\text{defs}.N \mid q)$ 
  shows  $L[q := l'] \in \text{defs}.states' s'$ 
using assms by (auto intro: states'_updI')

lemma states'_updI'':
  assumes
     $(L' \mid q, g, (a, c, m), f, l') \in \text{fst } (\text{defs}.N \mid q)$ 
     $(L' \mid q', g', (a', c', m'), f', l'') \in \text{fst } (\text{defs}.N \mid q')$ 
     $L' \in \text{defs}.states' s'' \quad q \neq q'$ 
  shows  $L'[q := l', q' := l''] \in \text{defs}.states' s'$ 
using assms by (auto intro: states'_updI')

lemma equiv_sound:
  assumes  $\text{step}: \text{state\_ta} \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
  shows  $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
using step proof cases
case (step_sn_t N d I)
then show ?thesis
  apply simp
  apply (subst A_unfold)
  apply (frule prod.A_simp(1))
  apply (frule prod.A_simp(2))
  apply (rule step_u_t)
  using inv'' bounded by (auto simp: inv_simp p_def)
next
case (step_sn_i l g a c m f l' N p r I)
then show ?thesis
  apply (simp)
  apply (frule prod.A_simp(1))
  apply (frule prod.A_simp(2))
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst A_unfold)
  apply (drule trans_state_taD)
  apply assumption
  unfolding state_trans_t_def
  apply safe
  apply (drule (2) P_steps_upd)
  apply (drule (3) P_steps_guard)
  apply safe
  apply (rule step_u_i)
  subgoals  $\langle \forall p < \text{length local}.N. \exists pc. \_ \rangle$ 
  apply safe
  apply (rule P_steps)
  apply (fastforce simp: p_def)
  apply (fastforce simp: p_def)
  by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  by (auto simp: inv_simp p_def Prod_TA_Defs.N_s_length intro!: P_bounded)
next
case (step_sn_s l1 g1 a ci mi f1 l1' N p l2 g2 co mo f2 l2' q r1 r2 I)
then show ?thesis
  apply (simp)
  apply (frule prod.A_simp(1))

```

```

apply (frule prod.A_simp(2))
apply (simp add: Prod_TA_Defs.N_s_length)
apply (subst A_unfold)
apply (drule trans_state_taD)
  apply assumption
apply (drule trans_state_taD)
  apply assumption
subgoal
  unfolding state_trans_t_def
  apply safe
  apply (drule (2) P_steps_upd)
  apply (drule (2) P_steps_upd)
  apply (drule (3) P_steps_guard)
  apply (drule (3) P_steps_guard)
  apply safe
  apply (rule step_u_s)
prefers ⟨_ ∨ _⟩
  apply (rule P_steps_reset; force)
subgoals ⟨∀ pa < length local.N. ∃ pc. _⟩
  apply safe
  apply (rule P_steps)
  apply (fastforce simp: p_def)
  apply (fastforce simp: p_def)
  by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  by (auto simp: inv_simp p_def Prod_TA_Defs.N_s_length intro!: P_bounded)
done
qed

lemma state_ta_unfold:
  state_ta = (defs.N, defs.P)
by simp

lemma equiv_complete:
assumes step: A ⊢n ⟨L, s, u⟩ →a ⟨L', s', u'⟩
shows state_ta ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩
using step proof cases
case (step_u_t N P d I)
note [simp] = A_simp[OF this(1)]
from step_u_t(2-) show ?thesis
  by (auto simp: state_ta_def p_def state_inv_def intro: step_sn_t)
next
case (step_u_i P pc_g uu uv uw ux pc_u uy uz va r N I l a l' p)
note [simp] = A_simp[OF this(1)]
from step_u_i(2-) show ?thesis
  apply -
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst state_ta_unfold)
  apply (frule steps_P_guard(1))
  apply assumption
  apply (simp; fail)
  apply (drule steps_P_guard(2))
  apply assumption
  apply (simp; fail)
  apply (frule steps_P_upd(1))

```

```

    apply assumption
    apply (simp; fail)
  apply (drule steps_P_upd(2))
    apply assumption
    apply (simp; fail)
  apply (drule N_transD)
    apply assumption
  apply (rule step_sn_i)
    apply assumption
    apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)+
    apply (simp add: Prod_TA_Defs.N_s_length; fail)
    apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)+
  by (auto 4 3 simp: p_def intro: steps_P intro!: states'_updI)
next
case (step_u_s P pc_g1 vb vc vd ve pc_g2 vf vg vh vi pc_u2 vj vk s1 vl r2 pc_u1 r1 vm vn
vo vp
  N I l1 a l1' p' l2 l2' q
)
note [simp] = A_simp[OF this(1)]
from ⟨q < length L⟩ have q < p by (simp add: Prod_TA_Defs.N_s_length)
from step_u_s(2-) show ?thesis
  apply -
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst state_ta_unfold)
  apply (frule steps_P_guard(1))
    apply assumption
    apply (simp; fail)
  apply (drule steps_P_guard(2))
    apply assumption
    apply (simp; fail)
  apply (frule steps_P_guard(1))
    apply (rule ⟨q < p⟩)
    apply (simp; fail)
  apply (drule steps_P_guard(2))
    apply (rule ⟨q < p⟩)
    apply (simp; fail)
  apply (frule steps_P_upd(1))
    apply (rule ⟨q < p⟩)
    apply (simp; fail)
  apply (drule steps_P_upd(2))
    apply (rule ⟨q < p⟩)
    apply (simp; fail)
  apply (drule steps_P_reset[simplified])
    apply assumption
    apply (simp; fail)
  apply (frule steps_P_upd(1))
    apply assumption
    apply (simp; fail)
  apply (drule steps_P_upd(2))
    apply assumption
    apply (simp; fail)
  apply (drule N_transD)
    apply assumption
  apply (drule N_transD)

```

```

    apply assumption
  apply (rule step_sn_s)
    apply assumption
    apply assumption
    apply (all <(auto; fail)?>)
    apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)
    apply (simp add: Prod_TA_Defs.N_s_length; fail)
    apply (simp add: Prod_TA_Defs.N_s_length; fail)
  apply (clarsimp simp: p_def)
  subgoal premises prems for p
    using prems(2, 6-)
    apply -
    apply (erule allE[where x = p], erule impE, rule prems)
    by (fastforce simp: p_def intro: steps_P intro!: states'_updI'')
  done
qed

lemma equiv_sound':
  assumes step: state_ta ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩
  shows A ⊢n ⟨L, s, u⟩ →a ⟨L', s', u'⟩ ∧ L' ∈ defs.states' s' ∧ (∀ q < p. ∃ pc st s'' rs pcs.
    exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
    Some ((pc, st, s'', True, rs), pcs))
  using step proof cases
  case (step_sn_t N d I)
  then show ?thesis
    apply simp
    apply (subst A_unfold)
    apply (frule prod.A_simp(1))
    apply (frule prod.A_simp(2))
    apply (rule conjI)
    apply (rule step_u_t)
    using inv inv'' bounded by (auto simp: inv_simp p_def)
  next
  case (step_sn_i l g a c m f l' N p r I)
  then show ?thesis
    apply (simp)
    apply (frule prod.A_simp(1))
    apply (frule prod.A_simp(2))
    apply (simp add: Prod_TA_Defs.N_s_length)
    apply (subst A_unfold)
    apply (drule trans_state_taD)
    apply assumption
  subgoal
    unfolding state_trans_t_def
    apply safe
    apply (drule (2) P_steps_upd)
    apply (drule (3) P_steps_guard)
    apply safe
    apply (rule step_u_i)
  subgoals <∀ p < length local.N. ∃ pc. __>
    apply safe
    apply (rule P_steps)
    apply (fastforce simp: p_def)
    apply (fastforce simp: p_def)

```

```

    by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  apply (solves ⟨auto simp: inv_simp p_def Prod_TA_Defs.N_s_length intro!: P_bounded⟩)+
  subgoal
    by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  apply simp
  subgoal premises prems for pc_g pc_u q
    using prems(9) ⟨q < _⟩ unfolding state_ta_def state_pred_def
    by (auto 4 3 simp: p_def split: option.splits)
  done
done
next
case (step_sn_s l1 g1 a ci mi f1 l1' N p l2 g2 co mo f2 l2' q r1 r2 I)
then show ?thesis
  apply simp
  apply (frule prod.A_simp(1))
  apply (frule prod.A_simp(2))
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst A_unfold)
  apply (drule trans_state_taD)
  apply assumption
  apply (drule trans_state_taD)
  apply assumption
  subgoal
    unfolding state_trans_t_def
    apply safe
    apply (drule (2) P_steps_upd)
    apply (drule (2) P_steps_upd)
    apply (drule (3) P_steps_guard)
    apply (drule (3) P_steps_guard)
    apply safe
    apply (rule step_u_s)
  prefers disj
    apply (rule P_steps_reset; force)
  subgoals ⟨∀ pa < length local.N. ∃ pc. _⟩
    apply safe
    apply (rule P_steps)
    apply (fastforce simp: p_def)
    apply (fastforce simp: p_def)
    by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  subgoals ⟨_ ∈ defs.states' s'⟩
    by (metis Prod_TA_Defs'.states'_simp Prod_TA_Defs'.states_step local.step states)
  subgoals ⟨∃ pc. _⟩
    apply simp
    subgoal premises prems for pc_g pc_ga pc_u pc_ua q'
      using prems(14) ⟨q' < _⟩ unfolding state_ta_def state_pred_def
      by (auto 4 3 simp: p_def split: option.splits)
    done
    apply (auto simp: inv_simp p_def Prod_TA_Defs.N_s_length intro!: P_bounded)
  done
done
qed

```

**lemma** *equiv\_complete'*:  
 assumes *step*:  $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u \rangle$

```

shows state_ta ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩ ∧ L' ∈ defs.states' s'
  ∧ (∀ q < p. (defs.P ! q) (L' ! q) s')
using step proof cases
case (step_u t N P d I)
note [simp] = A_simp[OF this(1)]
from step_u t(2-) show ?thesis
  apply safe
  subgoal
    by (auto simp: state_ta_def p_def state_inv_def intro: step_sn_t)
    by (fastforce simp: p_def intro: steps_P intro!: states'_updI)+
next
case (step_u i P pc_g uu uv uw ux pc_u uy uz va r N I l a l' p)
note [simp] = A_simp[OF this(1)]
from step_u i(2-) show ?thesis
  apply -
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst state_ta_unfold)
  apply (frule steps_P_guard(1))
  apply assumption
  apply (simp; fail)
  apply (drule steps_P_guard(2))
  apply assumption
  apply (simp; fail)
  apply (frule steps_P_upd(1))
  apply assumption
  apply (simp; fail)
  apply (drule steps_P_upd(2))
  apply assumption
  apply (simp; fail)
  apply (drule N_transD)
  apply assumption
  apply safe
  apply (rule step_sn_i)
  apply assumption
  apply (solves auto; fail)+
  apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)
  apply (solves auto; fail)+
  apply (simp add: Prod_TA_Defs.N_s_length; fail)
  apply (solves auto; fail)+
  by (fastforce simp: p_def intro: steps_P intro!: states'_updI)+
next
case (step_u s P pc_g1 vb vc vd ve pc_g2 vf vg vh vi pc_u2 vj vk s1 vl r2 pc_u1 r1 vm vn
vo vp N
  I l1 a l1' p' l2 l2' q
)
note [simp] = A_simp[OF this(1)]
from ⟨q < length L⟩ have q < p by (simp add: Prod_TA_Defs.N_s_length)
from step_u s(2-) show ?thesis
  apply -
  apply (simp add: Prod_TA_Defs.N_s_length)
  apply (subst state_ta_unfold)
  apply (frule steps_P_guard(1))
  apply assumption
  apply (simp; fail)

```

```

apply (drule steps_P_guard(2))
  apply assumption
apply (simp; fail)
apply (frule steps_P_guard(1))
  apply (rule < q < p >)
  apply (simp; fail)
apply (drule steps_P_guard(2))
  apply (rule < q < p >)
  apply (simp; fail)
apply (frule steps_P_upd(1))
  apply (rule < q < p >)
  apply (simp; fail)
apply (drule steps_P_upd(2))
  apply (rule < q < p >)
  apply (simp; fail)
apply (drule steps_P_reset[simplified])
  apply assumption
  apply (simp; fail)
apply (frule steps_P_upd(1))
  apply assumption
  apply (simp; fail)
apply (drule steps_P_upd(2))
  apply assumption
  apply (simp; fail)
apply (drule N_transD)
  apply assumption
apply (drule N_transD)
  apply assumption
  apply safe
apply (rule step_sn_s)
  apply assumption
  apply assumption
  apply (all <(auto; fail)?>)
apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)
apply (simp add: Prod_TA_Defs.N_s_length; fail)
apply (simp add: Prod_TA_Defs.N_s_length; fail)

subgoals <(defs.P ! _) (L[p' := l1', q := l2'] ! _) s'>
  by (metis Equiv_TA_Defs.p_def states states'_updI'' steps_P)
subgoal
  by simp (metis Equiv_TA_Defs.p_def states states'_updI'' steps_P)
apply (fastforce simp: p_def intro: steps_P intro!: states'_updI'')
done
qed

lemma equiv_complete'':
assumes step:  $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u \rangle p > 0$ 
shows  $(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$ 
   $exec \ PF \ n \ ((I \ ! \ q) \ (L' \ ! \ q), [], s', True, []) [] =$ 
   $Some \ ((pc, st, s'', True, rs), pcs)) \ (is \ ?A)$ 
   $bounded \ B \ s' \ (is \ ?B)$ 

proof –
from assms equiv_complete' have *:  $\forall q < p. (defs.P \ ! \ q) \ (L' \ ! \ q) \ s' \text{ by } simp$ 
then show ?A unfolding state_ta_def state_pred_def by (fastforce split: option.splits)

```

```

from  $\langle p > 0 \rangle$  * have ( $\text{defs}.P ! 0$ ) ( $L' ! 0$ )  $s'$  by auto
with  $\langle p > 0 \rangle$  show ?B unfolding state_ta_def state_pred_def by (auto split: option.splits)
qed

```

**lemma** *equiv\_steps\_sound'*:

**assumes** *step*:  $\text{state\_ta} \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$

**shows**  $A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \wedge L' \in \text{defs.states}' s' \wedge$

$(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$

$\text{exec } PF \ n \ ((I ! q) (L' ! q), [], s', \text{True}, []) [] =$

$\text{Some} ((pc, st, s'', \text{True}, rs), pcs)) \wedge \text{bounded } B \ s'$

**using** *step states inv*

**proof** (*induction*  $A \equiv \text{state\_ta } L \equiv L \ s \equiv s \ u \equiv u \ L' \ s' \ u'$  *arbitrary: rule: steps\_sn.induct*)

**case** (*refl*)

**with** *bounded* **show** ?*case* **by** *blast*

**next**

**case** *prems*: (*step*  $L' \ s' \ u' \ a \ L'' \ s'' \ u''$ )

**from** *prems* **have** \*:

$A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \ L' \in \text{defs.states}' s'$

$(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$

$\text{exec } PF \ n \ ((I ! q) (L' ! q), [], s', \text{True}, []) [] =$

$\text{Some} ((pc, st, s'', \text{True}, rs), pcs))$

$\text{bounded } B \ s'$

**by** *auto*

**interpret** *interp*: *Equiv\_TA*  $A \ n \ L' \ s'$

**using** *pred\_time\_indep upd\_time\_indep clock\_conj* \* **by** *unfold\_locales (auto simp: Len*

*intro!*: \*)

**from** *prems*(3) **have**

$A \vdash_n \langle L', s', u' \rangle \rightarrow_a \langle L'', s'', u'' \rangle \ L'' \in \text{defs.states}' s''$

$\forall q < p. \exists pc \ st \ s''' \ rs \ pcs.$

$\text{exec } PF \ n \ ((I ! q) (L'' ! q), [], s'', \text{True}, []) [] =$

$\text{Some} ((pc, st, s''', \text{True}, rs), pcs)$

$\text{bounded } B \ s''$

**by** (*force dest!*: *interp.equiv\_sound'*) +

**with** \* *interp.states* **show** ?*case*

**by** - (*assumption* | *rule conjI steps\_un.intros*) +

**qed**

**lemma** *equiv\_steps\_complete'*:

$\text{state\_ta} \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \wedge L' \in \text{defs.states}' s' \wedge$

$(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$

$\text{exec } PF \ n \ ((I ! q) (L' ! q), [], s', \text{True}, []) [] =$

$\text{Some} ((pc, st, s'', \text{True}, rs), pcs)) \wedge \text{bounded } B \ s'$

**if**  $A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \ p > 0$

**using** *that states inv* **proof** (*induction*  $A \equiv A \ n \equiv n \ L \equiv L \ s \equiv s \ u \equiv u \ \_\ \_\ \_\$  *rule:*

*steps\_un.induct*)

**case** *refl*

**with** *bounded* **show** ?*case* **by** *blast*

**next**

**case** *prems*: (*step*  $L' \ s' \ u' \ a \ L'' \ s'' \ u''$ )

**from** *prems* **have** \*:

$\text{state\_ta} \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \ L' \in \text{defs.states}' s'$

$(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$

$\text{exec } PF \ n \ ((I ! q) (L' ! q), [], s', \text{True}, []) [] =$

$\text{Some} ((pc, st, s'', \text{True}, rs), pcs))$

```

    bounded B s'
  by auto
interpret interp: Equiv_TA A n L' s'
  using pred_time_indep upd_time_indep clock_conj by unfold_locales (auto simp: Len
intro!: *)
  from interp.equiv_complete'[OF prems(3)] interp.equiv_complete''[OF prems(3) ⟨p > 0⟩]
have
  state_ta ⊢ ⟨L', s', u'⟩ →a ⟨L'', s'', u'⟩ L'' ∈ defs.states' s''
  ∀ q < p. ∃ pc st s''' rs pcs.
    exec PF n ((I ! q) (L'' ! q), [], s'', True, []) [] =
      Some ((pc, st, s''', True, rs), pcs)
  bounded B s''
  by auto
with * interp.states show ?case
  by auto
qed

lemmas equiv_steps_sound = equiv_steps_sound'[THEN conjunct1]
lemmas equiv_steps_complete = equiv_steps_complete'[THEN conjunct1]

lemma equiv_correct:
  state_ta ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩ ⟷ A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩ if p > 0
  using that equiv_steps_sound equiv_steps_complete by metis

lemma prod_correct:
  defs.prod_ta ⊢ ⟨⟨L, s⟩, u⟩ →* ⟨⟨L', s'⟩, u'⟩ ⟷ A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩ if p > 0
  by (metis prod.prod_correct equiv_correct that)

end

end
theory UPPAAL_State_Networks_Impl
  imports Munta_Base.Normalized_Zone_Semantics_Impl UPPAAL_State_Networks
begin

```

## 4 Implementation of UPPAAL Style Networks

**no\_notation** OR (infix or 60)

```

lemma step_resets:
  ∀ c ∈ set r'. ∃ x pc. Some (INSTR (STOREC c x)) = P pc
  if stepc cmd u (pc, st, s, f, r) = Some (pc', st', s', f', r')
  ∀ c ∈ set r. ∃ x pc. Some (INSTR (STOREC c x)) = P pc P pc = Some cmd
  using that
  apply -
  apply (erule stepc.elims)
  by (auto split: option.splits if_splits elim!: step.elims) metis+

lemma step_resets':
  ∀ c ∈ set r'. ∃ x pc. Some (INSTR (STOREC c x)) = P pc
  if step instr (pc, st, s, f, r) = Some (pc', st', s', f', r')
  ∀ c ∈ set r. ∃ x pc. Some (INSTR (STOREC c x)) = P pc P pc = Some (INSTR instr)
  using that

```

**by** (auto split: option.splits if\_splits elim!: step.elims) metis+

**lemma** step\_resets'':

$\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC } c \text{ } x) = P \text{ pc}$   
**if** step instr (pc, st, s, f, r) = Some (pc', st', s', f', r')  
 $\forall c \in \text{set } r. \exists x \text{ pc. Some (STOREC } c \text{ } x) = P \text{ pc } P \text{ pc} = \text{Some instr}$   
**using** that  
**by** (auto split: option.splits if\_splits elim!: step.elims) metis+

**lemma** steps\_reset:

$\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC } c \text{ } x) = P \text{ pc}$   
**if** steps P n (pc, st, s, f, r) (pc', st', s', f', r')  $\forall c \in \text{set } r. \exists x \text{ pc. Some (STOREC } c \text{ } x) = P \text{ pc}$   
**using** that  
**by** (induction P  $\equiv$  P n (pc, st, s, f, r :: nat list) (pc', st', s', f', r') arbitrary: pc st s f r rule: steps.induct)  
(auto dest!: step\_resets''[**where** P = P])

**lemma** exec\_reset:

$\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC } c \text{ } x) = P \text{ pc}$   
**if** Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, []) pcs  
**using** exec\_steps[OF that[symmetric]] steps\_reset **by** force

**lemma** exec\_pointers:

$\forall \text{ pc} \in \text{set pcs}'. \exists \text{ pc instr. Some instr} = P \text{ pc}$   
**if** Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, r) pcs  
 $\forall \text{ pc} \in \text{set pcs.} \exists \text{ pc instr. Some instr} = P \text{ pc}$   
**using** that  
**apply** (induction rule: exec.induct)  
**by** (auto split: option.splits if\_splits) metis+

**lemma** exec\_pointers':

$\forall \text{ pc} \in \text{set pcs}'. \exists \text{ pc instr. Some instr} = P \text{ pc}$   
**if** Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, r) []  
**using** that exec\_pointers **by** fastforce

**context** Prod\_TA\_Defs

**begin**

**lemma** finite\_range\_I':

**assumes**  $\forall A \in \{0..<p\}. \text{finite (range (snd (N ! A)))}$   
**shows** finite (range (I' s))  
**using** assms **unfolding** inv\_of\_def Product\_TA\_Defs.product\_ta\_def N\_s\_def  
**by** (auto simp: inv\_of\_def p\_def intro!: Product\_TA\_Defs.finite\_invariant\_of\_product)

**lemma** range\_prod\_invariant:

range prod\_invariant = range (I' s)  
**unfolding** prod\_invariant\_def **using** I'\_simp **by** auto

**lemma** finite\_rangeI:

**assumes**  $\forall A \in \{0..<p\}. \text{finite (range (snd (N ! A)))}$   
**shows** finite (range prod\_invariant)  
**using** assms **by** (metis finite\_range\_I' range\_prod\_invariant)

**end**

```

context Equiv_TA_Defs
begin

lemma states'_len_simp[simp]:
  length L = p if L ∈ defs.states' s
  using that
  using Product_TA_Defs.states_length defs.N_s_def state_ta_def by fastforce

lemma defs_N_p[simp]:
  length defs.N = p
  unfolding state_ta_def by simp

lemma defs_p[simp]:
  defs.p = p
  unfolding defs.p_def by simp

lemma P_Storec_iff:
  (Some (INSTR (STOREC x xa)) = P pc)  $\longleftrightarrow$  (Some (STOREC x xa) = PF pc)
  unfolding stripfp_def apply (cases P pc)
  apply force
  subgoal for a
  by (cases a) auto
  done

lemma product_trans_i_resets:
  collect_clkvt (Product_TA_Defs.product_trans_i (defs.N_s s))
   $\subseteq \{c. \exists x pc. \text{Some (INSTR (STOREC c x))} = P pc\}$ 
  unfolding collect_clkvt_def
  unfolding Product_TA_Defs.product_trans_i_def
  apply clarsimp
  unfolding defs.N_s_def
  unfolding trans_of_def
  unfolding defs.T_s_def
  unfolding state_ta_def
  unfolding state_trans_t_def
  unfolding make_f_def
  apply (clarsimp split: option.split_asm)
  by (auto dest: exec_reset simp: P_Storec_iff)

lemma product_trans_s_resets:
  collect_clkvt (Product_TA_Defs.product_trans_s (defs.N_s s))
   $\subseteq \{c. \exists x pc. \text{Some (INSTR (STOREC c x))} = P pc\}$ 
  unfolding collect_clkvt_def
  unfolding Product_TA_Defs.product_trans_s_def
  apply clarsimp
  unfolding defs.N_s_def
  unfolding trans_of_def
  unfolding defs.T_s_def
  unfolding state_ta_def
  unfolding state_trans_t_def

```

**unfolding** *make\_f\_def*  
**apply** (*clarsimp split: option.split\_asm*)  
**by** (*auto dest: exec\_reset simp: P\_Storec\_iff*)

**lemma** *product\_trans\_resets:*

$\text{collect\_clkvt } (\bigcup s. \text{defs}.T' s) \subseteq \{c. \exists x \text{ pc. Some } (\text{INSTR } (\text{STOREC } c \ x)) = P \text{ pc}\}$   
**unfolding** *trans\_of\_def*  
**unfolding** *Product\_TA\_Defs.product\_ta\_def*  
**apply** *simp*  
**unfolding** *Product\_TA\_Defs.product\_trans\_def*  
**unfolding** *collect\_clkvt\_def*  
**apply** *safe*  
**unfolding** *Product\_TA\_Defs.product\_trans\_i\_def Product\_TA\_Defs.product\_trans\_s\_def*  
**apply** *clarsimp\_all*  
**unfolding** *defs.N\_s\_def*  
**unfolding** *trans\_of\_def*  
**unfolding** *defs.T\_s\_def*  
**unfolding** *state\_ta\_def*  
**unfolding** *state\_trans\_t\_def*  
**unfolding** *make\_f\_def*  
**apply** (*clarsimp\_all split: option.split\_asm*)  
**by** (*auto dest: exec\_reset simp: P\_Storec\_iff*)

**lemma** *product\_trans\_guards:*

$\text{Timed\_Automata.collect\_clkt } (\bigcup s. \text{defs}.T' s)$   
 $\subseteq \{\text{constraint\_pair } ac \mid ac. \exists pc. \text{Some } (\text{CEXP } ac) = P \text{ pc}\}$   
**unfolding** *trans\_of\_def*  
**unfolding** *Product\_TA\_Defs.product\_ta\_def*  
**apply** *simp*  
**unfolding** *Product\_TA\_Defs.product\_trans\_def*  
**unfolding** *Timed\_Automata.collect\_clkt\_def collect\_clock\_pairs\_def*  
**apply** *safe*  
**unfolding** *Product\_TA\_Defs.product\_trans\_i\_def Product\_TA\_Defs.product\_trans\_s\_def*  
**apply** *clarsimp\_all*  
**unfolding** *defs.N\_s\_def*  
**unfolding** *trans\_of\_def*  
**unfolding** *defs.T\_s\_def*  
**unfolding** *state\_ta\_def*  
**unfolding** *state\_trans\_t\_def*  
**unfolding** *make\_g\_def*  
**apply** (*clarsimp\_all split: option.split\_asm*)  
**subgoal premises** *prems*  
**using** *prems(1,3-)*  
**unfolding** *set\_map\_filter*  
**by** (*smt (verit, best) instrc.simps(5,6) is\_instr.cases*  
*mem\_Collect\_eq option.exhaust option.inject*  
*option.simps(3,4,5))*+)

**subgoal premises** *prems*  
**using** *prems(1,2,4-)*  
**apply** *safe*  
**unfolding** *set\_map\_filter*  
**apply** (*drule exec\_pointers'*)  
**by** (*smt (verit, best) instrc.simps(5,6) is\_instr.cases*  
*mem\_Collect\_eq option.exhaust option.inject*)

```

    option.simps(3,4,5))+
done

end

datatype bexp =
  not bexp | and bexp bexp | or bexp bexp | imply bexp bexp | — Boolean connectives
  loc nat nat | — Is process p in location l?
  eq nat int — Does var i equal x? |
  le nat int |
  lt nat int |
  ge nat int |
  gt nat int

fun check_bexp :: bexp ⇒ nat list ⇒ int list ⇒ bool where
  check_bexp (not a) L s ⇔ ¬ check_bexp a L s |
  check_bexp (and a b) L s ⇔ check_bexp a L s ∧ check_bexp b L s |
  check_bexp (or a b) L s ⇔ check_bexp a L s ∨ check_bexp b L s |
  check_bexp (imply a b) L s ⇔ (check_bexp a L s → check_bexp b L s) |
  check_bexp (loc p l) L _ ⇔ L ! p = l |
  check_bexp (eq i x) _ s ⇔ s ! i = x |
  check_bexp (le i x) _ s ⇔ s ! i ≤ x |
  check_bexp (lt i x) _ s ⇔ s ! i < x |
  check_bexp (ge i x) _ s ⇔ s ! i ≥ x |
  check_bexp (gt i x) _ s ⇔ s ! i > x

datatype formula =
  EX bexp | EG bexp | AX bexp | AG bexp | Leadsto bexp bexp

abbreviation repeat x n ≡ map (λ _. x) [0..4.1 Pre-compiled Networks With States and Clocks as Natural Numbers

locale UPPAAL_Reachability_Problem_precompiled_defs =
  fixes p :: nat — Number of processes
  and m :: nat — Number of clocks

  and max_steps :: nat — Maximal number of execution for steps of programs in the automaton
  and inv :: (nat, int) cconstraint list list — Clock invariants on locations per process
  and pred :: addr list list — State invariants on locations per process
  and trans :: (addr * nat act * addr * nat) list list list
    — Transitions between states per process
  and prog :: int instrc option list

```

```

and formula :: formula — Model checking formula
and bounds :: (int * int) list
begin
definition clkp_set' ≡
  ∪ (collect_clock_pairs ' set (concat inv))
  ∪ {constraint_pair ac | ac. Some (CEXP ac) ∈ set prog}
definition clk_set'_def: clk_set' =
  (fst ' clkp_set' ∪ {c. ∃ x. Some (INSTR (STOREC c x)) ∈ set prog})

```

Definition of the corresponding network

```

definition I i l ≡ if l < length (inv ! i) then inv ! i ! l else []
definition T i ≡
  {(l, trans ! i ! l ! j) | l j. l < length (trans ! i) ∧ j < length (trans ! i ! l)}
definition P ≡ map (λ P l. P ! l) pred
definition PROG pc ≡ (if pc < length prog then prog ! pc else None)
definition N :: (nat, int, nat) unta where
  N ≡ (PROG, map (λ i. (T i, I i)) [0..p], P, bounds)
definition init ≡ repeat (0::nat) p
definition F ≡ hd_of_formula formula

```

**sublocale** *equiv*: *Equiv\_TA\_Defs N max\_steps* .

**abbreviation** *EA* ≡ *equiv.state\_ta*

**abbreviation** *A* ≡ *equiv.defs.prod\_ta*

```

lemma equiv_p_eq[simp]:
  equiv.p = p
unfolding equiv.p_def N_def Equiv_TA_Defs.p_def by simp

```

```

lemma length_N_s[simp]:
  length (equiv.defs.N_s s) = p
unfolding equiv.defs.N_s_def by simp

```

```

lemma length_N[simp]:
  length equiv.defs.N = p
by simp

```

```

lemma
  equiv.defs.I' s L = concat (map (λ q. if q < p then I q (L ! q) else []) [0..length L])
unfolding inv_of_def
unfolding Product_TA_Defs.product_ta_def
apply simp
unfolding Product_TA_Defs.product_invariant_def
unfolding equiv.defs.N_s_def inv_of_def
apply (rule arg_cong[where f = concat])
unfolding Equiv_TA_Defs.state_ta_def
apply simp
unfolding N_def Equiv_TA_Defs.state_inv_def
by simp

```

**end**

```

lemma snd_comp[simp]:

```

$\text{snd } o (\lambda i. (f i, g i)) = g$   
**by** *auto*

**locale** *UPPAAL\_Reachability\_Problem\_precompiled* =  
*UPPAAL\_Reachability\_Problem\_precompiled\_defs* +  
**assumes** *process\_length*:  $\text{length } \text{inv} = p \text{ length } \text{trans} = p \text{ length } \text{pred} = p$   
**and** *lengths*:  
 $\forall i < p. \text{length } (\text{pred } ! i) = \text{length } (\text{trans } ! i) \wedge \text{length } (\text{inv } ! i) = \text{length } (\text{trans } ! i)$   
**and** *state\_set*:  $\forall T \in \text{set } \text{trans}. \forall xs \in \text{set } T. \forall (\_, \_, \_, l) \in \text{set } xs. l < \text{length } T$   
**assumes** *consts\_nats*:  $\text{snd } ' \text{clkp\_set}' \subseteq \mathbb{N}$

**assumes** *clock\_set*:  $\text{clk\_set}' = \{1..m\}$   
**and** *p\_gt\_0*:  $p > 0$   
**and** *m\_gt\_0*:  $m > 0$   
**and** *processes\_have\_trans*:  $\forall i < p. \text{trans } ! i \neq []$  — Necessary for refinement  
**and** *start\_has\_trans*:  $\forall q < p. \text{trans } ! q ! 0 \neq []$  — Necessary for refinement

**assumes** *resets\_zero*:  $\forall x c. \text{Some } (\text{INSTR } (\text{STOREC } c \ x)) \in \text{set } \text{prog} \longrightarrow x = 0$

**begin**

**lemma** *consts\_nats'*:

$\forall I \in \text{set } \text{inv}. \forall cc \in \text{set } I. \forall (c, d) \in \text{collect\_clock\_pairs } cc. d \in \mathbb{N}$

$\forall ac. \text{Some } (\text{CEXP } ac) \in \text{set } \text{prog} \longrightarrow (\text{snd } (\text{constraint\_pair } ac) \in \mathbb{N})$

**using** *consts\_nats* **unfolding** *clkp\_set'\_def* **by** *force+*

**lemma** *clk\_pairs\_N\_inv*:

$\bigcup (\text{collect\_clock\_pairs } ' \text{range } (\text{snd } x)) \subseteq \bigcup (\text{collect\_clock\_pairs } ' \text{set } (\text{concat } \text{inv}))$

**if**  $x \in \text{set } \text{equiv.defs.N}$  **for**  $x$

**using** *that process\_length(1)*

**unfolding** *equiv.state\_ta\_def equiv.state\_inv\_def equiv.p\_def*

**unfolding** *N\_def I\_def*

**by** *clarsimp (auto split: if\_split\_asm dest: nth\_mem)+*

**lemma** *clkp\_set\_simp\_1*:

$\bigcup (\text{collect\_clock\_pairs } ' \text{set } (\text{concat } \text{inv})) \supseteq \text{Timed\_Automata.collect\_clki } (\text{snd } A)$

**unfolding** *equiv.defs.prod\_ta\_def inv\_of\_def*

**apply** (*rule subset\_trans*)

**apply** *simp*

**apply** (*rule equiv.defs.collect\_clki\_prod\_invariant*)

**unfolding** *Timed\_Automata.collect\_clki\_def* **using** *clk\_pairs\_N\_inv nth\_mem* **by** *blast*

**lemma** *clk\_set\_simp\_2*:

$\{c. \exists x. \text{Some } (\text{INSTR } (\text{STOREC } c \ x)) \in \text{set } \text{prog}\} \supseteq \text{collect\_clkvt } (\text{trans\_of } A)$

**unfolding** *equiv.defs.prod\_ta\_def trans\_of\_def*

**apply** (*rule subset\_trans*)

**apply** *simp*

**apply** (*rule equiv.defs.collect\_clkvt\_prod\_trans\_subs*)

**apply** (*rule subset\_trans*)

**apply** (*rule equiv.product\_trans\_resets*)

**unfolding** *N\_def PROG\_def* **by** (*auto dest!: nth\_mem*) *metis*

```

lemma clkp_set_simp_3:
  {constraint_pair ac | ac. Some (CEXP ac) ∈ set prog} ⊇ Timed_Automata.collect_clk
(trans_of A)
  unfolding equiv.defs.prod_ta_def trans_of_def
  apply (rule subset_trans)
  apply simp
  apply (rule equiv.defs.collect_clk_prod_trans_subs)
  apply (rule subset_trans)
  apply (rule equiv.product_trans_guards)
  unfolding N_def PROG_def by (auto dest!: nth_mem)

lemma clkp_set'_subs:
  Timed_Automata.clkp_set A ⊆ clkp_set'
  using clkp_set_simp_1 clkp_set_simp_3
  by (auto simp add: clkp_set'_def Timed_Automata.clkp_set_def inv_of_def)

lemma clk_set'_subs:
  clk_set A ⊆ clk_set'
  using clkp_set'_subs clk_set_simp_2 by (auto simp: clk_set'_def)

lemma clk_set:
  clk_set A ⊆ {1..m}
  using clock_set m_gt_0 clk_set'_subs by auto

lemma
  ∀(_, d) ∈ Timed_Automata.clkp_set A. d ∈ ℤ
  unfolding Ints_def by auto

lemma clkp_set'_consts_nat:
  ∀(_, d) ∈ clkp_set'. d ∈ ℕ
  using consts_nats' unfolding clkp_set'_def
  apply safe
  apply force
  by (metis snd_conv)

lemma clkp_set_consts_nat:
  ∀(_, d) ∈ Timed_Automata.clkp_set A. d ∈ ℕ
  using clkp_set'_subs clkp_set'_consts_nat by auto

lemma finite_clkp_set':
  finite clkp_set'
  unfolding clkp_set'_def
  using [[simproc add: finite_Collect]]
  by (auto simp: inj_on_def intro!: finite_vimageI)

lemma finite_clkp_set_A[intro, simp]:
  finite (Timed_Automata.clkp_set A)
  using clkp_set'_subs finite_clkp_set' by (rule finite_subset)

lemma clkp_set'_bounds:
  a ∈ {Suc 0..m} if (a, b) ∈ clkp_set'
  using that clock_set unfolding clk_set'_def by auto

lemma finite_range_inv_of_A[intro, simp]:

```

```

finite (range (inv_of A))
unfolding inv_of_def equiv.defs.prod_ta_def
apply simp
apply (rule equiv.defs.finite_rangeI)
  apply simp
unfolding Equiv_TA_Defs.state_ta_def
apply simp
unfolding equiv.state_inv_def
unfolding N_def
unfolding I_def
by (auto intro: finite_subset[where B = {[]}])

lemma Collect_fold_pair:
  {f a b | a b. P a b} = (λ (a, b). f a b) ‘ {(a, b). P a b} for P
by auto

lemma finite_T[intro, simp]:
  finite (trans_of A)
  unfolding trans_of_def equiv.defs.prod_ta_def fst_conv
proof (rule equiv.defs.finite_prod_trans, goal_cases)
  case (1 s)
  show ∀ l q. q < equiv.defs.p → (equiv.defs.P ! q) l s → (bounded equiv.B) s
  apply simp
  unfolding equiv.state_ta_def equiv.state_pred_def
  by (simp split: option.split)
next
  case 2
  show finite {s. bounded equiv.B s} by (rule equiv.bounded_finite)
next
  case 3
  show ?case
proof
  fix A assume A: A ∈ set equiv.defs.N
  have
    {(l, j). l < length (trans ! i) ∧ j < length (trans ! i ! l)}
    = ⋃ ((λ l. {(l, j) | j. j < length (trans ! i ! l)}) ‘ {l. l < length (trans ! i)}) for i
  by auto
  then have finite (T q) if q < p for q
  using that unfolding T_def by (fastforce simp: Collect_fold_pair)
  then have finite (fst (equiv.N ! q)) if q < p for q
  using that unfolding N_def by simp
  then have finite (equiv.state_trans q) if q < p for q
  using that
  unfolding Equiv_TA_Defs.state_trans_t_def
  using [[simproc add: finite_Collect]]
  by auto
  then show finite (fst A) using A unfolding Equiv_TA_Defs.state_ta_def by auto
qed
qed (auto simp: p_gt_0)

sublocale TA_Start_No_Ceiling A (init, s₀) m
  using clkp_set_consts_nat clk_set m_gt_0 by - (standard; blast)

lemma P_p[simp]:

```

```

length P = p
unfolding P_def using process_length(3) by simp

lemma length_I[simp]:
  length equiv.I = p
unfolding N_def by simp

lemma length_N[simp]:
  length equiv.N = p
unfolding N_def by simp

end

end
theory Program_Analysis
imports
  UPPAAL_State_Networks_Impl
  Munta_Base.More_Methods
begin

fun steps_approx :: nat ⇒ 't instrc option list ⇒ addr ⇒ addr set where
  steps_approx 0 prog pc = (if pc < length prog then {pc} else {}) |
  steps_approx (Suc n) prog pc =
    (
      if pc ≥ length prog
      then {}
      else
        case prog ! pc of
          None ⇒ {pc}
        | Some cmd ⇒
          let succs =
            (
              case cmd of
                CEXP ac ⇒ {pc + 1}
              | INSTR instr ⇒ (
                  case instr of
                    CALL ⇒ {..lemma bounded_less_simp[simp]:
  ∀ q ∈ {..by (rule eq_reflection) auto

context
fixes prog :: int instrc option list
and strip :: real instrc ⇒ instr
assumes instr_id[simp]:

```

```

strip (INSTR cmd) = cmd
strip (CEXP ac) ∉ {CALL, RETURN, HALT} ∪ (JMPZ ‘ UNIV)
begin

```

**private definition** [simp]:

```

P' ≡ map_option strip o (conv_prog (λ i. if i < length prog then prog ! i else None))

```

**lemma** steps\_out\_of\_range':

```

assumes steps P' n (pc, st, s, f, rs) (pc', st', s', f', rs') pc ≥ length prog
shows pc' = pc
using assms by cases (auto simp: striptp_def)

```

**lemmas** steps\_out\_of\_range = steps\_out\_of\_range'[unfolded striptp\_def P'\_def]

**lemma** steps\_steps\_approx':

```

assumes steps P' n (pc, st, s, f, rs) (pc', st', s', f', rs') pc' < length prog
shows pc' ∈ steps_approx n prog pc

```

**proof** –

```

{
  fix pc :: nat
  and st :: int list
  and m :: int list
  and f :: bool
  and rs :: nat list
  and a :: nat
  and aa :: int list
  and ab :: int list
  and ac :: bool
  and b :: nat list
  and n :: nat
  and za :: int instrc
  and x2 :: int instrc
  assume A:
    UPPAAL_Asm.step (strip (map_instrc real_of_int za)) (pc, st, m, f, rs)
    = Some (a, aa, ab, ac, b)
    UPPAAL_Asm.steps (map_option strip o
      (λpc. conv_prog (If (pc < length prog) (prog ! pc)) None)) n
      (a, aa, ab, ac, b) (pc', st', s', f', rs')
    pc' ∈ steps_approx n prog a
    pc' < length prog
    (if pc < length prog then prog ! pc else None) = Some za
    prog ! pc = Some x2
    ¬ (∃ x ∈ case x2 :: int instrc of INSTR (JMPZ pc') ⇒ {pc + 1, pc'}
      | INSTR CALL ⇒ {..proof (cases x2)
  case (INSTR x1)
  with A show pc' = pc
  apply (cases x1)
  apply (solves ⟨auto elim!: UPPAAL_Asm.step.elims split: if_split_asm⟩)+
  apply (auto elim!: UPPAAL_Asm.step.elims split: if_split_asm)[]

```

```

    subgoal for q _
      apply (cases q < length prog)
      apply force
      apply (drule steps_out_of_range; simp; fail)
    done
  subgoal for q _
    apply (cases q < length prog)
    apply force
    apply (drule steps_out_of_range; simp; fail)
  done
  apply (auto elim!: UPPAAL_Asm.step.elims split: if_split_asm)[]
subgoal for q
  apply (cases q + 1 < length prog)
  apply force
  apply (drule steps_out_of_range; simp)
done
subgoal for q
  apply (cases q + 1 < length prog)
  apply force
  apply (drule steps_out_of_range; simp)
done
  apply (solves ⟨auto elim!: UPPAAL_Asm.step.elims split: if_split_asm⟩)+
done
next
  case (CEXP x2)
  with A show ?thesis
    using instr_id by (fastforce split: if_split_asm elim!: UPPAAL_Asm.step.elims)
qed
} note * = this
show ?thesis
  using assms
  apply (
    induction P' n (pc, st, s, f, rs) (pc', st', s', f', rs')
    arbitrary: pc st s f rs rule: steps.induct
  )
  apply (solves ⟨simp split: option.split⟩)
  apply (clarsimp, rule conjI)
  apply (simp add: striptp_def split: if_split_asm; fail)
  apply (clarsimp split: option.split, rule conjI)
  apply (solves ⟨auto simp add: striptp_def split: if_split_asm⟩)
  apply safe
  by (rule *)
qed

lemmas steps_steps_approx = steps_steps_approx[unfolded P'_def]

end

context
  fixes prog :: int instrc option list
begin

private abbreviation P i ≡ if i < length prog then prog ! i else None

```

```

lemma stepsc_out_of_range:
  assumes stepsc (conv_prog P) n u (pc, st, s, f, rs) (pc', st', s', f', rs') pc ≥ length prog
  shows pc' = pc
  using assms by cases auto

lemma stepsc_steps_approx:
  assumes stepsc (conv_prog P) n u (pc, st, s, f, rs) (pc', st', s', f', rs') pc' < length prog
  shows pc' ∈ steps_approx n prog pc
proof –
{
  fix u :: nat ⇒ real
  and pc :: nat
  and st :: int list
  and m :: int list
  and f :: bool
  and rs :: nat list
  and a :: nat
  and aa :: int list
  and ab :: int list
  and ac :: bool
  and b :: nat list
  and n :: nat
  and z :: int instrc
  assume A: stepc (map_instrc real_of_int z) u (pc, st, m, f, rs) = Some (a, aa, ab, ac, b)
  stepc (conv_prog P) n u (a, aa, ab, ac, b) (pc', st', s', f', rs')
  pc' ∈ steps_approx n prog a
  pc' < length prog
  P pc = Some z
  have ∀ x2. prog ! pc = Some x2 ⟶ pc' = pc ∨
  (∃ x ∈ case x2 of INSTR (JMPZ pc') ⇒ {pc + 1, pc'} | INSTR CALL ⇒ {..length prog}
  | INSTR instr.RETURN ⇒ {..length prog} | INSTR HALT ⇒ {} | INSTR _ ⇒ {pc + 1}
  | CEXP ac ⇒ {pc + 1}. pc' ∈ steps_approx n prog x)
  proof(cases z)
  case (INSTR x1)
  with A show ?thesis
  apply (simp split: option.split_asm if_split_asm)
  apply (cases x1)
  apply (simp split: if_split_asm; fail)
  apply (solves ⟨auto elim!: UPPAAL_Asm.step.elims split: if_split_asm⟩⟩)
  apply (auto elim!: UPPAAL_Asm.step.elims split: if_split_asm)[]
  subgoal for q
  apply (cases q < length prog)
  apply force
  apply (drule stepsc_out_of_range; simp)
  done
  subgoal for q
  apply (cases q < length prog)
  apply force
  apply (drule stepsc_out_of_range; simp)
  done
  apply (auto elim!: UPPAAL_Asm.step.elims split: if_split_asm)[]
  subgoal for q
  apply (cases q + 1 < length prog)

```

```

    apply force
    apply (drule stepsc_out_of_range; simp)
  done
  subgoal for q
    apply (cases q + 1 < length prog)
    apply force
    apply (drule stepsc_out_of_range; simp)
    done
    apply (drule stepsc_out_of_range; simp)
    apply (solves ⟨auto elim!: UPPAAL_Asm.step.elims split: if_split_asm⟩)+
  done
next
  case (CEXP x2)
  with A show ?thesis
    by (force split: if_split_asm)
qed
} note * = this
show ?thesis
  using assms
  apply (
    induction conv_prog P n u (pc, st, s, f, rs) (pc', st', s', f', rs')
    arbitrary: pc st s f rs rule: stepsc.induct
  )
  apply (simp split: option.split)
  apply clarsimp
  apply (rule conjI)
  apply (simp split: if_split_asm; fail)
  apply (clarsimp split: option.split)
  apply (rule conjI)
  apply (simp split: if_split_asm; fail)
  by (rule *)
qed

```

#### definition

```

time_indep_check pc n ≡
  ∀ pc' ∈ steps_approx n prog pc. pc' < length prog
  → (case prog ! pc' of Some cmd ⇒ is_instr cmd | _ ⇒ True)

```

#### lemma time\_indep\_overapprox:

```

assumes
  time_indep_check pc n
shows time_indep (conv_prog P) n (pc, st, s, f, rs)
proof -
  { fix pc' st' s' f' rs' cmd u
    assume A:
      stepsc (conv_prog local.P) n u (pc, st, s, f, rs) (pc', st', s', f', rs')
      conv_prog local.P pc' = Some cmd
    have is_instr cmd
    proof (cases pc' < length prog)
      case True
      with A(2) obtain cmd' where prog ! pc' = Some cmd' by auto
      with True stepsc_steps_approx[OF A(1)] A(2) assms show ?thesis
        by (cases cmd') (auto simp: time_indep_check_def)
    next

```

```

    case False
    with A(2) show ?thesis by (auto split: option.split_asm)
  qed
}
then show ?thesis unfolding time_indep_def by blast
qed

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

definition
  collect_cexp' pc = {ac. Some (CEXP ac) ∈ (!) prog} ' steps_approx max_steps prog pc}

definition clkp_set'' i l ≡
  collect_clock_pairs (inv ! i ! l) ∪
  ∪ ((λ (g, _). constraint_pair ' collect_cexp' g) ' set (trans ! i ! l))

definition
  collect_cexp = {ac. Some (CEXP ac) ∈ set prog}

definition
  collect_store' pc =
  {(c, x). Some (INSTR (STOREC c x)) ∈ (!) prog} ' steps_approx max_steps prog pc}

end

lemma visited_resets_mono:
  set r ⊆ set r' if visited P n (pc, st, s, f, r) (pc', st', s', f', r') pcs
  using that
  apply (
    induction P ≡ P n (pc, st, s, f, r :: nat list) (pc', st', s', f', r') pcs
    arbitrary: pc st s f r rule: visited.induct
  )
  apply blast
  by (erule step.elims; force split: option.splits if_splits elim!: step.elims)+

lemma visitedc_resets_mono:
  set r ⊆ set r' if visitedc P n u (pc, st, s, f, r) (pc', st', s', f', r') pcs
  using that
  apply (
    induction P ≡ P n u (pc, st, s, f, r :: nat list) (pc', st', s', f', r') pcs
    arbitrary: pc st s f r rule: visitedc.induct
  )
  apply blast
  by (erule stepc.elims; force split: option.splits if_splits elim!: step.elims)+

lemma visited_reset:
  ∃ x. ∃ pc ∈ set pcs. Some (STOREC c x) = P pc
  if visited P n (pc, st, s, f, r) (pc', st', s', f', r') pcs c ∈ set r' - set r
  using that
  apply (
    induction P ≡ P n (pc, st, s, f, r :: nat list) (pc', st', s', f', r') pcs

```

```

    arbitrary: pc st s f r rule: visited.induct
  )
  apply blast
  by (erule step.elims; force split: option.split_asm if_split_asm elim!: step.elims)

lemma visitedc_reset:
   $\exists x. \exists pc \in \text{set pcs. Some (INSTR (STOREC c x)) = P pc}$ 
  if visitedc P n u (pc, st, s, f, r) (pc', st', s', f', r') pcs c  $\in \text{set } r' - \text{set } r$ 
  using that
  apply (
    induction P  $\equiv$  P n u (pc, st, s, f, r :: nat list) (pc', st', s', f', r') pcs
    arbitrary: pc st s f r rule: visitedc.induct
  )
  apply blast
  by (erule stepc.elims; force split: option.split_asm if_split_asm elim!: step.elims)

lemma visited_fuel_mono:
  visited P n' s s' pcs if visited P n s s' pcs  $n' \geq n$ 
  using that
  apply (induction arbitrary: n')
  subgoal for _ _ _ n'
    by (cases n') (auto intro: visited.intros)
  subgoal for _ _ _ _ _ _ _ _ n'
    by (cases n') (auto intro: visited.intros)
  done

lemma visitedc_fuel_mono:
  visitedc P n' u s s' pcs if visitedc P n u s s' pcs  $n' \geq n$ 
  using that
  apply (induction arbitrary: n')
  subgoal for _ _ _ _ _ n'
    by (cases n') (auto intro: visitedc.intros)
  subgoal for _ _ _ _ _ _ _ _ _ _ n'
    by (cases n') (auto intro: visitedc.intros)
  done

lemma visted_split:
  assumes visited P n (pc, st, s, f, r) s'' (pcs' @ pc' # pcs)
  obtains st' s' f' r' where
    visited P n (pc, st, s, f, r) (pc', st', s', f', r') pcs
    visited P n (pc', st', s', f', r') s'' (pcs' @ [pc'])
  apply atomize_elim
  using assms
  proof (induction _ _ _ _ _ pcs' @ pc' # pcs arbitrary: pcs rule: visited.induct)
    case (1 prog n u start)
    then show ?case by simp
  next
    case prems: (2 cmd pc st m f rs s prog n s' pcs pcs'')
    show ?case
    proof (cases pcs'' = [])
      case True
      with prems show ?thesis by (blast intro: visited.intros)
    next
      case False

```

```

with ⟨pcs @ [pc] = _⟩ obtain pcs3 where pcs'' = pcs3 @ [pc]
  by (metis append_butlast_last_id last_ConsR last_append last_snoc list.distinct(1))
with prems obtain st' s'a f' r' where
  visited prog n s (pc', st', s'a, f', r') pcs3
  visited prog n (pc', st', s'a, f', r') s' (pcs' @ [pc'])
  by (auto 9 2)
with prems ⟨pcs'' = _⟩ show ?thesis by (auto 4 6 intro: visited.intros visited_fuel_mono)
qed
qed

```

**lemma** *visited\_steps'*:

```

assumes visited P n (pc, st, s, f, r) s'' pcs pc' ∈ set pcs
obtains st' s' f' r' where steps P n (pc, st, s, f, r) (pc', st', s', f', r')
using assms by (force dest!: split_list dest: visited_steps elim: visted_split)

```

**lemma** *visitedc\_split*:

```

assumes visitedc P n u (pc, st, s, f, r) s'' (pcs' @ pc' # pcs)
obtains st' s' f' r' where
  visitedc P n u (pc, st, s, f, r) (pc', st', s', f', r') pcs
  visitedc P n u (pc', st', s', f', r') s'' (pcs' @ [pc'])
apply atomize_elim
using assms
proof (induction _ _ _ _ _ pcs' @ pc' # pcs arbitrary: pcs rule: visitedc.induct)
  case (1 prog n u start)
  then show ?case by simp
next
  case prems: (2 cmd u pc st m f rs s prog n s' pcs pcs'')
  show ?case
  proof (cases pcs'' = [])
    case True
    with prems show ?thesis by (blast intro: visitedc.intros)
  next
    case False
    with ⟨pcs @ [pc] = _⟩ obtain pcs3 where pcs'' = pcs3 @ [pc]
      by (metis append_butlast_last_id last_ConsR last_append last_snoc list.distinct(1))
    with prems obtain st' s'a f' r' where
      visitedc prog n u s (pc', st', s'a, f', r') pcs3
      visitedc prog n u (pc', st', s'a, f', r') s' (pcs' @ [pc'])
      by (auto 9 2)
    with prems ⟨pcs'' = _⟩ show ?thesis by (auto 4 6 intro: visitedc.intros visitedc_fuel_mono)
  qed
qed
qed

```

**lemma** *visitedc\_stepsc'*:

```

assumes visitedc P n u (pc, st, s, f, r) s'' pcs pc' ∈ set pcs
obtains st' s' f' r' where stepsc P n u (pc, st, s, f, r) (pc', st', s', f', r')
using assms by (force dest!: split_list dest: visitedc_stepsc elim: vistedc_split)

```

**lemma** *steps\_fuel\_mono*:

```

steps P n' s s' if steps P n s s' n' ≥ n
using that
apply (induction arbitrary: n')
subgoal for _ _ _ n'
  by (cases n') auto

```

```

subgoal for _ _ _ _ _ _ _ _ _ _ n'
  by (cases n') auto
done

lemma exec_steps':
  assumes exec_prog n s pcs = Some (s', pcs') pc' ∈ set pcs' - set pcs
  obtains st m f rs where steps_prog n s (pc', st, m, f, rs)
  apply atomize_elim
  using assms
proof (induction P ≡ prog n s pcs arbitrary: s' rule: exec.induct)
  case 1
  then show ?case by simp
next
  case (2 n pc st m f rs pcs)
  then obtain instr where prog pc = Some instr by (cases prog pc) auto
  show ?case
  proof (cases instr = HALT)
    case True
    with 2.prem1 ⟨prog pc = _⟩ show ?thesis by (auto 2 6)
  next
    case F: False
    show ?thesis
    proof (cases pc' = pc)
      case True
      with ⟨prog pc = _⟩ show ?thesis by (auto 2 5)
    next
      case False
      with ⟨prog pc = _⟩ 2(2,3) F show ?thesis
      by - (
        erule exec.elims;
        auto 5 6
        dest!: 2(1)[OF ⟨prog pc = _⟩ F, rotated, OF sym]
        simp: option.split_asm if_split_asm
      )
    qed
  qed
qed

lemma exec_visited:
  assumes exec_prog n s pcs = Some ((pc, st, m, f, rs), pcs')
  obtains pcs'' where
    visited_prog n s (pc, st, m, f, rs) pcs'' ∧ pcs' = pc # pcs'' @ pcs ∧ prog pc = Some HALT
  apply atomize_elim
  using assms proof (induction P ≡ prog n s pcs arbitrary: pc st m f rs rule: exec.induct)
  case 1
  then show ?case by simp
next
  case (2 n pc' st' m' f' rs' pcs')
  then obtain instr where prog pc' = Some instr by (cases prog pc') auto
  show ?case
  proof (cases instr = HALT)
    case True
    with 2.prem1 ⟨prog pc' = _⟩ show ?thesis by (auto elim: exec.elims intro: visited.intros)
  next

```

```

case False
with 2(2) ⟨prog pc' = _⟩ show ?thesis
  by - (
    erule exec.elims;
    auto split: option.split_asm if_split_asm intro: visited.intros
    dest!: 2(1)[OF ⟨prog pc' = _⟩ False, rotated, OF sym]
  )
qed
qed

lemma exec_reset':
  ∃ x. ∃ pc ∈ set pcs'. Some (STOREC c x) = P pc
  if exec P n (pc, st, s, f, r) pcs = Some ((pc', st', s', f', r'), pcs') c ∈ set r' - set r
proof -
  from exec_visited[OF that(1)] obtain pcs'' where *:
    visited P n (pc, st, s, f, r) (pc', st', s', f', r') pcs''
    pcs' = pc' # pcs'' @ pcs ∧ P pc' = Some HALT
  by auto
  from visited_reset[OF this(1) that(2)] obtain x pc where
    pc ∈ set pcs'' Some (STOREC c x) = P pc
  by auto
  with *(2) show ?thesis by auto
qed

lemma steps_approx_out_of_range:
  steps_approx n prog pc = {} if pc ≥ length prog
  using that by (induction n) auto

lemma steps_resets_mono:
  set r ⊆ set r' if steps P n (pc, st, s, f, r) (pc', st', s', f', r')
  using that
  by (induction (pc, st, s, f, r) (pc', st', s', f', r') arbitrary: pc st s f r;
    fastforce intro: visited.intros elim!: step.elims split: if_split_asm)

lemma resets_start:
  assumes
    ∀ pc ∈ {pc..pc'}. ∃ c x. prog ! pc = Some (INSTR (STOREC c x))
    steps
      (map_option stripf o (λpc. if pc < size prog then prog ! pc else None))
      n (pc, st, s, f, r) (pc_t, st', s', f', r')
    prog ! pc_t = Some (INSTR HALT)
  shows {c. ∃ x. ∃ pc ∈ {pc .. pc'}. prog ! pc = Some (INSTR (STOREC c x))} ⊆ set r'
  using assms(2,3,1)
  proof (induction
    (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) n (pc, st, s, f, r)
    (pc_t, st', s', f', r') arbitrary: pc st s f r
  )
    case prems: 1
    show ?case
    proof (cases pc' ≥ pc_t)
      case True
      with prems obtain c x where prog ! pc_t = Some (INSTR (STOREC c x))
      by fastforce
      with prems show ?thesis by simp

```

```

next
  case False
  with prems show ?thesis by auto
qed
next
case prems: (2 cmd pc st m f rs s n)
show ?case
proof (cases pc' ≥ pc)
  case True
  with prems(6) obtain c d where prog ! pc = Some (INSTR (STOREC c d))
  by fastforce
  moreover obtain pc1 st' s' f' r1 where s = (pc1, st', s', f', r1)
  using prod.exhaust by metis
  ultimately have pc1 = pc + 1
  using prems(1,2) by (auto elim!: step.elims split: if_split_asm)
  with prems(4)[OF ⟨s = _⟩ prems(5)] prems(6) have
    {c. ∃ x. ∃ pc ∈ {pc1..pc'}. prog ! pc = Some (INSTR (STOREC c x))} ⊆ set r'
  by auto
  moreover have c ∈ set r1
  using prems(1,2) ⟨s = _⟩ ⟨prog ! pc = _⟩ by (auto elim!: step.elims split: if_split_asm)
  moreover then have c ∈ set r'
  using prems(3) ⟨s = _⟩ by (auto dest: steps_resets_mono)
  ultimately show ?thesis
  using ⟨pc1 = _⟩ ⟨prog ! pc = _⟩ apply clarsimp
  subgoal for _ _ pc''
  by (cases pc'' = pc; force)
  done
next
case False
then show ?thesis by auto
qed
qed

unbundle lattice_syntax

function find_resets_start where
  find_resets_start prog pc =
  (
    if pc < length prog
    then
      case prog ! pc of
        Some (INSTR (STOREC c x)) ⇒ (Some pc ⊔ find_resets_start prog (pc + 1)) |
        _ ⇒ None
      else None
  )

by auto

termination
by (relation measure (λ (prog, pc). length prog - pc)) auto

lemma find_resets_start:
  ∀ pc ∈ {pc..pc'}. ∃ c x. prog ! pc = Some (INSTR (STOREC c x)) if
  find_resets_start prog pc = Some pc'

```

```

using that
proof (induction arbitrary: pc' rule: find_resets_start.induct)
case prems: (1 prog pc)
from prems(2) show ?case
  apply (simp split: if_split_asm)
  apply (
    auto 4 3 simp flip: not_less_eq_eq
    simp del: find_resets_start.simps simp: le_max_iff_disj sup_nat_def sup_option_def
    split: option.split_asm instr.split_asm instrc.split_asm dest!: prems(1)
  )
done
qed

```

```

lemmas resets_start' = resets_start[OF find_resets_start]

```

```

context UPPAAL_Reachability_Problem_precompiled_defs
begin

```

**definition**

```

collect_store'' pc ≡
case find_resets_start prog pc of
  None ⇒ {} |
  Some pc' ⇒
    {(c, x). Some (INSTR (STOREC c x)) ∈ (!) prog} ' {pc .. pc'}

```

**end**

**lemma** *berp\_atLeastAtMost\_iff*:

```

(∀ pc ∈ {pc_s..pc_t}. P pc) ⟷ (∀ pc. pc_s ≤ pc ∧ pc ≤ pc_t ⟶ P pc)
by auto

```

**lemma** *berp\_atLeastLessThan\_iff*:

```

(∀ pc ∈ {pc_s..

```

**lemma** *guaranteed\_execution*:

**assumes**

```

  ∀ pc ∈ {pc..

```

```

  ∀ pc ∈ {pc..

```

**shows**  $\exists st' s' f' r'. steps$

```

  (map_option stripf o (λpc. if pc < size prog then prog ! pc else None))
  n (pc, st, s, f, r) (pc_t, st', s', f', r')

```

**using** *assms*

**proof** (induction pc\_t - pc arbitrary: pc st s f r n rule: less\_induct)

**case** *less*

**let**  $?prog = (map\_option\ stripf\ o\ (\lambda pc. if\ pc < length\ prog\ then\ prog\ !\ pc\ else\ None))$

**from**  $\langle pc_t - pc < n \rangle$  **obtain**  $n'$  **where**  $[simp]: n = Suc\ n'$  **by** (cases n) auto

```

from less.prems(3-) have [simp]:  $pc < \text{length } prog \text{ } pc\_t < \text{length } prog$  by auto
show ?case
proof (cases  $pc\_t = pc$ )
  case True
  then show ?thesis by force
next
  case False
  with less.prems(1,4) have valid_instr:
     $prog ! pc \notin \text{Some } \text{'INSTR '}$ 
     $\{STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,$ 
instr.LE, instr.EQ\}
     $prog ! pc \neq None$ 
    by auto
  from  $\langle pc \leq \_ \rangle \langle pc\_t \neq \_ \rangle$  less.prems(2) have jumps:
     $pc' > pc \wedge pc' \leq pc\_t$  if  $prog ! pc = \text{Some } (INSTR (JMPZ pc'))$  for  $pc'$ 
    using that by fastforce
  from  $\langle pc \leq \_ \rangle \langle pc\_t \neq \_ \rangle$  less.prems(1) have stores:
     $d = 0$  if  $prog ! pc = \text{Some } (INSTR (STOREC c d))$  for  $c d$ 
    using that by fastforce
  show ?thesis
  proof (cases  $prog ! pc$ )
    case None
    from  $\langle pc\_t \neq \_ \rangle$  less.prems(1,4) have  $prog ! pc \neq None$ 
    by simp
    with None show ?thesis by auto
  next
  case (Some a)
  then show ?thesis
  proof (cases a)
    case (INSTR instr)
    have *:
       $\exists st' s' f' r'. \text{steps } ?prog \ n' (pc', st, s, f, r) (pc\_t, st', s', f', r')$ 
      if  $pc < pc' \ pc' \leq pc\_t$  for  $pc' \ st \ s \ f \ r$ 
      apply (rule less.hyps[of  $pc'$ ])
      subgoal
      using that by simp
      subgoal
      using that less.prems(1) by force
      subgoal
      apply clarsimp
      subgoal premises prems for  $pc\_s \ pc'$ 
      proof –
        from prems that have  $pc\_s \in \{pc..<pc\_t\}$  by simp
        with prems(1) less.prems(2) show ?thesis by blast
      qed
      done
      using  $\langle prog ! pc\_t = \_ \rangle \langle pc\_t - pc < n \rangle$  that by auto
    from  $\langle pc\_t \neq pc \rangle \langle pc \leq \_ \rangle$  have  $Suc \ pc \leq pc\_t$  by simp
    then obtain  $pc' \ st' \ s' \ f' \ r'$  where
       $step \ instr \ (pc, st, s, f, r) = \text{Some } (pc', st', s', f', r') \ pc < pc' \ pc' \leq pc\_t$ 
      apply atomize_elim
      apply (cases instr)
      using valid_instr  $\langle a = \_ \rangle \langle \_ = \text{Some } a \rangle$  by (auto simp: stores jumps)
      with  $\langle a = \_ \rangle \langle \_ = \text{Some } a \rangle$  show ?thesis by (force dest!: *)

```

```

next
  case (CEXP x2)
  have *:
     $\exists st' s' f' r'. \text{steps } ?prog \ n' \ (Suc \ pc, \ st, \ s, \ f, \ r) \ (pc\_t, \ st', \ s', \ f', \ r')$ 
    if  $Suc \ pc \leq pc\_t$  for  $st \ s \ f \ r$ 
    apply (rule less.hyps[of Suc pc])
    subgoal
      using  $\langle pc \leq \_ \rangle \ \langle pc\_t \neq pc \rangle$  by simp
    subgoal
      using less.premis(1) by force
    subgoal
      apply clarsimp
    subgoal premises prems for  $pc\_s \ pc'$ 
    proof -
      from prems have  $pc\_s \in \{pc..<pc\_t\}$  by simp
      with prems(1) less.premis(2) show ?thesis by blast
    qed
    done
    using  $\langle prog \ ! \ pc\_t = \_ \rangle \ \langle pc\_t - pc < n \rangle$  that by auto
    from  $\langle pc\_t \neq pc \rangle \ \langle pc \leq \_ \rangle$  have  $Suc \ pc \leq pc\_t$  by simp
    with  $\langle a = \_ \rangle \ \langle \_ = Some \ a \rangle$  show ?thesis by (force dest!: *)
  qed
qed
qed
qed

```

```

function find_next_halt where
  find_next_halt prog pc =
    (
      if  $pc < length \ prog$ 
      then
        case prog ! pc of
          Some (INSTR HALT)  $\Rightarrow$  Some pc |
          _  $\Rightarrow$  find_next_halt prog (pc + 1)
        else None
      )

```

by auto

**termination**

by (relation measure ( $\lambda (prog, pc). \text{length } prog - pc$ )) auto

**lemma** find\_next\_halt\_finds\_halt:

$prog \ ! \ pc' = Some \ (INSTR \ HALT) \wedge pc \leq pc' \wedge pc' < length \ prog$

if  $find\_next\_halt \ prog \ pc = Some \ pc'$

using that **proof** (induction prog pc rule: find\_next\_halt.induct)

case prems: (1 prog pc)

from prems(20) show ?case

by (

simp,

simp

split: if\_split\_asm option.split\_asm instrc.split\_asm instr.split\_asm

del: find\_next\_halt.simps;

fastforce dest: prems(1-19) simp del: find\_next\_halt.simps)

qed

**definition**

*guaranteed\_execution\_cond prog pc\_s n*  $\equiv$   
*case find\_next\_halt prog pc\_s of*  
*None*  $\Rightarrow$  *False* |  
*Some pc\_t*  $\Rightarrow$   
 (  
 $\forall pc \in \{pc\_s..<pc\_t\}.$   
 $prog ! pc \neq None$   
 $\wedge prog ! pc \notin \text{Some } 'INSTR'$   
 $\{STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,$   
 $instr.LE, instr.EQ\}$   
 $\wedge (\forall c d. prog ! pc = \text{Some } (INSTR (STOREC c d)) \longrightarrow d = 0)$   
 $) \wedge$   
 $(\forall pc \in \{pc\_s..<pc\_t\}. \forall pc'. prog ! pc = \text{Some } (INSTR (JMPZ pc')) \longrightarrow pc' > pc \wedge pc'$   
 $\leq pc\_t)$   
 $\wedge n > pc\_t - pc\_s$

**lemma** *guaranteed\_execution\_cond\_alt\_def[code]:*

*guaranteed\_execution\_cond prog pc\_s n*  $\equiv$   
*case find\_next\_halt prog pc\_s of*  
*None*  $\Rightarrow$  *False* |  
*Some pc\_t*  $\Rightarrow$   
 (  
 $\forall pc \in \{pc\_s..<pc\_t\}.$   
 $prog ! pc \neq None$   
 $\wedge prog ! pc \notin \text{Some } 'INSTR'$   
 $\{STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,$   
 $instr.LE, instr.EQ\}$   
 $\wedge (\text{case } prog ! pc \text{ of } \text{Some } (INSTR (STOREC c d)) \Rightarrow d = 0 \mid \_ \Rightarrow \text{True})$   
 $) \wedge$   
 $(\forall pc \in \{pc\_s..<pc\_t\}.$   
 $\text{case } prog ! pc \text{ of } \text{Some } (INSTR (JMPZ pc')) \Rightarrow pc' > pc \wedge pc' \leq pc\_t \mid \_ \Rightarrow \text{True})$   
 $\wedge n > pc\_t - pc\_s$

**proof** (*rule eq\_reflection, goal\_cases*)

**case 1**  
**have** \*:  
 $(\forall c d. prog ! pc = \text{Some } (INSTR (STOREC c d)) \longrightarrow d = 0) \longleftrightarrow$   
 $(\text{case } prog ! pc \text{ of } \text{Some } (INSTR (STOREC c d)) \Rightarrow d = 0 \mid \_ \Rightarrow \text{True})$  **for** *pc*  
**by** (*auto split: option.split instrc.split instr.split*)  
**have** \*\*:  
 $(\forall pc'. prog ! pc = \text{Some } (INSTR (JMPZ pc')) \longrightarrow pc' > pc \wedge pc' \leq pc\_t) \longleftrightarrow$   
 $(\text{case } prog ! pc \text{ of } \text{Some } (INSTR (JMPZ pc')) \Rightarrow pc' > pc \wedge pc' \leq pc\_t \mid \_ \Rightarrow \text{True})$  **for** *pc*  
*pc\_t*  
**by** (*auto split: option.split instrc.split instr.split*)  
**show** ?*case unfolding guaranteed\_execution\_cond\_def \* \*\* ..*  
 qed

**lemma** *guaranteed\_execution'!*:

$\exists pc\_t st' s' f' r' pcs'. \text{exec}$   
 $(\text{map\_option stripf } o (\lambda pc. \text{if } pc < \text{size } prog \text{ then } prog ! pc \text{ else } None))$

```

      n (pc, st, s, f, r) pcs = Some ((pc_t, st', s', f', r'), pcs')
    if guaranteed_execution_cond prog pc n
  proof -
    from that obtain pc_t where find_next_halt prog pc = Some pc_t
      unfolding guaranteed_execution_cond_def bexp_atLeastAtMost_iff
      by (auto split: option.split_asm)
    then have prog ! pc_t = Some (INSTR HALT)  $\wedge$  pc  $\leq$  pc_t  $\wedge$  pc_t < length prog
      by (rule find_next_halt_finds_halt)
    moreover then have  $\exists$  st' s' f' r'. steps
      (map_option stripf o ( $\lambda$ pc. if pc < size prog then prog ! pc else None))
      n (pc, st, s, f, r) (pc_t, st', s', f', r')
    using <_ = Some pc_t that
      unfolding guaranteed_execution_cond_def bexp_atLeastAtMost_iff
      by - (rule guaranteed_execution, auto)
    ultimately show ?thesis by (force dest: steps_exec)
  qed

```

```

context UPPAAL_Reachability_Problem_precompiled_defs
begin

```

```

lemma collect_cexp_alt_def:
  collect_cexp =
    set (List.map_filter
      ( $\lambda$  x. case x of Some (CEXP ac)  $\Rightarrow$  Some ac | _  $\Rightarrow$  None)
      prog)
  unfolding collect_cexp_def set_map_filter by (auto split: option.split_asm instrc.split_asm)

```

```

lemma clkp_set'_alt_def:
  clkp_set' =
     $\bigcup$  (collect_clock_pairs ' set (concat inv))  $\cup$  (constraint_pair ' collect_cexp)
  unfolding clkp_set'_def collect_cexp_def by auto

```

```

definition
  collect_store = {(c, x). Some (INSTR (STOREC c x))  $\in$  set prog}

```

```

lemma collect_store_alt_def:
  collect_store =
    set (List.map_filter
      ( $\lambda$  x. case x of Some (INSTR (STOREC c x))  $\Rightarrow$  Some (c, x) | _  $\Rightarrow$  None)
      prog)
  unfolding collect_store_def set_map_filter
  by (auto split: option.split_asm instrc.split_asm instr.split_asm)

```

```

lemma clk_set'_alt_def: clk_set' = (fst ' clkp_set'  $\cup$  fst ' collect_store)
  unfolding clk_set'_def collect_store_def by auto

```

```

end

```

```

fun conj_instr :: 't instrc  $\Rightarrow$  addr  $\Rightarrow$  bool where
  conj_instr (CEXP _) _ = True |
  conj_instr (INSTR COPY) _ = True |
  conj_instr (INSTR (JMPZ pc)) pc_t = (pc = pc_t) |

```

$\text{conj\_instr } (\text{INSTR instr.AND}) \_ = \text{True} \mid$   
 $\text{conj\_instr } \_ \_ = \text{False}$

**inductive**  $\text{is\_conj\_block}' :: 't \text{ instrc option list} \Rightarrow \text{addr} \Rightarrow \text{addr} \Rightarrow \text{bool}$  **where**  
 $\text{is\_conj\_block}' \text{ prog } pc \text{ pc}$  **if**  
 $pc < \text{length prog}$   
 $\text{prog} ! pc = \text{Some } (\text{INSTR HALT}) \mid$

$\text{is\_conj\_block}' \text{ prog } pc \text{ pc}'$  **if**  
 $pc' < \text{length prog}$   
 $\text{prog} ! pc = \text{Some } (\text{INSTR COPY}) \text{ prog} ! (pc + 1) = \text{Some } (\text{CEXP } ac)$   
 $\text{prog} ! (pc + 2) = \text{Some } (\text{INSTR instr.AND})$   
 $\text{is\_conj\_block}' \text{ prog } (pc + 3) \text{ pc}' \mid$   
 $\text{is\_conj\_block}' \text{ prog } pc \text{ pc}'$  **if**  
 $pc' < \text{length prog}$   
 $\text{prog} ! pc = \text{Some } (\text{INSTR COPY})$   
 $\text{prog} ! (pc + 1) = \text{Some } (\text{INSTR } (\text{JMPZ } pc'))$   
 $\text{prog} ! (pc + 2) = \text{Some } (\text{CEXP } ac)$   
 $\text{prog} ! (pc + 3) = \text{Some } (\text{INSTR instr.AND})$   
 $\text{is\_conj\_block}' \text{ prog } (pc + 4) \text{ pc}'$

**inductive\_cases**  $\text{stepscE} : \text{stepsc prog } n \text{ u } (pc, st, m, f, rs) (pc', st', m', f', rs')$

**function**  $\text{check\_conj\_block}' :: 't \text{ instrc option list} \Rightarrow \text{addr} \Rightarrow \text{addr option}$  **where**  
 $\text{check\_conj\_block}' \text{ prog } pc =$   
 $\text{if } pc \geq \text{length prog} \text{ then } \text{None}$   
 $\text{else if } \text{prog} ! pc = \text{Some } (\text{INSTR HALT}) \text{ then } \text{Some } pc$   
 $\text{else if}$   
 $\text{prog} ! pc = \text{Some } (\text{INSTR COPY}) \wedge (\text{case } \text{prog} ! (pc + 1) \text{ of } \text{Some } (\text{CEXP } ac) \Rightarrow \text{True} \mid$   
 $\_ \Rightarrow \text{False})$   
 $\wedge \text{prog} ! (pc + 2) = \text{Some } (\text{INSTR instr.AND})$   
 $\text{then } \text{check\_conj\_block}' \text{ prog } (pc + 3)$   
 $\text{else if}$   
 $\text{prog} ! pc = \text{Some } (\text{INSTR COPY}) \wedge (\text{case } \text{prog} ! (pc + 2) \text{ of } \text{Some } (\text{CEXP } ac) \Rightarrow \text{True} \mid$   
 $\_ \Rightarrow \text{False})$   
 $\wedge \text{prog} ! (pc + 3) = \text{Some } (\text{INSTR instr.AND})$   
 $\wedge (\text{case } \text{prog} ! (pc + 1) \text{ of } \text{Some } (\text{INSTR } (\text{JMPZ } pc')) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})$   
 $\text{then}$   
 $(\text{case } (\text{prog} ! (pc + 1), \text{check\_conj\_block}' \text{ prog } (pc + 4)) \text{ of } (\text{Some } (\text{INSTR } (\text{JMPZ } pc')),$   
 $\text{Some } pc'')$   
 $\Rightarrow \text{if } pc' = pc'' \text{ then } \text{Some } pc' \text{ else } \text{None} \mid \_ \Rightarrow \text{None})$   
 $\text{else } \text{None}$   
 $)$

**by**  $\text{pat\_completeness auto}$

**termination**

**by**  $(\text{relation measure } (\lambda (prog, pc). \text{length prog} - pc)) \text{ auto}$

**lemma**  $\text{is\_conj\_block}'\_len\_prog :$   
 $pc' < \text{length prog}$  **if**  $\text{is\_conj\_block}' \text{ prog } pc \text{ pc}'$

using that by induction auto

**lemma** *check\_conj\_block'*:

*check\_conj\_block' prog pc = Some pc'  $\implies$  is\_conj\_block' prog pc pc'*

**apply** (induction prog pc rule: *check\_conj\_block'.induct*)

**apply** (erule *check\_conj\_block'.elims*)

**by** (auto

*split: if\_split\_asm option.split\_asm instrc.split\_asm instr.split\_asm*

*simp del: check\_conj\_block'.simps*

*intro: is\_conj\_block'.intros is\_conj\_block'\_len\_prog*)

**lemma** *stepsc\_reverseE'*:

**assumes** *stepsc prog (Suc n) u s s'' s''  $\neq$  s*

**obtains** *pc' st' m' f' rs' cmd* **where**

*stepc cmd u (pc', st', m', f', rs') = Some s''*

*prog pc' = Some cmd*

*stepsc prog n u s (pc', st', m', f', rs')*

**apply** *atomize\_elim*

**using** *assms*

**proof** (induction prog Suc n u s s'' arbitrary: n rule: *stepsc.induct*)

**case** (1 prog n u start)

**then show** ?case **by** *simp*

**next**

**case** *prems: (2 cmd u pc st m f rs s prog n s')*

**then show** ?case

**proof** (*cases s' = s*)

**case** *True*

**with prems show** ?thesis

**apply** *simp*

**apply** *solve\_ex\_triv+*

**by** (auto *elim: stepsc.cases*)

**next**

**case** *False*

**with prems(3) have** *n > 0* **by** (auto *elim!: stepsc.cases*)

**then obtain** *n'* **where** *n = Suc n'* **by** (*cases n*) auto

**from** *prems(4)[OF this False]* **obtain** *cmd pc' st' m' f' rs'* **where**

*stepc cmd u (pc', st', m', f', rs') = Some s' prog pc' = Some cmd*

*stepsc prog n' u s (pc', st', m', f', rs')*

**by** *atomize\_elim*

**with prems show** ?thesis **unfolding**  $\langle n = \_ \rangle$  **by** *blast*

**qed**

**qed**

**lemma** *stepsc\_reverseE*:

**assumes** *stepsc prog n u s s'' s''  $\neq$  s*

**obtains** *n' pc' st' m' f' rs' cmd* **where**

*n = Suc n'*

*stepc cmd u (pc', st', m', f', rs') = Some s''*

*prog pc' = Some cmd*

*stepsc prog n' u s (pc', st', m', f', rs')*

**proof** –

**from** *assms* **have** *n > 0* **by** (auto *elim!: stepsc.cases*)

**then obtain** *n'* **where** *n = Suc n'* **by** (*cases n*) auto

**with** *assms* **show** ?thesis **by** (auto *elim!: stepsc\_reverseE' intro!: that*)

qed

**lemma**

$pc = pc' - 1$  **if**  
 $stepc (CEXP cc) u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$   
**using** *that* **by** *auto*

**lemma**

$pc = pc' - 1$  **if**  
 $stepc (INSTR instr) u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$   
 $\neg (\exists x. instr = JMPZ pc')$   $instr \neq RETURN$   $instr \neq CALL$   
**using** *that* **by** (*auto split: option.split\_asm if\_split\_asm elim!: step.elims*)

**lemma** *stepc\_pc\_no\_jump:*

$pc = pc' - 1$  **if**  
 $stepc cmd u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$   
 $cmd \neq INSTR (JMPZ pc')$   $cmd \neq INSTR RETURN$   $cmd \neq INSTR CALL$   
**using** *that* **by** (*cases cmd*) (*auto split: option.split\_asm if\_split\_asm elim!: step.elims*)

**inductive** *stepsn* ::  $'t \text{ programc} \Rightarrow \text{nat} \Rightarrow (\text{nat}, 't :: \text{time}) \text{ cval} \Rightarrow \text{state} \Rightarrow \text{state} \Rightarrow \text{bool}$

**for** *prog* **where**  
 $stepsn \text{ prog } 0 u \text{ start start} \mid$   
 $stepsn \text{ prog } (Suc n) u (pc, st, m, f, rs) s' \text{ if}$   
 $stepc cmd u (pc, st, m, f, rs) = Some s$   
 $prog \text{ pc} = Some cmd$   
 $stepsn \text{ prog } n u s s'$

**declare** *stepsn.intros*[*intro*]

**lemma** *stepsc\_stepsn:*

**assumes**  $stepsc P n u s s'$   
**obtains**  $n'$  **where**  $stepsn P n' u s s' n' < n$   
**using** *assms* **by** *induction auto*

**lemma** *stepsn\_stepsc:*

**assumes**  $stepsn P n' u s s' n' < n$   
**shows**  $stepsc P n u s s'$   
**using** *assms*  
**proof** (*induction arbitrary: n*)  
**case** ( $1 u \text{ start}$ )  
**then obtain**  $n'$  **where**  $n = Suc n'$  **by** (*cases n*) *auto*  
**then show** *?case* **by** *auto*

**next**

**case** ( $2 cmd u pc st m f rs s n s' n'$ )  
**from**  $\langle \_ < n' \rangle$  **have**  $n < n' - 1$  **by** *simp*  
**from**  $2(4)[OF \text{ this}] 2(1,2,3,5)$  **show** *?case*  
**proof**  $-$

**have**  $(\exists fa n fb p. P = fa \wedge n' = Suc n \wedge u = fb \wedge (pc, st, m, f, rs) = p \wedge s' = p) \vee (\exists i$   
 $fa n is isa b ns p fb na pa. P = fb \wedge n' = Suc na \wedge u = fa \wedge (pc, st, m, f, rs) = (n, is, isa, b,$   
 $ns) \wedge s' = pa \wedge stepc i fa (n, is, isa, b, ns) = Some p \wedge fb n = Some i \wedge stepsc fb na fa p pa)$

**using**  $2.prem\ Suc\_pred' \langle P \text{ pc} = Some cmd \rangle \langle stepc cmd u (pc, st, m, f, rs) = Some s \rangle$   
 $\langle stepsc P (n' - 1) u s s' \rangle \text{ gr\_implies\_not\_zero}$  **by** *blast*  
**then show** *?thesis*  
**by** *blast*

```

    qed
  qed

lemma stepsn_extend:
  assumes stepsn P n1 u s s1 stepsn P n2 u s s2 n1 ≤ n2
  shows stepsn P (n2 - n1) u s1 s2
using assms
proof (induction arbitrary: n2 s2)
  case (1 u start)
  then show ?case by simp
next
  case (2 cmd u pc st m f rs s n s')
  from 2(1,2,5-) have stepsn P (n2 - 1) u s s2 by (auto elim: stepsn.cases)
  from 2(4)[OF this] ‹Suc n ≤ n2› show ?case by simp
qed

lemma stepsc_halt:
  s' = (pc, s) if stepsc P n u (pc, s) s' P pc = Some (INSTR HALT)
  using that by (induction P n u (pc, s) s') auto

lemma stepsn_halt:
  s' = (pc, s) if stepsn P n u (pc, s) s' P pc = Some (INSTR HALT)
  apply (rule stepsc_halt[OF stepsn_stepsc[where n = Suc n]])
  using that by simp+

lemma is_conj_block'_pc_mono:
  pc ≤ pc' if is_conj_block' prog pc pc'
  using that by induction auto

lemma is_conj_block'_halt:
  prog ! pc' = Some (INSTR HALT) if is_conj_block' prog pc pc'
  using that by induction auto

lemma numeral_4_eq_4:
  4 = Suc (Suc (Suc (Suc 0)))
  by simp

lemma is_conj_block'_is_conj:
  assumes is_conj_block' P pc pc'
  and stepsn (λ i. if i < length P then P ! i else None) n u (pc, st, s, f, rs) (pc_t, st_t, s_t,
True, rs_t)
  and P ! pc_t = Some (INSTR HALT)

  shows f ∧ pc_t = pc'
  using assms
proof (induction arbitrary: n st s f rs)
  case (1 pc prog)
  with stepsn_halt[OF this(3)] show ?case by simp
next
  case prems: (2 pc' prog pc ac)
  let ?P = (λ i. if i < length prog then prog ! i else None)
  consider (0) n = 0 | (1) n = 1 | (2) n = 2 | (3) n ≥ 3 by force
  then show ?case
  proof cases

```

```

    case 0
    with prems show ?thesis by (auto elim!: stepsn.cases)
next
    case 1
    with prems show ?thesis by (auto elim!: stepsn.cases simp: int_of_def split: if_split_asm)
next
    case 2
    with prems show ?thesis by (auto elim!: stepsn.cases simp: int_of_def numeral_2_eq_2
split: if_split_asm)
next
    case 3
    from prems have pc + 2 < length prog by (auto dest: is_conj_block'_pc_mono)
    with prems(2-4) obtain st1 s1 rs1 where
      stepsn ?P 3 u (pc, st, s, f, rs) (pc + 3, st1, s1, f ∧ (u ⊢a ac), rs1)
    by (force simp: int_of_def numeral_3_eq_3)
    from stepsn_extend[OF this prems(7) 3] have
      stepsn ?P (n - 3) u (pc + 3, st1, s1, f ∧ (u ⊢a ac), rs1) (pc_t, st_t, s_t, True, rs_t) .
    from prems(6)[OF this ⟨prog ! pc_t = _⟩] have pc_t = pc' u ⊢a ac f by auto
    then show ?thesis by simp
qed
next
    case prems: (3 pc' prog pc ac)
    let ?P = (λi. if i < length prog then prog ! i else None)
    consider (0) n = 0 | (1) n = 1 | (2) n = 2 | (3) n = 3 | (4) n ≥ 4 by force
    then show ?case
    proof cases
      case 0
      with prems show ?thesis by (auto elim!: stepsn.cases)
    next
      case 1
      with prems show ?thesis by (auto elim!: stepsn.cases simp: int_of_def split: if_split_asm)
    next
      case 2
      with prems show ?thesis by (auto elim!: stepsn.cases simp: int_of_def numeral_2_eq_2
split: if_split_asm)
    next
      case 3
      with prems show ?thesis
      by (auto elim!: stepsn.cases simp: int_of_def numeral_3_eq_3 split: if_split_asm dest!:
is_conj_block'_halt)
    next
      case 4
      from prems have pc + 3 < length prog by (auto dest: is_conj_block'_pc_mono)
      show ?thesis
      proof (cases f)
        case True
        with ⟨pc + 3 < _⟩ prems(2-6) obtain st1 s1 rs1 where
          stepsn ?P 4 u (pc, st, s, f, rs) (pc + 4, st1, s1, f ∧ (u ⊢a ac), rs1)
        by (force simp: int_of_def numeral_3_eq_3 numeral_4_eq_4)
        from stepsn_extend[OF this prems(8) 4] have
          stepsn ?P (n - 4) u (pc + 4, st1, s1, f ∧ (u ⊢a ac), rs1) (pc_t, st_t, s_t, True, rs_t) .
        from prems(7)[OF this ⟨prog ! pc_t = _⟩] have pc_t = pc' u ⊢a ac f by auto
        then show ?thesis by simp
      next

```

```

case False
with  $\langle pc + 3 < \_ \rangle$  prems(1-6) obtain st1 s1 rs1 where
  stepsn ?P 2 u (pc, st, s, f, rs) (pc', st1, s1, False, rs1)
  by (force simp: int_of_def numeral_2_eq_2)
from stepsn_extend[OF this prems(8)] 4 have
  stepsn ?P (n - 2) u (pc', st1, s1, False, rs1) (pc_t, st_t, s_t, True, rs_t) by simp
from stepsn_halt[OF this] prems(1,6) show ?thesis by (auto dest: is_conj_block'_halt)
qed
qed
qed

```

```

lemma is_conj_block'_is_conj':
  assumes is_conj_block' P pc pc'
  and stepst ( $\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$ ) n u (pc, st, s, f, rs) (pc_t, st_t, s_t, True, rs_t)

  shows  $f \wedge pc\_t = pc'$ 
  using assms
  unfolding stepst_def by (auto split: if_split_asm elim!: is_conj_block'_is_conj stepsc_stepsn)

```

```

lemma is_conj_block'_is_conj2:
  assumes is_conj_block' P (pc + 1) pc' P ! pc = Some (CEXP ac)
  and stepst ( $\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$ ) n u (pc, st, s, f, rs) (pc_t, st_t, s_t, True, rs_t)
  shows ( $u \vdash_a ac$ )  $\wedge pc\_t = pc'$ 
proof -
  let ?P = ( $\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$ )
  from assms(1) have  $pc < pc' \wedge pc' < \text{length } P$ 
  by (auto dest: is_conj_block'_pc_mono is_conj_block'_len_prog)
  with assms(2,3) obtain st' s' rs' where
    stepst ?P (n - 1) u (pc + 1, st', s',  $u \vdash_a ac$ , rs') (pc_t, st_t, s_t, True, rs_t)
  unfolding stepst_def
  apply (clarsimp split: if_split_asm)
  apply (erule stepsc.cases)
  by auto
  from is_conj_block'_is_conj'[OF assms(1) this] show ?thesis .
qed

```

```

lemma is_conj_block'_is_conj3:
  assumes is_conj_block' P (pc + 2) pc' P ! pc = Some (CEXP ac) P ! (pc + 1) = Some (INSTR instr.AND)
  and stepst ( $\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$ ) n u (pc, st, s, f, rs) (pc_t, st_t, s_t, True, rs_t)
  shows ( $u \vdash_a ac$ )  $\wedge pc\_t = pc'$ 
proof -
  let ?P = ( $\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$ )
  from assms(1) have  $pc < pc' \wedge pc' < \text{length } P$ 
  by (auto dest: is_conj_block'_pc_mono is_conj_block'_len_prog)
  with assms(2,3,4) obtain st' s' rs' f where
    stepst ?P (n - 2) u (pc + 2, st', s',  $f \wedge (u \vdash_a ac)$ , rs') (pc_t, st_t, s_t, True, rs_t)
  unfolding stepst_def
  apply (clarsimp split: if_split_asm)
  apply (erule stepsc.cases)
  apply force

```

**apply** (*erule stepsc.cases*)  
**by** (*auto split: if\_split\_asm option.split\_asm elim!: UPPAAL\_Asm.step.elims*)  
**from** *is\_conj\_block'\_is\_conj* [*OF assms(1) this*] **show** *?thesis* **by** *simp*  
**qed**

**definition**

*is\_conj\_block* *P pc pc'*  $\equiv$   
 $(\exists ac. P ! pc = \text{Some } (CEXP\ ac)) \wedge is\_conj\_block' P (pc + 1) pc'$   
 $\vee (\exists ac.$   
 $P ! pc = \text{Some } (CEXP\ ac)) \wedge P ! (pc + 1) = \text{Some } (INSTR\ instr.AND)$   
 $\wedge is\_conj\_block' P (pc + 2) pc'$

**lemma** *is\_conj\_block\_alt\_def* [*code*]:

*is\_conj\_block* *P pc pc'*  $\equiv$   
 $(\text{case } P ! pc \text{ of } \text{Some } (CEXP\ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False}) \wedge is\_conj\_block' P (pc + 1) pc'$   
 $\vee (\text{case } P ! pc \text{ of } \text{Some } (CEXP\ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False}) \wedge P ! (pc + 1) = \text{Some } (INSTR\ instr.AND)$   
 $\wedge is\_conj\_block' P (pc + 2) pc'$   
**unfolding** *is\_conj\_block\_def*  
**by** (*rule eq\_reflection*) (*auto split: option.split\_asm instrc.split\_asm*)

**lemma** *is\_conj\_block\_is\_conj*:

**assumes** *is\_conj\_block* *P pc pc'*  $P ! pc = \text{Some } (CEXP\ ac)$   
**and**  
*stepst*  
 $(\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None})\ n\ u$   
 $(pc, st, s, f, rs)$   
 $(pc\_t, st\_t, s\_t, \text{True}, rs\_t)$   
**shows**  $(u \vdash_a ac) \wedge pc\_t = pc'$   
**using** *assms*  
**unfolding** *is\_conj\_block\_def*  
**apply** *safe*  
**by** ((*drule is\_conj\_block'\_is\_conj2 is\_conj\_block'\_is\_conj3; simp*), *simp*)+

**lemma** *is\_conj\_block'\_decomp*:

*is\_conj\_block* *P pc' pc''* **if**  
 $is\_conj\_block' P pc pc'' P ! pc' = \text{Some } (CEXP\ ac) \ pc \leq pc' \ pc' \leq pc''$   
**using** *that*  
**proof** (*induction arbitrary: pc'*)  
**case** (*1 pc prog*)  
**then show** *?case* **by** (*simp add: is\_conj\_block\_def*)  
**next**  
**case** *prems: (2 pc'' prog pc ac)*  
**with**  $\langle pc \leq \_ \rangle$  **consider**  $pc' = pc \mid pc' = \text{Suc } pc \mid pc' = \text{Suc } (\text{Suc } pc) \mid pc + 3 \leq pc'$   
**by** *force*  
**then show** *?case* **using** *prems* **by** *cases (auto simp add: numeral\_3\_eq\_3 is\_conj\_block\_def)*  
**next**  
**case** *prems: (3 pc'' prog pc ac)*  
**with**  $\langle pc \leq \_ \rangle$  **consider**  
 $pc' = pc \mid pc' = \text{Suc } pc \mid pc' = \text{Suc } (\text{Suc } pc) \mid pc' = \text{Suc } (\text{Suc } (\text{Suc } pc)) \mid pc + 4 \leq pc'$   
**by** *force*  
**then show** *?case* **using** *prems* **by** *cases (auto simp add: numeral\_3\_eq\_3 numeral\_4\_eq\_4 is\_conj\_block\_def)*+  
**qed**

```

lemma is_conj_block_decomp:
  is_conj_block P pc' pc'' if
    is_conj_block P pc pc'' P ! pc' = Some (CEXP ac) pc ≤ pc' pc' ≤ pc''
  using that
  apply (subst (asm) is_conj_block_def)
  apply safe
  using One_nat_def add_Suc_right is_conj_block'_decomp le_simps(3) that(1)
  apply (metis add.right_neutral le_neq_implies_less)
  by (metis
    One_nat_def Suc_1 add.right_neutral add_Suc_right instr.simps(4) is_conj_block'_decomp
    le_antisym not_less_eq_eq option.inject that(1))

```

```

lemma steps_approx_finite[intro,simp]:
  finite (steps_approx n P pc_s)
  by (induction rule: steps_approx.induct; clarsimp split: option.split instr.split instr.split)

```

```

abbreviation conv_P ≡ map (map_option (map_instr real_of_int))

```

```

lemma stepst_stepc_extend:
  stepst P n u (pc', s') (pc'', s'')
  if stepst P n u (pc, s) (pc'', s'') stepsc P n u (pc, s) (pc', s')
proof –
  from that have P pc'' = Some (INSTR HALT) unfolding stepst_def by auto
  from that obtain n1 n2 where *:
    stepsn P n1 u (pc, s) (pc'', s'') stepsn P n2 u (pc, s) (pc', s') n1 < n n2 < n
    unfolding stepst_def by (auto elim!: stepsc_stepsn)
  show ?thesis
  proof (cases n1 ≥ n2)
    case True
    from stepsn_extend[OF *(2,1) this] ⟨n1 < _⟩ ⟨n2 < _⟩ ⟨P pc'' = _⟩ show ?thesis
    unfolding stepst_def by (auto intro: stepsn_stepsc)
  next
    case False
    with stepsn_extend[OF *(1,2)] ⟨n1 < _⟩ ⟨n2 < _⟩ ⟨P pc'' = _⟩ show ?thesis
    unfolding stepst_def by (auto intro: stepsn_stepsc dest!: stepsn_halt)
  qed
qed

```

```

lemma conv_P_conj_block'[intro]:
  is_conj_block' (conv_P P) pc pc' if is_conj_block' P pc pc'
  using that
  apply induction
  apply (rule is_conj_block'.intros(1))
  apply (simp; fail)
  apply (simp; fail)
  apply (rule is_conj_block'.intros(2))
  apply (simp; fail)
  apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
  apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
  apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
  apply (simp; fail)
  apply (rule is_conj_block'.intros(3))
  apply (simp; fail)

```

```

    apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
    apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
    apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
    apply (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog)
    apply (simp; fail)
  done

lemma conv_P_conj_block[intro]:
  is_conj_block (conv_P P) pc pc' if is_conj_block P pc pc'
  using that[unfolded is_conj_block_def]
  apply safe
  by (frule is_conj_block'_pc_mono; force dest: is_conj_block'_len_prog simp: is_conj_block_def)+

context
  fixes P :: int instrc option list
  and pc_s :: addr
  and n :: nat
begin

private abbreviation prog i  $\equiv$  if i < length P then P ! i else None

lemma stepst_conv_P:
  stepst ( $\lambda$  i. if i < length (conv_P P) then conv_P P ! i else None) n u s s' if
  stepst (conv_prog prog) n u s s' using that unfolding stepst_def
  apply safe
  subgoal for pc a aa ab b z
    apply rotate_tac
    by (induction conv_prog prog n u s (pc, a, aa, ab, b) rule: stepsc.induct)
      (auto split: if_split_asm)
  by (auto split: if_split_asm)

lemma is_conj:
  fixes u :: nat  $\Rightarrow$  real
  defines S  $\equiv$  steps_approx n P pc_s
  defines pc_c  $\equiv$  Min {pc.  $\exists$  ac. pc  $\in$  S  $\wedge$  P ! pc = Some (CEXP ac)}
  assumes is_conj_block P pc_c (Max S)
  and stepst (conv_prog prog) n u (pc_s, st, s, f, rs) (pc_t, st_t, s_t, True, rs_t)
  and stepsc (conv_prog prog) n u (pc_s, st, s, f, rs) (pc', st', s', f', rs')
  and P ! pc' = Some (CEXP ac) pc' < length P
  shows (u  $\vdash_a$  conv_ac ac)  $\wedge$  pc_t = Max S
proof -
  from stepst_stepc_extend[OF assms(4,5)] have *:
    stepst (conv_prog prog) n u (pc', st', s', f', rs') (pc_t, st_t, s_t, True, rs_t) .
  from  $\langle$ stepsc _ _ _ _  $\rangle$   $\langle$ P ! pc' = _  $\rangle$   $\langle$ pc' < _  $\rangle$  have pc_c  $\leq$  pc' pc'  $\leq$  Max S
  unfolding pc_c_def S_def by (auto intro: stepsc_steps_approx Min_le Max_ge)
  from is_conj_block_decomp[OF assms(3)  $\langle$ P ! pc' = _  $\rangle$  this] have
    is_conj_block P pc' (Max S) .
  then have is_conj_block (conv_P P) pc' (Max S) by auto
  from is_conj_block_is_conj[OF this _ stepst_conv_P[OF *]]  $\langle$ P ! pc' = _  $\rangle$   $\langle$ pc' < _  $\rangle$ 
  show ?thesis
  by auto
qed

lemma is_conj':

```

```

fixes  $u :: \text{nat} \Rightarrow \text{real}$ 
defines  $S \equiv \text{steps\_approx } n \ P \ pc\_s$ 
assumes  $\{pc. \exists \ ac. pc \in S \wedge P ! pc = \text{Some } (CEXP \ ac)\} = \{\}$ 
  and  $\text{stepsc } (\text{conv\_prog } prog) \ n \ u \ (pc\_s, st, s, f, rs) \ (pc', st', s', f', rs')$ 
  and  $P ! pc' = \text{Some } (CEXP \ ac) \ pc' < \text{length } P$ 
shows False
using  $\text{stepsc\_steps\_approx}[OF \ \text{assms}(3,5)] \ \text{assms}(4) \ \text{assms}(2) \ \text{unfolding } S\_def \ \text{by } auto$ 

```

**definition**

```

 $\text{check\_conj\_block } pc \ pc' \equiv$ 
   $(\text{case } P ! pc \text{ of } \text{Some } (CEXP \ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False}) \wedge \text{check\_conj\_block}' P (pc + 1) =$ 
 $\text{Some } pc'$ 
   $\vee (\text{case } P ! pc \text{ of } \text{Some } (CEXP \ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False}) \wedge P ! (pc + 1) = \text{Some } (\text{INSTR}$ 
 $\text{instr.AND})$ 
   $\wedge \text{check\_conj\_block}' P (pc + 2) = \text{Some } pc'$ 

```

**lemma**  $\text{check\_conj\_block}$ :

```

 $\text{check\_conj\_block } pc \ pc' \implies \text{is\_conj\_block } P \ pc \ pc'$ 
unfolding  $\text{is\_conj\_block\_alt\_def } \text{check\_conj\_block\_def}$ 
by  $(auto \ \text{dest}!: \text{check\_conj\_block}' \ \text{simp} \ \text{del}: \text{check\_conj\_block}'.\text{sims})$ 

```

**definition**

```

 $\text{conjunction\_check} \equiv$ 
   $\text{let } S = \text{steps\_approx } n \ P \ pc\_s; S' = \{pc. \exists \ ac. pc \in S \wedge P ! pc = \text{Some } (CEXP \ ac)\} \text{ in}$ 
 $S' = \{\} \vee \text{check\_conj\_block } (\text{Min } S') (\text{Max } S')$ 

```

**lemma**  $\text{conjunction\_check\_alt\_def}[code]$ :

```

 $\text{conjunction\_check} =$ 
  (
    let
       $S = \text{steps\_approx } n \ P \ pc\_s;$ 
       $S' = \{pc. pc \in S \wedge (\text{case } P ! pc \text{ of } \text{Some } (CEXP \ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})\}$ 
    in
       $S' = \{\} \vee \text{check\_conj\_block } (\text{Min } S') (\text{Max } S')$ 
  )

```

**proof** –

```

let  $?S = \text{steps\_approx } n \ P \ pc\_s$ 
have
   $\{pc. pc \in ?S \wedge (\text{case } P ! pc \text{ of } \text{Some } (CEXP \ ac) \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})\}$ 
 $= \{pc. \exists \ ac. pc \in ?S \wedge P ! pc = \text{Some } (CEXP \ ac)\}$ 
by  $\text{safe } (auto \ \text{split}: \text{option.splits } \text{instr.splits})$ 
show  $?thesis \ \text{unfolding } \text{conjunction\_check\_def } \text{Let\_def } \langle \_ = \_ \rangle ..$ 
qed

```

**lemma**  $\text{conjunction\_check}$ :

```

fixes  $u :: \text{nat} \Rightarrow \text{real}$ 
assumes  $\text{conjunction\_check}$ 
  and  $\text{stepst } (\text{conv\_prog } prog) \ n \ u \ (pc\_s, st, s, f, rs) \ (pc\_t, st\_t, s\_t, \text{True}, rs\_t)$ 
  and  $\text{stepsc } (\text{conv\_prog } prog) \ n \ u \ (pc\_s, st, s, f, rs) \ (pc', st', s', f', rs')$ 
  and  $P ! pc' = \text{Some } (CEXP \ ac) \ pc' < \text{length } P$ 
shows  $u \vdash_a \text{conv\_ac } ac$ 
using  $\text{assms}$ 

```

```

unfolding conjunction_check_def Let_def
apply -
apply (erule disjE)
  apply (drule is_conj'; simp)
    apply (drule check_conj_block)
  apply (subst is_conj; simp)
done

end

end

theory UPPAAL_State_Networks_Impl_Refine
imports
  Program_Analysis
  Munta_Base.Normalized_Zone_Semantics_Impl_Refine
  Munta_Base.TA_Impl_Misc
  Munta_Base.TA_Syntax_Bundles
begin

unbundle no_library_syntax

```

## 5 Imperative Implementation of UPPAAL Style Networks

### 5.1 Executable Successor Computation

```

lemma exec_state_length:
  assumes exec_prog n (pc, st, s, f, rs) pcs = Some ((pc', st', s', f', rs'), pcs')
  shows length s = length s'
  using assms
proof (induction prog n (pc, st, s, f, rs) pcs arbitrary: pc st s f rs pcs rule: exec.induct)
  case 1
  then show ?case by simp
next
  case prems: (2 prog n pc st m f rs pcs)
  from prems(2) show ?case
    apply (clarsimp split: option.split_asm if_split_asm)
    apply (drule prems(1), assumption+)
    by (erule step.elims; simp split: if_split_asm)
qed

locale UPPAAL_Reachability_Problem_precompiled_defs' =
  UPPAAL_Reachability_Problem_precompiled_defs +
  fixes na :: nat
begin

```

Definition of implementation auxiliaries (later connected to the automaton via proof)

```

definition
  trans_i_map =
    map (map (List.map_filter
      (λ (g, a, m, l'). case a of Sil a ⇒ Some (g, a, m, l') | _ ⇒ None)
    )) trans

```

**definition**

```

trans_in_map ≡
  map (map (map
    (λ (g, a, m, l'). case a of In a ⇒ (g, a, m, l') o filter (λ (g, a, m, l').
      case a of In a ⇒ True | _ ⇒ False))
    ) trans

```

**definition**

```

trans_out_map ≡
  map (map (map
    (λ (g, a, m, l'). case a of Out a ⇒ (g, a, m, l') o filter (λ (g, a, m, l').
      case a of Out a ⇒ True | _ ⇒ False))
    ) trans

```

**abbreviation**

```

nested_list_to_iarray xs ≡ IArray (map IArray xs)

```

**definition**

```

actions_by_state i ≡ fold (λ t acc. acc[fst (snd t) := (i, t) # (acc ! fst (snd t))])

```

**definition**

```

all_actions_by_state t L ≡
  fold (λ i. actions_by_state i (t !! i !! (L ! i))) [0..<p] (repeat [] na)

```

**definition**  $PROG' pc \equiv (if\ pc < length\ prog\ then\ (IArray\ prog) !! pc\ else\ None)$

**abbreviation**  $PF \equiv stripfp\ PROG'$

**abbreviation**  $PT \equiv striptp\ PROG'$

**definition**  $runf\ pc\ s \equiv exec\ PF\ max\_steps\ (pc, [], s, True, []) []$

**definition**  $runt\ pc\ s \equiv exec\ PT\ max\_steps\ (pc, [], s, True, []) []$

**lemma**  $PROG'_{\_}PROG [simp]:$

$PROG' = PROG$

**unfolding**  $PROG'_{\_}def\ PROG_{\_}def$  **by** (rule ext) simp

**definition**  $bounded' s \equiv$

$(\forall i < length\ s. fst\ (IArray\ bounds\ !!\ i) < s\ !\ i \wedge s\ !\ i < snd\ (IArray\ bounds\ !!\ i))$

**definition**

```

check_pred L s ≡
  list_all
    (λ q.
      case runf (pred ! q ! (L ! q)) s of
        Some ((_, _, _, f, _), _) ⇒ f ∧ bounded' s
      | None ⇒ False
    )
  [0..<p]

```

**definition**

```

make_constr pcs = List.map_filter
  (λ pc.
    case PROG pc of
      Some (CEXP ac) ⇒ Some ac
    | _ ⇒ None

```

)  
*pcs*

**definition**

*check\_g pc s*  $\equiv$   
*case runt pc s of*  
*Some ((\_, \_, \_, True, \_), pcs)  $\Rightarrow$  Some (make\_cconstr pcs)*  
*| \_  $\Rightarrow$  None*

**definition**

*trans\_i\_from*  $\equiv \lambda (L, s) i.$   
*List.map\_filter* ( $\lambda (g, a, m, l').$   
*case check\_g g s of*  
*Some cc  $\Rightarrow$*   
*(case runf m s of*  
*Some ((\_, \_, s', \_, r), \_)  $\Rightarrow$*   
*if check\_pred (L[i := l']) s'*  
*then Some (cc, a, r, (L[i := l'], s'))*  
*else None*  
*| \_  $\Rightarrow$  None)*  
*| \_  $\Rightarrow$  None)*  
*((IArray (map IArray trans\_i\_map)) !! i !! (L ! i))*

**definition**

*trans\_i\_fun L*  $\equiv \text{concat } (\text{map } (\text{trans\_i\_from } L) [0..<p])$

**definition**

*make\_reset m1 s*  $\equiv$   
*case runf m1 s of*  
*Some ((\_, \_, \_, \_, r1), \_)  $\Rightarrow$  r1*  
*| None  $\Rightarrow$  []*

**definition**

*pairs\_by\_action*  $\equiv \lambda (L, s) \text{ OUT. concat o}$   
*map* ( $\lambda (i, g1, a, m1, l1). \text{List.map\_filter}$   
 $\lambda (j, g2, a, m2, l2).$   
*if i = j then None else*  
*case (check\_g g1 s, check\_g g2 s) of*  
*(Some cc1, Some cc2)  $\Rightarrow$  (*  
*case runf m2 s of*  
*Some ((\_, \_, s1, \_, r2), \_)  $\Rightarrow$  (*  
*case runf m1 s1 of*  
*Some ((\_, \_, s', \_, \_), \_)  $\Rightarrow$*   
*if check\_pred (L[i := l1, j := l2]) s'*  
*then Some (cc1 @ cc2, a, make\_reset m1 s @ r2, (L[i := l1, j := l2], s'))*  
*else None*  
*| \_  $\Rightarrow$  None)*  
*| \_  $\Rightarrow$  None)*  
*| \_  $\Rightarrow$  None*  
*)*  
*OUT)*

**definition**

$trans\_s\_fun \equiv \lambda (L, s).$   
*let*  
      $In = all\_actions\_by\_state (nested\_list\_to\_iarray\ trans\_in\_map)\ L;$   
      $Out = all\_actions\_by\_state (nested\_list\_to\_iarray\ trans\_out\_map)\ L$   
*in*  
      $concat\ (map\ (\lambda\ a.\ pairs\_by\_action\ (L, s)\ (Out\ !\ a)\ (In\ !\ a))\ [0..<na])$

**definition**

$trans\_fun\ L \equiv trans\_s\_fun\ L\ @\ trans\_i\_fun\ L$

**lemma**  $trans\_i\_fun\_trans\_fun:$ 

**assumes**  $(g, a, r, L') \in set\ (trans\_i\_fun\ L)$   
**shows**  $(g, a, r, L') \in set\ (trans\_fun\ L)$   
**using** *assms* **unfolding**  $trans\_fun\_def$  **by** *auto*

**lemma**  $trans\_s\_fun\_trans\_fun:$ 

**assumes**  $(g, a, r, L') \in set\ (trans\_s\_fun\ L)$   
**shows**  $(g, a, r, L') \in set\ (trans\_fun\ L)$   
**using** *assms* **unfolding**  $trans\_fun\_def$  **by** *auto*

**end**

**context**  $Prod\_TA\_Defs$

**begin**

**lemma**  $prod\_trans\_i\_alt\_def:$ 

$prod\_trans\_i =$   
 $\{((L, s), g, a, r, (L', s')) \mid L\ s\ g\ c\ a\ r\ m\ L'\ s'.$   
 $(L, g, (a, Networks.label.Act\ (c, m)), r, L') \in Product\_TA\_Defs.product\_trans\_i\ (N\_s\ s)$   
 $\wedge$   
 $(\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s')$   
 $\wedge\ c\ s \wedge Some\ s' = m\ s\}$

**unfolding**

$prod\_trans\_i\_def\ trans\_of\_def\ Product\_TA\_Defs.product\_ta\_def$   
 $Product\_TA\_Defs.product\_trans\_def$   
 $Product\_TA\_Defs.product\_trans\_i\_def\ Product\_TA\_Defs.product\_trans\_s\_def$   
**by** (*safe; simp; metis*)

**lemma**  $prod\_trans\_s\_alt\_def:$ 

$prod\_trans\_s =$   
 $\{((L, s), g, a, r, (L', s')) \mid L\ s\ g\ ci\ co\ a\ r\ mi\ mo\ L'\ s1\ s'.$   
 $ci\ s \wedge co\ s$   
 $\wedge (\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s')$   
 $\wedge (L, g, (a, Networks.label.Syn\ (ci, mi)\ (co, mo)), r, L')$   
 $\in Product\_TA\_Defs.product\_trans\_s\ (N\_s\ s)$   
 $\wedge Some\ s' = mi\ s1 \wedge Some\ s1 = mo\ s$   
 $\}$

**unfolding**

$prod\_trans\_s\_def\ trans\_of\_def\ Product\_TA\_Defs.product\_ta\_def$   
 $Product\_TA\_Defs.product\_trans\_def$

```

    Product_TA_Defs.product_trans_i_def
    Product_TA_Defs.product_trans_s_def
  by (safe; simp; metis)

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

lemma T_s_unfold_1:
  fst 'equiv.defs.T_s q s = fst 'fst (equiv.N ! q) if q < p
  using 'q < p'
  unfolding equiv.defs.T_s_def
  unfolding equiv.state_ta_def
  unfolding equiv.state_trans_t_def
  by force

lemma T_s_unfold_2:
  (snd o snd o snd o snd) 'equiv.defs.T_s q s = (snd o snd o snd o snd) 'fst (equiv.N ! q)
  if q < p
  using 'q < p'
  unfolding equiv.defs.T_s_def
  unfolding equiv.state_ta_def
  unfolding equiv.state_trans_t_def
  by force

end

lemma (in Equiv_TA_Defs) p_p:
  defs.p = p
  by simp

context
  Prod_TA_Defs
begin

lemma states'_alt_def:
  states' s =
  {L. length L = p ∧
  (∀ q < p. (L ! q) ∈ fst 'fst (fst A ! q)) ∪ (snd o snd o snd o snd) 'fst (fst A ! q))}
  unfolding trans_of_def Product_TA_Defs.product_ta_def N_s_def
  unfolding Product_TA_Defs.product_trans_def
  unfolding Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def
  apply simp
  apply safe
  unfolding T_s_def
  apply (fastforce simp: Product_TA_Defs.states_def trans_of_def p_def)
  apply (force simp: Product_TA_Defs.states_def trans_of_def p_def)
  by (fastforce simp: Product_TA_Defs.states_def trans_of_def p_def image_iff)

end

context UPPAAL_Reachability_Problem_precompiled

```

**begin**

**lemma** *PF\_unfold*:

*equiv.PF* = *stripfp* (*conv\_prog PROG*)  
**using** *[[show\_abbrevs=false]]*  
**unfolding** *N\_def stripfp\_def*  
**apply** (*rule ext*)  
**subgoal for** *x*  
**apply** (*cases PROG x*)  
**apply** (*solves simp*)  
**subgoal for** *a*  
**by** (*cases a*) *auto*  
**done**  
**done**

**lemma** *PT\_unfold*:

*equiv.PT* = *striptp* (*conv\_prog PROG*)  
**using** *[[show\_abbrevs=false]]*  
**unfolding** *N\_def striptp\_def*  
**apply** (*rule ext*)  
**subgoal for** *x*  
**apply** (*cases PROG x*)  
**apply** (*solves simp*)  
**subgoal for** *a*  
**by** (*cases a*) *auto*  
**done**  
**done**

**lemma** *states'I*:

*l* ∈ *equiv.defs.states' s* **if** *A* ⊢ (*l*, *s*)  $\longrightarrow^{g,a,r}$  (*l'*, *s'*)  
**using** *equiv.defs.prod\_ta\_cases[OF that]*  
**unfolding** *equiv.defs.prod\_trans\_i\_alt\_def equiv.defs.prod\_trans\_s\_alt\_def*  
**unfolding** *Product\_TA\_Defs.product\_trans\_def*  
**unfolding** *Product\_TA\_Defs.product\_trans\_i\_def Product\_TA\_Defs.product\_trans\_s\_def*  
**by** *fastforce*

**lemma** *A\_lengthD*:

*length l* = *p* **if** *A* ⊢ (*l*, *s*)  $\longrightarrow^{g,a,r}$  (*l'*, *s'*)  
**using** *that* **by** (*auto dest: states'I*)

**lemma** *N\_s\_state\_trans*:

**assumes** *equiv.defs.N\_s s ! q* ⊢ *l ! q*  $\longrightarrow^{g,(a,c,m'),r}$  *l' q* < *p*  
**obtains** *f' g'* **where**  
(*l ! q*, *g'*, (*a*, *c*, *m'*), *f'*, *l'*) ∈ *equiv.state\_trans q g* = *g' s r* = *f' s*  
**using** *assms*  
**unfolding** *equiv.defs.N\_s\_def trans\_of\_def equiv.defs.T\_s\_def*  
**unfolding** *equiv.state\_ta\_def* **by** *auto*

**lemma** *make\_f\_collect\_store*:

**assumes** (*l*, *pc\_g*, *a*, *pc\_u*, *l'*) ∈ *fst (equiv.N ! q)* *c* ∈ *set (equiv.make\_f pc\_u s)* *q* < *p*  
**shows** *c* ∈ *fst 'collect\_store' pc\_u*

**proof** –

**from** *assms*(1) <*q* < *p*> **have** (*pc\_g*, *a*, *pc\_u*, *l'*) ∈ *set (trans ! q ! l)*  
**unfolding** *N\_def T\_def* **by** (*auto dest!: nth\_mem*)

```

from assms obtain pc x2 x3 x4 pcs r where exec:
  exec equiv.PF max_steps (pc_u, [], s, True, []) [] = Some ((pc, x2, x3, x4, r), pcs)
  unfolding equiv.make_f_def by (auto split: option.split_asm) metis
with assms have c ∈ set r unfolding equiv.make_f_def by auto
with exec_reset'[OF exec] obtain pc' d where Some (STOREC c d) = equiv.PF pc' pc' ∈ set
pcs
  by force
with exec obtain y2 y3 y4 y5 where steps:
  steps equiv.PF max_steps (pc_u, [], s, True, []) (pc', y2, y3, y4, y5)
  by (auto intro: exec_steps')
from ⟨_ = equiv.PF pc'⟩ have pc' < length prog
  unfolding N_def PROG_def stripfp_def by (simp split: if_split_asm)
from steps have pc' ∈ steps_approx max_steps prog pc_u
  unfolding PF_unfold
  unfolding stripfp_def
  by (auto simp: PROG_def intro: steps_steps_approx[of stripf, OF _ _ _ ⟨pc' < length prog⟩])
with ⟨_ = equiv.PF pc'⟩ ⟨_ ∈ set (trans ! q ! l)⟩ show ?thesis
  unfolding collect_store'_def stripfp_def N_def PROG_def
  apply (clarsimp split: if_split_asm)
  apply (cases prog ! pc')
  apply (simp; fail)
  subgoal for x
    by (cases x; force)
  done
qed

```

```

lemma resets_approx:
  set r ⊆
   $\bigcup \{fst \text{ 'collect\_store' } r \mid i \ g \ a \ r. (g, a, r, (l' ! i)) \in set (trans ! i ! (l ! i))\}$ 
  if  $A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$ 

```

```

proof –
from that have [simp]: length l = p by (auto dest: A_lengthD)
show ?thesis using that
  apply clarsimp
  apply (drule equiv.defs.prod_ta_cases)
  apply safe
  subgoal for x
    unfolding equiv.defs.prod_trans_i_alt_def
    apply simp
    unfolding Product_TA_Defs.product_trans_def
    apply safe
    unfolding Product_TA_Defs.product_trans_i_def
    apply clarsimp
    apply (erule N_s_state_trans, assumption)
    unfolding equiv.state_trans_t_def
    apply clarsimp
    subgoal for q l'' pc_g pc_u
      apply (frule make_f_collect_store, assumption+)
      unfolding N_def T_def
      apply (clarsimp dest!: nth_mem)
      subgoal premises prems for b j
      proof –
        from prems(6,8,11–) have
           $(pc_g, Sil \ a, pc_u, l[q := l''] ! q) \in set (trans ! q ! (l ! q))$ 

```

```

      by simp
      with prems(6,8,11-) show ?thesis by blast
    qed
  done
done
subgoal for x
  unfolding equiv.defs.prod_trans_s_alt_def
  apply simp
  unfolding Product_TA_Defs.product_trans_def
  apply safe
  unfolding Product_TA_Defs.product_trans_s_def
  apply clarsimp
  apply (erule N_s_state_trans, assumption)
  apply (erule N_s_state_trans, assumption)
  unfolding equiv.state_trans_t_def
  apply clarsimp
  apply (erule disjE)
  subgoal for s1 p' q l'' l'aa pc_g pc_ga pc_u pc_ua
    apply (frule make_f_collect_store, assumption+)
    unfolding N_def T_def
    apply (clarsimp dest!: nth_mem)
    subgoal premises prems for b j j'
      proof -
        from prems(9-) have
          (pc_g, In a, pc_u, l[p' := l'', q := l'aa] ! p') ∈ set (trans ! p' ! (l ! p'))
          by simp
        with prems(9-) show ?thesis by blast
      qed
    done
  subgoal for s1 q p' l'aa l'' pc_ga pc_g pc_ua pc_u
    apply (frule make_f_collect_store, assumption+)
    unfolding N_def T_def
    apply (clarsimp dest!: nth_mem)
    subgoal premises prems for b j j'
      proof -
        from prems(9-) have
          (pc_g, Out a, pc_u, l[q := l'aa, p' := l''] ! p') ∈ set (trans ! p' ! (l ! p'))
          by simp
        with prems(9-) show ?thesis by blast
      qed
    done
  done
done
qed

```

**lemma** *make\_g\_clkp\_set''*:

```

  fixes x
  assumes
    (l, pc_g, a, pc_u, l') ∈ fst (equiv.N ! q) x ∈ collect_clock_pairs (equiv.make_g pc_g s)
    q < p
  shows x ∈ clkp_set'' q l

```

**proof** –

```

  from asms(1) ⟨q < p⟩ have (pc_g, a, pc_u, l') ∈ set (trans ! q ! l)
  unfolding N_def T_def by (auto dest!: nth_mem)

```

```

from assms obtain pc x2 x3 x4 r pcs where exec:
  exec equiv.PT max_steps (pc_g, [], s, True, []) [] = Some ((pc, x2, x3, x4, r), pcs)
  unfolding equiv.make_g_def by (auto split: option.split_asm)
with assms have x ∈ collect_clock_pairs (List.map_filter (λ pc.
  case equiv.P pc of
    Some (CEXP ac) ⇒ Some ac
  | _ ⇒ None
  )
  pcs)
  unfolding equiv.make_g_def by auto
then obtain pc' ac where
  equiv.P pc' = Some (CEXP ac) x = constraint_pair ac pc' ∈ set pcs
  unfolding equiv.make_g_def collect_clock_pairs_def set_map_filter
  by (clarsimp split: option.split_asm; auto split: instrc.split_asm)
with exec obtain y2 y3 y4 y5 where steps:
  steps equiv.PT max_steps (pc_g, [], s, True, []) (pc', y2, y3, y4, y5)
  by (auto intro: exec_steps')
from ⟨equiv.P pc' = _⟩ have pc' < length prog
  unfolding N_def PROG_def by (simp split: if_split_asm)
from steps have pc' ∈ steps_approx max_steps prog pc_g
  unfolding PT_unfold
  unfolding striptp_def
  by (auto simp: PROG_def intro: steps_steps_approx[of stript, OF _ _ _ ⟨pc' < length prog⟩])
with ⟨equiv.P pc' = _⟩ ⟨_ ∈ set (trans ! q ! l)⟩ ⟨x = _⟩ show ?thesis
  unfolding clkp_set''_def collect_cexp'_def N_def PROG_def by (force split: if_split_asm)
qed

lemma guard_approx:
  collect_clock_pairs g ⊆
  
$$\bigcup \{ \text{clkp\_set'' } i \ (l ! i) \mid i \ g \ a \ r. \\ (g, a, r, (l' ! i)) \in \text{set } (\text{trans } ! \ i ! \ (l ! i)) \wedge l \in \text{equiv.defs.states'} \ s \wedge i < p \}$$

  if  $A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$ 
proof –
  from that have [simp]: length l = p by (auto dest: A_lengthD)
  show ?thesis using that
    apply clarsimp
    apply (drule equiv.defs.prod_ta_cases)
    apply safe
    subgoal for x b
      unfolding equiv.defs.prod_trans_i_alt_def
      apply simp
      unfolding Product_TA_Defs.product_trans_def
      apply safe
      unfolding Product_TA_Defs.product_trans_i_def
      apply clarsimp
      apply (erule N_s_state_trans, assumption)
      unfolding equiv.state_trans_t_def
      apply clarsimp
      subgoal for q l'' pc_g pc_u
        apply (frule make_g_clkp_set'', assumption+)
        unfolding N_def T_def
        apply (clarsimp dest!: nth_mem)
        subgoal premises prems for j

```

```

proof -
  from prems(6,8,11-) have
    (pc_g, Sil a, pc_u, l[q := l'] ! q) ∈ set (trans ! q ! (l ! q))
  by simp
  with prems(6,8,11-) prems show ?thesis by blast
qed
done
done
subgoal for x b
  unfolding equiv.defs.prod_trans_s_alt_def
  apply simp
  unfolding Product_TA_Defs.product_trans_def
  apply safe
  unfolding Product_TA_Defs.product_trans_s_def
  apply clarsimp
  apply (erule N_s_state_trans, assumption)
  apply (erule N_s_state_trans, assumption)
  unfolding equiv.state_trans_t_def
  apply clarsimp
  apply (drule collect_clock_pairs_append_cases)
  apply (erule disjE)
  subgoal for s1 p' q l'' l'aa pc_g pc_ga pc_u pc_ua
    unfolding equiv.make_c_def
    apply (clarsimp split: option.split_asm)
    apply (frule make_g_clkp_set'', assumption+)
    unfolding N_def T_def
    apply (clarsimp dest!: nth_mem)
    subgoal premises prems for j j'
      proof -
        from prems(9-) have
          (pc_g, In a, pc_u, l[p' := l'', q := l'aa] ! p') ∈ set (trans ! p' ! (l ! p'))
        by simp
        with prems(9-) show ?thesis by blast
      qed
    done
  subgoal for s1 q p' l'aa l'' pc_ga pc_g pc_ua pc_u
    apply (frule make_g_clkp_set'', assumption+)
    unfolding N_def T_def
    apply (clarsimp dest!: nth_mem)
    subgoal premises prems for j j'
      proof -
        from prems(9-) have
          (pc_g, Out a, pc_u, l[q := l'aa, p' := l'] ! p') ∈ set (trans ! p' ! (l ! p'))
        by simp
        with prems(9-) show ?thesis by blast
      qed
    done
  done
done
qed
end

```

**abbreviation**  $\text{conv } B \equiv (\text{conv\_prog } (\text{fst } B), (\text{map conv\_A'} (\text{fst } (\text{snd } B))), \text{snd } (\text{snd } B))$

**context** *UPPAAL\_Reachability\_Problem\_precompiled*  
**begin**

**sublocale** *defs'*:

*Equiv\_TA\_Defs conv N max\_steps .*

**lemma** *equiv\_states'\_alt\_def*:

*equiv.defs.states' s =*

*{L. length L = p ∧*

*(∀ q < p. L ! q ∈ fst 'fst (equiv.N ! q)*

*∨ L ! q ∈ (snd o snd o snd o snd) 'fst (equiv.N ! q))}*

**unfolding** *Product\_TA\_Defs.states\_def*

**unfolding** *equiv.defs.N\_s\_def trans\_of\_def*

**using** *T\_s\_unfold\_1 T\_s\_unfold\_2 by simp*

**lemma** *init\_states*:

*init ∈ equiv.defs.states' s<sub>0</sub>*

**using** *processes\_have\_trans start\_has\_trans*

**unfolding** *equiv\_states'\_alt\_def*

**unfolding** *init\_def N\_def T\_def by force*

**lemma** *p\_p[simp]*:

*defs'.p = p*

**unfolding** *defs'.p\_def by simp*

**lemma** *T\_s\_unfold\_1'*:

*fst 'defs'.defs.T\_s q s = fst 'fst (defs'.N ! q) if q < p*

**using** *⟨q < p⟩*

**unfolding** *defs'.defs.T\_s\_def*

**unfolding** *defs'.state\_ta\_def*

**unfolding** *defs'.state\_trans\_t\_def p\_p*

**by force**

**lemma** *T\_s\_unfold\_2'*:

*(snd o snd o snd o snd) 'defs'.defs.T\_s q s = (snd o snd o snd o snd) 'fst (defs'.N ! q)*

**if** *q < p*

**using** *⟨q < p⟩*

**unfolding** *defs'.defs.T\_s\_def*

**unfolding** *defs'.state\_ta\_def*

**unfolding** *defs'.state\_trans\_t\_def p\_p*

**by force**

**lemma** *product\_states'\_alt\_def*:

*defs'.defs.states' s =*

*{L. length L = p ∧*

*(∀ q < p. L ! q ∈ fst 'fst (defs'.N ! q)*

*∨ L ! q ∈ (snd o snd o snd o snd) 'fst (defs'.N ! q))}*

**unfolding** *Product\_TA\_Defs.states\_def*

**unfolding** *defs'.defs.N\_s\_def trans\_of\_def*

**using** *T\_s\_unfold\_1' T\_s\_unfold\_2'*

**by force**

```

lemma states'_conv[simp]:
  defs'.defs.states' s = equiv.defs.states' s
  unfolding product_states'_alt_def equiv_states'_alt_def
  unfolding N_def T_def by simp

lemma init_states_defs'[intro]:
  init  $\in$  defs'.defs.states' s0
  using init_states by simp

lemma
  defs'.I = equiv.I
  by simp

lemma PF_PF[simp]:
  defs'.PF = equiv.PF
  apply simp
  unfolding stripfp_def
  apply (rule ext)
  apply clarsimp
  subgoal for x
    apply (cases equiv.P x)
    apply simp
    subgoal for a
      by (cases a) auto
    done
  done

lemma PF_PROG[simp]:
  equiv.PF = stripfp PROG
  unfolding N_def by simp

lemma I_simp[simp]:
  (equiv.I ! q) l = pred ! q ! l if q < p
  unfolding N_def P_def using  $\langle q < p \rangle$  process_length(3) by simp

lemma
  defs'.P = conv_prog PROG
  by (simp add: N_def)

lemma states_len[intro]:
  assumes
    q < p L  $\in$  equiv.defs.states' s
  shows
    L ! q < length (trans ! q)
  using assms unfolding Product_TA_Defs.states_def
  apply simp
  unfolding trans_of_def equiv.defs.N_s_def
  apply (simp add: T_s_unfold_1[simplified] T_s_unfold_2[simplified])
  unfolding N_def
  apply simp
  unfolding T_def
    using state_set
  unfolding process_length(2)[symmetric]
  apply safe

```

```

    apply (erule allE)
    apply (erule impE)
    apply assumption
    apply safe
    by (clarsimp dest!: nth_mem; force)+

end

locale UPPAAL_Reachability_Problem_precompiled_ceiling =
  UPPAAL_Reachability_Problem_precompiled +
  fixes  $k :: \text{nat list list list}$ 
  assumes  $k\_ceiling$ :
     $\forall i < p. \forall l < \text{length}(\text{trans } i). \forall (x, m) \in \text{clkp\_set}'' i l. m \leq k ! i ! l ! x$ 
     $\forall i < p. \forall l < \text{length}(\text{trans } i). \forall (x, m) \in \text{collect\_clock\_pairs}(\text{inv } i ! l).$ 
     $m \leq k ! i ! l ! x$ 
  and  $k\_resets$ :
     $\forall i < p. \forall l < \text{length}(\text{trans } i). \forall (g, a, r, l') \in \text{set}(\text{trans } i ! l).$ 
     $\forall c \in \{0..<m+1\} - \text{fst } 'collect\_store'' r. k ! i ! l' ! c \leq k ! i ! l ! c$ 
  and  $k\_length$ :
     $\text{length } k = p \ \forall i < p. \text{length } (k ! i) = \text{length}(\text{trans } i)$ 
     $\forall xs \in \text{set } k. \forall xxs \in \text{set } xs. \text{length } xxs = m + 1$ 
  and  $k\_0$ :
     $\forall i < p. \forall l < \text{length}(\text{trans } i). k ! i ! l ! 0 = 0$ 
  and  $guaranteed\_resets$ :
     $\forall i < p. \forall l < \text{length}(\text{trans } i). \forall (g, a, r, l') \in \text{set}(\text{trans } i ! l).$ 
     $\text{guaranteed\_execution\_cond } \text{prog } r \ \text{max\_steps}$ 

begin

definition  $k\_fun \ l \ c \equiv \text{if } c > 0 \wedge c \leq m \text{ then } \text{Max } \{k ! i ! (\text{fst } l ! i) ! c \mid i . i < p\} \text{ else } 0$ 

lemma  $p\_p'$ :
   $\text{equiv}.p = p$ 
  by  $\text{simp}$ 

lemma  $\text{clkp\_set\_clk\_set\_subs}$ :
   $\text{fst } 'clkp\_set' A \ (l, s) \subseteq \text{clk\_set } A$ 
  unfolding  $\text{TA\_clkp\_set\_unfold}$  by  $\text{auto}$ 

lemma  $k\_ceiling\_1$ :
   $\forall l. \forall (x, m) \in \text{clkp\_set } A \ l. m \leq k\_fun \ l \ x$ 

  apply  $\text{safe}$ 
  subgoal  $\text{premises } \text{prems} \text{ for } l \ s \ x \ d$ 
  proof –
    from  $\langle (x, d) \in \_ \rangle$  have  $0 < x \ x \leq m$ 
    using  $\text{clkp\_set\_clk\_set\_subs}[of \ l \ s] \ \text{clk\_set}$  by  $\text{force}+$ 
    from  $\text{prems}$  show  $?thesis$ 
    unfolding  $\text{clkp\_set\_def}$ 
    apply  $\text{safe}$ 
    subgoal
      unfolding  $\text{collect\_clki\_def}$ 

```

```

unfolding inv_of_def
unfolding equiv.defs.prod_ta_def
unfolding equiv.defs.prod_invariant_def
unfolding inv_of_def
  Product_TA_Defs.product_ta_def
  Product_TA_Defs.product_invariant_def
  equiv.defs.N_s_def
unfolding length_N
unfolding equiv.state_ta_def
unfolding p_p'
unfolding equiv.state_inv_def
unfolding N_def
unfolding collect_clock_pairs_def
apply (clarsimp cong: if_cong simp: I_def)
subgoal premises prems for l' i
proof -
  have nat d ≤ k ! i ! (l ! i) ! x
    using prems lengths k_ceiling(2)
    unfolding collect_clock_pairs_def
    by (auto 4 4)
  also from ⟨_ < p⟩ have ... ≤ Max {k ! i ! (l ! i) ! x | i. i < p}
    by (auto intro: Max_ge)
  finally show ?thesis
    unfolding k_fun_def using ⟨0 < x⟩ ⟨x ≤ m⟩ by auto
qed
done
subgoal
  unfolding collect_clk_def
  apply clarsimp
  subgoal premises prems for g a r l' s'
  proof -
    from guard_approx[OF prems(2)] prems(1) obtain i g a r where *:
      (x, d) ∈ clkp_set'' i (l ! i) (g, a, r, l' ! i) ∈ set (trans ! i ! (l ! i))
      l ∈ equiv.defs.states' s i < p
    by auto
    from ⟨i < p⟩ ⟨l ∈ _⟩ have l ! i < length (trans ! i)
      by auto
    with k_ceiling(1) * have nat d ≤ k ! i ! (l ! i) ! x
      by force
    also from ⟨_ < p⟩ have ... ≤ Max {k ! i ! (l ! i) ! x | i. i < p}
      by (auto intro: Max_ge)
    finally show ?thesis
      unfolding k_fun_def using ⟨0 < x⟩ ⟨x ≤ m⟩ by auto
  qed
done
done
qed
done
lemma k_ceiling_2:
  ∀ l g a r l' c. A ⊢ l →g,a,r l' ∧ c ∉ set r → k_fun l' c ≤ k_fun l c
  unfolding trans_of_def equiv.defs.prod_ta_def equiv.defs.prod_trans_def
  apply clarsimp
  apply safe

```

subgoal premises *prems* for *l s g a r l' s' c*

proof –

from *prems* obtain *p' l'' pc\_g pc\_u* where \*:

*p' < p l' = l[p' := l'']*

*r = equiv.make\_f pc\_u s Some s' = equiv.make\_mf pc\_u s*

*(l ! p', pc\_g, Sil a, pc\_u, l'') ∈ fst (equiv.N ! p')*

*l ∈ equiv.defs.states' s*

apply *atomize\_elim*

unfolding *equiv.defs.prod\_trans\_i\_def*

unfolding *Product\_TA\_Defs.product\_ta\_def Product\_TA\_Defs.product\_trans\_def trans\_of\_def*

apply *clarsimp*

apply *safe*

unfolding *Product\_TA\_Defs.product\_trans\_i\_def*

unfolding *trans\_of\_def*

apply *clarsimp*

unfolding *equiv.defs.N\_s\_def*

unfolding *equiv.defs.T\_s\_def*

unfolding *Equiv\_TA\_Defs.state\_ta\_def*

unfolding *equiv.state\_trans\_t\_def*

unfolding *Product\_TA\_Defs.product\_trans\_s\_def*

by *auto*

from *⟨l ∈ \_⟩* have [*simp*]: *length l = p*

by *simp*

from *⟨r = \_⟩* have *fst 'collect\_store'' pc\_u ⊆ set r*

supply *find\_resets\_start.simps[simp del]*

unfolding *collect\_store''\_def equiv.make\_f\_def*

apply (*clarsimp split: option.split\_asm*)

subgoal

using *⟨Some s' = \_⟩* unfolding *equiv.make\_mf\_def*

by (*auto split: option.split\_asm*)

subgoal premises *prems* for *pc' g st f pcs c d pc\_t pc''*

proof –

from *prems* have

*steps (map\_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max\_steps*

*(pc\_u, [], s, True, []) (pc', g, st, f, r)*

*prog ! pc' = Some (INSTR HALT)*

unfolding *PF\_unfold stripfp\_def N\_def PROG\_def*

by (*auto dest!: exec\_steps split: if\_split\_asm elim!: stripf.elims*)

with *prems* show *?thesis*

by (*force intro: sym dest!: resets\_start'*)

qed

done

with *⟨c ∉ \_⟩* have *c ∉ fst 'collect\_store'' pc\_u* by *blast*

show *?thesis*

proof (*cases c > m*)

case *True*

then show *?thesis*

unfolding *k\_fun\_def* by *auto*

next

case *False*

with *⟨c ∉ fst 'collect\_store'' pc\_u⟩* have *c ∈ {0..<m+1} – fst 'collect\_store'' pc\_u*

by *auto*

from \* have *(l ! p') < length (trans ! p')*

unfolding *N\_def T\_def* by *auto*

```

from * have (pc_g, Sil a, pc_u, l'') ∈ set (trans ! p' ! (l ! p'))
  (l ! p') < length (trans ! p')
  unfolding N_def T_def by auto
with k_resets ⟨c ∈ _⟩ ⟨p' < _⟩ have k ! p' ! l'' ! c ≤ k ! p' ! (l ! p') ! c
  unfolding k_fun_def by force
with ⟨l' = _⟩ show ?thesis
  unfolding k_fun_def
  apply clarsimp
  apply (rule Max.boundedI)
  apply force
  using p_gt_0 apply force
  apply clarsimp
  subgoal for i
    apply (cases i = p')
    apply simp
    apply (rule le_trans)
    by (auto intro: Max_ge)
  done
qed
qed
subgoal premises prems for l s g a r l' s' c
proof -
  from prems obtain p1 l1 pc_g1 pc_u1 p2 l2 pc_g2 pc_u2 s'' where *:
    p1 < p p2 < p l' = l[p1 := l1, p2 := l2]
    r = equiv.make_f pc_u1 s @ equiv.make_f pc_u2 s
    Some s' = equiv.make_mf pc_u1 s'' Some s'' = equiv.make_mf pc_u2 s
    (l ! p1, pc_g1, In a, pc_u1, l1) ∈ fst (equiv.N ! p1)
    (l ! p2, pc_g2, Out a, pc_u2, l2) ∈ fst (equiv.N ! p2)
    l ∈ equiv.defs.states' s
  apply atomize_elim
  unfolding equiv.defs.prod_trans_s_def
unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def trans_of_def
  apply clarsimp
  apply safe
  subgoal
    unfolding Product_TA_Defs.product_trans_i_def
    by auto
  unfolding Product_TA_Defs.product_trans_s_def
  unfolding trans_of_def
  apply clarsimp
  unfolding equiv.defs.N_s_def
  unfolding equiv.defs.T_s_def
  unfolding Equiv_TA_Defs.state_ta_def
  unfolding equiv.state_trans_t_def
  apply clarsimp
  by blast
from ⟨l ∈ _⟩ have [simp]: length l = p
  by simp
from * have **:
  (pc_g1, In a, pc_u1, l1) ∈ set (trans ! p1 ! (l ! p1)) (l ! p1) < length (trans ! p1)
  (pc_g2, Out a, pc_u2, l2) ∈ set (trans ! p2 ! (l ! p2)) (l ! p2) < length (trans ! p2)
  unfolding N_def T_def by auto
with ⟨p1 < p⟩ guaranteed_resets have guaranteed_execution:
  guaranteed_execution_cond prog pc_u1 max_steps

```

```

by blast
from ⟨r = _⟩ have fst ‘collect_store’’ pc_u1 ⊆ set r
supply find_resets_start.simps[simp del]
unfolding collect_store’’_def
equiv.make_f_def
apply (clarsimp split: option.split_asm)
subgoal
using ⟨Some s’’ = _⟩ unfolding equiv.make_mf_def
by (auto split: option.split_asm)
subgoal
using ⟨Some s’’ = _⟩ unfolding equiv.make_mf_def
by (auto split: option.split_asm)
subgoal
using ⟨Some s’ = _⟩ unfolding equiv.make_mf_def
using guaranteed_execution'[OF guaranteed_execution, of [] s True [] []]
unfolding stripfp_def PROG_def by auto
subgoal premises prems for _ _ _ _ r2 _ pc’ g st f r1 pcs c d pc_t pc’’
proof –
from prems have
steps (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max_steps
(pc_u1, [], s, True, []) (pc’, g, st, f, r1)
prog ! pc’ = Some (INSTR HALT) r = r1 @ r2
unfolding PF_unfold stripfp_def N_def PROG_def
by (auto dest!: exec_steps split: if_split_asm elim!: stripf.elims)
with prems show ?thesis
by (force intro: sym dest!: resets_start’)
qed
done
moreover from ⟨r = _⟩ have fst ‘collect_store’’ pc_u2 ⊆ set r
supply find_resets_start.simps[simp del]
unfolding collect_store’’_def
equiv.make_f_def
apply (clarsimp split: option.split_asm)
subgoal
using ⟨Some s’’ = _⟩ unfolding equiv.make_mf_def
by (auto split: option.split_asm)
subgoal
using ⟨Some s’’ = _⟩ unfolding equiv.make_mf_def
by (auto split: option.split_asm)
subgoal
using ⟨Some s’ = _⟩ unfolding equiv.make_mf_def
using guaranteed_execution'[OF guaranteed_execution, of [] s True [] []]
unfolding stripfp_def PROG_def by auto
subgoal premises prems for pc’ g st f r1 pcs _ _ _ _ r2 _ c d pc_t pc’’
proof –
from prems have
steps (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max_steps
(pc_u2, [], s, True, []) (pc’, g, st, f, r1)
prog ! pc’ = Some (INSTR HALT) r = r2 @ r1
unfolding PF_unfold stripfp_def N_def PROG_def
by (auto dest!: exec_steps split: if_split_asm elim!: stripf.elims)
with prems show ?thesis
by (force intro: sym dest!: resets_start’)
qed

```

```

done
ultimately have  $c\_not\_elem$ :  $c \notin fst \text{ 'collect\_store'' } pc\_u1$   $c \notin fst \text{ 'collect\_store'' } pc\_u2$ 
  using  $\langle c \notin \_ \rangle$  by auto
show ?thesis
proof (cases  $c > m$ )
  case True
  then show ?thesis
    unfolding  $k\_fun\_def$  by auto
next
case False
with  $c\_not\_elem$  have
   $c \in \{0..<m+1\} - fst \text{ 'collect\_store'' } pc\_u1$ 
   $c \in \{0..<m+1\} - fst \text{ 'collect\_store'' } pc\_u2$ 
  by auto
with  $** k\_resets \langle p1 < \_ \rangle \langle p2 < \_ \rangle$  have
   $k ! p1 ! l1 ! c \leq k ! p1 ! (l ! p1) ! c$   $k ! p2 ! l2 ! c \leq k ! p2 ! (l ! p2) ! c$ 
  by (auto split: prod.split_asm)
with  $\langle l' = \_ \rangle$  show ?thesis
  unfolding  $k\_fun\_def$ 
  apply clarsimp
  apply (rule Max.boundedI)
  apply force
  using  $p\_gt\_0$  apply force
  apply clarsimp
  subgoal for  $i$ 
    apply (cases  $i = p2$ )
    subgoal
      apply simp
      apply (rule le_trans)
      by (auto intro: Max_ge)
    apply (cases  $i = p1$ )
    apply simp
    apply (rule le_trans)
    by (auto intro: Max_ge)
  done
qed
qed
done

lemma
shows  $k\_ceiling'$ :
 $\forall l. \forall (x,m) \in clkp\_set A. l. m \leq k\_fun l x$ 
 $\forall l g a r l' c. A \vdash l \xrightarrow{g,a,r} l' \wedge c \notin set r \longrightarrow k\_fun l' c \leq k\_fun l c$ 
and  $k\_bound'$ :
 $\forall l. \forall i > m. k\_fun l i = 0$ 
and  $k\_0'$ :
 $\forall l. k\_fun l 0 = 0$ 
using  $k\_ceiling\_1$   $k\_ceiling\_2$  unfolding  $k\_fun\_def$  by auto

sublocale Reachability_Problem A (init,  $s_0$ )  $m$   $k\_fun$  PR_CONST  $(\lambda (l, s). F l s)$ 
  by (standard; rule  $k\_ceiling'$   $k\_bound'$   $k\_0'$ )

end

```

```

locale UPPAAL_Reachability_Problem_precompiled_start_state =
  UPPAAL_Reachability_Problem_precompiled _ _ _ pred
for pred :: nat list list +
fixes s0 :: int list
assumes start_pred:
  ∀ q < p. ∃ pc st s' rs pcs.
    exec (stripfp PROG) max_steps ((pred ! q ! (init ! q)), [], s0, True, []) []
  = Some ((pc, st, s', True, rs), pcs)
  and bounded: bounded bounds s0
  and pred_time_indep: ∀ x ∈ set pred. ∀ pc ∈ set x. time_indep_check prog pc max_steps
  and upd_time_indep:
    ∀ T ∈ set trans. ∀ xs ∈ set T. ∀ (_, _, pc_u, _) ∈ set xs.
      time_indep_check prog pc_u max_steps
  and clock_conj:
    ∀ T ∈ set trans. ∀ xs ∈ set T. ∀ (pc_g, _, _, _) ∈ set xs.
      conjunction_check prog pc_g max_steps
begin

lemma B0[intro]:
  bounded defs'.B s0
  using bounded unfolding bounded_def N_def by simp

lemma equiv_P_simp:
  equiv.P = PROG
  unfolding N_def by simp

lemma time_indep_P[intro]:
  time_indep (conv_prog equiv.P) max_steps (pred ! q ! (L ! q), [], s, True, [])
  if q < p L ∈ equiv.defs.states' s
proof –
  from that_lengths process_length have q < length pred L ! q < length (pred ! q) by auto
  then have pred ! q ∈ set pred pred ! q ! (L ! q) ∈ set (pred ! q) by auto
  with pred_time_indep time_indep_overapprox show ?thesis
  by (auto simp: PROG_def equiv_P_simp)
qed

lemma time_indep_PROG[intro]:
  time_indep (conv_prog PROG) max_steps (pc_u, [], s, True, [])
  if q < p (l, pc_g, a, pc_u, l') ∈ T q
proof –
  from that_lengths process_length have
    q < length trans l < length (trans ! q) (pc_g, a, pc_u, l') ∈ set (trans ! q ! l)
  unfolding T_def by auto
  moreover then have trans ! q ∈ set trans trans ! q ! l ∈ set (trans ! q) by auto
  ultimately show ?thesis using upd_time_indep time_indep_overapprox
  unfolding PROG_def by blast
qed

lemma facts[intro]:
  u ⊢a ac if
  q < defs'.p
  (l, pc_g, a, pc_u, l') ∈ fst (defs'.N ! q)
  stepst defs'.P max_steps u (pc_g, [], s, True, []) (pc_t, st_t, s_t, True, rs_t)

```

```

steps def's'.P max_steps u (pc_g, [], s, True, []) (pc', st, s', f', rs)
def's'.P pc' = Some (CEXP ac)
proof -
  let ?P = conv_P prog
  from that(5) obtain ac' where
    ac = conv_ac ac' prog ! pc' = Some (CEXP ac') pc' < length prog
  apply (clarsimp split: option.split_asm if_split_asm simp add: PROG_def N_def)
  subgoal for z
    by (cases z) auto
  done
  with that have u ⊢a conv_ac ac'
  apply -
  apply (rule conjunction_check)
  using clock_conj apply simp_all
  unfolding N_def apply simp_all
  using lengths process_length(2) by (force dest!: nth_mem simp: PROG_def N_def T_def)+
  with ⟨ac = _⟩ show ?thesis by simp
qed

```

```

sublocale product':
  Equiv_TA conv N max_steps init s0
  apply standard
    apply standard
    apply (simp; blast)
  subgoal
    apply clarsimp
    apply (force simp: N_def)
  done
  apply blast
  apply (simp; fail)
  unfolding PF_PF using start_pred apply simp
  by (rule B0)

```

```

lemma fst_S[simp]:
  fst ' (λ(l, g, a, r, l'). (l, map conv_ac g, a, r, l')) ' S = fst ' S
  by force

```

```

lemma snd_S[simp]:
  (snd ∘ snd ∘ snd ∘ snd) ' (λ(l, g, a, r, l'). (l, map conv_ac g, a, r, l')) ' S
  = (snd ∘ snd ∘ snd ∘ snd) ' S
  by force

```

end

```

locale UPPAAL_Reachability_Problem_precompiled' =
  UPPAAL_Reachability_Problem_precompiled_start_state +
  UPPAAL_Reachability_Problem_precompiled_defs' +
  UPPAAL_Reachability_Problem_precompiled_ceiling +
  assumes action_set:
    ∀ T ∈ set trans. ∀ xs ∈ set T. ∀ (_, a, _) ∈ set xs. pred_act (λ a. a < na) a
begin

```

```

sublocale Reachability_Problem_Impl_Defs __ A (init, s0) m k_fun PR_CONST (λ (l, s).

```

$F\ l\ s)$  .

**definition**

$states' = \{(L, s). L \in equiv.defs.states' s \wedge check\_pred\ L\ s \wedge length\ s = length\ bounds\}$

**lemma** *in\_trans\_in\_mapI*:

**assumes**

$q < p\ l < length\ (trans\ !\ q)\ i < length\ (trans\ !\ q\ !\ l)$

$(g1, In\ a, r1) = trans\ !\ q\ !\ l\ !\ i$

**shows**  $(g1, a, r1) \in set\ (IArray\ (map\ IArray\ trans\_in\_map)\ !!\ q\ !!\ l)$

**using** *assms process\_length(2) unfolding trans\_in\_map\_def*

**by** *(force dest: nth\_mem intro!: image\_eqI[where x = (g1, In a, r1)])*

**lemma** *in\_trans\_out\_mapI*:

**assumes**

$q < p\ l < length\ (trans\ !\ q)\ i < length\ (trans\ !\ q\ !\ l)$

$(g1, Out\ a, r1) = trans\ !\ q\ !\ l\ !\ i$

**shows**  $(g1, a, r1) \in set\ (IArray\ (map\ IArray\ trans\_out\_map)\ !!\ q\ !!\ l)$

**using** *assms process\_length(2) unfolding trans\_out\_map\_def*

**by** *(force dest: nth\_mem intro!: image\_eqI[where x = (g1, Out a, r1)])*

**lemma** *in\_trans\_in\_mapD*:

**assumes**

$(g1, a, r1) \in set\ (IArray\ (map\ IArray\ trans\_in\_map)\ !!\ q\ !!\ l)$

$q < p\ l < length\ (trans\ !\ q)$

**obtains** *i where*

$i < length\ (trans\ !\ q\ !\ l) \wedge trans\ !\ q\ !\ l\ !\ i = (g1, In\ a, r1)$

**using** *assms process\_length(2) unfolding trans\_in\_map\_def*

**by** *(fastforce dest: mem\_nth split: act.split\_asm)*

**lemma** *in\_trans\_out\_mapD*:

**assumes**

$(g1, a, r1) \in set\ (IArray\ (map\ IArray\ trans\_out\_map)\ !!\ q\ !!\ l)$

$q < p\ l < length\ (trans\ !\ q)$

**obtains** *i where*

$i < length\ (trans\ !\ q\ !\ l) \wedge trans\ !\ q\ !\ l\ !\ i = (g1, Out\ a, r1)$

**using** *assms process\_length(2) unfolding trans\_out\_map\_def*

**by** *(fastforce dest: mem\_nth split: act.split\_asm)*

**lemma** *in\_actions\_by\_stateI*:

**assumes**

$(g1, a, r1) \in set\ xs\ a < length\ acc$

**shows**

$(q, g1, a, r1) \in set\ (actions\_by\_state\ q\ xs\ acc\ !\ a)$

$\wedge a < length\ (actions\_by\_state\ q\ xs\ acc)$

**using** *assms unfolding actions\_by\_state\_def*

**apply** *(induction xs arbitrary: acc)*

**apply** *(simp; fail)*

**apply** *simp*

**apply** *(erule disjE)*

**apply** *(rule fold\_acc\_preserv*

*[where P =  $\lambda\ acc. (q, g1, a, r1) \in set\ (acc\ !\ a) \wedge a < length\ acc$*

*)*

**apply** *(subst list\_update\_nth\_split; auto)*

by auto

lemma in\_actions\_by\_state\_preserv:

assumes

$(q, g1, a, r1) \in \text{set } (acc ! a) \ a < \text{length } acc$

shows

$(q, g1, a, r1) \in \text{set } (\text{actions\_by\_state } y \ xs \ acc ! a)$

$\wedge a < \text{length } (\text{actions\_by\_state } y \ xs \ acc)$

using assms unfolding actions\_by\_state\_def

apply -

apply (rule fold\_acc\_preserv

[where  $P = \lambda acc. (q, g1, a, r1) \in \text{set } (acc ! a) \wedge a < \text{length } acc$ ])

apply (subst list\_update\_nth\_split; auto)

by auto

lemma length\_actions\_by\_state\_preserv[simp]:

shows  $\text{length } (\text{actions\_by\_state } y \ xs \ acc) = \text{length } acc$

unfolding actions\_by\_state\_def by (auto intro: fold\_acc\_preserv simp: list\_update\_nth\_split)

lemma in\_all\_actions\_by\_stateI:

assumes

$a < na \ q < p \ (g1, a, r1) \in \text{set } (M !! q !! (L ! q))$

shows

$(q, g1, a, r1) \in \text{set } (\text{all\_actions\_by\_state } M \ L ! a)$

unfolding all\_actions\_by\_state\_def

apply (rule fold\_acc\_ev\_preserv

[where  $P = \lambda acc. (q, g1, a, r1) \in \text{set } (acc ! a)$  and  $Q = \lambda acc. a < \text{length } acc$ ,  
THEN conjunct1])

)

apply (rule in\_actions\_by\_state\_preserv[THEN conjunct1])

using assms by (auto intro: in\_actions\_by\_stateI[THEN conjunct1])

lemma actions\_by\_state\_inj:

assumes  $j < \text{length } acc$

shows  $\forall (q, a) \in \text{set } (\text{actions\_by\_state } i \ xs \ acc ! j). (q, a) \notin \text{set } (acc ! j) \longrightarrow i = q$

unfolding actions\_by\_state\_def

apply (rule fold\_acc\_preserv

[where  $P = \lambda acc'. (\forall (q, a) \in \text{set } (acc' ! j). (q, a) \notin \text{set } (acc ! j) \longrightarrow i = q) \wedge j < \text{length } acc'$ ,  
THEN conjunct1])

subgoal for  $x \ acc$

by (cases fst (snd x) = j; simp)

using assms by auto

lemma actions\_by\_state\_inj':

assumes  $j < \text{length } acc \ (q, a) \notin \text{set } (acc ! j) \ (q, a) \in \text{set } (\text{actions\_by\_state } i \ xs \ acc ! j)$

shows  $i = q$

using actions\_by\_state\_inj[OF assms(1)] assms(2-) by fast

lemma in\_actions\_by\_stateD:

assumes

$(q, g, a, t) \in \text{set } (\text{actions\_by\_state } i \ xs \ acc ! j) \ (q, g, a, t) \notin \text{set } (acc ! j)$

$j < \text{length } acc$

**shows**  
 $(g, a, t) \in \text{set } xs \wedge j = a$   
**using** *assms* **unfolding** *actions\_by\_state\_def*  
**apply** –  
**apply** (*drule fold\_evD*  
 $[\text{where } y = (g, a, t) \text{ and } Q = \lambda acc'. \text{length } acc' = \text{length } acc$   
 $\text{and } R = \lambda (\_, a', t). a' = j]$   
 $)$   
**apply** *assumption*  
**apply** (*subst (asm) list\_update\_nth\_split[of j]; force*)  
**apply** *simp* +  
**apply** (*subst (asm) list\_update\_nth\_split[of j]; force*)  
**by** *auto*

**lemma** *in\_all\_actions\_by\_stateD*:  
**assumes**  
 $(q, g1, a, r1) \in \text{set } (\text{all\_actions\_by\_state } M L ! a') \ a' < na$   
**shows**  
 $(g1, a, r1) \in \text{set } (M !! q !! (L ! q)) \wedge q < p \wedge a' = a$   
**using** *assms*  
**unfolding** *all\_actions\_by\_state\_def*  
**apply** –  
**apply** (*drule fold\_evD''[where y = q and Q = \lambda acc. length acc = na]*)  
**apply** (*simp; fail*)  
**apply** (*drule actions\_by\_state\_inj'[rotated]*)  
**apply** (*simp; fail*) +  
**apply** *safe*  
**apply** (*drule in\_actions\_by\_stateD*)  
**apply** *assumption*  
**apply** (*rule fold\_acc\_preserv*)  
**apply** (*simp; fail*) +  
**subgoal** *premises prems*  
**proof** –  
**from** *prems(2)* **have**  $q \in \text{set } [0..<p]$  **by** *auto*  
**then show** *?thesis* **by** *simp*  
**qed**  
**by** (*auto intro: fold\_acc\_preserv dest!: in\_actions\_by\_stateD*)

**lemma** *length\_all\_actions\_by\_state\_preserv*:  
 $\text{length } (\text{all\_actions\_by\_state } M L) = na$   
**unfolding** *all\_actions\_by\_state\_def* **by** (*auto intro: fold\_acc\_preserv*)

**lemma** *less\_naI*:  
**assumes**  
 $q < p$   
 $(g1, a, r1) = \text{trans } ! q ! l ! j$   
 $l < \text{length } (\text{trans } ! q)$   
 $j < \text{length } (\text{trans } ! q ! l)$   
**shows** *pred\_act*  $(\lambda a. a < na) \ a$   
**using** *action\_set assms process\_length(2)* **by** (*force dest!: nth\_mem*)

**lemma** *in\_actions\_trans\_in\_mapI*:  
**assumes**  
 $pa < p$

```

    (g1, In a, r1) = trans ! pa ! (L ! pa) ! j
    L ! pa < length (trans ! pa)
    j < length (trans ! pa ! (L ! pa))
  shows (pa, g1, a, r1) ∈ set (all_actions_by_state (IArray (map IArray trans_in_map))) L !
a)
  apply (rule in_all_actions_by_stateI)
  using assms action_set process_length(2) apply (fastforce dest!: nth_mem)
  using assms by (fastforce intro: in_trans_in_mapI)+

lemma in_actions_trans_out_mapI:
  assumes
    pa < p
    (g1, Out a, r1) = trans ! pa ! (L ! pa) ! j
    L ! pa < length (trans ! pa)
    j < length (trans ! pa ! (L ! pa))
  shows (pa, g1, a, r1) ∈ set (all_actions_by_state (IArray (map IArray trans_out_map))) L
! a)
  apply (rule in_all_actions_by_stateI)
  using assms action_set process_length(2) apply (fastforce dest!: nth_mem)
  using assms by (fastforce intro: in_trans_out_mapI)+

lemma in_pairs_by_actionD2:
  assumes
    (g, a, r, L', s') ∈ set (pairs_by_action (L, s) xs ys)
    ∀ (q, g, a'', m, l) ∈ set xs. a'' = a'
    ∀ (q, g, a'', m, l) ∈ set ys. a'' = a'
  shows check_pred L' s'
  using assms(1) unfolding pairs_by_action_def using assms(2,3)
  by (clarsimp split: option.split_asm simp: set_map_filter) (clarsimp split: if_split_asm)

lemma in_pairs_by_actionD1:
  assumes
    (g, a, r, L', s') ∈ set (pairs_by_action (L, s) xs ys)
    ∀ (q, g, a'', m, l) ∈ set xs. a'' = a'
    ∀ (q, g, a'', m, l) ∈ set ys. a'' = a'
  obtains
    pa q pc_g1 pc_g2 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
    x1 x2 x3 x4 x5 y1 y2 y3 y4
  where
    pa ≠ q
    (pa, pc_g1, a, pc_u1, l1') ∈ set ys
    (q, pc_g2, a, pc_u2, l2') ∈ set xs
    L' = L[pa := l1', q := l2']
    runf pc_u1 s1 = Some ((x1, x2, s', x3, x4), x5)
    runf pc_u2 s = Some ((y1, y2, s1, y3, r2), y4)
    check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
    r1 = make_reset pc_u1 s
    g = g1 @ g2 r = r1 @ r2
proof -
  obtain
    pa q pc_g1 pc_g2 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
    x1 x2 x3 x4 x5 y1 y2 y3 y4
  where
    (q, pc_g1, a, pc_u1, l1') ∈ set ys (pa, pc_g2, a, pc_u2, l2') ∈ set xs q ≠ pa

```

```

  check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
  runf pc_u1 s1 = Some ((x1, x2, s', x3, x4), x5)
  runf pc_u2 s = Some ((y1, y2, s1, y3, r2), y4)
  r1 = make_reset pc_u1 s
  Some (g1 @ g2, a, r1 @ r2, L[q := l1', pa := l2'], s') = Some (g, a, r, L', s')
proof -
  from assms(1) show ?thesis
  unfolding pairs_by_action_def using assms(2,3)
  apply clarsimp
  unfolding set_map_filter
  apply clarsimp
  apply (clarsimp split: option.split_asm if_split_asm)
  by (force intro!: that)
qed
then show ?thesis by (fast intro: that)
qed

lemma in_pairs_by_actionD:
  assumes
    (g, a, r, L', s') ∈ set (pairs_by_action (L, s) xs ys)
    ∀ (q, g, a'', m, l) ∈ set xs. a'' = a'
    ∀ (q, g, a'', m, l) ∈ set ys. a'' = a'
  obtains
    pa q pc_g1 pc_g2 p1 p2 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
    x1 x2 x3 x4 x5 y1 y2 y3 y4
  where
    pa ≠ q
    (pa, pc_g1, a, pc_u1, l1') ∈ set ys
    (q, pc_g2, a, pc_u2, l2') ∈ set xs
    L' = L[pa := l1', q := l2']

    runf pc_u1 s1 = Some ((x1, x2, s', x3, x4), x5)
    runf pc_u2 s = Some ((y1, y2, s1, y3, r2), y4)
    check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
    r1 = make_reset pc_u1 s
    g = g1 @ g2 r = r1 @ r2

    check_pred L' s'

  using in_pairs_by_actionD1[OF assms] in_pairs_by_actionD2[OF assms] by metis

lemma in_trans_funD:
  assumes y ∈ set (trans_fun L)
  shows y ∈ set (trans_s_fun L) ∨ y ∈ set (trans_i_fun L)
  using assms unfolding trans_fun_def by auto

lemma states'_states'[intro]:
  L ∈ equiv.defs.states' s if (L, s) ∈ states'
  using that unfolding states'_def by auto

lemma bounded'_bounded:
  bounded' s ⟷ bounded bounds s if length s = length bounds
  using that unfolding bounded'_def bounded_def by simp

```

**lemma** *bounded\_bounded'*:

*bounded bounds s  $\implies$  bounded' s*

**unfolding** *bounded'\_def bounded\_def* **by** *simp*

**lemma** *P\_unfold*:

$(\forall q < p. (equiv.defs.P ! q) (L ! q) s) \longleftrightarrow (check\_pred L s) \text{ if } length\ s = length\ bounds$

**unfolding** *equiv.state\_ta\_def equiv.state\_pred\_def check\_pred\_def* **using** *process\_length(3)*

*that*

**apply** *simp*

**unfolding** *list\_all\_iff*

**unfolding** *N\_def*

**unfolding** *runf\_def P\_def*

**by** (*fastforce split: option.splits simp: bounded'\_bounded*)

**lemma** *P\_unfold\_1*:

$(\forall q < p. (equiv.defs.P ! q) (L ! q) s) \implies (check\_pred L s)$

**unfolding** *equiv.state\_ta\_def equiv.state\_pred\_def check\_pred\_def* **using** *process\_length(3)*

**apply** *simp*

**unfolding** *list\_all\_iff*

**unfolding** *N\_def*

**unfolding** *runf\_def P\_def*

**by** (*fastforce split: option.split\_asm simp: bounded\_bounded'*)

**lemma** *equiv\_PT[simp]*:

*equiv.PT = PT*

**unfolding** *striptp\_def N\_def* **by** *simp*

**lemmas** [*simp*] = *equiv\_P\_simp*

**lemma** *transD*:

**assumes**

$(pc\_g, a, pc\_u, l') = trans ! q ! (L ! q) ! j$

$L ! q < length\ (trans ! q) \ j < length\ (trans ! q ! (L ! q))$

$q < p$

**shows**  $(L ! q, pc\_g, a, pc\_u, l') \in fst\ (equiv.N ! q)$

**using** *assms* **unfolding** *N\_def T\_def* **by** *simp solve\_ex\_triv*

**lemma** *trans\_ND*:

**assumes**

$(L ! q, pc\_g, a, pc\_u, l') \in fst\ (equiv.N ! q)$

$q < p$

**shows**

$equiv.defs.N\_s\ s ! q \vdash L ! q$

$\longrightarrow equiv.make\_g\ pc\_g\ s, (a, equiv.make\_c\ pc\_g, equiv.make\_mf\ pc\_u), equiv.make\_f\ pc\_u\ s\ l'$

**using** *assms*

**unfolding** *equiv.defs.N\_s\_def trans\_of\_def equiv.defs.T\_s\_def*

**unfolding** *equiv.state\_ta\_def equiv.state\_trans\_t\_def*

**by** *clarsimp solve\_ex\_triv+*

**lemma** *make\_f\_unfold*:

*equiv.make\_f pc s = make\_reset pc s*

**unfolding** *make\_reset\_def equiv.make\_f\_def runf\_def* **by** *simp*

**lemma** *make\_g\_simp*:

```

assumes check_g pc_g s = Some g1
shows g1 = equiv.make_g pc_g s
using assms unfolding check_g_def equiv.make_g_def runt_def
by (clarsimp split: option.splits bool.splits simp: make_cconstr_def)

```

**lemma** *make\_c\_simp:*

```

assumes check_g pc_g s = Some g1
shows equiv.make_c pc_g s
using assms unfolding check_g_def equiv.make_c_def runt_def
by (clarsimp split: option.splits bool.splits simp: make_cconstr_def)

```

**lemma** *make\_reset\_simp:*

```

assumes runf pc_u s = Some ((y1, y2, s1, y3, r2), y4)
shows make_reset pc_u s = r2
using assms unfolding runf_def make_reset_def by (auto split: option.splits)

```

**lemma** *make\_mf\_simp:*

```

assumes runf pc_u s = Some ((y1, y2, s1, y3, r2), y4)
shows equiv.make_mf pc_u s = Some s1
using assms unfolding runf_def equiv.make_mf_def by (auto split: option.splits)

```

**lemma** *trans\_fun\_trans\_of':*

```

(trans_fun, trans_of A)  $\in$  transition_rel states'
unfolding transition_rel_def T_def
apply simp
unfolding trans_of_def
apply safe
subgoal for L s g a r L' s'
  unfolding equiv.defs.prod_ta_def equiv.defs.prod_trans_def
  apply simp
  apply safe
  subgoal
    apply (rule trans_i_fun_trans_fun)
    unfolding equiv.defs.prod_trans_i_alt_def
    apply safe
    unfolding trans_fun_def trans_i_from_def trans_i_fun_def
    unfolding Product_TA_Defs.product_trans_i_def
    apply clarsimp
    subgoal premises prems for c m p' l'
    proof –
      from prems have L ! p' < length (trans ! p') by auto
      from prems obtain pc_g pc_u where
        (L ! p', pc_g, Sil a, pc_u, l')  $\in$  T p'
        g = equiv.make_g pc_g s r = equiv.make_f pc_u s
        c = equiv.make_c pc_g m = equiv.make_mf pc_u
      unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
      unfolding equiv.state_ta_def equiv.state_trans_t_def
      apply clarsimp
      unfolding N_def T_def by clarsimp
      from this(1) have (pc_g, Sil a, pc_u, l')  $\in$  set (trans ! p' ! (L ! p'))
      unfolding T_def by auto
      moreover have check_g pc_g s = Some g
      using <g = _> <c = _> <c s>
      unfolding check_g_def equiv.make_g_def equiv.make_c_def

```

```

    by (auto split: option.splits simp: runt_def make_cconstr_def)
  moreover obtain x1 x2 x3 pcs where runf pc_u s = Some ((x1, x2, s', x3, r), pcs)
    using ⟨r = _⟩ ⟨m = _⟩ prems(5)
    unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
    by (auto split: option.splits)
  moreover have check_pred (L[p' := l']) s'
    using prems(3) by (auto intro: P_unfold_1)
  ultimately show ?thesis using process_length(2) ⟨p' < _⟩ ⟨L ! p' < _⟩
    by (force simp: set_map_filter trans_i_map_def)
qed
done
subgoal
  apply (rule trans_s_fun_trans_fun)
    unfolding equiv.defs.prod_trans_s_alt_def
    apply safe
  unfolding trans_fun_def trans_s_fun_def
  unfolding Product_TA_Defs.product_trans_s_def
  apply clarsimp
  subgoal premises prems for ci co mi mo s1 q1 q2 g1 g2 r1 r2 l1' l2'
  proof -
    from prems have L ! q1 < length (trans ! q1) L ! q2 < length (trans ! q2) by auto
    from prems obtain pc_g1 pc_u1 where
      (L ! q1, pc_g1, In a, pc_u1, l1') ∈ T q1
      g1 = equiv.make_g pc_g1 s r1 = equiv.make_f pc_u1 s
      ci = equiv.make_c pc_g1 mi = equiv.make_mf pc_u1
    unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
    unfolding equiv.state_ta_def equiv.state_trans_t_def
    apply clarsimp
    unfolding N_def T_def by clarsimp
    from prems obtain pc_g2 pc_u2 where
      (L ! q2, pc_g2, Out a, pc_u2, l2') ∈ T q2
      g2 = equiv.make_g pc_g2 s r2 = equiv.make_f pc_u2 s
      co = equiv.make_c pc_g2 mo = equiv.make_mf pc_u2
    unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
    unfolding equiv.state_ta_def equiv.state_trans_t_def
    apply clarsimp
    unfolding N_def T_def by clarsimp
    from ⟨_ ∈ T q1⟩ have (pc_g1, In a, pc_u1, l1') ∈ set (trans ! q1 ! (L ! q1))
      unfolding T_def by auto
    from ⟨_ ∈ T q2⟩ have (pc_g2, Out a, pc_u2, l2') ∈ set (trans ! q2 ! (L ! q2))
      unfolding T_def by auto
    moreover have check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
      using ⟨g1 = _⟩ ⟨ci = _⟩ ⟨ci s⟩ ⟨g2 = _⟩ ⟨co = _⟩ ⟨co s⟩
      unfolding check_g_def equiv.make_g_def equiv.make_c_def
      by (auto split: option.splits simp: runt_def make_cconstr_def)
    moreover obtain x1 x2 x3 pcs where runf pc_u2 s = Some ((x1, x2, s1, x3, r2), pcs)
      using ⟨r2 = _⟩ ⟨mo = _⟩ ⟨Some s1 = _⟩
      unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
      by (auto split: option.splits)
    moreover obtain x1 x2 x3 x4 pcs where runf pc_u1 s1 = Some ((x1, x2, s', x3, x4),
      pcs)
      using ⟨r1 = _⟩ ⟨mi = _⟩ ⟨Some s' = _⟩
      unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
      by (auto split: option.splits)
  end
end

```

```

moreover have  $r1 = \text{make\_reset } pc\_u1 \ s$ 
  using  $\langle r1 = \_ \rangle$  unfolding  $\text{make\_reset\_def equiv.make\_f\_def runf\_def}$  by auto
moreover have  $\text{check\_pred } (L[q1 := l1', q2 := l2']) \ s'$ 
  using  $\text{prems}(5)$  by (auto intro: P\_unfold\_1)
moreover have  $a < na$ 
  using  $\text{action\_set } \langle \_ \in \text{set } (trans ! q1 ! (L ! q1)) \rangle \langle q1 < \_ \rangle \langle L ! q1 < \_ \rangle$ 
     $\text{process\_length}(2)$ 
  by (fastforce dest!: nth\_mem)
moreover have
  ( $q1, pc\_g1, a, pc\_u1, l1'$ )
   $\in \text{set } (\text{all\_actions\_by\_state } (\text{nested\_list\_to\_iarray } trans\_in\_map) \ L ! a)$ 
  using  $\langle L ! q1 < \_ \rangle \langle \_ \in \text{set } (trans ! q1 ! (L ! q1)) \rangle \langle q1 < p \rangle$ 
  by (force intro: in\_actions\_trans\_in\_mapI dest: mem\_nth)
moreover have
  ( $q2, pc\_g2, a, pc\_u2, l2'$ )
   $\in \text{set } (\text{all\_actions\_by\_state } (\text{nested\_list\_to\_iarray } trans\_out\_map) \ L ! a)$ 
  using  $\langle L ! q2 < \_ \rangle \langle \_ \in \text{set } (trans ! q2 ! (L ! q2)) \rangle \langle q2 < p \rangle$ 
  by (force intro: in\_actions\_trans\_out\_mapI dest: mem\_nth)
ultimately show ?thesis
  using  $\text{process\_length}(2) \langle q1 < \_ \rangle \langle L ! q1 < \_ \rangle \langle q2 < \_ \rangle \langle L ! q2 < \_ \rangle \langle \_ \neq \_ \rangle$ 
  unfolding  $\text{pairs\_by\_action\_def}$ 
  apply –
  apply (rule bexI[where  $x = a$ ])
  by auto (force simp: set\_map\_filter)
qed
done
done
subgoal for  $L \ s \ g \ a \ r \ L' \ s'$ 
apply (drule in\_trans\_funD)
apply (erule disjE)
unfolding  $\text{equiv.defs.prod\_ta\_def equiv.defs.prod\_trans\_def}$ 
apply simp
apply (rule disjI2)
subgoal
  unfolding  $\text{equiv.defs.prod\_trans\_s\_alt\_def}$ 
  apply safe
  unfolding  $\text{trans\_s\_fun\_def}$ 
  apply clarsimp
  subgoal for  $x$ 
  apply (erule in\_pairs\_by\_actionD[where  $a' = x$ ])
  apply (solves  $\langle \text{auto dest: in\_all\_actions\_by\_stateD} \rangle$ )
    apply (solves  $\langle \text{auto dest: in\_all\_actions\_by\_stateD} \rangle$ )
    apply (drule in\_all\_actions\_by\_stateD, assumption)
  apply (drule in\_all\_actions\_by\_stateD, assumption)
  apply safe
  apply (erule in\_trans\_in\_mapD)
    apply (simp; fail)
  apply blast
  apply (erule in\_trans\_out\_mapD)
  apply blast
  apply blast
  apply (simp only: ex\_simps[symmetric])
  unfolding  $\text{states'\_def}$ 
  apply (clarsimp)

```

```

    apply (subst P_unfold, assumption)
    apply (subst P_unfold)
    subgoal
      unfolding runf_def by (auto dest!: exec_state_length)
    apply simp
    apply (drule transD[rotated 2], solve_triv+, blast)
    apply (drule transD[rotated 2], solve_triv+, blast)
    apply (drule_tac s = s in trans_ND, assumption)
    apply (drule_tac s = s in trans_ND, assumption)
    unfolding Product_TA_Defs.product_trans_s_def
    apply clarsimp
    unfolding trans_of_def
    subgoal
      apply (simp only: ex_simps[symmetric])
      apply defer_ex
      apply defer_ex
      apply solve_ex_triv
      apply solve_ex_triv
      apply solve_ex_triv
      unfolding make_f_unfold
    by (auto simp add: make_c_simp make_mf_simp make_reset_simp make_g_simp[symmetric])
    done
  done
subgoal
  apply simp
  apply (rule disjI1)
  using process_length(2)
  unfolding equiv.defs.prod_trans_i_alt_def
  apply simp
  unfolding P_unfold
  unfolding trans_i_fun_def trans_i_from_def states'_def
  apply simp
  apply (erule bexE)

  unfolding set_map_filter
  apply simp
  subgoal premises prems for q
  proof -
    from prems have len:  $q < \text{length } \text{trans } L \mid q < \text{length } (\text{trans } ! q)$  by auto
    from prems(4) obtain pc_g pc_u l' x1 x2 x3 x4 where
      (pc_g, a, pc_u, l')  $\in \text{set } (\text{IArray.list\_of } (\text{map IArray trans\_i\_map } ! q) ! (L ! q))$ 
      check_g pc_g s = Some g
      r = make_reset pc_u s
      runf pc_u s = Some ((x1, x2, s', x3, r), x4)
      check_pred (L[q := l']) s'
      L' = L[q := l']
    apply atomize_elim
    unfolding make_reset_def
    by (force split: option.splits if_split_asm)
  moreover then have
    (L, g, (a, Networks.label.Act (equiv.make_c pc_g, equiv.make_mf pc_u)), r, L')
     $\in \text{Product\_TA\_Defs.product\_trans\_i } (\text{equiv.defs.N\_s } s)$ 
  unfolding Product_TA_Defs.product_trans_i_def
  apply clarsimp

```

```

    apply solve_ex_triv
    apply safe
    using prems apply simp
    unfolding trans_i_map_def
    using len ⟨q < p⟩ apply (clarsimp simp: set_map_filter)
      apply (clarsimp split: act.split_asm)
    apply (frule make_c_simp)
    apply (drule mem_nth)
    apply safe
    apply (drule transD[rotated], solve_triv+)
    apply (drule trans_ND)
    apply solve_triv
    apply (subst make_g_simp)
    using ⟨q < p⟩ prems(1) by (auto simp add: make_f_unfold)
ultimately show ?thesis
  apply (subst P_unfold)
  subgoal
    using prems(1) by fast
  apply (subst P_unfold)
  subgoal
    using ⟨runf _ _ = _⟩ prems(1)
    unfolding runf_def by (auto dest!: exec_state_length)
  using prems(1) by (force simp: make_mf_simp dest: make_c_simp)
qed
done
done
done

```

```

lemma transition_rel_mono:
  (a, b) ∈ transition_rel B if (a, b) ∈ transition_rel C B ⊆ C
  using that unfolding transition_rel_def b_rel_def fun_rel_def by auto

```

end

```

context
  Equiv_TA_Defs
begin

```

```

lemma state_set_subs:
  state_set (trans_of (defs.product s''))
  ⊆ {L. length L = defs.p ∧ (∀ q < defs.p. L ! q ∈ State_Networks.state_set (fst (defs.N ! q)))}
  unfolding defs.states'_alt_def[symmetric]
  unfolding defs.N_s_def
  unfolding state_set_def Product_TA_Defs.states_def
  unfolding trans_of_def
  unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
  unfolding Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def
  unfolding defs.T_s_def
  unfolding Product_TA_Defs.states_def trans_of_def
  apply simp
  apply safe
  apply (simp; fail)
  using [[goals_limit = 1]]

```

```

      apply force
      apply force
      apply (force simp: image_iff)
      apply force
      subgoal for x _ _ _ _ _ L p g _ _ _ r l' q
      apply (cases p = q)
      apply (force simp: image_iff)
      apply (force simp: image_iff)
      done
      apply force
      subgoal for _ _ _ _ _ _ _ p q g1 g2 _ _ _ _ _ r1 r2 l1' l2' qa
      apply (cases qa = q)
      apply force
      apply (cases qa = p)
      by (force simp: image_iff)+
      done

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

lemma N_s_state_indep:
  assumes (L ! q, g, a, r, l') ∈ map trans_of (equiv.defs.N_s s) ! q q < p
  obtains g r where (L ! q, g, a, r, l') ∈ map trans_of (equiv.defs.N_s s') ! q
  using assms unfolding trans_of_def equiv.defs.N_s_def equiv.defs.T_s_def by force

lemma fst_product_state_indep:
  fst 'fst (equiv.defs.product s) = fst 'fst (equiv.defs.product s')
  unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
  unfolding Product_TA_Defs.product_trans_s_def Product_TA_Defs.product_trans_i_def
  apply simp
  unfolding equiv.defs.states'_alt_def
  apply clarsimp
  apply (rule equalityI)
  subgoal
    apply (rule subsetI)
    apply clarsimp
    apply (erule disjE)
    apply (erule conjE exE)+
    apply (erule N_s_state_indep)
    apply (simp; fail)
    apply (rule img_fst)
    apply (rule Set.UnI1)
    apply (subst mem_Collect_eq)
    apply solve_ex_triv+
    apply (erule conjE exE)+
    apply (erule N_s_state_indep, (simp; fail))
    apply (erule N_s_state_indep, (simp; fail))
    apply (rule img_fst)
    apply (rule Set.UnI2)
    apply (subst mem_Collect_eq)
    apply (rule exI)+
    apply (rule conjI)

```

```

    defer
    by solve_ex_triv+
subgoal
  apply (rule subsetI)
  apply clarsimp
  apply (erule disjE)
  apply (erule conjE exE)+
  apply (erule N_s_state_indep)
  apply (simp; fail)
  apply (rule img_fst)
  apply (rule Set.UnI1)
  apply (subst mem_Collect_eq)
  apply solve_ex_triv+
  apply (erule conjE exE)+
  apply (erule N_s_state_indep, (simp; fail))
  apply (erule N_s_state_indep, (simp; fail))
  apply (rule img_fst)
  apply (rule Set.UnI2)
  apply (subst mem_Collect_eq)
  apply (rule exI)+
  apply (rule conjI)
  defer
  by solve_ex_triv+
done

```

```

lemma last_product_state_indep:
  (snd o snd o snd o snd) 'fst (equiv.defs.product s)
= (snd o snd o snd o snd) 'fst (equiv.defs.product s')
unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
unfolding Product_TA_Defs.product_trans_s_def Product_TA_Defs.product_trans_i_def
apply simp
unfolding equiv.defs.states'_alt_def
apply clarsimp
apply (rule equalityI)
subgoal
  apply (rule subsetI)
  apply clarsimp
  apply (erule disjE)
  apply (erule conjE exE)+
  apply (erule N_s_state_indep)
  apply (simp; fail)
  apply (rule image_eqI)
  prefer 2
  apply (rule Set.UnI1)
  apply (subst mem_Collect_eq)
  apply solve_ex_triv+
  apply (erule conjE exE)+
  apply (erule N_s_state_indep, (simp; fail))
  apply (erule N_s_state_indep, (simp; fail))
  apply (rule image_eqI)
  defer
  apply (rule Set.UnI2)
  apply (subst mem_Collect_eq)
  apply (rule exI)+

```

```

  apply (rule conjI)
  defer
  apply solve_ex_triv+
  defer
  apply (rule HOL.refl)
  by simp
subgoal
  apply (rule subsetI)
  apply clarsimp
  apply (erule disjE)
  apply (erule conjE exE)+
  apply (erule N_s_state_indep)
  apply (simp; fail)
  apply (rule image_eqI)
  prefer 2
  apply (rule Set.UnI1)
  apply (subst mem_Collect_eq)
  apply solve_ex_triv+
  apply (erule conjE exE)+
  apply (erule N_s_state_indep, (simp; fail))
  apply (erule N_s_state_indep, (simp; fail))
  apply (rule image_eqI)
  defer
  apply (rule Set.UnI2)
  apply (subst mem_Collect_eq)
  apply (rule exI)+
  apply (rule conjI)
  defer
  apply solve_ex_triv+
  defer
  apply (rule HOL.refl)
  by simp
done

```

```

lemma state_set_T':
  state_set (equiv.defs.T' s'')  $\supseteq$  fst ' state_set (trans_of A)
  unfolding trans_of_def
  unfolding state_set_def
  unfolding Prod_TA_Defs.prod_ta_def equiv.defs.prod_trans_def
  apply simp
  unfolding equiv.defs.prod_trans_i_def equiv.defs.prod_trans_s_def
  unfolding trans_of_def
  apply safe
  apply (subst fst_product_state_indep; force)
  apply (subst fst_product_state_indep; force)
  apply (subst (asm) last_product_state_indep[simplified]; force)
  by (subst (asm) last_product_state_indep[simplified]; force)

```

```

lemma state_set_T'2:
  length L = equiv.defs.p
   $\forall q < \text{equiv.defs.p}. L ! q \in \text{State\_Networks.state\_set (fst (equiv.defs.N ! q))}$ 
  if (L, s)  $\in$  state_set (trans_of A)
  using subset_trans [OF state_set_T' equiv.state_set_subs] that by blast+

```

**lemma** *state\_set\_states'*:

$L \in \text{equiv.defs.states}' s$  **if**  $(L, s) \in \text{state\_set} (\text{trans\_of } A)$   
**using** *state\_set\_T'2*[*OF that*] **unfolding** *equiv.defs.states'\_alt\_def* **by** *simp*

**lemma** *state\_set\_pred*:

$\forall q < p. (\text{equiv.defs.P} ! q) (L ! q) s$  **if**  $(L, s) \in \text{state\_set} (\text{trans\_of } A)$   
**using** *that*  
**unfolding** *Normalized\_Zone\_Semantics\_Impl\_Refine.state\_set\_def*  
**unfolding** *trans\_of\_def Prod\_TA\_Defs.prod\_ta\_def Prod\_TA\_Defs.prod\_trans\_def*  
**unfolding** *Prod\_TA\_Defs.prod\_trans\_i\_def Prod\_TA\_Defs.prod\_trans\_s\_def*  
**by** *force*

**end**

**context** *UPPAAL\_Reachability\_Problem\_precompiled'*

**begin**

**lemma** *bounded\_bounded''*:

$\text{bounded bounds } s \implies \text{length } s = \text{length bounds}$   
**unfolding** *bounded'\_def bounded\_def* **by** *simp*

**lemma** *P\_bounded*:

$(\forall q < p. (\text{equiv.defs.P} ! q) (L ! q) s) \implies \text{bounded bounds } s$   
**unfolding** *equiv.state\_ta\_def equiv.state\_pred\_def check\_pred\_def* **using** *process\_length(3)*  
*p\_gt\_0*  
**apply** *simp*  
**unfolding** *list\_all\_iff*  
**unfolding** *N\_def*  
**unfolding** *runf\_def P\_def*  
**apply** (*drule spec[of \_ 0]*)  
**by** (*simp split: option.split; auto split: option.split\_asm*)

**lemma** *P\_state\_length*:

$(\forall q < p. (\text{equiv.defs.P} ! q) (L ! q) s) \implies \text{length } s = \text{length bounds}$   
**by** (*intro P\_bounded bounded\_bounded''*)

**lemma** *state\_set\_state\_length*:

$\text{length } s = \text{length bounds}$  **if**  $(L, s) \in \text{state\_set} (\text{trans\_of } A)$   
**using** *that* **unfolding** *state\_set\_def*  
**apply** (*safe dest!: equiv.defs.prod\_ta\_cases*)  
**unfolding** *equiv.defs.prod\_trans\_i\_alt\_def equiv.defs.prod\_trans\_s\_alt\_def*  
**by** *safe (auto dest: P\_state\_length)*

**lemma** *state\_set\_states*:

$\text{state\_set} (\text{trans\_of } A) \subseteq \text{states}'$   
**using** *state\_set\_states' state\_set\_pred* **unfolding** *states'\_def*  
**by** (*auto intro: P\_unfold\_1 state\_set\_state\_length*)

**lemma** *p\_p\_2[simp]*:

$\text{defs'.defs.p} = p$   
**unfolding** *defs'.p\_p p\_p ..*

**lemma** *len\_product'\_N[simp]*:

```

length defs'.defs.N = p
unfolding defs'.defs.p_def[symmetric] by (rule p_p_2)

lemma len_equiv_N:
  length equiv.defs.N = p
  unfolding equiv.defs.p_def[symmetric] by simp

lemma
  defs'.p = p
  unfolding defs'.p_def by simp

lemma equiv_p_p: equiv.p = p
  by simp

lemma init_states:
  init ∈ equiv.defs.states' s0
  using processes_have_trans start_has_trans
  unfolding equiv_states'_alt_def
  unfolding init_def N_def T_def by force

lemma start_pred':
  check_pred init s0
  using start_pred bounded unfolding check_pred_def runf_def list_all_iff
  by (fastforce split: option.split intro: bounded_bounded')

lemma start_states':
  (init, s0) ∈ states'
  using start_pred' init_states bounded unfolding states'_def bounded_def by auto

lemma trans_fun_trans_of[intro, simp]:
  (trans_fun, trans_of A) ∈ transition_rel states
  using trans_fun_trans_of' state_set_states start_states' unfolding transition_rel_def by
blast

definition
  inv_fun ≡ λ (L, _). concat (map (λ i. IArray (map IArray inv) !! i !! (L ! i)) [0..lemma states_states':
  states ⊆ states'
  using state_set_states start_states' by auto

lemma states'_length[simp]:
  length L = p if (L, s) ∈ states'
  using that unfolding states'_def by auto

lemma inv_simp:
  I q (L ! q) = inv ! q ! (L ! q) if q < p (L, s) ∈ states'
  unfolding I_def using that states'_states'[OF that(2)] lengths by (auto dest!: states_len)

lemma inv_fun_inv_of':
  (inv_fun, inv_of A) ∈ inv_rel Id states'
  unfolding inv_rel_def
  apply (clarsimp simp: equiv.defs.inv_of_simp Product_TA_Defs.inv_of_product)

```

```

using process_length(1)
unfolding inv_fun_def Product_TA_Defs.product_invariant_def init_def
unfolding equiv.defs.N_s_def
apply simp
apply (rule arg_cong[where f = concat])
unfolding inv_of_def Equiv_TA_Defs.state_ta_def apply simp
unfolding equiv.state_inv_def N_def Equiv_TA_Defs.state_inv_def
by (auto simp: inv_simp)

lemma inv_rel_mono:
   $(a, b) \in \text{inv\_rel } Id \ B \text{ if } (a, b) \in \text{inv\_rel } Id \ C \ B \subseteq C$ 
  using that unfolding inv_rel_def b_rel_def fun_rel_def by auto

lemma inv_fun_inv_of[intro, simp]:
   $(\text{inv\_fun}, \text{inv\_of } A) \in \text{inv\_rel } Id \ \text{states}$ 
  using inv_fun_inv_of' states_states' by (rule inv_rel_mono)

definition final_fun  $\equiv \lambda (L, s). \text{hd\_of\_formula } \text{formula } L \ s$ 

lemma final_fun_final':
   $(\text{final\_fun}, (\lambda (l, s). F \ l \ s)) \in \text{inv\_rel } Id \ \text{states'}$ 
  unfolding F_def final_fun_def inv_rel_def list_ex_iff
  by (force dest!: states'_states')

lemma final_fun_final[intro, simp]:
   $(\text{final\_fun}, (\lambda (l, s). F \ l \ s)) \in \text{inv\_rel } Id \ \text{states}$ 
  using final_fun_final' states_states' by (rule inv_rel_mono)

lemma fst_clkp_setD:
  assumes  $(c, d) \in \text{clkp\_set } A \ l$ 
  shows  $c > 0 \ c \leq m \ d \in \text{range } \text{int}$ 
  using assms clock_set consts_nats clkp_set'_subs
  unfolding Nats_def clk_set'_def TA_clkp_set_unfold by force+

lemma init_has_trans:
   $(\text{init}, s_0) \in \text{fst } '(\text{trans\_of } A) \longleftrightarrow \text{trans\_fun } (\text{init}, s_0) \neq []$ 
  apply standard
  using trans_fun_trans_of unfolding transition_rel_def apply force
  using trans_fun_trans_of' start_states' unfolding transition_rel_def by fast

end

context UPPAAL_Reachability_Problem_precompiled'
begin

abbreviation k_i  $\equiv \text{IArray } (\text{map } (\text{IArray } o (\text{map } (\text{IArray } o \text{ map int}))) \ k)$ 

definition
  k_impl  $\equiv \lambda (l, \_). \text{IArray } (\text{map } (\lambda c. \text{Max } \{k\_i \ !! \ i \ !! \ (l \ ! \ i) \ !! \ c \mid i. i < p\}) \ [0..<m+1])$ 

lemma k_impl_alt_def:
  k_impl =
   $(\lambda (l, \_). \text{IArray } (\text{map } (\lambda c. \text{Max } ((\lambda i. k\_i \ !! \ i \ !! \ (l \ ! \ i) \ !! \ c) \ ' \ \{0..<p\})) \ [0..<m+1]))$ 
proof -

```

```

have {i. i < p} = {0..<p}
  by auto
then show ?thesis unfolding k_impl_def setcompr_eq_image by auto
qed

lemma k_length_alt:
   $\forall i < p. \forall j < \text{length } (k ! i). \text{length } (k ! i ! j) = m + 1$ 
  using k_length(1,3) by (auto dest: nth_mem)

lemma Max_int_commute:
   $\text{int } (\text{Max } S) = \text{Max } (\text{int } ' S) \text{ if finite } S \ S \neq \{\}$ 
  apply (rule mono_Max_commute)
  apply standard
  using that by auto

lemma k_impl_k'[intro]:
   $k\_impl (l, s) = IArray (k' (l, s)) \text{ if } (l, s) \in \text{states}'$ 
proof -
  have l_len[simp]:  $l ! i < \text{length } (\text{trans } ! i) \text{ if } i < p \text{ for } i$ 
    using  $\langle i < p \rangle \langle (l, s) \in \_ \rangle$  by auto
  have *:  $k\_i !! i !! (l ! i) !! c = k ! i ! (l ! i) ! c$ 
    if  $c \leq m \ i < p$  for  $c \ i$ 
  proof -
    from k_length_alt that k_length(1,2) have  $\text{length } (k ! i ! (l ! i)) = m + 1$ 
      by auto
    with that k_length process_length(2) processes_have_trans start_has_trans show ?thesis
      unfolding init_def by auto
  qed
show ?thesis
  unfolding k_impl_def k'_def k_fun_def

  apply clarsimp
  apply safe
  subgoal
    apply (subst Max_int_commute)
    subgoal
      by auto
    subgoal
      using p_gt_0 by auto
    apply (rule arg_cong[where f = Max])
    apply safe
    using * apply (solves auto)
    by (solves  $\langle \text{auto simp add: } *[\text{symmetric}] \rangle$ )

  subgoal
    apply (rule Max_eqI)
    apply (solves auto)
    using k_length_alt k_length processes_have_trans k_0 p_gt_0 unfolding init_def
    apply (solves auto)

    using k_length_alt k_length processes_have_trans k_0 p_gt_0 unfolding init_def
    apply clarsimp
    apply (rule exI[where x = 0])
    by simp

```

```

subgoal
  apply (subst Max_int_commute)
subgoal
  by auto
subgoal
  using p_gt_0 by auto
  apply (rule arg_cong[where f = Max])
  apply safe
  using * apply (solves auto)
  by (solves ⟨auto simp add: *[symmetric]⟩)
done
qed

lemma k_impl_k'2[intro]:
  k_impl (l, s) = IArray (k' (l, s)) if
  (l, s) ∈ Normalized_Zone_Semantics_Impl_Refine.state_set (trans_of A)
  using that states_states' by auto

lemma k_impl_k'_0[intro]:
  k_impl (init, s0) = IArray (k' (init, s0))
  using states_states' by auto

sublocale impl:
  Reachability_Problem_Impl
  where trans_fun = trans_fun
  and trans_impl = trans_fun
  and inv_fun = inv_fun
  and F_fun = final_fun
  and ceiling = k_impl
  and A = A
  and l0 = (init, s0)
  and l0i = (init, s0)
  and F = PR_CONST ((λ (l, s). F l s))
  and n = m
  and k = k_fun
  and loc_rel = Id
  and show_clock = show
  and show_state = show
  and states' = states
  unfolding PR_CONST_def
  apply standard
  apply (fastforce simp: inv_rel_def b_rel_def)
subgoal
  by auto (metis IdI list_rel_id_simp relAPP_def)
  by (fastforce simp: inv_rel_def b_rel_def)+

lemma F_reachable_correct':
  impl.op.F_reachable
  ↔ (∃ L' s' u u'.
    conv_A A ⊢' ⟨(init, s0), u⟩ →* ⟨(L', s'), u⟩
    ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
  ) if formula = formula.EX φ
  using that E_op''.E_from_op_reachability_check[of F_rel PR_CONST (λ(x, y). F x y),

```

```

    unfolded F_rel_def, OF HOL.refl]
  reachability_check
unfolding impl.E_op F_reachable E_op''.F_reachable_def E_op''.reachable_def
unfolding F_rel_def unfolding F_def by force

```

```

lemma PT_PT:
  defs'.PT = equiv.PT
apply simp
unfolding striptp_def
apply (rule ext)
apply clarsimp
subgoal for x
  apply (cases PROG x)
  apply (simp; fail)
  subgoal for a
    by (cases a) auto
  done
done

```

```

lemma P_P[simp]:
  defs'.defs.P = equiv.defs.P
unfolding Equiv_TA_Defs.state_ta_def
unfolding Equiv_TA_Defs.p_def
unfolding Equiv_TA_Defs.state_pred_def
using PF_PF by (auto split: option.split)

```

```

lemma map_map_filter:
  map f (List.map_filter g xs) = List.map_filter (map_option f o g) xs
by (induction xs; simp add: List.map_filter_simps split: option.split)

```

```

lemma make_g_conv:
  defs'.make_g = conv_cc oo equiv.make_g
unfolding Equiv_TA_Defs.make_g_def PT_PT PF_PF apply simp
apply (intro ext)
apply (clarsimp split: option.splits simp: map_map_filter)
apply (rule arg_cong2[where f = List.map_filter])
by (auto split: option.split instrc.split)

```

```

lemma make_c_conv:
  defs'.make_c = equiv.make_c
unfolding Equiv_TA_Defs.make_c_def PT_PT by simp

```

```

lemma make_f_conv:
  defs'.make_f = equiv.make_f
unfolding Equiv_TA_Defs.make_f_def PF_PF by simp

```

```

lemma make_mf_conv:
  defs'.make_mf = equiv.make_mf
unfolding Equiv_TA_Defs.make_mf_def PF_PF by simp

```

```

lemmas make_convs = make_g_conv make_c_conv make_f_conv make_mf_conv

```

```

lemma state_trans_conv:
  Equiv_TA_Defs.state_trans_t (conv N) max_steps q

```

```

= (λ (a, b, c). (a, λ x. conv_cc (b x), c)) ‘equiv.state_trans q if ⟨q < p⟩
unfolding Equiv_TA_Defs.state_trans_t_def image_Collect
using ⟨q < _⟩ make_convs by (force split: prod.splits)+

```

```

lemma map_conv_t:
  map_trans_of (defs'.defs.N_s s) ! q = conv_t ‘(map_trans_of (equiv.defs.N_s s) ! q)
  if ⟨q < p⟩
  using ⟨q < p⟩
  apply (subst nth_map)
  unfolding defs'.defs.N_s_length p_p_2
  apply assumption
  apply (subst nth_map)
  unfolding equiv.defs.N_s_length
  apply simp
  unfolding trans_of_def Prod_TA_Defs.N_s_def
  unfolding len_equiv_N len_product'_N
  apply simp
  unfolding Prod_TA_Defs.T_s_def
  unfolding image_Collect
  unfolding Equiv_TA_Defs.state_ta_def Equiv_TA_Defs.p_def
  apply simp
  using state_trans_conv[of q]
  apply simp
  apply safe
  apply force
  apply solve_ex_triv+
  unfolding image_iff by force

```

```

lemma product_trans_t_conv:
  Product_TA_Defs.product_trans_s (defs'.defs.N_s s)
  = conv_t ‘Product_TA_Defs.product_trans_s (equiv.defs.N_s s)
  unfolding Product_TA_Defs.product_trans_s_def
  apply (simp only: states'_conv)
  apply safe
  apply (simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
  unfolding image_Collect
  apply (simp split: prod.split)
  apply safe
  subgoal
    by defer_ex solve_ex_triv+
  subgoal
    by solve_ex_triv+ (force simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
  done

```

```

lemma product_trans_t_conv':
  Product_TA_Defs.product_trans_i (defs'.defs.N_s s)
  = conv_t ‘Product_TA_Defs.product_trans_i (equiv.defs.N_s s)
  unfolding Product_TA_Defs.product_trans_i_def
  apply (simp only: states'_conv)
  apply safe
  apply (simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
  unfolding image_Collect
  apply (simp split: prod.split)
  apply safe

```

```

subgoal
  by defer_ex solve_ex_triv+
subgoal
  by solve_ex_triv+ (force simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
done

```

```

lemma prod_trans_s_conv:
  defs'.defs.prod_trans_s = conv_t 'equiv.defs.prod_trans_s
  unfolding defs'.defs.prod_trans_s_alt_def
  unfolding equiv.defs.prod_trans_s_alt_def
  unfolding product_trans_t_conv
  unfolding p_p_2 P_P
  apply simp
  apply safe
  unfolding p_p P_P
  apply (simp add: image_Collect)
  apply solve_ex_triv
subgoal
  apply defer_ex
  apply defer_ex
  by solve_ex_triv+
subgoal
  apply defer_ex
  apply defer_ex
  by solve_ex_triv+
done

```

```

lemma prod_trans_i_conv:
  defs'.defs.prod_trans_i = conv_t 'equiv.defs.prod_trans_i
  unfolding defs'.defs.prod_trans_i_alt_def
  unfolding equiv.defs.prod_trans_i_alt_def
  unfolding product_trans_t_conv'
  unfolding p_p_2 P_P
  apply simp
  apply safe
  unfolding p_p P_P
  apply (simp add: image_Collect)
  apply solve_ex_triv
subgoal
  apply defer_ex
  apply defer_ex
  by solve_ex_triv+
subgoal
  apply defer_ex
  apply defer_ex
  by solve_ex_triv+
done

```

```

lemma prod_trans_conv:
  defs'.defs.prod_trans = conv_t 'equiv.defs.prod_trans
  unfolding defs'.defs.prod_trans_def
  unfolding equiv.defs.prod_trans_def
  unfolding prod_trans_s_conv
  unfolding prod_trans_i_conv image_Un ..

```

```

lemma prod_invariant_conv:
  defs'.defs.prod_invariant = (map conv_ac  $\circ\circ$  Prod_TA_Defs.prod_invariant) EA
  apply (rule ext)
  apply safe
  unfolding defs'.defs.prod_invariant_def equiv.defs.prod_invariant_def
  unfolding Product_TA_Defs.product_ta_def inv_of_def
  apply simp
  unfolding Product_TA_Defs.product_invariant_def List.map_concat
  apply (simp add: Prod_TA_Defs.N_s_length)
  unfolding Equiv_TA_Defs.p_p Equiv_TA_Defs.p_def apply simp
  apply (rule cong[where f = concat])
  apply (rule HOL.refl)
  unfolding Prod_TA_Defs.N_s_def inv_of_def Equiv_TA_Defs.state_ta_def
  unfolding Equiv_TA_Defs.p_def unfolding Equiv_TA_Defs.state_inv_def
  by (simp split: prod.split)

```

```

lemma prod_conv: defs'.defs.prod_ta = conv_A A
  unfolding defs'.defs.prod_ta_def
  unfolding equiv.defs.prod_ta_def
  unfolding conv_A_def
  by (simp add: prod_invariant_conv[symmetric] prod_trans_conv[symmetric])

```

```

lemma F_reachable_correct:
  impl.op.F_reachable
   $\longleftrightarrow (\exists L' s' u u'.
    \text{conv } N \vdash_m ax\_steps \langle init, s_0, u \rangle \rightarrow^* \langle L', s', u' \rangle$ 
   $\wedge (\forall c \in \{1..m\}. u\ c = 0) \wedge check\_bexp\ \varphi\ L'\ s'$ 
   $)$  if formula = formula.EX  $\varphi$  start_inv_check
  unfolding F_reachable_correct'[OF that(1)]
  apply (subst product'.prod_correct[symmetric])
  using prod_conv p_p p_gt_0 apply simp
  using prod_conv p_p p_gt_0 apply simp
  using F_reachable_equiv[OF that(2)]
  by (simp add: F_def, simp add: that(1))

```

**definition**

```

reachability_checker_old  $\equiv$ 
  worklist_algo2_impl
  impl.subsumes_impl impl.a0_impl impl.F_impl impl.succs_impl impl.emptyness_check_impl

```

**definition**

```

reachability_checker'  $\equiv$ 
  pw_impl
  (return o fst) impl.state_copy_impl impl.tracei_impl.subsumes_impl impl.a0_impl impl.F_impl
  impl.succs_impl impl.emptyness_check_impl

```

**theorem** *reachability\_check'*:

```

(uncurry0 reachability_checker',
  uncurry0 (
    Refine_Basic.RETURN ( $\exists L' s' u u'.
      \text{conv}_A\ A \vdash' \langle (init, s_0), u \rangle \rightarrow^* \langle (L', s'), u' \rangle$ 
 $\wedge (\forall c \in \{1..m\}. u\ c = 0) \wedge check\_bexp\ \varphi\ L'\ s'$ 
    )
  )

```

```

)
)
 $\in \text{unit\_assn}^k \rightarrow_a \text{bool\_assn}$  if  $\text{formula} = \text{formula.EX } \varphi$ 
using  $\text{impl.pw\_impl\_hnr\_F\_reachable}$ 
unfolding  $\text{reachability\_checker'}\_def$   $\text{F\_reachable\_correct'}$ [OF that] .

```

**corollary**  $\text{reachability\_checker'}\_hoare$ :

```

<emp>  $\text{reachability\_checker'}$ 
< $\lambda r. \uparrow(r = (\exists L' s' u u'. \text{conv}_A A \vdash' \langle \text{init}, s_0 \rangle, u) \rightarrow^* \langle (L', s'), u \rangle$ 
 $\wedge (\forall c \in \{1..m\}. u c = 0) \wedge \text{check\_bexp } \varphi L' s'$ 
 $\rangle)$ 
> $_t$  if  $\text{formula} = \text{formula.EX } \varphi$ 
apply ( $\text{rule cons\_post\_rule}$ )
using  $\text{reachability\_check'}$ [OF that, to_hnr] apply ( $\text{simp add: hn\_refine\_def}$ )
by ( $\text{sep\_auto simp: pure\_def}$ )

```

**definition**  $\text{reachability\_checker}$  **where**

```

 $\text{reachability\_checker} \equiv \text{do}$ 
{
   $\text{init\_sat} \leftarrow \text{impl.start\_inv\_check\_impl}$ ;
  if  $\text{init\_sat}$  then do
    {  $x \leftarrow \text{reachability\_checker'}$ ;
      return (if  $x$  then  $\text{REACHABLE}$  else  $\text{UNREACHABLE}$ )
    }
  else
    return  $\text{INIT\_INV\_ERR}$ 
}

```

**theorem**  $\text{reachability\_check}$ :

```

( $\text{uncurry0 reachability\_checker}$ ,
 $\text{uncurry0}$  (
   $\text{Refine\_Basic.RETURN}$  (
    if  $\text{start\_inv\_check}$ 
    then
      if
        (
           $\exists L' s' u u'. \text{conv } N \vdash_m \text{ax\_steps } \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle (L', s'), u \rangle$ 
           $\wedge (\forall c \in \{1..m\}. u c = 0) \wedge \text{check\_bexp } \varphi L' s'$ 
        )
        then  $\text{REACHABLE}$ 
        else  $\text{UNREACHABLE}$ 
      else  $\text{INIT\_INV\_ERR}$ 
    )
  )
))
 $\in \text{unit\_assn}^k \rightarrow_a \text{id\_assn}$  if  $\text{formula} = \text{formula.EX } \varphi$ 
apply ( $\text{simp only: F\_reachable\_correct'}$ [OF that, symmetric]  $\text{cong: if\_cong}$ )
supply
   $\text{impl.pw\_impl\_hnr\_F\_reachable}$ 
  [ $\text{unfolded reachability\_checker'}\_def$ [symmetric], to_hnr,  $\text{unfolded hn\_refine\_def}$ ,
   $\text{rule\_format}$ ,  $\text{sep\_heap\_rules}$ ]
supply
   $\text{impl.start\_inv\_check\_impl.refine}$ [to_hnr,  $\text{unfolded hn\_refine\_def}$ ,  $\text{rule\_format}$ ,  $\text{sep\_heap\_rules}$ ]

```

**unfolding** *reachability\_checker\_def*  
**by** *sepref\_to\_hoare* (*sep\_auto simp: pure\_def*)

**corollary** *reachability\_checker\_hoare*:

*<emp> reachability\_checker*  
*<λ r. ↑(r =*  
 (  
*if start\_inv\_check*  
*then*  
*if*  
 (  
 $\exists L' s' u u'.$   
 $conv\ N \vdash_m ax\_steps\ \langle init, s_0, u \rangle \rightarrow^* \langle L', s', u' \rangle$   
 $\wedge (\forall c \in \{1..m\}. u\ c = 0) \wedge check\_bexp\ \varphi\ L'\ s'$   
 )  
*then REACHABLE*  
*else UNREACHABLE*  
*else INIT\_INV\_ERR*  
 )  
 )  
*><sub>t</sub> if formula = formula.EX φ*  
**apply** (*rule cons\_post\_rule*)  
**using** *reachability\_check*[*OF that, to\_hnr*] **apply** (*simp add: hn\_refine\_def*)  
**by** (*sep\_auto simp: pure\_def*)

**lemma** *list\_all\_concat*:

*list\_all Q (concat xs)  $\longleftrightarrow$  ( $\forall xs \in set\ xs. list\_all\ Q\ xs$ )*  
**unfolding** *list\_all\_iff* **by** *auto*

**lemma** *inv\_of\_init\_unfold*:

$u \vdash inv\_of\ (conv\_A\ A)\ (init, s_0) \longleftrightarrow (\forall i < p. u \vdash conv\_cc\ (inv\ !\ i\ 0))$

**proof** –

**have** \*:  $inv\_of\ (conv\_A\ A)\ (init, s_0) = conv\_cc\ (equiv.defs.I'\ s_0\ init)$   
**using** *equiv.defs.inv\_of\_simp*[*of init s<sub>0</sub>*]  
**unfolding** *inv\_of\_def conv\_A\_def* **by** (*auto split: prod.split*)  
**have**  $u \vdash inv\_of\ (conv\_A\ A)\ (init, s_0) \longleftrightarrow (\forall i < p. u \vdash conv\_cc\ (I\ i\ 0))$   
**unfolding** \* *Product\_TA\_Defs.inv\_of\_product Product\_TA\_Defs.product\_invariant\_def*  
**apply** (*simp only: product'.prod.length\_L p\_p\_2 cong: list.map\_cong\_simp*)  
**unfolding** *equiv.defs.N\_s\_def length\_N*  
**apply** (*simp cong: list.map\_cong\_simp*)  
**unfolding** *inv\_of\_def*  
**apply** (*simp cong: list.map\_cong\_simp*)  
**unfolding** *init\_def*  
**apply** (*simp cong: list.map\_cong\_simp*)  
**unfolding** *Equiv\_TA\_Defs.state\_ta\_def*  
**apply** (*simp cong: list.map\_cong\_simp*)  
**unfolding** *equiv.state\_inv\_def*  
**unfolding** *N\_def*  
**by** (*force simp: map\_concat list\_all\_concat clock\_val\_def cong: list.map\_cong\_simp*)  
**also have**  $(\forall i < p. u \vdash conv\_cc\ (I\ i\ 0)) \longleftrightarrow (\forall i < p. u \vdash conv\_cc\ (inv\ !\ i\ 0))$   
**unfolding** *I\_def* **using** *lengths\_processes\_have\_trans* **by** *fastforce*  
**finally show** *?thesis* .  
**qed**

**corollary** *reachability\_checker\_hoare'*:

*<emp> reachability\_checker*

*<λ r. ↑(r =*

*(*  
*if (∀ u. (∀ c ∈ {1..m}. u c = 0) → (∀ i < p. u ⊢ conv\_cc (inv ! i ! 0)))*  
*then*  
*if*  
*(*  
*∃ L' s' u u'.*  
*conv N ⊢<sub>m</sub> ax\_steps ⟨init, s<sub>0</sub>, u⟩ →\* ⟨L', s', u'⟩*  
*∧ (∀ c ∈ {1..m}. u c = 0) ∧ check\_bexp φ L' s'*  
*)*  
*then REACHABLE*  
*else UNREACHABLE*  
*else INIT\_INV\_ERR*  
*)*  
*)*

*><sub>t</sub> if formula = formula.EX φ*

**using** *reachability\_checker\_hoare*[OF that] **unfolding** *start\_inv\_check\_correct inv\_of\_init\_unfold*

**Post-processing** **schematic\_goal** *succs\_impl\_alt\_def*:

*impl.succs\_impl ≡ ?impl*

**unfolding** *impl.succs\_impl\_def*

**unfolding** *k\_impl\_alt\_def*

**apply** (*abstract let*

*λ (l, \_ :: int list). IArray (map (λ c. MAX i ∈ {0..<p}. k\_i !! i !! (l ! i) !! c) [0..<m+1])*  
*k\_i*

*)*

**apply** (*abstract let inv\_fun :: nat list × int list ⇒ (nat, int) acconstraint list inv\_fun*)

**apply** (*abstract let trans\_fun trans\_fun*)

**unfolding** *inv\_fun\_def*[*abs\_def*] *trans\_fun\_def*[*abs\_def*] *trans\_s\_fun\_def* *trans\_i\_fun\_def*

*trans\_i\_from\_def*

**apply** (*abstract let IArray (map IArray inv) inv*)

**apply** (*abstract let IArray (map IArray trans\_out\_map) trans\_out\_map*)

**apply** (*abstract let IArray (map IArray trans\_in\_map) trans\_in\_map*)

**apply** (*abstract let IArray (map IArray trans\_in\_map) trans\_in\_map*)

**by** (*rule Pure.reflexive*)

**lemma** *reachability\_checker'\_alt\_def*:

*reachability\_checker' ≡*

*let*

*key = return ∘ fst;*

*sub = impl.subsumes\_impl;*

*copy = impl.state\_copy\_impl;*

*start = impl.a<sub>0</sub>\_impl;*

*final = impl.F\_impl;*

*succs = impl.succs\_impl;*

*empty = impl.emptyness\_check\_impl;*

*trace = impl.tracei*

*in pw\_impl key copy trace sub start final succs empty*

**unfolding** *reachability\_checker'\_def* **by** *simp*

```

schematic_goal reachability_checker_alt_def:
  reachability_checker  $\equiv$  ?impl
  unfolding reachability_checker_def
  unfolding reachability_checker'_alt_def' impl.succs_impl_def
  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
  unfolding k_impl_alt_def
  apply (abstract_let k_i k_i)
  apply (abstract_let inv_fun :: nat list  $\times$  int list  $\Rightarrow$  (nat, int) acconstraint list )
  apply (abstract_let trans_fun trans_fun)
  unfolding impl.init_dbm_impl_def impl.a0_impl_def
  unfolding impl.F_impl_def
  unfolding final_fun_def[abs_def]
  unfolding impl.subsumes_impl_def
  unfolding impl.emptyness_check_impl_def
  unfolding impl.state_copy_impl_def
  by (rule Pure.reflexive)

```

**end**

```

lemmas [code] = UPPAAL_Reachability_Problem_precompiled'.k_impl_def

```

**Some post refinements**   **code\_thms** fw\_upd'

**code\_thms** fw\_impl'

**code\_thms** fw\_impl

**abbreviation** plus\_int :: int  $\Rightarrow$  int  $\Rightarrow$  int **where**

plus\_int a b  $\equiv$  a + b

**fun** dbm\_add\_int :: int DBMEntry  $\Rightarrow$  int DBMEntry  $\Rightarrow$  int DBMEntry

**where**

```

  dbm_add_int  $\infty$  _ =  $\infty$  |
  dbm_add_int _  $\infty$  =  $\infty$  |
  dbm_add_int (Le a) (Le b) = (Le (plus_int a b)) |
  dbm_add_int (Le a) (Lt b) = (Lt (plus_int a b)) |
  dbm_add_int (Lt a) (Le b) = (Lt (plus_int a b)) |
  dbm_add_int (Lt a) (Lt b) = (Lt (plus_int a b))

```

**lemma** dbm\_add\_int:

dbm\_add = dbm\_add\_int

**apply** (rule ext)+

**subgoal for** x y

by (cases x; cases y) auto

**done**

**definition**

```

  fw_upd'_int m k i j =
    Refine_Basic.RETURN
      (op_mtx_set m (i, j)
        (min (op_mtx_get m (i, j)) (dbm_add_int (op_mtx_get m (i, k)) (op_mtx_get m (k,
          j)))))

```

**definition**

```

fw_upd_impl_int n ≡ λai bib bia bi. do {
  xa ← mtx_get (Suc n) ai (bia, bib);
  xb ← mtx_get (Suc n) ai (bib, bi);
  x ← mtx_get (Suc n) ai (bia, bi);
  let e = (dbm_add_int xa xb);
  if e < x then mtx_set (Suc n) ai (bia, bi) e else Heap_Monad.return ai
}

```

**lemma** *fw\_upd\_impl\_int\_eq*:  
*fw\_upd\_impl\_int* = *fw\_upd\_impl*  
**unfolding** *fw\_upd\_impl\_int\_def* *fw\_upd\_impl\_def*  
**unfolding** *dbm\_add\_int* *add*  
**unfolding** *Let\_def* ..

**definition**  
*fw\_impl\_int* n ≡  
*imp\_for'* 0 (n + 1)  
(λxb. *imp\_for'* 0 (n + 1)  
(λxd. *imp\_for'* 0 (n + 1) (λxf σ'''''. *fw\_upd\_impl\_int* n σ'''''. xb xd xf)))

**lemma** *fw\_impl'\_int*:  
*fw\_impl* = *fw\_impl\_int*  
**unfolding** *fw\_impl\_def* *fw\_impl\_int\_def*  
**unfolding** *fw\_upd\_impl\_int\_eq* ..

**context** *UPPAAL\_Reachability\_Problem\_precompiled\_defs'*  
**begin**

**definition** *run\_impl* program pc s ≡ *exec* program *max\_steps* (pc, [], s, True, []) []

**lemma** *runf\_impl*:  
*runf* = *run\_impl* PF  
**unfolding** *runf\_def* *run\_impl\_def* ..

**lemma** *runt\_impl*:  
*runt* = *run\_impl* PT  
**unfolding** *runt\_def* *run\_impl\_def* ..

**definition**  
*make\_constr\_impl* program pcs =  
*List.map\_filter*  
(λpc. case program pc of None ⇒ None | Some (INSTR x) ⇒ *Map.empty* x  
| Some (CEXP ac) ⇒ Some ac)  
pcs

**lemma** *make\_constr\_impl*:  
*make\_constr* = *make\_constr\_impl* PROG  
**unfolding** *make\_constr\_def* *make\_constr\_impl\_def* ..

**definition**  
*check\_g\_impl* programf program pc s ≡  
case *run\_impl* programf pc s of None ⇒ None  
| Some ((x, xa, xb, True, xc), pcs) ⇒ Some (*make\_constr\_impl* program pcs)  
| Some ((x, xa, xb, False, xc), pcs) ⇒ None

**lemma** *check\_g\_impl*:  
*check\_g* = *check\_g\_impl* *PT PROG'*  
**unfolding** *check\_g\_impl\_def check\_g\_def runt\_impl PROG'\_PROG make\_cconstr\_impl ..*

**definition**  
*make\_reset\_impl* *program m1 s*  $\equiv$   
*case run\_impl* *program m1 s* of  
*Some* ((\_, \_, \_, \_, *r1*), \_)  $\Rightarrow$  *r1*  
| *None*  $\Rightarrow$  []

**lemma** *make\_reset\_impl*:  
*make\_reset* = *make\_reset\_impl* *PF*  
**unfolding** *make\_reset\_def make\_reset\_impl\_def runf\_impl ..*

**definition**  
*check\_pred\_impl* *program bnds L s*  $\equiv$   
*list\_all*  
( $\lambda q$ . *case run\_impl* *program* (*pred* ! *q* ! (*L* ! *q*)) *s* of *None*  $\Rightarrow$  *False*  
| *Some* ((*x*, *xa*, *xb*, *f*, *xc*), *xd*)  $\Rightarrow$   
*f*  $\wedge$  ( $\forall i < \text{length } s. \text{fst } (\text{bnds} !! i) < s ! i \wedge s ! i < \text{snd } (\text{bnds} !! i)$ ))  
[0..*p*])

**lemma** *check\_pred\_impl*:  
*check\_pred* = *check\_pred\_impl* *PF (IArray bounds)*  
**unfolding** *check\_pred\_def check\_pred\_impl\_def runf\_impl bounded'\_def ..*

**definition**  
*pairs\_by\_action\_impl* *pf pt porig bnds*  $\equiv$   $\lambda (L, s)$  *OUT. concat o*  
*map* ( $\lambda (i, g1, a, m1, l1). \text{List.map\_filter}$   
( $\lambda (j, g2, a, m2, l2).$   
*if* *i = j* *then None* *else*  
*case* (*check\_g\_impl* *pt porig g1 s, check\_g\_impl pt porig g2 s*) of  
(*Some* *cc1*, *Some* *cc2*)  $\Rightarrow$   
(*case run\_impl* *pf m2 s* of  
*Some* ((\_, \_, *s1*, \_, *r2*), \_)  $\Rightarrow$   
(*case run\_impl* *pf m1 s1* of  
*Some* ((\_, \_, *s'*, \_, \_), \_)  $\Rightarrow$   
*if* *check\_pred\_impl pf bnds* (*L*[*i* := *l1*, *j* := *l2*]) *s'*  
*then* *Some* (*cc1* @ *cc2*, *a*, *make\_reset\_impl pf m1 s* @ *r2*, (*L*[*i* := *l1*, *j* := *l2*], *s'*))  
*else None*  
| \_  $\Rightarrow$  *None*)  
| \_  $\Rightarrow$  *None*)  
| \_  $\Rightarrow$  *None*  
)  
*OUT*)

**lemma** *pairs\_by\_action\_impl*:  
*pairs\_by\_action* = *pairs\_by\_action\_impl* *PF PT PROG' (IArray bounds)*  
**unfolding** *pairs\_by\_action\_def pairs\_by\_action\_impl\_def*  
**unfolding** *check\_g\_impl make\_reset\_impl check\_pred\_impl runf\_impl ..*

**definition**

*all\_actions\_by\_state\_impl* *upt\_p empty\_ran i L*  $\equiv$   
*fold* ( $\lambda ia. \text{actions\_by\_state } ia \ (i \ !! \ ia \ !! \ (L \ ! \ ia))$ ) *upt\_p empty\_ran*

**lemma** *all\_actions\_by\_state\_impl*:

*all\_actions\_by\_state* = *all\_actions\_by\_state\_impl* [0..*p*] (*repeat* [] *na*)  
**unfolding** *all\_actions\_by\_state\_def* *all\_actions\_by\_state\_impl\_def* ..

**definition**

*trans\_i\_from\_impl* *programf programt program bnds trans\_i\_array*  $\equiv$   
 $\lambda(L, s) \ i.$   
*List.map\_filter*  
 $(\lambda(g, a, m, l').$   
 $\text{case check\_g\_impl } programt \ program \ g \ s \text{ of } None \Rightarrow None$   
 $| \text{Some } cc \Rightarrow$   
 $(\text{case run\_impl } programf \ m \ s \text{ of } None \Rightarrow None$   
 $| \text{Some } ((xx, xa, s', xb, r), xc) \Rightarrow$   
 $\text{if check\_pred\_impl } programf \ (IArray \ bnds) \ (L[i := l']) \ s'$   
 $\text{then Some } (cc, a, r, L[i := l'], s') \text{ else None}))$   
 $(trans\_i\_array \ !! \ i \ !! \ (L \ ! \ i))$

**lemma** *trans\_i\_from\_impl*:

*trans\_i\_from* = *trans\_i\_from\_impl* *PF PT PROG'* (*IArray bnds*) (*IArray* (*map* *IArray* *trans\_i\_map*))  
**unfolding** *trans\_i\_from\_def* *trans\_i\_from\_impl\_def*  
**unfolding** *check\_g\_impl* *runf\_impl* *check\_pred\_impl* ..

**end****context** *UPPAAL\_Reachability\_Problem\_precompiled'***begin****lemma** *PF\_alt\_def*:

*PF* = ( $\lambda pc. \text{if } pc < \text{length } prog \text{ then } (IArray \ (\text{map} \ (\text{map\_option } stripf) \ prog)) \ !! \ pc \text{ else } None$ )  
**unfolding** *stripfp\_def* *PROG'\_def* **by** *auto*

**lemma** *PT\_alt\_def*:

*PT* = ( $\lambda pc. \text{if } pc < \text{length } prog \text{ then } (IArray \ (\text{map} \ (\text{map\_option } stript) \ prog)) \ !! \ pc \text{ else } None$ )  
**unfolding** *striptp\_def* *PROG'\_def* **by** *auto*

**schematic\_goal** *reachability\_checker\_alt\_def\_refined*:

*reachability\_checker*  $\equiv$  *?impl*  
**unfolding** *reachability\_checker\_alt\_def*  
**unfolding** *fw\_impl'\_int*  
**unfolding** *inv\_fun\_def* *trans\_fun\_def* *trans\_s\_fun\_def* *trans\_i\_fun\_def*  
**unfolding** *trans\_i\_from\_impl*  
**unfolding** *runf\_impl* *runt\_impl* *check\_g\_impl* *pairs\_by\_action\_impl* *check\_pred\_impl*  
**apply** (*abstract\_let* *IArray* (*map* *IArray* *inv*) *inv*)  
**apply** (*abstract\_let* *IArray* (*map* *IArray* *trans\_out\_map*) *trans\_out\_map*)  
**apply** (*abstract\_let* *IArray* (*map* *IArray* *trans\_in\_map*) *trans\_in\_map*)  
**apply** (*abstract\_let* *IArray* (*map* *IArray* *trans\_i\_map*) *trans\_i\_map*)  
**apply** (*abstract\_let* *IArray* *bnds* *bnds*)  
**apply** (*abstract\_let* *PF* *PF*)

```

apply (abstract_let PT PT)
unfolding PF_alt_def PT_alt_def
apply (abstract_let PROG' PROG')
unfolding PROG'_def
apply (abstract_let length prog len_prof)
apply (abstract_let IArray (map (map_option stripf) prog) prog_f)
apply (abstract_let IArray (map (map_option stript) prog) prog_t)
apply (abstract_let IArray prog prog)
unfolding all_actions_by_state_impl
apply (abstract_let [0..apply (abstract_let [0..apply (abstract_let {0..apply (abstract_let [0..by (rule Pure.reflexive)

```

**end**

## 5.2 Check Preconditions

**context** UPPAAL\_Reachability\_Problem\_precompiled\_defs  
**begin**

**abbreviation**

$check\_nat\_subs \equiv \forall (\_, d) \in clkp\_set'. d \geq 0$

**lemma** check\_nat\_subs:

$check\_nat\_subs \longleftrightarrow snd \text{ ` } clkp\_set' \subseteq \mathbb{N}$   
**unfolding** Nats\_def **apply** safe  
**subgoal for**  $\_ \_ b$  **using** rangeI[of int nat b] **by** auto  
**by** auto

**definition**

$check\_resets \equiv \forall x \ c. \text{ Some } (INSTR (STOREC \ c \ x)) \in set \ prog \longrightarrow x = 0$

**lemma** check\_resets\_alt\_def:

$check\_resets =$   
 $(\forall (c, x) \in collect\_store. x = 0)$   
**unfolding** check\_resets\_def collect\_store\_def **by** auto

**definition**

$check\_pre \equiv$   
 $length \ inv = p \wedge length \ trans = p \wedge length \ pred = p$   
 $\wedge (\forall i < p. length \ (pred \ ! \ i) = length \ (trans \ ! \ i) \wedge length \ (inv \ ! \ i) = length \ (trans \ ! \ i))$   
 $\wedge (\forall T \in set \ trans. \forall xs \in set \ T. \forall (\_, \_, \_, l) \in set \ xs. l < length \ T)$   
 $\wedge p > 0 \wedge m > 0$   
 $\wedge (\forall i < p. trans \ ! \ i \neq []) \wedge (\forall q < p. trans \ ! \ q \ ! \ 0 \neq [])$   
 $\wedge check\_nat\_subs \wedge clk\_set' = \{1..m\}$   
 $\wedge check\_resets$

**lemma** finite\_clkp\_set'[intro, simp]:

$finite \ clkp\_set'$   
**unfolding** clkp\_set'\_def  
**using** [[simproc add: finite\_Collect]]

```

    by (auto intro!: finite_vimageI simp: inj_on_def)

lemma check_pre:
  UPPAAL_Reachability_Problem_precompiled p m inv pred trans prog  $\longleftrightarrow$  check_pre
  unfolding
    UPPAAL_Reachability_Problem_precompiled_def
    check_pre_def check_nat_subs check_resets_def
  by auto

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

context
  fixes k :: nat list list list
begin

definition
  check_ceiling  $\equiv$ 
    UPPAAL_Reachability_Problem_precompiled_ceiling_axioms p m max_steps inv trans prog
k

lemma check_axioms:
  UPPAAL_Reachability_Problem_precompiled_ceiling p m max_steps inv pred trans prog k
 $\longleftrightarrow$  check_pre  $\wedge$  check_ceiling
  unfolding UPPAAL_Reachability_Problem_precompiled_ceiling_def check_pre check_ceiling_def
  by auto

end

end

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.collect_store_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.check_resets_alt_def

export_code UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp in SML mod-
ule_name Test

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs.check_pre
  UPPAAL_Reachability_Problem_precompiled_defs.check_axioms
  UPPAAL_Reachability_Problem_precompiled_defs.clkp_set'_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.clk_set'_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.check_pre_def
  UPPAAL_Reachability_Problem_precompiled_defs.check_ceiling_def
  UPPAAL_Reachability_Problem_precompiled_defs.init_def

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs'.trans_out_map_def
  UPPAAL_Reachability_Problem_precompiled_defs'.trans_in_map_def

```

```

UPPAAL_Reachability_Problem_precompiled_defs'.trans_i_map_def
UPPAAL_Reachability_Problem_precompiled_defs'.all_actions_by_state_def
UPPAAL_Reachability_Problem_precompiled_defs'.actions_by_state_def
UPPAAL_Reachability_Problem_precompiled_defs'.pairs_by_action_def

```

```
code_pred clock_val_a .
```

```

concrete_definition reachability_checker_impl
  uses UPPAAL_Reachability_Problem_precompiled'.reachability_checker_alt_def_refined

```

```

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs'.make_cconstr_def
  UPPAAL_Reachability_Problem_precompiled_defs'.make_reset_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_pred_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_g_def
  UPPAAL_Reachability_Problem_precompiled_defs'.runf_def
  UPPAAL_Reachability_Problem_precompiled_defs'.runt_def
  UPPAAL_Reachability_Problem_precompiled_defs.PROG_def

```

```

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs.P_def

```

```

lemma exec_code[code]:
  exec prog n (pc, st, m, f, rs) pcs =
    (case n of 0 ⇒ None
    | Suc n ⇒
      (case prog pc of None ⇒ None
      | Some instr ⇒
        if instr = HALT
        then Some ((pc, st, m, f, rs), pc # pcs)
        else
          (case UPPAAL_Asm.step instr (pc, st, m, f, rs) of
            None ⇒ None | Some s ⇒ exec prog n s (pc # pcs))))
  by (cases n) auto

```

```

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled'.axioms_def
  UPPAAL_Reachability_Problem_precompiled'.def
  pred_act_def

```

```

definition
  init_pred_check ≡ λ p prog max_steps pred s0.
    (∀ q < p.
      case (exec
        (stripfp (UPPAAL_Reachability_Problem_precompiled_defs.PROG prog))
        max_steps
        ((pred ! q ! (UPPAAL_Reachability_Problem_precompiled_defs.init p ! q)), [], s0, True,
        []))
      of Some ((pc, st, s', True, rs), pcs) ⇒ True | _ ⇒ False)

```

```

definition
  time_indep_check1 ≡ λ pred prog max_steps.

```

$(\forall x \in \text{set } \text{pred}. \forall pc \in \text{set } x. \text{time\_indep\_check } \text{prog } pc \text{ max\_steps})$

**definition**

$\text{time\_indep\_check2} \equiv \lambda \text{ trans prog max\_steps}.$   
 $(\forall T \in \text{set } \text{trans}. \forall xs \in \text{set } T. \forall (\_, \_, pc\_u, \_) \in \text{set } xs. \text{time\_indep\_check } \text{prog } pc\_u \text{ max\_steps})$

**definition**

$\text{conjunction\_check2} \equiv \lambda \text{ trans prog max\_steps}.$   
 $(\forall T \in \text{set } \text{trans}. \forall xs \in \text{set } T. \forall (pc\_g, \_, \_, \_) \in \text{set } xs. \text{conjunction\_check } \text{prog } pc\_g \text{ max\_steps})$

**lemma** *start\_pred*[code]:

*UPPAAL\_Reachability\_Problem\_precompiled\_start\_state\_axioms* =  $(\lambda p \text{ max\_steps trans prog}$   
*bounds pred s<sub>0</sub>.*

*init\_pred\_check* *p prog max\_steps pred s<sub>0</sub>*  
 $\wedge$  *bounded* *bounds s<sub>0</sub>*  
 $\wedge$  *time\_indep\_check1* *pred prog max\_steps*  
 $\wedge$  *time\_indep\_check2* *trans prog max\_steps*  
 $\wedge$  *conjunction\_check2* *trans prog max\_steps*  
 $)$

**unfolding** *UPPAAL\_Reachability\_Problem\_precompiled\_start\_state\_axioms\_def*

**unfolding** *init\_pred\_check\_def bounded\_def time\_indep\_check1\_def time\_indep\_check2\_def*  
*conjunction\_check2\_def*

**apply** (*rule ext*)+

**apply** *safe*

**apply** (*fastforce split: option.split\_asm bool.split\_asm*)

**subgoal** *premises prems*

**using** *prems(1,7)* **by** (*fastforce split: option.split\_asm bool.split\_asm*)

**done**

**export\_code** *UPPAAL\_Reachability\_Problem\_precompiled\_start\_state\_axioms*

**context** *UPPAAL\_Reachability\_Problem\_precompiled\_defs*

**begin**

**lemma** *collect\_store''\_alt\_def*:

*collect\_store'' pc*  $\equiv$

*case find\_resets\_start prog pc of*

*None*  $\Rightarrow \{\}$  |

*Some pc'*  $\Rightarrow$

$\bigcup$  (

$(\lambda \text{ cmd. case cmd of Some (INSTR (STOREC c x))} \Rightarrow \{(c, x)\} \mid \_ \Rightarrow \{\})$  ‘

$((!) \text{ prog})$  ‘  $\{pc \dots pc'\}$

)

**unfolding** *collect\_store''\_def*

**apply** (*rule eq\_reflection*)

**apply** *safe*

**subgoal**

**apply** (*simp del: find\_resets\_start.simps split: option.split\_asm*)

**by** (*auto intro: sym intro!: bexI split: option.split*)

**subgoal**

**apply** (*clarsimp simp del: find\_resets\_start.simps split: option.split\_asm*)

```

    by (auto intro: sym split: instrc.split_asm instr.split_asm)
done

lemma collect_cexp'_alt_def:
  collect_cexp' pc  $\equiv$ 
     $\bigcup ((\lambda \text{cmd. case cmd of Some (CEXP ac) } \Rightarrow \{ac\} \mid \_ \Rightarrow \{\}) \text{ '}$ 
       $((!) \text{ prog}) \text{ ' steps\_approx max\_steps prog pc}$ 
    )
  unfolding collect_cexp'_def
  by (auto 4 3 intro!: eq_reflection bexI intro: sym split: option.splits instrc.split_asm)

end

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs'.PROG'_def
  UPPAAL_Reachability_Problem_precompiled_start_state_def
  UPPAAL_Reachability_Problem_precompiled_ceiling_axioms_def
  UPPAAL_Reachability_Problem_precompiled_defs.N_def
  UPPAAL_Reachability_Problem_precompiled_defs.collect_store''_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.clkp_set''_def
  UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp'_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs'.pairs_by_action_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.make_reset_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_g_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.run_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.make_constr_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_pred_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.all_actions_by_state_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.trans_i_from_impl_def

lemmas [code] =
  Equiv_TA_Defs.state_ta_def Prod_TA_Defs.N_s_def Product_TA_Defs.states_def

export_code UPPAAL_Reachability_Problem_precompiled'_axioms in SML module_name
Test

export_code UPPAAL_Reachability_Problem_precompiled' in SML module_name Test

hide_const check_and_verify

definition [code]:
  check_and_verify p m k max_steps I T prog final bounds P s0 na  $\equiv$ 
    if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na k
    then
      reachability_checker_impl p m max_steps I T prog bounds P s0 na k final
       $\gg (\lambda x. \text{return (Some x)})$ 
    else return None

abbreviation N  $\equiv$  UPPAAL_Reachability_Problem_precompiled_defs.N

theorem reachability_check:
  (uncurry0 (check_and_verify p m k max_steps I T prog (formula.EX formula) bounds P s0

```

```

na),
  uncurry0 (
    Refine_Basic.RETURN (
      if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na
    k
      then Some (
        if (∀ u. (∀ c ∈ {1..m}. u c = 0) → (∀ i < p. u ⊢ conv_cc (I ! i ! 0)))
        then
          if
            (
              ∃ L' s' u u'.
                conv (N p I P T prog bounds) ⊢m ax_steps
                ⟨repeat 0 p, s0, u⟩ →* ⟨L', s', u^⟩
                ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp formula L' s'
            )
            then REACHABLE
            else UNREACHABLE
            else INIT_INV_ERR
          )
        else None
      )
    )
  )
  ∈ unit_assnk →a id_assn
proof –
define A where A ≡ conv (N p I P T prog bounds)
define start_inv where
  start_inv ≡ (∀ u. (∀ c ∈ {1..m}. u c = 0) → (∀ i < p. u ⊢ conv_cc (I ! i ! 0)))
define reach where
  reach ≡
    ∃ L' s' u u'.
      A ⊢m ax_steps
      ⟨repeat 0 p, s0, u⟩ →* ⟨L', s', u^⟩
      ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp formula L' s'
note [sep_heap_rules] =
  UPPAAL_Reachability_Problem_precompiled'.reachability_checker_hoare'
  [ OF__HOL.refl[of formula.EX formula],
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    of p m max_steps I T prog bounds P s0 na k,
    unfolded A_def[symmetric] start_inv_def[symmetric] reach_def[symmetric]
  ]
show ?thesis
unfolding A_def[symmetric] start_inv_def[symmetric] reach_def[symmetric]
unfolding check_and_verify_def
by sepref_to_hoare (sep_auto simp: reachability_checker_impl.refine[symmetric])
qed

export_code open
  check_and_verify_init_pred_check time_indep_check1 time_indep_check1 conjunction_check2
  checking SML_imp

end
theory UPPAAL_Model_Checking
imports

```

```

UPPAAL_State_Networks_Impl_Refine
Munta_Base.TA_More
Munta_Base.Abstract_Term
begin

hide_const models

inductive step_u' ::
  ('a, 't :: time, 's) unta  $\Rightarrow$  nat  $\Rightarrow$  's list  $\Rightarrow$  int list  $\Rightarrow$  (nat, 't) cval
 $\Rightarrow$  's list  $\Rightarrow$  int list  $\Rightarrow$  (nat, 't) cval  $\Rightarrow$  bool
( $\_ \vdash \langle \_, \_, \_ \rangle \rightarrow \langle \_, \_, \_ \rangle$  [61,61,61,61,61,61] 61) where
  A  $\vdash^n \langle L, s, u \rangle \rightarrow \langle L'', s'', u'' \rangle$  if
  A  $\vdash_n \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$  a  $\neq Del$  A  $\vdash_n \langle L', s', u' \rangle \rightarrow_a \langle L'', s'', u'' \rangle$ 

inductive steps_un' ::
  ('a, 't :: time, 's) unta  $\Rightarrow$  nat  $\Rightarrow$  's list  $\Rightarrow$  int list  $\Rightarrow$  (nat, 't) cval
 $\Rightarrow$  's list  $\Rightarrow$  int list  $\Rightarrow$  (nat, 't) cval  $\Rightarrow$  bool
( $\_ \vdash \langle \_, \_, \_ \rangle \rightarrow^* \langle \_, \_, \_ \rangle$  [61,61,61,61,61,61] 61)
where
  refl: A  $\vdash^n \langle L, s, u \rangle \rightarrow^* \langle L, s, u \rangle$  |
  step: A  $\vdash^n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \Longrightarrow A \vdash^n \langle L', s', u' \rangle \rightarrow \langle L'', s'', u'' \rangle$ 
 $\Longrightarrow A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$ 

declare steps_un'.intros[intro]

lemma stepI2:
  A  $\vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$  if
  A  $\vdash^n \langle L', s', u' \rangle \rightarrow^* \langle L'', s'', u'' \rangle$  A  $\vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ 
  using that by induction auto

context Equiv_TA
begin

lemma prod_correct'_action:
  ( $\exists$  a. defs.prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ ) =
  ( $\exists$  a. state_ta  $\vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq Del$ )
  apply standard
  subgoal
    by (blast elim: prod.prod_sound'_action)
  apply clarify
  subgoal for a
    apply (cases a; simp; erule prod.prod_complete_silent prod.prod_complete_sync, fast)
  done
done

lemma prod_correct'_delay:
  ( $\exists$  d. defs.prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$ ) =
  state_ta  $\vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$ 
  by (blast dest: prod.prod_sound'_delay elim: prod.prod_complete_delay)

lemma equiv_correct:
  state_ta  $\vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle = A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
  by (blast intro!: equiv_sound equiv_complete)

```

**lemma** *prod\_correct\_action*:

$(\exists a. \text{defs.prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle) =$   
 $(\exists a. A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq \text{State\_Networks.label.Del})$   
**unfolding** *prod\_correct'\_action equiv\_correct* ..

**lemma** *prod\_correct\_delay*:

$(\exists d. \text{defs.prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle) =$   
 $A \vdash_n \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$   
**unfolding** *prod\_correct'\_delay equiv\_correct* ..

**lemma** *prod\_correct*:

$\text{defs.prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle =$   
 $(\exists a. A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle)$   
**apply** *standard*  
**subgoal**  
**using** *prod\_correct\_action*[of  $u \ L' \ s' \ u'$ ] *prod\_correct\_delay*[of  $u \ L' \ s' \ u'$ ]  
*Timed\_Automata.step.cases* **by** *metis*  
**subgoal**  
**apply** *clarify*  
**subgoal for**  $a$   
**apply** (*cases*  $a$ )  
**using** *prod\_correct\_action*[of  $u \ L' \ s' \ u'$ ] *prod\_correct\_delay*[of  $u \ L' \ s' \ u'$ ]  
*Timed\_Automata.step.intros* **apply** *metis* +  
**done**  
**done**  
**done**

**context**

**assumes**  $0 < p$

**begin**

**lemmas** *equiv\_complete'' = equiv\_complete'*[OF  $\langle 0 < p \rangle$ ]

**definition**

*all\_prop*  $L' \ s' \equiv$   
 $(\forall q < p. \exists pc \ st \ s'' \ rs \ pcs.$   
 $\text{exec PF } n \ ((I \ ! \ q) \ (L' \ ! \ q), [], s', \text{True}, []) \ [] =$   
 $\text{Some} ((pc, st, s'', \text{True}, rs), pcs)$   
 $) \wedge \text{bounded } B \ s' \wedge L' \in \text{defs.states}' \ s'$

**lemma** *step\_u\_inv*:

*all\_prop*  $L' \ s'$  **if**  $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$   
**using** *equiv\_complete''*[OF *that*] *equiv\_complete'*[OF *that*] **unfolding** *all\_prop\_def* **by** *auto*

**lemma** *step\_inv*:

*all\_prop*  $L' \ s'$  **if**  $\text{state\_ta} \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$   
**using** *step\_u\_inv*[OF *equiv\_sound*[OF *that*]] .

**lemma** *Equiv\_TA\_I*:

*Equiv\_TA*  $A \ n \ L' \ s'$  **if**  $*[\text{unfolded } \text{all\_prop\_def}]: \text{all\_prop } L' \ s'$   
**using**  $*$  **by**  $-(\text{standard}, \text{auto intro!}: \text{pred\_time\_indep upd\_time\_indep clock\_conj Len})$

```

lemma step_u'_inv:
  all_prop L'' s''  $\wedge$  defs.prod_ta  $\vdash'$   $\langle (L, s), u \rangle \rightarrow \langle (L'', s''), u' \rangle$ 
  if  $A \vdash^n \langle L, s, u \rangle \rightarrow \langle L'', s'', u' \rangle$ 
using that proof cases
case prems: (1 L' s' u')
from step_u_inv[OF prems(1)] have *[unfolded all_prop_def]: all_prop L' s' .
interpret equiv: Equiv_TA A n L' s'
  using Equiv_TA_I[OF step_u_inv[OF prems(1)]] .
from equiv.step_u_inv[OF  $\langle 0 < p \rangle$  prems(3)] show ?thesis
  using prems prod_correct_delay[of u L' s' u'] equiv.prod_correct_action[of u' L'' s'' u']
    Timed_Automata.step'.intros
  by metis
qed

lemma step'_inv:
  all_prop L'' s''  $\wedge$   $A \vdash^n \langle L, s, u \rangle \rightarrow \langle L'', s'', u' \rangle$ 
  if defs.prod_ta  $\vdash'$   $\langle (L, s), u \rangle \rightarrow \langle (L'', s''), u' \rangle$ 
using that proof cases
case prems: (step' d l' u' a)
obtain L' s' where l' = (L', s')
  by force
from step_inv prod_correct'_delay prems(1) have *:
  all_prop L' s'
  unfolding  $\langle l' = \_ \rangle$  by fast
interpret equiv: Equiv_TA A n L' s'
  by (rule Equiv_TA_I[OF *])
from equiv.step_inv[OF  $\langle 0 < p \rangle$ ] equiv.prod_correct'_action prems(2)[unfolded  $\langle l' = \_ \rangle$ ] have
  all_prop L'' s''
  by metis
then show ?thesis
  using prems prod_correct_delay[of u L' s' u'] equiv.prod_correct_action[of u' L'' s'' u']
    step_u'.intros
  unfolding  $\langle l' = \_ \rangle$  by metis
qed

lemma prod_correct_step':
  defs.prod_ta  $\vdash'$   $\langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle =$ 
   $A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ 
  using step'_inv step_u'_inv by blast

lemma all_prop_start:
  all_prop L s
  using Equiv_TA_axioms unfolding Equiv_TA_def all_prop_def by auto

lemma steps_u'_inv:
  all_prop L'' s''  $\wedge$  defs.prod_ta  $\vdash'$   $\langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u' \rangle$ 
  if  $A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u' \rangle$ 
  using that
proof (induction  $A \equiv A \ n \equiv n \ L \equiv L \ s \equiv s \ u \ L'' \ s'' \ u'$ )
case (refl u)
show ?case using all_prop_start by auto
next
case (step u L' s' u' L'' s'' u')
then interpret equiv: Equiv_TA A n L' s'

```

by (blast intro: *Equiv\_TA\_I*)  
 from *equiv.step\_u'\_inv*[*OF*  $\langle 0 < p \rangle$  *step.hyps*(3)] *step.hyps*(1-2) **show** ?*case*  
 by (blast intro: *steps'\_altI*)  
**qed**

**lemma** *steps'\_inv*:  
*all\_prop*  $L'' s'' \wedge A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$   
 if *defs.prod\_ta*  $\vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u'' \rangle$   
 using *that all\_prop\_start*  
**proof** (*induction* *defs.prod\_ta* ( $L, s$ )  $u$  ( $L'', s''$ )  $u''$  *arbitrary: L s*)  
 case (*refl'*  $u$ )  
 then **show** ?*case* using *all\_prop\_start* **by** *auto*  
**next**  
 case (*step'*  $u$   $l'$   $u'$   $u''$   $L s$ )  
 obtain  $L' s'$  **where**  $l' = (L', s')$  **by** *force*  
 from *step'* **interpret** *equiv: Equiv\_TA A n L s*  
 by (blast intro: *Equiv\_TA\_I*)  
 from *equiv.step'\_inv*[*OF*  $\langle 0 < p \rangle$  *step'(1)*[*unfolded*  $\langle l' = \_ \rangle$ ]] *step'(3)*[*OF*  $\langle l' = \_ \rangle$ ] **show** ?*case*  
 by (*auto* intro: *stepI2*)  
**qed**

**lemma** *steps\_un'\_complete*:  
*defs.prod\_ta*  $\vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u'' \rangle$   
 if  $A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$   
 using *steps\_u'\_inv*[*OF that*] ..

**lemma** *steps'\_sound*:  
 $A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$   
 if *defs.prod\_ta*  $\vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u'' \rangle$   
 using *steps'\_inv*[*OF that*] ..

**lemma** *prod\_reachable\_correct*:  
*defs.prod\_ta*  $\vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \longleftrightarrow A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$   
 using *steps'\_sound* *steps\_un'\_complete* **by** *fast*

**lemma** *Bisimulation\_Invariant\_I*:  
*Bisimulation\_Invariant*  
 $(\lambda (L, s, u) (L', s', u'). \text{defs.prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 (=)  
 $(\lambda (L, s, u). \text{all\_prop } L s)$   
 $(\lambda (L, s, u). \text{all\_prop } L s)$   
**apply** (*standard; clarsimp*)  
**apply** (*simp add: Equiv\_TA.prod\_correct\_step' Equiv\_TA\_I*  $\langle 0 < p \rangle$ ) +  
 using *Equiv\_TA.step\_u'\_inv* *Equiv\_TA\_I*  $\langle 0 < p \rangle$  **apply** *blast+*  
**done**

**interpretation** *Bisimulation\_Invariant*  
 $\lambda (L, s, u) (L', s', u'). \text{defs.prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$   
 $\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$   
 (=)  
 $\lambda (L, s, u). \text{all\_prop } L s$   
 $\lambda (L, s, u). \text{all\_prop } L s$   
**by** (*rule Bisimulation\_Invariant\_I*)

end

end

**definition** *models* ( $\_,\_ \models\_ \_$  [61,61] 61) **where**

$A, a_0 \models_n \Phi \equiv (\text{case } \Phi \text{ of}$   
 $\text{formula.EX } \varphi \Rightarrow$   
 $\text{Graph\_Defs.Ex\_ev}$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \varphi L s)$   
 $| \text{formula.EG } \varphi \Rightarrow$   
 $\text{Graph\_Defs.Ex\_alw}$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \varphi L s)$   
 $| \text{formula.AX } \varphi \Rightarrow$   
 $\text{Graph\_Defs.Alw\_ev}$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \varphi L s)$   
 $| \text{formula.AG } \varphi \Rightarrow$   
 $\text{Graph\_Defs.Alw\_alw}$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \varphi L s)$   
 $| \text{formula.Leadsto } \varphi \psi \Rightarrow$   
 $\text{Graph\_Defs.leadsto}$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \varphi L s)$   
 $(\lambda (L, s, \_). \text{check\_bexp } \psi L s)$   
 $) a_0$

**definition**

$\text{has\_deadlock } A \ n \ a_0 \equiv$   
 $\text{Graph\_Defs.deadlock } (\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle) \ a_0$

**lemmas** *models\_iff* = *models\_def*[*unfolded* *Graph\_Defs.Ex\_alw\_iff* *Graph\_Defs.Alw\_alw\_iff*]

**context** *Reachability\_Problem*

**begin**

**lemma** *reaches\_steps'*:

$\text{reaches } (l, u) (l', u') \longleftrightarrow \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$   
**apply** *standard*  
**subgoal** *premises* *prems*  
**using** *prems* **by** (*induction*  $(l, u) (l', u')$  *arbitrary: l' u'*) (*auto intro: steps'\_altI*)  
**subgoal** *premises* *prems*  
**using** *prems* **by** *induction* (*auto intro: converse\_rtranclp\_into\_rtranclp*)  
**done**

**lemma** *clocks\_I*:

$(\forall \ c. c \in \text{clk\_set } (\text{conv\_A } A) \longrightarrow u \ c = u' \ c) \text{ if } \forall \ c \in \{1..n\}. u \ c = u' \ c$   
**unfolding** *clk\_set\_conv\_A* **using** *clocks\_n* **using** *that* **by** *auto*

**lemma** *init\_dbm\_reaches\_iff*:

$$\begin{aligned}
& (\exists u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle) \\
& \longleftrightarrow ([\text{curry (init\_dbm :: real DBM')}]_{v,n} \neq \{\}) \wedge \\
& (\forall u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle)
\end{aligned}$$

**proof** –

**interpret** *ta\_bisim*: *Bisimulation\_Invariant*  
 $(\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(l, u) (l', u'). l' = l \wedge (\forall c. c \in \text{clk\_set } (\text{conv\_A } A) \longrightarrow u c = u' c))$   
 $(\lambda_. \text{True}) (\lambda_. \text{True})$   
**by** (*rule ta\_bisimulation*[of *conv\_A A*])  
**show** *?thesis*  
**apply** *safe*  
**apply** *force*  
**subgoal** **for** *u1 u' u2*  
**unfolding** *init\_dbm\_semantics\_reaches\_steps'*[*symmetric*]  
**apply** (*drule ta\_bisim.A\_B.simulation\_reaches*[of *\_ \_ (l0, u2)*])  
**subgoal**  
**using** *clocks\_I*[of *u1 u2*] **by** *fastforce*  
**by** *auto*  
**subgoal** **for** *u*  
**by** *blast*  
**done**  
**qed**

**theorem** *reachable\_decides\_emptiness\_new*:

$$\begin{aligned}
& (\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry (conv\_M } D')]_{v,n} \neq \{\}) \\
& \longleftrightarrow [\text{curry (init\_dbm :: real DBM')}]_{v,n} \neq \{\} \wedge \\
& (\forall u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle) \\
& \text{unfolding } \text{reachable\_decides\_emptiness } \text{init\_dbm\_reaches\_iff} \dots
\end{aligned}$$

**lemma** *reachable\_decides\_emptiness'\_new*:

$$\begin{aligned}
& (\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry (conv\_M } D')]_{v,n} \neq \{\}) \\
& \longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle)) \\
& \text{unfolding } \text{reachable\_decides\_emptiness\_new} \\
& \text{using } \text{init\_dbm\_semantics'} \text{ init\_dbm\_semantics'' } \text{init\_dbm\_non\_empty} \text{ by } \text{blast}
\end{aligned}$$

**lemma** *reachability\_check\_new\_aux*:

$$\begin{aligned}
& (\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry (conv\_M } D')]_{v,n} \neq \{\} \wedge F l') \\
& \longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle \wedge F l')) \\
& \text{using } \text{reachable\_decides\_emptiness'_new}[\text{of } l'] \text{ by } \text{fast}
\end{aligned}$$

**theorem** *reachability\_check\_new*:

$$\begin{aligned}
& (\exists D'. E^{**} a_0 (l', D') \wedge F\_rel (l', D')) \\
& \longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle \wedge F l')) \\
& \text{using } \text{reachability\_check\_new\_aux}[\text{of } l'] \text{ check\_diag\_empty\_spec } \text{reachable\_empty\_check\_diag} \\
& \text{unfolding } F\_rel\_def \text{ by } \text{auto}
\end{aligned}$$

**lemma** *init\_state\_in\_state\_set*:

$l_0 \in \text{state\_set } (\text{trans\_of } A) \text{ if } \neg \text{deadlock } (l_0, u_0)$

**proof** –

**obtain** *l u* **where**  $\text{conv\_A } A \vdash' \langle l_0, u_0 \rangle \rightarrow \langle l, u \rangle$   
**using**  $\neg \text{deadlock\_}$  **unfolding** *deadlock\_def* *deadlocked\_def* **by** *force*  
**then have**  $l_0 \in \text{state\_set } (\text{trans\_of } (\text{conv\_A } A))$

```

    unfolding state_set_def
    by cases (auto elim!: step_a.cases step_t.cases)
  then show ?thesis
    unfolding state_set_def unfolding trans_of_def conv_A_def by (cases A) force
qed

lemma init_state_in_state_set':
  l0 ∈ state_set (trans_of A) if (∀ u0. (∀ c ∈ {1..n}. u0 c = 0) → ¬ deadlock (l0, u0))
  using init_state_in_state_set that by auto

end

context Reachability_Problem_Impl
begin

context
  fixes Q :: 's ⇒ bool and Q_fun
  assumes Q_fun: (Q_fun, Q) ∈ inv_rel loc_rel states'
begin

lemma leadsto_spec_refine:
  leadsto_spec_alt Q
  ≤ SPEC (λ r. ¬ r ↔
    (∄ x. (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b))** (l0, init_dbm) x ∧
      F (fst x) ∧ Q (fst x) ∧
      (∃ a. (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))** x a ∧
        (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))++ a a)
    ))
proof -
  have *:
    (λ x y. (case y of (l', M') ⇒ E_op''.E_from_op x (l', M') ∧ ¬ check_diag n M') ∧
      ¬ (case y of (l, M) ⇒ check_diag n M))
    = (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b))
  by (intro ext) auto
  have **:
    (λ x y. (case y of (l', M') ⇒ E_op''.E_from_op x (l', M') ∧ ¬ check_diag n M') ∧
      (case y of (l, M) ⇒ Q l) ∧ ¬ (case y of (l, M) ⇒ check_diag n M))
    = (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))
  by (intro ext) auto
  have ***: ¬ check_diag n b
  if (λ a b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b))** a0 (a, b) for a b
  using that by cases (auto simp: a0_def)
  show ?thesis
    unfolding leadsto_spec_alt_def[OF Q_fun]
    unfolding PR_CONST_def a0_def[symmetric] by (auto dest: *** simp: ** *)
qed

lemma leadsto_impl_hnr':
  (uncurry0
    (leadsto_impl state_copy_impl
      (succs_P_impl' Q_fun) a0_impl subsumes_impl (return ∘ fst)
      succs_impl' emptiness_check_impl F_impl (Q_impl Q_fun) tracei),
  uncurry0
  (SPEC

```

```

    (λr. (∀ u0. (∀ c∈{1..n}. u0 c = 0) → ¬ deadlock (l0, u0)) →
      (¬ r) =
      (∀ u0. (∀ c∈{1..n}. u0 c = 0) →
        leadsto (λ(l, u). F l) (λ(l, u). ¬ Q l) (l0, u0))))
  ∈ unit_assnk →a bool_assn
proof -
  define p1 where p1 ≡
    (∄ x. (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b))** (l0, init_dbm) x ∧
      F (fst x) ∧ Q (fst x) ∧
      (∃ a. (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))** x a ∧
        (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))++ a a))
  define p2 where p2 ≡ (∀ u0. (∀ c∈{1..n}. u0 c = 0) → ¬ deadlock (l0, u0))
  define p3 where
    p3 ≡ (∀ u0. (∀ c∈{1..n}. u0 c = 0) → leadsto (λ(l, u). F l) (λ(l, u). ¬ Q l) (l0, u0))
  show ?thesis
  unfolding state_set_eq[symmetric]
  using Reachability_Problem_Impl_Op.leadsto_impl_hnr[OF Reachability_Problem_Impl_Op_axioms,
    OF Q_fun precondition_a0,
    FCOMP leadsto_spec_refine[THEN Id_SPEC_refine, THEN nres_relI], to_hnr, unfolded
    hn_refine_def,
    of show_state show_clock
  ]
  using init_state_in_state_set'
  using leadsto_mc[of F Q]
  unfolding p1_def[symmetric] p2_def[symmetric] p3_def[symmetric]
  apply (simp del: state_set_eq)
  apply seprel_to_hoare
  apply sep_auto
  apply (erule cons_post_rule)
  apply sep_auto
  done
qed

end

end

context UPPAAL_Reachability_Problem_precompiled'
begin

lemma F_reachable_correct'_new:
  impl.op.F_reachable
  ↔ (∃ L' s'. ∀ u. (∀ c ∈ {1..m}. u c = 0) → (∃ u'.
    conv_A A ⊢' ⟨(init, s0), u⟩ →* ⟨(L', s'), u'⟩
    ∧ check_bexp φ L' s'))
  ) if formula = formula.EX φ
  using
    that E_op''.E_from_op_reachability_check[of F_rel PR_CONST (λ(x, y). F x y)]
    reachability_check_new
  unfolding impl.E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
  unfolding F_rel_def unfolding F_def by force

lemma F_reachable_correct'_new':
  impl.op.F_reachable

```

$\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv\_A } A \vdash' \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \neg \text{check\_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.AG } \varphi \\
\text{using} \\
\text{that } E\_op''.E\_from\_op\_reachability\_check[\text{of } F\_rel\ PR\_CONST\ (\lambda(x, y). F\ x\ y)] \\
\text{reachability\_check\_new} \\
\text{unfolding } \text{impl.E\_op\_F\_reachable } E\_op''.F\_reachable\_def\ E\_op''.reachable\_def \\
\text{unfolding } F\_rel\_def\ \text{unfolding } F\_def\ \text{by force}$

**lemma** *F\_reachable\_correct\_new*:  
*impl.op.F\_reachable*  
 $\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv } N \vdash^m \text{ax\_steps } \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \text{check\_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.EX } \varphi \\
\text{unfolding } F\_reachable\_correct\_new[\text{OF that}] \\
\text{apply } (\text{subst product'}.prod\_reachable\_correct[\text{symmetric}]) \\
\text{using prod\_conv } p\_p\ p\_gt\_0\ \text{by simp+}$

**lemma** *F\_reachable\_correct\_new'*:  
*impl.op.F\_reachable*  
 $\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv } N \vdash^m \text{ax\_steps } \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \neg \text{check\_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.AG } \varphi \\
\text{unfolding } F\_reachable\_correct\_new'[\text{OF that}] \\
\text{apply } (\text{subst product'}.prod\_reachable\_correct[\text{symmetric}]) \\
\text{using prod\_conv } p\_p\ p\_gt\_0\ \text{by simp+}$

**definition**

$\text{Alw\_ev\_checker} = \text{dfs\_map\_impl'}$   
 $(\text{impl.succs\_P\_impl' final\_fun})\ \text{impl.a}_0\_impl\ \text{impl.subsumes\_impl}\ (\text{return} \circ \text{fst})$   
 $\text{impl.state\_copy\_impl}$

**definition**

$\text{leadsto\_checker } \psi = \text{do } \{$   
 $r \leftarrow \text{leadsto\_impl}$   
 $\text{impl.state\_copy\_impl } (\text{impl.succs\_P\_impl' } (\lambda (L, s). \neg \text{check\_bexp } \psi\ L\ s))$   
  
 $\text{impl.a}_0\_impl\ \text{impl.subsumes\_impl}\ (\text{return} \circ \text{fst})$   
 $\text{impl.succs\_impl' impl.empty\_check\_impl impl.F\_impl}$   
 $(\text{impl.Q\_impl } (\lambda (L, s). \neg \text{check\_bexp } \psi\ L\ s))$   
 $\text{impl.tracei};$   
 $\text{return } (\neg r)$   
 $\}$

**definition**

$\text{model\_checker} = ($   
 $\text{case formula of}$   
 $\text{formula.EX } \_ \Rightarrow \text{reachability\_checker' } |$   
 $\text{formula.AG } \_ \Rightarrow \text{do } \{$   
 $r \leftarrow \text{reachability\_checker'}$   
 $\text{return } (\neg r)$   
 $\}$

```

} |
formula.AX _  $\Rightarrow$  do {
  r  $\leftarrow$  if PR_CONST ( $\lambda(x, y). F x y$ ) (init, s0)
  then Alw_ev_checker
  else return False;
  return ( $\neg$  r)
} |
formula.EG _  $\Rightarrow$ 
  if PR_CONST ( $\lambda(x, y). F x y$ ) (init, s0)
  then Alw_ev_checker
  else return False |
formula.Leadsto _  $\psi \Rightarrow$  leadsto_checker  $\psi$ 
)

```

**lemma**  $p\_gt\_0$ :  
 $0 < \text{defs'.}p$   
**unfolding**  $p\_p$  **by** (rule  $p\_gt\_0$ )

**interpretation** *Bisim\_A: Bisimulation\_Invariant*  
 $(\lambda(L, s, u) (L', s', u').$   
 $\text{defs'.defs.prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$   
 $(\lambda(L, s, u) (L', s', u').$   
 $\text{conv } N \vdash^m \text{ax\_steps} \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(=) (\lambda(L, s, u). \text{product'.all\_prop } L s)$   
 $(\lambda(L, s, u). \text{product'.all\_prop } L s)$   
**by** (rule *product'.Bisimulation\_Invariant\_I*) (rule  $p\_gt\_0$ )

**interpretation** *Bisim\_B: Bisimulation\_Invariant*  
 $(\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(L, s, u) (L', s', u').$   
 $\text{defs'.defs.prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$   
 $(\lambda(l, u') (L, s, u). (l, u') = ((L, s), u)) \lambda \_ . \text{True}$   
 $(\lambda(L, s, u). \text{product'.all\_prop } L s)$   
**unfolding** *prod\_conv[symmetric]*  
**apply** *standard*  
**apply** (*solves auto*)  
**by** (rule *Bisim\_A.A\_invariant*)

**interpretation** *Bisimulation\_Invariant*  
 $(\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(L, s, u) (L', s', u').$   
 $\text{conv } N \vdash^m \text{ax\_steps} \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda(l, u') (L, s, u). (l, u') = ((L, s), u)) \lambda \_ . \text{True}$   
 $(\lambda(L, s, u). \text{product'.all\_prop } L s)$

**proof** –

**interpret** *bisim: Bisimulation\_Invariant*  
 $(\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(L, s, u) (L', s', u').$   
 $\text{conv } N \vdash^m \text{ax\_steps} \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda a c. \exists b. (\text{case } b \text{ of } (L, s, u) \Rightarrow \text{product'.all\_prop } L s) \wedge$   
 $(\text{case } a \text{ of } (l, u') \Rightarrow \lambda(L, s, u). (l, u') = ((L, s), u)) b \wedge b = c)$   
 $\lambda \_ . \text{True}$   
 $(\lambda(L, s, u). \text{product'.all\_prop } L s)$

```

using Bisimulation_Invariant_composition[OF
  Bisim_B.Bisimulation_Invariant_axioms Bisim_A.Bisimulation_Invariant_axioms
]
.
show Bisimulation_Invariant
  ( $\lambda (l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
  ( $\lambda (L, s, u) (L', s', u').$ 
     $\text{conv } N \vdash^m \text{ax\_steps } \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ )
  ( $\lambda (l, u') (L, s, u). (l, u') = ((L, s), u) (\lambda \_. \text{True}) (\lambda (L, s, u). \text{product'}.all\_prop L s)$ )
apply standard
subgoal for a b a'
  apply (drule bisim.A_B_step[of a b a'])
  apply auto
  done
subgoal for a b a'
  apply (drule bisim.B_A_step[of b a' a])
  apply auto
  done
apply simp
apply (drule bisim.B_invariant)
apply auto
done
qed

lemma models_correct:
   $\text{conv } N, (init, s_0, u_0) \models_m \text{ax\_steps } \Phi = (\text{case } \Phi \text{ of}$ 
    formula.EX  $\varphi \Rightarrow$ 
      ( $\lambda ((L, s), u). \exists L' s' u'. \text{conv\_A } A \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge \text{check\_bexp } \varphi L' s'$ )
    | formula.EG  $\varphi \Rightarrow$ 
      Not o Graph_Defs.Alw_ev
      ( $\lambda (l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
      ( $\lambda ((L, s), \_). \neg \text{check\_bexp } \varphi L s$ )
    | formula.AX  $\varphi \Rightarrow$ 
      Graph_Defs.Alw_ev
      ( $\lambda (l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
      ( $\lambda ((L, s), \_). \text{check\_bexp } \varphi L s$ )
    | formula.AG  $\varphi \Rightarrow$ 
      Not o ( $\lambda ((L, s), u).$ 
         $\exists L' s' u'. \text{conv\_A } A \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge \neg \text{check\_bexp } \varphi L' s'$ 
      )
    | formula.Leadsto  $\varphi \psi \Rightarrow$ 
      Graph_Defs.leadsto
      ( $\lambda (l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
      ( $\lambda ((L, s), \_). \text{check\_bexp } \varphi L s$ )
      ( $\lambda ((L, s), \_). \text{check\_bexp } \psi L s$ )
  ) ((init, s0), u0) if  $\neg \text{deadlock } ((init, s_0), u_0)$ 
proof -
  have *: ((Not  $\circ$  case_prod) ( $\lambda (L, s) \_. \text{check\_bexp } \varphi L s$ )) =
    ( $\lambda ((L, s), \_). \neg \text{check\_bexp } \varphi L s$ ) for  $\varphi$  by auto

show ?thesis
  apply (subst models_def)
  apply (cases  $\Phi$ )
  subgoal for  $\varphi$ 

```

```

apply simp

apply (subst Ex_ev_iff[
  of ( $\lambda((L, s), \_).$  check_bexp  $\varphi$   $L$   $s$ )  $\_$  ((init,  $s_0$ ),  $u_0$ ), symmetric, simplified
])
  apply (drule equiv'_D[simplified], force)
using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
apply (subst Ex_ev[OF that])
unfolding reaches_steps'[symmetric]
apply auto
done
subgoal for  $\varphi$ 
  apply simp
  apply (subst Ex_alw_iff[
    of ( $\lambda((L, s), \_).$  check_bexp  $\varphi$   $L$   $s$ )  $\_$  ((init,  $s_0$ ),  $u_0$ ), symmetric, simplified
  ])
    apply (drule equiv'_D[simplified]; force)
  using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
  unfolding Graph_Defs.Ex_alw_iff * ..
subgoal for  $\varphi$ 
  apply simp
  apply (subst Alw_ev_iff[
    of ( $\lambda((L, s), \_).$  check_bexp  $\varphi$   $L$   $s$ )  $\_$  ((init,  $s_0$ ),  $u_0$ ), symmetric, simplified
  ])
    apply (drule equiv'_D[simplified]; force)
  using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
  unfolding Graph_Defs.Ex_alw_iff * ..
subgoal for  $\varphi$ 
  apply simp
  unfolding Graph_Defs.Alw_alw_iff
  apply (subst Ex_ev_iff[
    of ( $\lambda((L, s), \_).$   $\neg$ check_bexp  $\varphi$   $L$   $s$ )  $\_$  ((init,  $s_0$ ),  $u_0$ ), symmetric, simplified
  ])
    apply (drule equiv'_D[simplified], subst *[symmetric], force)
  using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
  apply (subst Ex_ev[OF that])
  unfolding reaches_steps'[symmetric]
  apply auto
  done
subgoal for  $\varphi$   $\psi$ 
  apply simp
  apply (subst Leadsto_iff[
    of ( $\lambda((L, s), \_).$  check_bexp  $\varphi$   $L$   $s$ )  $\_$ 
      ( $\lambda((L, s), \_).$  check_bexp  $\psi$   $L$   $s$ )  $\_$  ((init,  $s_0$ ),  $u_0$ ), symmetric, simplified
  ])
    apply (drule equiv'_D[simplified]; force)
    apply (drule equiv'_D[simplified]; force)
  using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
  ..

```

done  
qed

**lemma** *final\_fun\_final'*:  
 $((\lambda (L, s). P L s), (\lambda (L, s). P L s)) \in \text{inv\_rel } Id \text{ states' for } P$   
**unfolding** *F\_def final\_fun\_def inv\_rel\_def list\_ex\_iff*  
**by** (*force dest! states'\_states'*)

**lemma** *final\_fun\_final[intro, simp]*:  
 $((\lambda (L, s). P L s), (\lambda (L, s). P L s)) \in \text{inv\_rel } Id \text{ states for } P$   
**using** *final\_fun\_final' states\_states'* **by** (*rule inv\_rel\_mono*)

**abbreviation**  $u_0 \equiv (\lambda \_. 0 :: \text{real})$

**lemma** *deadlock\_start\_iff*:  
 $\text{Bisim\_A.B.deadlock } (init, s_0, \lambda \_. 0) \longleftrightarrow \text{deadlock } ((init, s_0), u_0)$   
**using** *product'.all\_prop\_start[OF p'\_gt\_0]*  
**by** – (*rule deadlock\_iff[of \_ (init, s\_0, u\_0), symmetric]; simp*)

**theorem** *model\_check'*:  
 $(\text{uncurry0 model\_checker},$   
 $\text{uncurry0 } ($   
 $\text{SPEC } (\lambda r.$   
 $\neg \text{Graph\_Defs.deadlock}$   
 $(\lambda (L, s, u) (L', s', u'). \text{conv } N \vdash^m \text{ax\_steps } \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle) (init, s_0, u_0) \longrightarrow$   
 $r = (\text{conv } N, (init, s_0, u_0) \models_m \text{ax\_steps formula})$   
 $)$   
 $)$   
 $)$   
 $\in \text{unit\_assn}^k \rightarrow_a \text{bool\_assn}$

**proof** –  
**have** \*:  $(\lambda(l, u). \neg (\text{case } l \text{ of } (L, s) \Rightarrow (\text{Not } \circ \circ \circ \text{check\_bexp}) \varphi L s))$   
 $= (\lambda((L, s), \_). \text{check\_bexp } \varphi L s) \text{ for } \varphi$   
**by auto**  
**have** \*\*:  $(\lambda(l, u). \neg (\text{case } l \text{ of } (L, s) \Rightarrow \text{check\_bexp } \varphi L s)) = (\lambda((L, s), \_). \neg \text{check\_bexp } \varphi L s)$   
**for } \varphi \text{ by auto}**  
**have** \*\*\*:  $(\lambda(l, u). \text{case } l \text{ of } (L, s) \Rightarrow \varphi L s) = (\lambda((L, s), \_). \varphi L s) \text{ for } \varphi$   
**by auto**  
**have** \*\*\*\*:  $(\lambda(L, y). (\text{Not } \circ \circ \circ \text{check\_bexp}) \psi L y) = (\lambda(L, y). \neg \text{check\_bexp } \psi L y) \text{ for } \psi$   
**by auto**  
**have** \*\*\*\*\*:  $\text{return True} = (\text{return False} \gg \text{return o Not})$   
**by auto**

**interpret** *ta\_bisim*: *Bisimulation\_Invariant*

$(\lambda(l, u) (l', u').$   
 $\text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(l, u) (l', u').$   
 $\text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
 $(\lambda(l, u) (l', u').$   
 $l' = l \wedge$   
 $(\forall c. c \in \text{clk\_set } (\text{conv\_A } A) \longrightarrow$

$u\ c = u'\ c))$   
 $(\lambda\_. \text{True})\ (\lambda\_. \text{True})$   
**by** (rule ta\_bisimulation[of conv\_A A])

**have** bisim2:

$(\exists u_0. (\forall c \in \{\text{Suc } 0..m\}. u_0\ c = 0) \wedge$   
 $\neg \text{Alw\_ev } (\lambda(l, u). \varphi\ l)\ ((\text{init}, s_0), u_0))$   
 $\longleftrightarrow (\neg \text{Alw\_ev } (\lambda(l, u). \varphi\ l)\ ((\text{init}, s_0), u_0))$   
**for**  $\varphi$   
**apply** safe  
**subgoal for**  $u$   
**apply** (subst (asm) ta\_bisim.Alw\_ev\_iff[of  $\_$   $(\lambda(l, u). \varphi\ l)\ \_ ((\text{init}, s_0), \lambda\_. 0))$ )  
**using** clocks\_I[of  $u\ \lambda\_. 0$ ] **unfolding** ta\_bisim.A\_B.equiv'\_def  
**apply** auto  
**done**  
**by** force

**have** bisim1:

$(\exists u_0. (\forall c \in \{\text{Suc } 0..m\}. u_0\ c = 0) \wedge \neg \text{Alw\_ev } (\lambda((L, s), \_). \neg \text{check\_bexp } \varphi\ L\ s)\ ((\text{init}, s_0),$   
 $u_0)) =$   
 $(\neg \text{Alw\_ev } (\lambda((L, s), \_). \neg \text{check\_bexp } \varphi\ L\ s)\ ((\text{init}, s_0), u_0))$  **for**  $\varphi$   
**using** bisim2[of  $\lambda\ (L, s). \neg \text{check\_bexp } \varphi\ L\ s$ ] **unfolding** \*\*\* .

**have** bisim3:

$(\forall u_0. (\forall c \in \{\text{Suc } 0..m\}. u_0\ c = 0) \longrightarrow$   
 $\text{leadsto } (\lambda((L, s), \_). \text{check\_bexp } \varphi\ L\ s)\ (\lambda((L, s), \_). \text{check\_bexp } \psi\ L\ s)\ ((\text{init}, s_0), u_0)) =$   
 $\text{leadsto } (\lambda((L, s), \_). \text{check\_bexp } \varphi\ L\ s)\ (\lambda((L, s), \_). \text{check\_bexp } \psi\ L\ s)\ ((\text{init}, s_0), u_0))$   
**for**  $\varphi\ \psi$   
**apply** safe  
**apply** force  
**subgoal for**  $u$   
**apply** (subst (asm) ta\_bisim.Leadsto\_iff[of  
 $\_$   $(\lambda((L, s), \_). \text{check\_bexp } \varphi\ L\ s)\ \_ (\lambda((L, s), \_). \text{check\_bexp } \psi\ L\ s)$   
 $\_ ((\text{init}, s_0), u)$   
 $])$ )  
**using** clocks\_I[of  $u\ \lambda\_. 0$ ] **unfolding** ta\_bisim.A\_B.equiv'\_def **by** auto  
**done**

**have** bisim4:

$(\forall u_0. (\forall c \in \{\text{Suc } 0..m\}. u_0\ c = 0) \longrightarrow \neg \text{deadlock } ((\text{init}, s_0), u_0))$   
 $\longleftrightarrow \neg \text{deadlock } ((\text{init}, s_0), u_0)$

**apply** safe  
**apply** fast  
**subgoal for**  $u$   
**apply** (subst (asm) ta\_bisim.deadlock\_iff[of  $\_ ((\text{init}, s_0), u)$ ])  
**using** clocks\_I[of  $u\ \lambda\_. 0$ ] **unfolding** ta\_bisim.A\_B.equiv'\_def **by** auto  
**done**

**have** bisim5:

$(\forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle (\text{init}, s_0), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge \varphi\ L'\ s'))$   
 $\longleftrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle (\text{init}, s_0), u_0 \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge \varphi\ L'\ s')$   
**for**  $\varphi\ L'\ s'$   
**unfolding** reaches\_steps'[symmetric]

```

apply safe
apply fast
subgoal for  $u' u$ 
  apply (drule ta_bisim.bisim.A_B_reaches[of  $\_ \_ ((init, s_0), u)$ ])
  subgoal
    using clocks_I[of  $u \lambda \_ . 0$ ] unfolding ta_bisim.equiv'_def by auto
    unfolding ta_bisim.equiv'_def by auto
done

```

```

define protect where
  protect = ( $(\lambda(l, u) (l', u')).$ 
     $conv\_A A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ ))

```

```

show ?thesis
unfolding deadlock_start_iff
using models_correct
apply simp
unfolding protect_def[symmetric]
apply sepref_to_hoare
apply sep_auto
unfolding model_checker_def reachability_checker'_def Alw_ev_checker_def leadsto_checker_def
apply (cases formula; simp)

```

— *EX*

```

subgoal premises prems for  $\varphi$ 
using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
apply (subst (asm) (2) F_reachable_correct'_new)
apply (rule prems; fail)
apply (subst (asm) bisim5)
apply sep_auto
unfolding final_fun_def F_def prems
apply (erule cons_post_rule)
apply (sep_auto simp: pure_def)
done

```

— *EG*

```

subgoal premises prems for  $\varphi$ 
using impl.Alw_ev_impl_hnr[
  to_hnr, unfolded hn_refine_def
]
unfolding final_fun_def F_def prems(2)
  UPPAAL_Reachability_Problem_precompiled'.final_fun_def[
    OF UPPAAL_Reachability_Problem_precompiled'_axioms
  ]
  UPPAAL_Reachability_Problem_precompiled_defs.F_def
apply sep_auto
unfolding **
subgoal
  apply (subst (asm) bisim1)
  apply (erule cons_post_rule)
  using init_state_in_state_set[of  $u_0$ ]
  apply (sep_auto simp: pure_def protect_def ***)
done
subgoal

```

```

    apply (subst (asm) bisim1)
    apply simp
    apply (erule cons_post_rule)
    using init_state_in_state_set[of u0]
    apply (sep_auto simp: pure_def protect_def ***)
  done
done

— AX
subgoal premises prems for  $\varphi$ 
  using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def]
  unfolding final_fun_def F_def
  unfolding UPPAAL_Reachability_Problem_precompiled_defs.F_def
  apply (subst
    UPPAAL_Reachability_Problem_precompiled'.final_fun_def[
      OF UPPAAL_Reachability_Problem_precompiled'_axioms
    ])
  apply (safe; clarsimp simp: prems(2))
  unfolding bisim2
  unfolding * ***
  subgoal
    using init_state_in_state_set[of u0]
    by (sep_auto simp: pure_def protect_def)
  subgoal
    unfolding protect_def *****
    apply (erule bind_rule)
    using init_state_in_state_set[of u0]
    apply (sep_auto simp: pure_def)
  done
done

— AG
subgoal premises prems for  $\varphi$ 
  using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
  apply (subst (asm) (2) F_reachable_correct'_new')
  apply (rule prems; fail)
  apply (subst (asm) bisim5)
  unfolding final_fun_def F_def prems
  apply (sep_auto simp: pure_def)
done

— Leadsto
subgoal premises prems for  $\varphi \psi$ 
  using impl.leadsto_impl_hnr'[
    OF final_fun_final, of Not oo check_bexp  $\psi$ ,
    to_hnr, unfolded hn_refine_def
  ]
  unfolding * F_def
  apply (simp add: prems(2))
  unfolding *** ***** bisim3 bisim4
  apply (erule bind_rule)
  apply (sep_auto simp: pure_def protect_def)
done
done

```

qed

**theorem** *model\_check'\_hoare*:

```

<emp>
  model_checker
< $\lambda r. \uparrow ((\neg \text{Bisim\_A.B.deadlock (init, s}_0, \lambda_. 0)) \longrightarrow r = ($ 
  conv  $N, (init, s_0, u_0) \models_m ax\_steps formula$ 
  )>t
using model_check'[to_hnr, unfolded hn_refine_def]
by (sep_auto simp: pure_def elim!: cons_post_rule)

```

**lemma** *Alw\_ev\_checker\_alt\_def'*:

```

Alw_ev_checker  $\equiv$ 
do {
  x  $\leftarrow$  let
    key = return  $\circ$  fst;
    sub = impl.subsumes_impl;
    copy = impl.state_copy_impl;
    start = impl.a0_impl;
    succs = impl.succs_P_impl' final_fun
  in dfs_map_impl' succs start sub key copy;
  _  $\leftarrow$  return ();
  return x
}
unfolding Alw_ev_checker_def by simp

```

**lemma** *leadsto\_checker\_alt\_def'*:

```

leadsto_checker  $\psi \equiv$ 
do {
  r  $\leftarrow$  let
    key = return  $\circ$  fst;
    sub = impl.subsumes_impl;
    copy = impl.state_copy_impl;
    start = impl.a0_impl;
    final = impl.F_impl;
    final' = (impl.Q_impl ( $\lambda(L, s). \neg \text{check\_bexp } \psi \ L \ s$ ));
    succs = impl.succs_P_impl' ( $\lambda(L, s). \neg \text{check\_bexp } \psi \ L \ s$ );
    succs' = impl.succs_impl';
    empty = impl.emptyness_check_impl;
    trace = impl.tracei
  in
    leadsto_impl copy succs start sub key succs' empty final final' trace;
  return ( $\neg r$ )
}
unfolding leadsto_checker_def by simp

```

**schematic\_goal** *succs\_P\_impl\_alt\_def*:

```

impl.succs_P_impl ( $\lambda(L, s). P \ L \ s$ )  $\equiv$  ?impl for P
unfolding impl.succs_P_impl_def[OF final_fun_final]
unfolding k_impl_alt_def
apply (abstract_let trans_fun trans_fun)
unfolding inv_fun_def[abs_def] trans_fun_def[abs_def] trans_s_fun_def trans_i_fun_def
trans_i_from_def
apply (abstract_let IArray (map IArray inv) inv_a)

```

```

apply (abstract_let IArray (map IArray trans_out_map) trans_out_map)
apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
apply (abstract_let IArray (map IArray trans_i_map) trans_i_map)
by (rule Pure.reflexive)

```

```

lemmas succs_P'_impl_alt_def =
  impl.succs_P_impl'_def[OF final_fun_final, unfolded succs_P_impl_alt_def]

```

```

lemmas succs_impl'_alt_def =
  impl.succs_impl'_def[unfolded succs_impl_alt_def]

```

```

lemma reachability_checker'_alt_def':
  reachability_checker'  $\equiv$ 
  do {
    x  $\leftarrow$  do {
      let key = return  $\circ$  fst;
      let sub = impl.subsumes_impl;
      let copy = impl.state_copy_impl;
      let start = impl.a0_impl;
      let final = impl.F_impl;
      let succs = impl.succs_impl;
      let empty = impl.emptyness_check_impl;
      let tracei = impl.tracei;
      pw_impl key copy tracei sub start final succs empty
    };
    _  $\leftarrow$  return ();
    return x
  }
  unfolding reachability_checker'_def by simp

```

```

schematic_goal reachability_checker'_alt_def:
  reachability_checker'  $\equiv$  ?impl
  unfolding reachability_checker'_alt_def' impl.succs_impl_def
  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
  unfolding k_impl_alt_def
  apply (abstract_let k_i k_i)

```

```

apply (abstract_let trans_fun trans_fun)
unfolding impl.init_dbm_impl_def impl.a0_impl_def
unfolding impl.F_impl_def
unfolding final_fun_def[abs_def]
unfolding impl.subsumes_impl_def
unfolding impl.emptyness_check_impl_def
unfolding impl.state_copy_impl_def
by (rule Pure.reflexive)

```

```

schematic_goal Alw_ev_checker_alt_def:
  Alw_ev_checker  $\equiv$  ?impl
  unfolding Alw_ev_checker_alt_def' final_fun_def
  impl.succs_P_impl_def[OF final_fun_final] impl.succs_P_impl'_def[OF final_fun_final]
  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def

```

**unfolding** *k\_impl\_alt\_def*  
**apply** (*abstract\_let k\_i k\_i*)

**apply** (*abstract\_let trans\_fun trans\_fun*)  
**unfolding** *impl.init\_dbm\_impl\_def impl.a0\_impl\_def*  
**unfolding** *impl.F\_impl\_def*  
**unfolding** *final\_fun\_def[abs\_def]*  
**unfolding** *impl.subsumes\_impl\_def*  
**unfolding** *impl.emptyness\_check\_impl\_def*  
**unfolding** *impl.state\_copy\_impl\_def*  
**by** (*rule Pure.reflexive*)

**schematic\_goal** *leadsto\_checker\_alt\_def*:

*leadsto\_checker*  $\equiv$  *?impl*  
**unfolding** *leadsto\_checker\_alt\_def'*  
**unfolding** *impl.F\_impl\_def impl.Q\_impl\_def[OF final\_fun\_final]*  
**unfolding** *impl.succs\_P\_impl'\_def[OF final\_fun\_final]*  
**unfolding** *impl.succs\_impl'\_def impl.succs\_impl\_def impl.succs\_P\_impl\_def[OF final\_fun\_final]*  
**unfolding** *impl.E\_op''\_impl\_def impl.abstr\_repair\_impl\_def impl.abstra\_repair\_impl\_def*  
**unfolding** *impl.start\_inv\_check\_impl\_def unbounded\_dbm\_impl\_def unbounded\_dbm'\_def*  
**unfolding** *k\_impl\_alt\_def*  
**apply** (*abstract\_let k\_i k\_i*)

**apply** (*abstract\_let trans\_fun trans\_fun*)  
**unfolding** *impl.init\_dbm\_impl\_def impl.a0\_impl\_def*  
**unfolding** *final\_fun\_def*  
**unfolding** *impl.subsumes\_impl\_def*  
**unfolding** *impl.emptyness\_check\_impl\_def*  
**unfolding** *impl.state\_copy\_impl\_def*  
**by** (*rule Pure.reflexive*)

**schematic\_goal** *reachability\_checker'\_alt\_def\_refined*:

*reachability\_checker'*  $\equiv$  *?impl*  
**unfolding** *reachability\_checker'\_alt\_def*  
**unfolding** *fw\_impl'\_int*  
**unfolding** *inv\_fun\_def trans\_fun\_def trans\_s\_fun\_def trans\_i\_fun\_def*  
**unfolding** *trans\_i\_from\_impl*  
**unfolding** *runf\_impl runt\_impl check\_g\_impl pairs\_by\_action\_impl check\_pred\_impl*  
**apply** (*abstract\_let IArray (map IArray inv) inv\_ia*)  
**apply** (*abstract\_let IArray (map IArray trans\_out\_map) trans\_out\_map*)  
**apply** (*abstract\_let IArray (map IArray trans\_in\_map) trans\_in\_map*)  
**apply** (*abstract\_let IArray (map IArray trans\_i\_map) trans\_i\_map*)  
**apply** (*abstract\_let IArray bounds bounds\_ia*)  
**apply** (*abstract\_let PF PF*)  
**apply** (*abstract\_let PT PT*)  
**unfolding** *PF\_alt\_def PT\_alt\_def*  
**apply** (*abstract\_let PROG' PROG'*)  
**unfolding** *PROG'\_def*  
**apply** (*abstract\_let length prog len\_prog*)  
**apply** (*abstract\_let IArray (map (map\_option stripf) prog) progf\_ia*)  
**apply** (*abstract\_let IArray (map (map\_option stript) prog) progt\_ia*)  
**apply** (*abstract\_let IArray prog prog\_ia*)  
**unfolding** *all\_actions\_by\_state\_impl*  
**apply** (*abstract\_let [0..<p]*)

```

apply (abstract_let [0..na])
apply (abstract_let {0..p})
apply (abstract_let [0..m+1])
by (rule Pure.reflexive)

```

**schematic\_goal** *Alw\_ev\_checker\_alt\_def\_refined*:

```

Alw_ev_checker  $\equiv$  ?impl
unfolding Alw_ev_checker_alt_def
unfolding fw_impl'_int
unfolding inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def
unfolding trans_i_from_impl
unfolding runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl
apply (abstract_let IArray (map IArray inv) inv_ia)
apply (abstract_let IArray (map IArray trans_out_map) trans_out_map)
apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
apply (abstract_let IArray (map IArray trans_i_map) trans_i_map)
apply (abstract_let IArray bounds bounds)
apply (abstract_let PF PF)
apply (abstract_let PT PT)
unfolding PF_alt_def PT_alt_def
apply (abstract_let PROG' PROG')
unfolding PROG'_def
apply (abstract_let length prog len_prog)
apply (abstract_let IArray (map (map_option stripf) prog) progf_ia)
apply (abstract_let IArray (map (map_option stript) prog) progt_ia)
apply (abstract_let IArray prog prog)
unfolding all_actions_by_state_impl
apply (abstract_let [0..p])
apply (abstract_let [0..na])
apply (abstract_let {0..p})
apply (abstract_let [0..m+1])
by (rule Pure.reflexive)

```

**schematic\_goal** *leadsto\_checker\_alt\_def\_refined*:

```

leadsto_checker  $\equiv$  ?impl
unfolding leadsto_checker_alt_def
unfolding fw_impl'_int
unfolding inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def
unfolding trans_i_from_impl
unfolding runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl
apply (abstract_let IArray (map IArray inv) inv_ia)
apply (abstract_let IArray (map IArray trans_out_map) trans_out_map)
apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
apply (abstract_let IArray (map IArray trans_i_map) trans_i_map)
apply (abstract_let IArray bounds bounds_ia)
apply (abstract_let PF PF)
apply (abstract_let PT PT)
unfolding PF_alt_def PT_alt_def
apply (abstract_let PROG' PROG')
unfolding PROG'_def
apply (abstract_let length prog len_prog)
apply (abstract_let IArray (map (map_option stripf) prog) progf_ia)
apply (abstract_let IArray (map (map_option stript) prog) progt_ia)
apply (abstract_let IArray prog prog_ia)

```

```

    unfolding all_actions_by_state_impl
    apply (abstract_let [0..<p])
    apply (abstract_let [0..<na])
    apply (abstract_let {0..<p})
    apply (abstract_let [0..<m+1])
    by (rule Pure.reflexive)

end

concrete_definition reachability_checker'
  uses UPPAAL_Reachability_Problem_precompiled'.reachability_checker'_alt_def_refined

concrete_definition Alw_ev_checker
  uses UPPAAL_Reachability_Problem_precompiled'.Alw_ev_checker_alt_def_refined

concrete_definition leadsto_checker
  uses UPPAAL_Reachability_Problem_precompiled'.leadsto_checker_alt_def_refined

context UPPAAL_Reachability_Problem_precompiled'
begin

lemmas model_checker_def_refined = model_checker_def[unfolded
  reachability_checker'.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
  Alw_ev_checker.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
  leadsto_checker.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
]

end

concrete_definition model_checker uses
  UPPAAL_Reachability_Problem_precompiled'.model_checker_def_refined

definition [code]:
  precondition mc p m k max_steps I T prog final bounds P s0 na ≡
  if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na k
  then
    model_checker p m max_steps I T prog bounds P s0 na k final
    ≫ (λ x. return (Some x))
  else return None

theorem model_check:
  <emp> precondition mc p m k max_steps I T prog formula bounds P s0 na
  <λ Some r ⇒ ↑(
    UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na k
  ∧
    (¬ has_deadlock (conv (N p I P T prog bounds)) max_steps (repeat 0 p, s0, λ_. 0)) →
    r = conv (N p I P T prog bounds), (repeat 0 p, s0, λ_. 0) ⊨max_steps formula
  ))
  | None ⇒ ↑(¬ UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds
  P s0 na k)
  >t
proof -
  define A where A ≡ conv (N p I P T prog bounds)
  define check where check ≡ A, (repeat 0 p, s0, λ_. 0) ⊨max_steps formula

```

```

note [sep_heap_rules] =
  UPPAAL_Reachability_Problem_precompiled'.model_check'_hoare[
    of p m max_steps I T prog bounds P s0 na k formula,
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    folded A_def check_def has_deadlock_def
  ]
show ?thesis
  unfolding UPPAAL_Reachability_Problem_precompiled_defs.init_def
  unfolding A_def[symmetric] check_def[symmetric]
  unfolding precondition_def by (sep_auto simp: model_checker.refine[symmetric])
qed

theorem model_check_alt:
  <emp> precondition p m k max_steps I T prog formula bounds P s0 na
  <λ r. ↑ (
    if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na k
    then r ≠ None ∧
      (¬ has_deadlock (conv (N p I P T prog bounds)) max_steps (repeat 0 p, s0, λ_. 0) →
        r = Some (
          conv (N p I P T prog bounds), (repeat 0 p, s0, λ_. 0) ⊨m ax_steps formula
        ))
    else r = None
  )>t
proof -
  define A where A ≡ conv (N p I P T prog bounds)
  define check where check ≡ A, (repeat 0 p, s0, λ_. 0) ⊨m ax_steps formula
  note [sep_heap_rules] =
    UPPAAL_Reachability_Problem_precompiled'.model_check'_hoare[
      of p m max_steps I T prog bounds P s0 na k formula,
      unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
      folded A_def check_def has_deadlock_def
    ]
  show ?thesis
  unfolding UPPAAL_Reachability_Problem_precompiled_defs.init_def
  unfolding A_def[symmetric] check_def[symmetric]
  unfolding precondition_def by (sep_auto simp: model_checker.refine[symmetric])
qed

prepare_code_thms dfs_map_impl'_def leadsto_impl_def

lemmas [code] =
  reachability_checker'_def
  Alw_ev_checker_def
  leadsto_checker_def
  model_checker_def[unfolded UPPAAL_Reachability_Problem_precompiled_defs.F_def PR_CONST_def]

export_code
  precondition Pure.type init_pred_check time_indep_check1 time_indep_check1 conjunction_check2
  checking SML

end
theory Simple_Expressions
  imports Main
begin

```

```

datatype ('a, 'b) bexp =
  true |
  not ('a, 'b) bexp |
  and ('a, 'b) bexp ('a, 'b) bexp |
  or ('a, 'b) bexp ('a, 'b) bexp |
  imply ('a, 'b) bexp ('a, 'b) bexp | — Boolean connectives
  eq ('a, 'b) exp ('a, 'b) exp |
  le ('a, 'b) exp ('a, 'b) exp |
  lt ('a, 'b) exp ('a, 'b) exp |
  ge ('a, 'b) exp ('a, 'b) exp |
  gt ('a, 'b) exp ('a, 'b) exp
and ('a, 'b) exp =
  const 'b | var 'a | if_then_else ('a, 'b) bexp ('a, 'b) exp ('a, 'b) exp |
  binop 'b  $\Rightarrow$  'b  $\Rightarrow$  'b ('a, 'b) exp ('a, 'b) exp | unop 'b  $\Rightarrow$  'b ('a, 'b) exp

inductive check_bexp :: ('a  $\rightarrow$  'b)  $\Rightarrow$  ('a, 'b :: linorder) bexp  $\Rightarrow$  bool  $\Rightarrow$  bool and is_val
  where
    check_bexp s true True |
    check_bexp s (not e) ( $\neg$  b) if check_bexp s e b |
    check_bexp s (and e1 e2) (a  $\wedge$  b) if check_bexp s e1 a check_bexp s e2 b |
    check_bexp s (or e1 e2) (a  $\vee$  b) if check_bexp s e1 a check_bexp s e2 b |
    check_bexp s (imply e1 e2) (a  $\longrightarrow$  b) if check_bexp s e1 a check_bexp s e2 b |
    check_bexp s (eq a b) (x = v) if is_val s a v is_val s b x |
    check_bexp s (le a b) (v  $\leq$  x) if is_val s a v is_val s b x |
    check_bexp s (lt a b) (v < x) if is_val s a v is_val s b x |
    check_bexp s (ge a b) (v  $\geq$  x) if is_val s a v is_val s b x |
    check_bexp s (gt a b) (v > x) if is_val s a v is_val s b x |
    is_val s (const c) c |
    is_val s (var x) v if s x = Some v |
    is_val s (if_then_else b e1 e2) (if bv then v1 else v2)
    if is_val s e1 v1 is_val s e2 v2 check_bexp s b bv |
    is_val s (binop f e1 e2) (f v1 v2) if is_val s e1 v1 is_val s e2 v2 |
    is_val s (unop f e) (f v) if is_val s e v

inductive_simps check_bexp_simps:
  check_bexp s bexp.true bv
  check_bexp s (bexp.not b) bv
  check_bexp s (bexp.and b1 b2) bv
  check_bexp s (bexp.or b1 b2) bv
  check_bexp s (bexp.imply b1 b2) bv
  check_bexp s (le i x) bv
  check_bexp s (lt i x) bv
  check_bexp s (ge i x) bv
  check_bexp s (eq i x) bv
  check_bexp s (gt i x) bv

inductive_simps is_val_simps:
  is_val s (const c) d
  is_val s (var x) v
  is_val s (if_then_else b e1 e2) v
  is_val s (binop f e1 e2) v
  is_val s (unop f e) v

```

**inductive\_cases** *check\_bexp\_elims*:

*check\_bexp s bexp.true bv*  
*check\_bexp s (bexp.not b) bv*  
*check\_bexp s (bexp.and b1 b2) bv*  
*check\_bexp s (bexp.or b1 b2) bv*  
*check\_bexp s (bexp.imply b1 b2) bv*  
*check\_bexp s (le i x) bv*  
*check\_bexp s (lt i x) bv*  
*check\_bexp s (ge i x) bv*  
*check\_bexp s (eq i x) bv*  
*check\_bexp s (gt i x) bv*

**inductive\_cases** *is\_val\_elims*:

*is\_val s (const c) d*  
*is\_val s (var x) v*  
*is\_val s (if\_then\_else b e1 e2) v*  
*is\_val s (binop f e1 e2) v*  
*is\_val s (unop f e) v*

**type\_synonym**

$(a, b) \text{ upd} = (a * (a, b) \text{ exp}) \text{ list}$

**definition** *is\_upd* **where**

$\text{is\_upd } s \text{ upd } s' \equiv \exists x e v. \text{ upd} = (x, e) \wedge \text{is\_val } s e v \wedge s' = s(x := \text{Some } v) \text{ for } \text{upd}$

**inductive** *is\_upds* **where**

$\text{is\_upds } s [] s$   
 $\text{is\_upds } s (x \# xs) s'' \text{ if } \text{is\_upd } s x s' \text{ is\_upds } s' xs s''$

**inductive\_cases** *is\_upds\_NilE*:

$\text{is\_upds } s [] s'$

**and** *is\_upds\_ConsE*:

$\text{is\_upds } s (e \# es) s'$

**inductive\_simps** *is\_upds\_Nil\_iff*:  $\text{is\_upds } s [] s'$

**and** *is\_upds\_Cons\_iff*:  $\text{is\_upds } s (f \# upds) s'$

**lemma** *is\_upds\_appendI*:

**assumes**  $\text{is\_upds } s \text{ upds1 } s' \text{ is\_upds } s' \text{ upds2 } s''$   
**shows**  $\text{is\_upds } s (\text{upds1} @ \text{upds2}) s''$   
**using** *assms* **by induction** (auto intro: *is\_upds.intros*)

**lemma** *is\_upds\_appendE*:

**assumes**  $\text{is\_upds } s (\text{upds1} @ \text{upds2}) s''$   
**obtains**  $s'$  **where**  $\text{is\_upds } s \text{ upds1 } s' \text{ is\_upds } s' \text{ upds2 } s''$   
**using** *assms* **by** (induction *upds1* arbitrary: *s*) (auto intro: *is\_upds.intros elim!*: *is\_upds\_ConsE*)

**lemma** *is\_upds\_NilD*:

$s' = s$  **if**  $\text{is\_upds } s [] s'$   
**using** *that* **by** (rule *is\_upds\_NilE*)

**lemma** *is\_upds\_all\_NilD*:

```

assumes is_upds s (concat upds) s' ( $\forall xs \in \text{set } upds. xs = []$ )
shows s' = s
using assms by (simp del: Nil_eq_concat_conv flip: concat_eq_Nil_conv) (rule is_upds_NilD)

lemma is_upd_const_simp:
  is_upd s (x, const c) s'  $\longleftrightarrow$  s' = s(x := Some c)
  unfolding is_upd_def by (simp add: is_val_simps)

end
theory Simple_Network_Language
imports Main
  Munta_Model_Checker.State_Networks
  Munta_Model_Checker.UPPAAL_State_Networks_Impl
  FinFun.FinFun
  Simple_Expressions
  Munta_Tagging
begin

```

## 6 Simple Networks of Automata With Broadcast Channels and Committed Locations

```

no_notation top_assn (true)

abbreviation concat_map where
  concat_map f xs  $\equiv$  concat (map f xs)

type_synonym
  ('c, 't, 's) invassn  $=$  's  $\Rightarrow$  ('c, 't) cconstraint

type_synonym
  ('a, 's, 'c, 't, 'x, 'v) transition  $=$ 
  's  $\times$  ('x, 'v) bexp  $\times$  ('c, 't) cconstraint  $\times$  'a  $\times$  ('x, 'v) upd  $\times$  'c list  $\times$  's

type_synonym
  ('a, 's, 'c, 't, 'x, 'v) sta  $=$ 
  's set  $\times$  's set  $\times$  ('a, 's, 'c, 't, 'x, 'v) transition set  $\times$  ('c, 't, 's) invassn

type_synonym
  ('a, 's, 'c, 't, 'x, 'v) nta  $=$  'a set  $\times$  ('a act, 's, 'c, 't, 'x, 'v) sta list  $\times$  ('x  $\rightarrow$  'v  $*$  'v)

context begin

qualified definition conv_t where conv_t  $\equiv$   $\lambda (l, b, g, a, f, r, l'). (l, b, \text{conv\_cc } g, a, f, r, l')$ 

qualified definition conv_A where conv_A  $\equiv$   $\lambda (C, U, T, I). (C, U, \text{conv\_t } T, \text{conv\_cc } o I)$ 

definition conv where
  conv  $\equiv$   $\lambda (\text{broadcast}, \text{automata}, \text{bounds}). (\text{broadcast}, \text{map conv\_A automata}, \text{bounds})$ 

end

```

**datatype** 'b label = Del | Internal 'b | Bin 'b | Broad 'b

**definition** bounded **where**

bounded bounds s  $\equiv$  dom s = dom bounds  $\wedge$   
 $(\forall x \in \text{dom } s. \text{fst } (\text{the } (\text{bounds } x)) \leq \text{the } (s \ x) \wedge \text{the } (s \ x) \leq \text{snd } (\text{the } (\text{bounds } x)))$

**definition** committed :: ('a, 's, 'c, 't, 'x, 'v) sta  $\Rightarrow$  's set **where**

committed A  $\equiv$  fst A

**definition** urgent :: ('a, 's, 'c, 't, 'x, 'v) sta  $\Rightarrow$  's set **where**

urgent A  $\equiv$  fst (snd A)

**definition** trans :: ('a, 's, 'c, 't, 'x, 'v) sta  $\Rightarrow$  ('a, 's, 'c, 't, 'x, 'v) transition set  
**where**

trans A  $\equiv$  fst (snd (snd A))

**definition** inv :: ('a, 's, 'c, 't, 'x, 'v) sta  $\Rightarrow$  ('c, 't, 's) invassn **where**

inv A  $\equiv$  snd (snd (snd A))

**no\_notation** step\_sn ( $\_ \vdash \langle \_, \_, \_ \rangle \rightarrow \_ \langle \_, \_, \_ \rangle$  [61,61,61,61,61] 61)

**no\_notation** steps\_sn ( $\_ \vdash \langle \_, \_, \_ \rangle \rightarrow^* \langle \_, \_, \_ \rangle$  [61, 61, 61,61,61] 61)

**datatype** 'a tag = ANY 'a | TRANS 'a | SEL 'a | SEND 'a | RECV 'a

**inductive** step\_u ::

(*'a, 's, 'c, 't :: time, 'x, 'v :: linorder*) nta  $\Rightarrow$  's list  $\Rightarrow$  ('x  $\rightarrow$  'v)  $\Rightarrow$  ('c, 't) cval  
 $\Rightarrow$  'a label  $\Rightarrow$  's list  $\Rightarrow$  ('x  $\rightarrow$  'v)  $\Rightarrow$  ('c, 't) cval  $\Rightarrow$  bool  
 $(\_ \vdash \langle \_, \_, \_ \rangle \rightarrow \_ \langle \_, \_, \_ \rangle$  [61,61,61,61,61] 61)

**where**

step\_t:

[[  
 "target invariant" |  $\forall p < \text{length } N. u \oplus d \vdash \text{inv } (N ! p) (L ! p);$   
 "nonnegative" |  $d \geq 0;$   
 "urgent" |  $(\exists p < \text{length } N. L ! p \in \text{urgent } (N ! p)) \longrightarrow d = 0;$   
 "bounded" | bounded B s  
 ]]

$\Longrightarrow (\text{broadcast}, N, B) \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L, s, u \oplus d \rangle$  |

step\_int:

[[  
 TRANS "silent" |  $(l, b, g, \text{Sil } a, f, r, l') \in \text{trans } (N ! p);$   
 "committed" |  $l \in \text{committed } (N ! p) \vee (\forall p < \text{length } N. L ! p \notin \text{committed } (N ! p));$   
 "bexp" | check\_bexp s b True;  
 "guard" |  $u \vdash g;$   
 "target invariant" |  $\forall p < \text{length } N. u' \vdash \text{inv } (N ! p) (L' ! p);$   
 "loc" |  $L!p = l;$   
 "range" |  $p < \text{length } L;$   
 "new loc" |  $L' = L[p := l'];$   
 "new valuation" |  $u' = [r \rightarrow 0]u;$   
 "is\_upd" | is\_upds s f s';  
 "bounded" | bounded B s'  
 ]]

$\Longrightarrow (\text{broadcast}, N, B) \vdash \langle L, s, u \rangle \rightarrow_{\text{Internal } a} \langle L', s', u' \rangle$  |

step\_bin:

$$\begin{aligned} & \llbracket \\ & \quad \text{"not broadcast"} \mid (a \notin \text{broadcast}); \\ & \quad \text{TRANS "in"} \mid (l1, b1, g1, \text{In } a, f1, r1, l1') \in \text{trans } (N ! p); \\ & \quad \text{TRANS "out"} \mid (l2, b2, g2, \text{Out } a, f2, r2, l2') \in \text{trans } (N ! q); \\ & \quad \text{"committed"} \mid \\ & \quad \quad l1 \in \text{committed } (N ! p) \vee l2 \in \text{committed } (N ! q) \vee (\forall p < \text{length } N. L ! p \notin \text{committed } \\ & \quad (N ! p)); \\ & \quad \text{"bexp"} \mid \text{check\_bexp } s \ b1 \ \text{True}; \text{"bexp"} \mid \text{check\_bexp } s \ b2 \ \text{True}; \\ & \quad \text{"guard"} \mid u \vdash g1; \text{"guard"} \mid u \vdash g2; \\ & \quad \text{"target invariant"} \mid \forall p < \text{length } N. u' \vdash \text{inv } (N ! p) \ (L' ! p); \\ & \quad \text{RECV "loc"} \mid L!p = l1; \text{SEND "loc"} \mid L!q = l2; \\ & \quad \text{RECV "range"} \mid p < \text{length } L; \text{SEND "range"} \mid q < \text{length } L; \\ & \quad \text{"different"} \mid p \neq q; \\ & \quad \text{"new loc"} \mid L' = L[p := l1', q := l2']; \\ & \quad \text{"new valuation"} \mid u' = [r1 @ r2 \rightarrow 0]u; \\ & \quad \text{"upd"} \mid \text{is\_upds } s \ f1 \ s'; \text{"upd"} \mid \text{is\_upds } s' \ f2 \ s''; \\ & \quad \text{"bounded"} \mid \text{bounded } B \ s'' \\ & \rrbracket \\ & \implies (\text{broadcast}, N, B) \vdash \langle L, s, u \rangle \rightarrow_{\text{Bin } a} \langle L', s'', u' \rangle \mid \\ & \text{step\_broad:} \\ & \llbracket \\ & \quad \text{"broadcast"} \mid a \in \text{broadcast}; \\ & \quad \text{TRANS "out"} \mid (l, b, g, \text{Out } a, f, r, l') \in \text{trans } (N ! p); \\ & \quad \text{TRANS "in"} \mid (\forall p \in \text{set } ps. (L ! p, bs \ p, gs \ p, \text{In } a, fs \ p, rs \ p, ls' \ p) \in \text{trans } (N ! p)); \\ & \quad \text{"committed"} \mid (l \in \text{committed } (N ! p) \vee (\exists p \in \text{set } ps. L ! p \in \text{committed } (N ! p)) \\ & \quad \vee (\forall p < \text{length } N. L ! p \notin \text{committed } (N ! p))); \\ & \quad \text{"bexp"} \mid \text{check\_bexp } s \ b \ \text{True}; \\ & \quad \text{"bexp"} \mid \forall p \in \text{set } ps. \text{check\_bexp } s \ (bs \ p) \ \text{True}; \\ & \quad \text{"guard"} \mid u \vdash g; \\ & \quad \text{"guard"} \mid \forall p \in \text{set } ps. u \vdash gs \ p; \\ & \quad \text{"maximal"} \mid \forall q < \text{length } N. q \notin \text{set } ps \wedge p \neq q \\ & \quad \quad \rightarrow (\forall b \ g \ f \ r \ l'. (L!q, b, g, \text{In } a, f, r, l') \in \text{trans } (N ! q)) \\ & \quad \quad \rightarrow \neg \text{check\_bexp } s \ b \ \text{True} \vee \neg u \vdash g; \\ & \quad \text{"target invariant"} \mid \forall p < \text{length } N. u' \vdash \text{inv } (N ! p) \ (L' ! p); \\ & \quad \text{SEND "loc"} \mid L!p = l; \\ & \quad \text{SEND "range"} \mid p < \text{length } L; \\ & \quad \text{SEL "range"} \mid \text{set } ps \subseteq \{0..<\text{length } N\}; \\ & \quad \text{SEL "not sender"} \mid p \notin \text{set } ps; \\ & \quad \text{SEL "distinct"} \mid \text{distinct } ps; \\ & \quad \text{SEL "sorted"} \mid \text{sorted } ps; \\ & \quad \text{"new loc"} \mid L' = (\text{fold } (\lambda p \ L. L[p := ls' \ p]) \ ps \ L)[p := l']; \\ & \quad \text{"new valuation"} \mid u' = [r @ \text{concat } (\text{map } rs \ ps) \rightarrow 0]u; \\ & \quad \text{"upd"} \mid \text{is\_upds } s \ f \ s'; \\ & \quad \text{"upds"} \mid \text{is\_upds } s' \ (\text{concat\_map } fs \ ps) \ s''; \\ & \quad \text{"bounded"} \mid \text{bounded } B \ s'' \\ & \rrbracket \\ & \implies (\text{broadcast}, N, B) \vdash \langle L, s, u \rangle \rightarrow_{\text{Broad } a} \langle L', s'', u' \rangle \\ & \text{lemmas [intro?]} = \text{step\_u.intros[unfolded TAG\_def]}
\end{aligned}$$

Comments:

- Should there be an error transition + state if states run of bounds or updates are undefined?  
Then one could run a reachability check for the error state.
- Should the state be total?

- Note that intermediate states are allowed to run out of bounds

**definition** *steps\_u* ::

$(\text{'a}, \text{'s}, \text{'c}, \text{'t} :: \text{time}, \text{'x}, \text{'v} :: \text{linorder}) \text{ nta} \Rightarrow \text{'s list} \Rightarrow (\text{'x} \rightarrow \text{'v}) \Rightarrow (\text{'c}, \text{'t}) \text{ cval}$   
 $\Rightarrow \text{'s list} \Rightarrow (\text{'x} \rightarrow \text{'v}) \Rightarrow (\text{'c}, \text{'t}) \text{ cval} \Rightarrow \text{bool}$   
 $(\_ \vdash \langle \_, \_, \_ \rangle \rightarrow^* \langle \_, \_, \_ \rangle [61, 61, 61, 61, 61] \ 61)$

**where**

$A \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \equiv$   
 $(\lambda (L, s, u) (L', s', u'). \exists a. A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle)^{**} (L, s, u) (L', s', u')$

**Misc lemma** *clock\_val\_concat\_iff*:

$\langle u \vdash \text{concat } gs \longleftrightarrow (\forall g \in \text{set } gs. u \vdash g) \rangle$

**by** (*auto intro: guard\_concat\_concat\_guard*)

**lemma** *clock\_val\_append\_iff*:

$\langle u \vdash g1 @ g2 \longleftrightarrow u \vdash g1 \wedge u \vdash g2 \rangle$

**proof** —

**have**  $u \vdash g1 @ g2 \longleftrightarrow u \vdash \text{concat } [g1, g2]$

**by** *simp*

**also have**  $\dots \longleftrightarrow u \vdash g1 \wedge u \vdash g2$

**unfolding** *clock\_val\_concat\_iff* **by** *simp*

**finally show** *?thesis* .

**qed**

## 6.1 Product Construction

**locale** *Prod\_TA\_Defs* =

**fixes**  $A :: (\text{'a}, \text{'s}, \text{'c}, \text{'t}, \text{'x}, \text{'v} :: \text{linorder}) \text{ nta}$

**begin**

**definition**

*broadcast* = *fst A*

**definition**

$N \ i = \text{fst } (\text{snd } A) ! i$

**definition**

*bounds* = *snd (snd A)*

**definition** — Number of processes

$n\_ps = \text{length } (\text{fst } (\text{snd } A))$

**definition** *states* :: *'s list set* **where**

$\text{states} \equiv \{L. \text{length } L = n\_ps \wedge$   
 $(\forall i. i < n\_ps \longrightarrow L ! i \in (\bigcup (l, e, g, a, r, u, l') \in (\text{trans } (N \ i)). \{l, l'\})))\}$

**definition**

$\text{prod\_inv} \equiv \lambda(L, s). \text{if } L \in \text{states} \text{ then } \text{concat } (\text{map } (\lambda i. \text{inv } (N \ i) (L ! i)) [0..<n\_ps]) \text{ else } []$

**definition**

*trans\_int* =  
 $\{((L, s), g, \text{Internal } a, r, (L', s')) \mid L \ s \ l \ b \ g \ f \ p \ a \ r \ l' \ L' \ s'.$   
 $(l, b, g, \text{Sil } a, f, r, l') \in \text{trans } (N \ p) \wedge$   
 $(l \in \text{committed } (N \ p) \vee (\forall p < n\_ps. L ! p \notin \text{committed } (N \ p))) \wedge$

$$\begin{aligned}
& L!p = l \wedge p < \text{length } L \wedge L' = L[p := l'] \wedge \text{is\_upds } s \text{ f } s' \wedge \text{check\_bexp } s \text{ b } \text{True} \wedge \\
& L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s' \\
& \}
\end{aligned}$$

**definition**

$$\begin{aligned}
\text{trans\_bin} = & \\
& \{((L, s), g1 @ g2, \text{Bin } a, r1 @ r2, (L', s')) \mid \\
& \quad L \text{ s } L' \text{ s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'. \\
& \quad a \notin \text{broadcast} \wedge \\
& \quad (l1, b1, g1, \text{In } a, f1, r1, l1') \in \text{trans } (N \text{ p}) \wedge \\
& \quad (l2, b2, g2, \text{Out } a, f2, r2, l2') \in \text{trans } (N \text{ q}) \wedge \\
& \quad (l1 \in \text{committed } (N \text{ p}) \vee l2 \in \text{committed } (N \text{ q}) \vee (\forall p < n\_ps. L!p \notin \text{committed } (N \text{ p}))) \wedge \\
& \quad L!p = l1 \wedge L!q = l2 \wedge p < \text{length } L \wedge q < \text{length } L \wedge p \neq q \wedge \\
& \quad \text{check\_bexp } s \text{ b1 } \text{True} \wedge \text{check\_bexp } s \text{ b2 } \text{True} \wedge \\
& \quad L' = L[p := l1', q := l2'] \wedge \text{is\_upds } s \text{ f1 s' } \wedge \text{is\_upds } s' \text{ f2 s'' } \wedge \\
& \quad L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s'' \\
& \}
\end{aligned}$$

**definition**

$$\begin{aligned}
\text{trans\_broad} = & \\
& \{((L, s), g @ \text{concat } (\text{map } gs \text{ ps}), \text{Broad } a, r @ \text{concat } (\text{map } rs \text{ ps}), (L', s')) \mid \\
& \quad L \text{ s } L' \text{ s' s'' a p l b g f r l' bs gs fs rs ls' ps. \\
& \quad a \in \text{broadcast} \wedge \\
& \quad (l, b, g, \text{Out } a, f, r, l') \in \text{trans } (N \text{ p}) \wedge \\
& \quad (\forall p \in \text{set } ps. (L!p, bs \text{ p}, gs \text{ p}, \text{In } a, fs \text{ p}, rs \text{ p}, ls' \text{ p}) \in \text{trans } (N \text{ p})) \wedge \\
& \quad (l \in \text{committed } (N \text{ p}) \vee (\exists p \in \text{set } ps. L!p \in \text{committed } (N \text{ p}))) \wedge \\
& \quad \vee (\forall p < n\_ps. L!p \notin \text{committed } (N \text{ p}))) \wedge \\
& \quad (\forall q < n\_ps. q \notin \text{set } ps \wedge p \neq q \longrightarrow \\
& \quad \quad \neg (\exists b \text{ g f r l'. } (L!q, b, g, \text{In } a, f, r, l') \in \text{trans } (N \text{ q}) \wedge \text{check\_bexp } s \text{ b } \text{True})) \wedge \\
& \quad L!p = l \wedge \\
& \quad p < \text{length } L \wedge \text{set } ps \subseteq \{0..<n\_ps\} \wedge p \notin \text{set } ps \wedge \text{distinct } ps \wedge \text{sorted } ps \wedge \\
& \quad \text{check\_bexp } s \text{ b } \text{True} \wedge (\forall p \in \text{set } ps. \text{check\_bexp } s \text{ (bs p) } \text{True}) \wedge \\
& \quad L' = (\text{fold } (\lambda p \text{ L. } L[p := ls' \text{ p}]) \text{ ps } L)[p := l'] \wedge \\
& \quad \text{is\_upds } s \text{ f s' } \wedge \text{is\_upds } s' (\text{concat\_map } fs \text{ ps}) \text{ s'' } \wedge \\
& \quad L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s'' \\
& \}
\end{aligned}$$

**definition**

$$\text{trans\_prod} = \text{trans\_int} \cup \text{trans\_bin} \cup \text{trans\_broad}$$

**definition**

$$\text{prod\_ta} = (\text{trans\_prod}, \text{prod\_inv} :: ('s \text{ list} \times ('x \Rightarrow 'v \text{ option}) \Rightarrow \_))$$

**lemma** *inv\_of\_prod[simp]:*

$$\text{inv\_of prod\_ta} = \text{prod\_inv}$$

**unfolding** *prod\\_ta\_def* *inv\_of\_def* **by** *simp*

**lemma** *trans\_of\_prod[simp]:*

$$\text{trans\_of prod\_ta} = \text{trans\_prod}$$

**unfolding** *prod\\_ta\_def* *trans\_of\_def* **by** *simp*

**lemma** *A\_split:*

$$\langle A = (\text{broadcast}, \text{map } N \text{ [0..<n\_ps]}, \text{bounds}) \rangle$$

**unfolding** *broadcast\_def* *N\_def* *bounds\_def* *n\\_ps\_def* **by** (*cases* *A*) (*simp* *add: List.map\_nth*)

```

lemma map_N_n_ps_simp:
  map N [0.. $n\_ps$ ] ! p = N p
  unfolding N_def n_ps_def List.map_nth ..

lemma N_split_simp[simp]:
  assumes A = (broadcast', N', B)
  shows N' ! i = N i
  unfolding N_def unfolding assms by simp

lemma state_preservation_updI:
  assumes  $l' \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N\ p). \{l, l'\})\ L \in \text{states}$ 
  shows  $L[p := l'] \in \text{states}$ 
  using assms unfolding states_def by (fastforce simp: nth_list_update')

lemma state_preservation_fold_updI:
  assumes  $\forall p \in \text{set } ps. ls' p \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N\ p). \{l, l'\})\ L \in \text{states}$ 
  shows fold ( $\lambda p\ L. L[p := ls' p]$ ) ps L  $\in \text{states}$ 
  using assms by (induction ps arbitrary: L) (auto intro: state_preservation_updI)

lemma state_set_states:
  Simulation_Graphs_TA.state_set prod_ta  $\subseteq \{(l, s). l \in \text{states}\}$ 
  unfolding prod_ta_def state_set_def
  unfolding trans_of_def trans_prod_def
  unfolding trans_int_def trans_bin_def trans_broad_def
  by auto (auto intro: state_preservation_updI state_preservation_fold_updI)

lemma trans_prod_bounded_inv:
   $\langle \text{bounded bounds } s' \rangle$  if  $\langle ((L, s), g, a, r, (L', s')) \in \text{trans\_prod} \rangle$ 
  using that unfolding bounds_def trans_prod_def trans_int_def trans_bin_def trans_broad_def
  by (auto simp: bounds_def)

lemma trans_prod_states_inv:
   $\langle L' \in \text{states} \rangle$  if  $\langle ((L, s), g, a, r, (L', s')) \in \text{trans\_prod} \rangle$   $\langle L \in \text{states} \rangle$ 
  using that
  unfolding bounds_def trans_prod_def trans_int_def trans_bin_def trans_broad_def
  apply clarsimp
  apply safe
    apply (force intro!: state_preservation_updI)
    apply (force intro!: state_preservation_updI)
    apply (force intro!: state_preservation_updI)
    apply (force intro!: state_preservation_updI)
    apply (force intro!: state_preservation_updI)
    apply (fastforce intro!: state_preservation_updI state_preservation_fold_updI)
  subgoal
    apply (rule state_preservation_updI)
    apply force
    apply (force intro!: state_preservation_fold_updI)
    done
  apply (fastforce intro!: state_preservation_updI state_preservation_fold_updI)
  done

end

```

```

locale Prod_TA_sem =
  Prod_TA_Defs A for A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder) nta
begin

lemma prod_invI[intro]:
   $\langle u \vdash \text{prod\_inv } (L, s) \rangle$  if  $\langle \forall p < n\_ps. u \vdash \text{Simple\_Network\_Language.inv } (N p) (L ! p) \rangle$ 
  using that unfolding prod_inv_def by (auto intro!: guard_concat)

lemma prod_invD[dest]:
   $\langle \forall p < n\_ps. u \vdash \text{Simple\_Network\_Language.inv } (N p) (L ! p) \rangle$  if  $\langle u \vdash \text{prod\_inv } (L, s) \rangle$   $\langle L \in \text{states} \rangle$ 
  using that unfolding prod_inv_def by (auto elim: concat_guard)

lemma delay_sound:
  assumes prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$   $L \in \text{states}$  bounded bounds s
     $\forall N \in \text{set } (\text{fst } (\text{snd } A)). \text{urgent } N = \{\}$ 
  shows  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$ 
proof -
  from assms(4) have  $l \notin \text{urgent } (N p)$  if  $p < n\_ps$  for  $p$   $l$ 
    using that unfolding N_def n_ps_def by auto
  then show ?thesis
    by (subst A_split) (use assms in  $\langle \text{cases}, \text{auto intro!}: \text{step\_t simp: TAG\_def} \rangle$ )
qed

lemma action_sound:
   $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$  if prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ 
  using that
proof cases
  case prems: (1 g r)
  note [simp add] = map_N_n_ps_simp clock_val_append_iff clock_val_concat_iff
  from  $\langle \text{prod\_ta} \vdash (L, s) \rightarrow^{g,a,r} (L', s') \rangle$  [THEN state_setI2] have  $L' \in \text{states}$ 
    using state_set_states that by fast
  from  $\langle \text{prod\_ta} \vdash (L, s) \rightarrow^{g,a,r} (L', s') \rangle$  show ?thesis
    unfolding trans_of_prod trans_prod_def
  proof safe
  assume  $((L, s), g, a, r, L', s') \in \text{trans\_int}$ 
  then show  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
    unfolding trans_int_def
    apply clarsimp
    using prems  $\langle L' \in \_ \rangle$ 
    by (subst A_split) (intro step_int; simp add: TAG_def; elim prod_invD; assumption)
  next
  assume  $((L, s), g, a, r, L', s') \in \text{trans\_bin}$ 
  then show  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 
    unfolding trans_bin_def
    using prems  $\langle L' \in \_ \rangle$ 
    apply clarsimp
    apply (subst A_split, standard)
    apply (assumption | simp; elim prod_invD; assumption) +
  done
next
  assume  $((L, s), g, a, r, L', s') \in \text{trans\_broad}$ 
  then show  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$ 

```

```

    using prems  $\langle L' \in \_ \rangle$ 
    unfolding trans_broad_def inv_of_prod
    apply clarsimp
    apply (subst A_split)
    apply standard
    apply (simp; elim prod_invD; assumption)+
    apply fastforce+
  done
qed
qed

lemma bounded_inv:
   $\langle \text{bounded bounds } s' \rangle$  if  $\langle A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \rangle$ 
  using that unfolding bounds_def by cases (simp add: TAG_def)+

lemma states_inv:
   $\langle L' \in \text{states} \rangle$  if  $\langle A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \rangle$   $\langle L \in \text{states} \rangle$ 
  using that unfolding bounds_def
proof cases
  case (step_t N d B broadcast)
  with  $\langle L \in \text{states} \rangle$  show ?thesis
  by simp
next
  case prems[unfolded TAG_def]: (step_int l b g a f r l' N' p B broadcast)
  from  $\langle A = \_ \rangle$  prems(3) have  $l' \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N p). \{l, l'\})$ 
  by force
  with  $\langle L \in \text{states} \rangle$  show ?thesis
  unfolding  $\langle L' = \_ \rangle$  by (intro state_preservation_updI)
next
  case prems[unfolded TAG_def]:
    (step_bin broadcast a l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s' B)
  from  $\langle A = \_ \rangle$  prems(4, 5) have
     $l1' \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N p). \{l, l'\})$ 
     $l2' \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N q). \{l, l'\})$ 
  by force+
  with  $\langle L \in \text{states} \rangle$  show ?thesis
  unfolding  $\langle L' = \_ \rangle$  by (intro state_preservation_updI)
next
  case prems[unfolded TAG_def]: (step_broad a broadcast l b g f r l' N' p ps bs gs fs rs ls' s' B)
  from  $\langle A = \_ \rangle$  prems(4, 5) have
     $l' \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N p). \{l, l'\})$ 
     $\forall q \in \text{set } ps. ls' q \in (\bigcup (l, b, g, a, r, u, l') \in \text{trans } (N q). \{l, l'\})$ 
  by force+
  with  $\langle L \in \text{states} \rangle$  show ?thesis
  unfolding  $\langle L' = \_ \rangle$  by (intro state_preservation_updI state_preservation_fold_updI)
qed

end

locale Prod_TA =
  Prod_TA_sem A for A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder) nta +
  assumes broadcast_receivers_unguarded:
     $\forall p < n\_ps. \forall l b g a f r l'. (l, b, g, \text{In } a, f, r, l') \in \text{trans } (N p) \wedge a \in \text{broadcast} \longrightarrow g = []$ 

```

```

begin

lemma action_complete:
  prod_ta ⊢ ⟨(L, s), u⟩ →a ⟨(L', s'), u'⟩
  if A ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩ a ≠ Del L ∈ states bounded bounds s
using that(1) proof cases
  case (step_t N d B broadcast)
  then show ?thesis
    using that(2) by auto
next
case prems[unfolded TAG_def]: (step_int l b g a' f r l' N' p B broadcast')
have [simp]:
  B = bounds broadcast' = broadcast length N' = n_ps
  unfolding bounds_def broadcast_def n_ps_def unfolding prems(1) by simp+
have [simp]:
  N' ! i = N i for i
  unfolding N_def unfolding prems(1) by simp
have prod_ta ⊢ (L, s) →g, Internal a', r (L', s')
proof -
  from prems ⟨L ∈ states⟩ ⟨bounded _ s⟩ have ((L, s), g, Internal a', r, (L', s')) ∈ trans_int
  unfolding trans_int_def
  by simp (rule exI conjI HOL.refl | assumption | (simp; fail))+
  then show ?thesis
    unfolding prod_ta_def trans_of_def trans_prod_def by simp
qed
moreover have u ⊢ g
  by (rule prems)
moreover have u' ⊢ inv_of prod_ta (L', s')
  using prems(7) by auto
moreover have u' = [r → 0]u
  by (rule prems)
ultimately show ?thesis
  unfolding ⟨a = _⟩ ..
next
case prems[unfolded TAG_def]:
  (step_bin a' broadcast' l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s'' B)
have [simp]:
  B = bounds broadcast' = broadcast length N' = n_ps
  unfolding bounds_def broadcast_def n_ps_def unfolding prems(1) by simp+
have [simp]:
  N' ! i = N i for i
  unfolding N_def unfolding prems(1) by simp
have prod_ta ⊢ (L, s) →g1 @ g2, Bin a', r1 @ r2 (L', s')
proof -
  from prems ⟨L ∈ states⟩ ⟨bounded bounds s⟩ have
    ((L, s), g1 @ g2, Bin a', r1 @ r2, (L', s')) ∈ trans_bin
  unfolding trans_bin_def
  using [[simproc add: ex_reorder4]]
  by simp (rule exI conjI HOL.refl | assumption | fast)+
  then show ?thesis
    unfolding prod_ta_def trans_of_def trans_prod_def by simp
qed
moreover have u ⊢ g1 @ g2

```

```

    using  $\langle u \vdash g1 \rangle \langle u \vdash g2 \rangle$  by (rule guard_append)
  moreover have  $u' \vdash \text{inv\_of prod\_ta } (L', s')$ 
    using prems by auto
  moreover have  $u' = [r1 @ r2 \rightarrow 0]u$ 
    by (rule prems)
  ultimately show ?thesis
    unfolding  $\langle a = \_ \rangle$  ..
next
  case prems[unfolded TAG_def]: (step_broad a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s''
B)
  have [simp]:
     $B = \text{bounds broadcast}' = \text{broadcast length } N' = n\_ps$ 
    unfolding bounds_def broadcast_def n_ps_def unfolding prems(1) by simp+
  have [simp]:
     $N' ! i = N i$  for  $i$ 
    unfolding N_def unfolding prems(1) by simp
  let  $?r = r @ \text{concat } (\text{map rs ps})$  and  $?g = g @ \text{concat } (\text{map gs ps})$ 
  have  $\text{prod\_ta} \vdash (L, s) \longrightarrow^{?g, \text{Broad } a', ?r} (L', s')$ 
  proof -
    have *:  $\neg u \vdash g \longleftrightarrow \text{False}$  if
       $p < n\_ps$   $(l, b, g, \text{In } a', f, r, l') \in \text{Simple\_Network\_Language.trans } (N p)$ 
       $a' \in \text{broadcast}$ 
      for  $l b g a' f r l' p$ 
    proof -
      from that broadcast_receivers_unguarded have  $\langle g = [] \rangle$ 
      by blast
      then show ?thesis
        by auto
    qed
  from prems  $\langle L \in \text{states} \rangle \langle \text{bounded bounds } s \rangle$  have  $((L, s), ?g, \text{Broad } a', ?r, (L', s')) \in \text{trans\_broad}$ 
    unfolding trans_broad_def
    by clarsimp
    (intro exI conjI HOL.refl; (rule HOL.refl | assumption | fastforce simp: *))
  then show ?thesis
    unfolding prod_ta_def trans_of_def trans_prod_def by simp
qed
moreover have  $u \vdash ?g$ 
  using prems by (auto intro!: guard_append guard_concat)
moreover have  $u' \vdash \text{inv\_of prod\_ta } (L', s')$ 
  using prems by auto
moreover have  $u' = [?r \rightarrow 0]u$ 
  by (rule prems)
ultimately show ?thesis
  unfolding  $\langle a = \_ \rangle$  ..
qed

lemma delay_complete:
  assumes  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$ 
  obtains  $d$  where  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$ 
  using assms
proof cases
  case prems: (step_t N' d B broadcast)
  have [simp]:

```

```

    length N' = n_ps
    unfolding n_ps_def unfolding prems(1) by simp+
  have [simp]:
    N' ! i = N i for i
    unfolding N_def unfolding prems(1) by simp
  from prems show ?thesis
  by (intro that[of d]; unfold ⟨u' = _⟩ ⟨L' = L⟩ ⟨s' = s⟩ TAG_def)
    (rule step_t.intros; (unfold inv_of_prod; rule prod_invI)?; simp)
qed

lemma step_iff:
  (∃ a. A ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩) ⟷ prod_ta ⊢ ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩
  if bounded bounds s L ∈ states ∀ N ∈ set (fst (snd A)). urgent N = {}
  using that(1,2)
  apply safe
  subgoal for a
    by (cases a; blast dest: action_complete elim: delay_complete)
  by (auto intro: action_sound delay_sound[OF _ _ _ that(3)])

end

end

theory TA_Impl_Misc2
  imports
    Manta_Base.TA_Impl_Misc
    HOL-Library.Sublist
    List-Index.List_Index
    Automatic_Refinement.Misc
begin

lemma mem_nth:
  x ∈ set xs ⟹ ∃ i < length xs. xs ! i = x
  by (metis index_less_size_conv nth_index)

lemma union_subsetI:
  A ∪ B ⊆ C ∪ D if A ⊆ C B ⊆ D
  using that by blast

lemma map_eq_imageD:
  f ' set xs = set ys if map f xs = ys
  using that by auto

lemma if_contract:
  (if a then x else if b then x else y) = (if a ∨ b then x else y) for a x b y
  by (rule SMT.z3_rule)

More intros   named__theorems more__intros
named__theorems more__elims
lemmas [more__intros] =
  image_eqI[rotated] CollectI subsetI

lemmas [more__elims] =
  CollectE

```

**Finiteness lemma** *finite\_prodI*:

*finite*  $\{(a,b). P\ a \wedge Q\ b\}$  **if** *finite*  $\{a. P\ a\}$  *finite*  $\{a. Q\ a\}$   
**using** *that* **by** *simp*

**lemma** *finite\_prodI3*:

*finite*  $\{(a,b,c). P\ a \wedge Q\ b \wedge Q1\ c\}$   
**if** *finite*  $\{a. P\ a\}$  *finite*  $\{a. Q\ a\}$  *finite*  $\{a. Q1\ a\}$   
**using** *that* **by** *simp*

**lemma** *finite\_prodI4*:

*finite*  $\{(a,b,c,d). P\ a \wedge Q\ b \wedge Q1\ c \wedge Q2\ d\}$   
**if** *finite*  $\{a. P\ a\}$  *finite*  $\{a. Q\ a\}$  *finite*  $\{a. Q1\ a\}$  *finite*  $\{a. Q2\ a\}$   
**using** *that* **by** *simp*

**lemma** *finite\_prodI5*:

*finite*  $\{(a,b,c,d,e). P\ a \wedge Q\ b \wedge Q1\ c \wedge Q2\ d \wedge Q3\ e\}$   
**if** *finite*  $\{a. P\ a\}$  *finite*  $\{a. Q\ a\}$  *finite*  $\{a. Q1\ a\}$  *finite*  $\{a. Q2\ a\}$  *finite*  $\{a. Q3\ a\}$   
**using** *that* **by** *simp*

**named\_theorems** *finite\_intros*

**named\_theorems** *more\_finite\_intros*

**lemmas** [*finite\_intros*] =

*finite\_UnI* *finite\_Union* *finite\_imageI*  
*finite\_lists\_length\_eq* *finite\_lists\_length\_le*  
*finite\_subset\_distinct* *distinct\_finite\_set*

**lemmas** [*more\_finite\_intros*] =

*finite\_prodI* *finite\_prodI3* *finite\_prodI4* *finite\_prodI5*

**Lists lemma** *fold\_evD2*:

**assumes**

$P\ y\ (fold\ f\ xs\ acc) \rightarrow P\ y\ acc$   
 $\bigwedge acc\ x. \neg P\ y\ acc \implies Q\ acc \implies P\ y\ (f\ x\ acc) \implies x \in set\ xs \implies x = y$   
 $Q\ acc \bigwedge acc\ x. Q\ acc \implies Q\ (f\ x\ acc)$   
 $\bigwedge acc\ x. \neg P\ y\ acc \implies Q\ acc \implies P\ y\ (f\ x\ acc) \implies R\ y$

**shows**  $\exists\ ys\ zs. xs = ys @ y \# zs \wedge \neg P\ y\ (fold\ f\ ys\ acc) \wedge P\ y\ (f\ y\ (fold\ f\ ys\ acc)) \wedge R\ y$

**proof** –

**from** *fold\_evD'[OF assms(2,1)]* **obtain**  $x\ ys\ zs$  **where** \*:

$xs = ys @ x \# zs \rightarrow P\ y\ (fold\ f\ ys\ acc) \wedge P\ y\ (f\ x\ (fold\ f\ ys\ acc))$

**by** *auto*

**moreover from** *assms(4–)* **have**  $Q\ (fold\ f\ ys\ acc)$  **by** (*auto intro: fold\_acc\_preserv*)

**moreover from**  $\langle xs = \_ \rangle$  **have**  $x \in set\ xs$

**by** *auto*

**ultimately show** *?thesis* **using** *assms(3,6)* **by** *auto*

**qed**

**lemmas** *fold\_evD2'* = *fold\_evD2* [**where**  $R = \lambda \_ . True$ , *simplified*]

**lemma** *filter\_map\_map\_filter*:

*filter*  $P\ (map\ f\ xs) = List.map\_filter\ (\lambda x. let\ y = f\ x\ in\ if\ P\ y\ then\ Some\ y\ else\ None)\ xs$   
**by** (*induction xs; simp add: Let\_def List.map\_filter\_simps*)

```

lemma distinct_map_filterI:
  distinct (List.map_filter f xs)
  if  $\forall x \in \text{set } xs. \forall y \in \text{set } xs. \forall a. f x = \text{Some } a \wedge f y = \text{Some } a \longrightarrow x = y$  distinct xs
  using that by (induction xs) (auto simp: map_filter_simps set_map_filter split: option.split)

lemma filter_eqI:
  assumes
    subseq ys xs  $\forall x \in \text{set } ys. P x$ 
     $\forall zs. \text{subseq } zs xs \wedge \text{length } zs > \text{length } ys \longrightarrow (\exists x \in \text{set } zs. \neg P x)$ 
  shows filter P xs = ys
  using assms
proof (induction xs arbitrary: ys rule: list.induct)
  case Nil
  then show ?case
    by - (cases ys; simp)
next
  case (Cons x xs)
  show ?case
  proof (cases P x)
    case True
    show ?thesis
    proof (cases ys)
      case Nil
      have subseq [x] (x # xs)
        by auto
      with Cons.prem1 Nil  $\langle P x \rangle$  show ?thesis
        by fastforce
    next
      case (Cons y ys')
      have x = y
      proof (rule ccontr)
        assume  $x \neq y$ 
        with  $\langle \text{subseq } ys (x \# xs) \rangle \langle ys = \_ \rangle$  have subseq (x # ys) (x # xs)
          by simp
        with Cons.prem2(2-)  $\langle P x \rangle$  show False
          by fastforce
      qed
      have  $\exists x \in \text{set } zs. \neg P x$  if subseq zs xs and  $\text{length } ys' < \text{length } zs$  for zs
      proof -
        from  $\langle \text{subseq } zs xs \rangle$  have subseq (x # zs) (x # xs)
          by simp
        with  $\langle \text{length } ys' < \text{length } zs \rangle$  Cons.prem3(3)  $\langle ys = \_ \rangle$  have  $\exists x \in \text{set } (x \# zs). \neg P x$ 
          by (intro Cons.prem3(3)[rule_format]; simp)
        with  $\langle P x \rangle$  show ?thesis
          by auto
      qed
      with Cons.prem1  $\langle P x \rangle \langle ys = \_ \rangle \langle x = y \rangle$  show ?thesis
        by (auto intro!: Cons.IH)
    qed
  next
    case False
    with Cons.prem1 show ?thesis
      by (cases ys) (auto split: if_split_asm intro!: Cons.IH)
  qed

```

qed

**lemma** *filter\_greatest\_subseqD*:

$\exists x \in \text{set } zs. \neg P x$  **if**  $\text{subseq } zs \text{ } xs \text{ } \text{length } zs > \text{length } (\text{filter } P \text{ } xs)$   
**using** *that* **by** (*metis filter\_id\_conv not\_subseq\_length subseq\_filter*)

**lemma** *filter\_eq\_iff\_greatest\_subseq*:

$\text{filter } P \text{ } xs = ys \longleftrightarrow$   
 $\text{subseq } ys \text{ } xs \wedge (\forall x \in \text{set } ys. P x) \wedge$   
 $(\forall zs. \text{subseq } zs \text{ } xs \wedge \text{length } zs > \text{length } ys \longrightarrow (\exists x \in \text{set } zs. \neg P x))$   
**using** *filter\_greatest\_subseqD filter\_eqI* **by** *auto*

**lemma** *subseq\_subsetD*:

$\text{set } xs \subseteq \text{set } ys$  **if**  $\text{subseq } xs \text{ } ys$   
**using** *that*  
**by** (*intro subsetI*) (*unfold subseq\_singleton\_left[symmetric], erule subseq\_order.trans*)

**lemma** *subseq\_distinct*:

*distinct*  $xs$  **if** *distinct*  $ys$   $\text{subseq } xs \text{ } ys$   
**using** *subseqs\_distinctD* **that** **by** *simp*

**lemma** *finite\_subseqs*:

*finite*  $\{xs. \text{subseq } xs \text{ } ys\}$  (**is** *finite*  $?S$ )

**proof**  $-$

**have**  $?S \subseteq \{xs. \text{set } xs \subseteq \text{set } ys \wedge \text{length } xs \leq \text{length } ys\}$   
**using** *not\_subseq\_length* **by** (*force dest: subseq\_subsetD*)  
**also have** *finite*  $\dots$   
**by** (*auto intro: finite\_lists\_length\_le*)  
**finally** (*finite\_subset*) **show**  $?thesis$  .

qed

**lemma** *filter\_distinct\_eqI*:

**assumes**  
 $\text{subseq } ys \text{ } xs \forall x \in \text{set } ys. P x \forall x \in \text{set } xs. x \notin \text{set } ys \longrightarrow \neg P x$  *distinct*  $xs$   
**shows**  $\text{filter } P \text{ } xs = ys$

**proof** (*intro filter\_eqI, safe*)

**fix**  $zs$  **assume** *prems*:  $\text{subseq } zs \text{ } xs \text{ } \text{length } ys < \text{length } zs$

**obtain**  $x$  **where**  $x \in \text{set } zs \wedge x \notin \text{set } ys$

**proof** (*atomize\_elim, rule ccontr*)

**assume**  $\nexists x. x \in \text{set } zs \wedge x \notin \text{set } ys$

**then have**  $\text{set } zs \subseteq \text{set } ys$

**by** *auto*

**moreover from** *prems* **assms** **have** *distinct*  $zs$  *distinct*  $ys$

**by** (*blast intro: subseq\_distinct*) $+$

**ultimately show** *False*

**using**  $\langle \text{length } ys < \text{length } zs \rangle$

**by** (*auto dest: card\_mono[rotated] simp: distinct\_card[symmetric]*)

qed

**with** *prems* *assms* **show**  $\exists x \in \text{set } zs. \neg P x$

**by** (*auto 4 3 dest: subseq\_subsetD*)

qed (*use* *assms* **in** *blast*) $+$

**lemma** *subseq\_sorted\_wrt*:

*sorted\_wrt*  $R \text{ } xs$  **if** *sorted\_wrt*  $R \text{ } ys$   $\text{subseq } xs \text{ } ys$

```

using that
by (induction xs arbitrary: ys)
  (auto 0 4 dest: subseq_subsetD list_emb_ConsD subseq_Cons' simp: sorted_wrt_append)

lemma subseq_sorted:
  sorted xs if sorted ys subseq xs ys
  using that by (rule subseq_sorted_wrt)

lemma sorted_distinct_subset_subseqI:
  assumes sorted xs distinct xs sorted ys set xs  $\subseteq$  set ys
  shows subseq xs ys
  using assms
proof (induction ys arbitrary: xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons y ys xs)
  from Cons.prem1 show ?case
    by (cases xs; simp) (safe; rule Cons.IH; auto 4 4)
qed

lemma sorted_distinct_subseq_iff:
  assumes sorted ys distinct ys
  shows subseq xs ys  $\longleftrightarrow$  (sorted xs  $\wedge$  distinct xs  $\wedge$  set xs  $\subseteq$  set ys)
  using assms
  by safe
    (erule
      subseq_subsetD[THEN subsetD] sorted_distinct_subset_subseqI subseq_distinct subseq_sorted;
      assumption
    )+

lemma subseq_mapE:
  assumes subseq xs (map f ys)
  obtains xs' where subseq xs' ys map f xs' = xs
  using assms
  by (induct x1  $\equiv$  xs x2  $\equiv$  map f ys arbitrary: xs ys rule: list_emb.induct)
    (auto, metis map_consI(1) subseq_Cons2)

lemma list_all2_map_fst_aux:
  assumes list_all2 ( $\lambda x y. x \in \text{Pair } y \text{ ' (zs y)}$ ) xs ys
  shows list_all2 (=) (map fst xs) ys
  using assms by (smt fstI imageE list.rel_mono_strong list_all2_map1)

lemma list_all2_fst_aux:
  map fst xs = ys if list_all2 ( $\lambda x y. \text{fst } x = y$ ) xs ys
  using that by (induction) auto

Stronger version of  $\llbracket \text{inj } ?f; \text{map\_of } ?t \text{ } ?k = \text{Some } ?x \rrbracket \implies \text{map\_of } (\text{map } (\lambda(k, y). (?f k, y))) ?t$ 
 $(?f ?k) = \text{Some } ?x$ 

theorem map_of_mapk_SomeI':
  assumes inj_on f (fst ' set t)
  and map_of t k = Some x
  shows map_of (map ( $\lambda(k, y). (f k, g y)$ ) t) (f k) = Some (g x)

```

```

using assms
apply (induction t)
  apply (solves simp)
  apply (clarsimp split: if_split_asm)
  apply (metis DiffI imageI imgfst map_of_SomeD singletonD)
done

theorem map_of_mapk_SomeI:
  assumes inj f
  and map_of t k = Some x
  shows map_of (map ( $\lambda(k, y). (f k, g y)$ ) t) (f k) = Some (g x)
  using assms by - (rule map_of_mapk_SomeI', erule inj_on_subset, auto)

lemma list_all2_map_eq_iff:
  list_all2 ( $\lambda x y. f x = g y$ ) xs ys  $\longleftrightarrow$  map f xs = map g ys
proof
  assume list_all2 ( $\lambda x y. f x = g y$ ) xs ys
  then show map f xs = map g ys
    by induction auto
next
  assume map f xs = map g ys
  then have length xs = length ys
    by (rule map_eq_imp_length_eq)
  then show list_all2 ( $\lambda x y. f x = g y$ ) xs ys
    using  $\langle \text{map } f \text{ xs} = \_ \rangle$  by (induction rule: list_induct2; simp)
qed

end
theory TA_More2
  imports Munta_Base.TA_More
begin

lemma collect_clock_pairs_concat:
  collect_clock_pairs (concat xxs) = ( $\bigcup x \in \text{set } xxs. \text{collect\_clock\_pairs } x$ )
  unfolding collect_clock_pairs_def by auto

end
theory TA_Equivalences
  imports
    Timed_Automata.Timed_Automata
    Munta_Base.Normalized_Zone_Semantics_Impl_Refine
begin

locale TA_Equiv =
  fixes A B :: ( $'a, 'c, 't :: \text{time}, 's$ ) ta
  fixes S ::  $'s \text{ set}$ 
  assumes state_set_trans_of: state_set (trans_of A)  $\subseteq$  S
  assumes invs:  $\forall l \in S. \text{inv\_of } A \ l = \text{inv\_of } B \ l$ 
  assumes trans_eq: trans_of A = trans_of B
begin

lemma delay1:
  A  $\vdash \langle l, u \rangle \rightarrow^d \langle l', u \rangle$  if  $l \in S \ B \vdash \langle l, u \rangle \rightarrow^d \langle l', u \rangle$ 
  using that invs by (auto elim!: step_t.cases)

```

```

lemma action1:
  A ⊢ ⟨l, u⟩ →a ⟨l', u'⟩ if B ⊢ ⟨l, u⟩ →a ⟨l', u'⟩
proof -
  from that have l ∈ S l' ∈ S
    using state_set_trans_of by (auto elim!: step_a.cases simp: trans_eq state_set_def)
  with that invs show ?thesis
    by (auto elim!: step_a.cases simp: trans_eq intro: step_a.intros)
qed

lemma delay2:
  B ⊢ ⟨l, u⟩ →d ⟨l', u'⟩ if l ∈ S A ⊢ ⟨l, u⟩ →d ⟨l', u'⟩
  using that invs by (auto elim!: step_t.cases)

lemma action2:
  B ⊢ ⟨l, u⟩ →a ⟨l', u'⟩ if A ⊢ ⟨l, u⟩ →a ⟨l', u'⟩
proof -
  from that have l ∈ S l' ∈ S
    using state_set_trans_of by (auto elim!: step_a.cases simp: trans_eq[symmetric] state_set_def)
  with that invs show ?thesis
    by (auto elim!: step_a.cases simp: trans_eq [symmetric] intro: step_a.intros)
qed

lemma step1:
  A ⊢ ⟨l, u⟩ → ⟨l', u'⟩ if l ∈ S B ⊢ ⟨l, u⟩ → ⟨l', u'⟩
  using that(2,1) by cases (auto dest: action1 delay1)

lemma step2:
  B ⊢ ⟨l, u⟩ → ⟨l', u'⟩ if l ∈ S A ⊢ ⟨l, u⟩ → ⟨l', u'⟩
  using that(2,1) by cases (auto dest: action2 delay2)

lemma S_inv:
  l' ∈ S if A ⊢ ⟨l, u⟩ → ⟨l', u'⟩ l ∈ S
  using that state_set_trans_of
  by (auto elim!: step.cases step_a.cases step_t.cases simp: state_set_def)

interpretation interp: Bisimulation_Invariant
  λ (l, u) (l', u'). A ⊢ ⟨l, u⟩ → ⟨l', u'⟩
  λ (l, u) (l', u'). B ⊢ ⟨l, u⟩ → ⟨l', u'⟩
  λ lu lu'. lu' = lu λ (l, u). l ∈ S λ (l, u). l ∈ S
  by standard (auto dest: step1 step2 intro: S_inv)

end

end

theory Simple_Network_Language_Impl
imports
  Simple_Network_Language
  Munta_Base.Normalized_Zone_Semantics_Impl_Refine
  HOL-Library.IArray HOL-Library.AList
  Munta_Base.More_Methods
  Munta_Base.Bijective_Embedding

```

```

    TA_Impl_Misc2
    TA_More2
    TA_Equivalences
    HOL-Eisbach.Eisbach_Tools
    Munta_Base.TA_Syntax_Bundles
begin

unbundle no_library_syntax

Default maps definition default_map_of :: 'b  $\Rightarrow$  ('a  $\times$  'b) list  $\Rightarrow$  'a  $\Rightarrow$  'b where
  default_map_of a xs  $\equiv$  FinFun.map_default a (map_of xs)

lemma default_map_of_alt_def:
  default_map_of a xs = (if  $x \in \text{dom } (\text{map\_of } xs)$  then the  $(\text{map\_of } xs \ x)$  else a)
unfolding default_map_of_def map_default_def by (auto split: option.split_asm)

lemma range_default_map_of:
  range (default_map_of x xs)  $\subseteq$  snd ` set xs  $\cup \{x\}$ 
unfolding default_map_of_def map_default_def
by (auto split: option.split_asm) (meson img_snd map_of_SomeD)

lemma finite_range_default_map_of:
  finite (range (default_map_of x m))
proof -
  have range (default_map_of x m)  $\subseteq$  the ` range (map_of m)  $\cup \{x\}$ 
    unfolding default_map_of_def FinFun.map_default_def
    by (auto split: option.splits) (metis image_eqI option.sel rangeI)
  also have finite ...
    by (blast intro: finite_range_map_of)
  finally show ?thesis .
qed

lemma map_index'_inj:
  L = L'
  if length L = length L' map_index' n f L = map_index' n g L' set L  $\subseteq$  S set L'  $\subseteq$  T
     $\forall i < \text{length } L + n. \forall x \in S. \forall y \in T. f \ i \ x = g \ i \ y \longrightarrow x = y$ 
  using that
proof (induction length L arbitrary: L L' n)
  case 0
  then show ?case
    by simp
next
  case (Suc x)
  from Suc.premis obtain a b L1 L1' where *:
    length L1 = x length L1' = x L = a # L1 L' = b # L1'
  by (smt Suc.hyps(2) length_Suc_conv)
show ?case
  unfolding  $\langle L = \_ \rangle \langle L' = \_ \rangle$ 
  apply (clarsimp, rule conjI)
  subgoal
  by (smt *(3,4) Suc.hyps(2) Suc.premis Suc_less_eq add_Suc_shift less_add_Suc2 list.set_intros(1)
list_tail_coinc map_index'.sims(2) set_mp)
  subgoal

```

```

    apply (rule Suc.hyps)
    using Suc.premis * by auto
  done
qed

```

```

lemma map_index_inj:
  L = L'
  if map_index f L = map_index g L' set L ⊆ S set L' ⊆ T
    ∀ i < length L. ∀ x ∈ S. ∀ y ∈ T. f i x = g i y ⟶ x = y
  using that by - (rule map_index'_inj, auto dest: map_index_eq_imp_length_eq)

```

```

lemma map_index_inj1:
  L = L'
  if map_index f L = map_index g L'
    ∀ i < length L. f i (L ! i) = g i (L' ! i) ⟶ L ! i = L' ! i
proof (intros add: nth_equalityI)
  from that(1) show ⟨length L = length L'⟩
    by (rule map_index_eq_imp_length_eq)
  fix i :: ⟨nat⟩
  assume ⟨i < length L⟩
  with that have map_index f L ! i = map_index g L' ! i
    by auto
  with ⟨i < length L⟩ ⟨length L = length L'⟩ have f i (L ! i) = g i (L' ! i)
    by simp
  with ⟨i < length L⟩ that(2) show ⟨L ! i = L' ! i⟩
    by simp
qed

```

```

lemma map_index_update:
  map_index f (xs[i := a]) = (map_index f xs)[i := f i a]
  by (rule nth_equalityI) (auto simp: nth_list_update')

```

```

lemma map_trans_broad_aux1:
  map_index map_loc (fold (λp L. L[p := ls' p]) ps L) =
  fold (λp L. L[p := map_loc p (ls' p)]) ps (map_index map_loc L)
  by (induction ps arbitrary: L) (auto simp: map_index_update)

```

```

lemma InD2:
  fixes map_action
  assumes inj map_action In (map_action a) = map_act map_action a'
  shows a' = In a
  using assms(2) by (cases a') (auto simp: injD[OF assms(1)])

```

```

lemma OutD2:
  fixes map_action
  assumes inj map_action Out (map_action a) = map_act map_action a'
  shows a' = Out a
  using assms(2) by (cases a') (auto simp: injD[OF assms(1)])

```

```

lemma (in Prod_TA_Defs) trans_broad_alt_def:
  trans_broad =
  {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L', s')) |
  L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
  a ∈ broadcast ∧

```

```

    (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
    (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
    (l ∈ committed (N p) ∨ (∃ p ∈ set ps. L ! p ∈ committed (N p)))
    ∨ (∀ p < n_ps. L ! p ∉ committed (N p))) ∧
    (∀ q < n_ps. q ∉ set ps ∧ p ≠ q →
      ¬ (∃ b g f r l'. (L!q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
    L!p = l ∧
    p < length L ∧ set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
    check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
    L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'] ∧
    is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
    L ∈ states ∧ bounded bounds s ∧ bounded bounds s'' ∧
    (∀ p. p ∉ set ps → bs p = bexp.true) ∧ (∀ p. p ∉ set ps → gs p = []) ∧
    (∀ p. p ∉ set ps → fs p = []) ∧ (∀ p. p ∉ set ps → rs p = [])
  }
unfolding trans_broad_def
proof ((intro Collect_eqI iffI; elims add: more_elims), goal_cases)
case prems: (1 x L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps)
let ?f = λgs p. if p ∈ set ps then gs p else []
let ?bs = λp. if p ∈ set ps then bs p else bexp.true
let ?gs = ?f gs let ?fs = ?f fs let ?rs = ?f rs
have [simp]: map gs ps = map ?gs ps map rs ps = map ?rs ps map fs ps = map ?fs ps
  by (simp cong: map_cong)+
with prems show ?case
  by (inst_existentials L s L' s' s'' a p l b g f r l' ?bs ?gs ?fs ?rs ls' ps; (assumption | simp))
next
case (2 x L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps)
then show ?case
  by blast
qed

```

### definition

```

conv_automaton ≡ λ(committed, urgent, trans, inv).
  (committed,
   urgent,
   map (λ(l, b, g, a, f, r, l'). (l, b, conv_cc g, a, f, r, l')) trans,
   map (λ(s, cc). (s, conv_cc cc)) inv)

```

### definition

```

automaton_of ≡
  λ(committed, urgent, trans, inv). (set committed, set urgent, set trans, default_map_of [] inv)

```

**locale** Simple\_Network\_Impl\_Defs =

```

fixes automata ::
  ('s list × 's list × ('a act, 's, 'c, 't, 'x, int) transition list
   × ('s × ('c, 't) cconstraint) list) list
and broadcast :: 'a list
and bounds' :: ('x × (int × int)) list

```

**begin**

**definition** — Number of state variables

```

n_vs = length bounds'

```

**definition**

$B\ x \equiv \text{if } x \in \text{dom } (\text{map\_of } \text{bounds}') \text{ then the } (\text{map\_of } \text{bounds}'\ x) \text{ else } (0, 0)$

**sublocale** *Prod\_TA\_Defs*

(*set broadcast*, *map automaton\_of automata*, *map\_of bounds'*) .

**lemma** *L\_len*[*intro*, *dest*]:

*length L = n\_ps* **if** *L ∈ states*

**using** *that* **unfolding** *states\_def* **by** *simp*

**lemma** *N\_eq*:

$\langle N\ i = \text{automaton\_of } (\text{automata } !\ i) \rangle$  **if**  $\langle i < n\_ps \rangle$

**using** *that* **unfolding** *N\_def n\_ps\_def fst\_conv snd\_conv* **by** (*intro nth\_map*; *simp*)

**end**

**locale** *Simple\_Network\_Impl* =

**fixes** *automata* ::

(*'s list* × *'s list* × (*'a act*, *'s*, *'c*, *int*, *'x*, *int*) *transition list*  
× (*'s* × (*'c*, *int*) *cconstraint*) *list*) *list*

**and** *broadcast* :: *'a list*

**and** *bounds'* :: (*'x* × (*int* × *int*)) *list*

**begin**

**sublocale** *Simple\_Network\_Impl\_Defs* *automata broadcast bounds'* .

**end**

**Mapping through the product construction** **lemma** *f\_the\_inv\_f*:

$f\ (\text{the\_inv } f\ x) = x$  **if**  $\text{inj } f\ x \in \text{range } f$

**using** *that* **by** (*auto simp: the\_inv\_f\_f*)

**method** *fprem* =

(*match premises in R: \_* ⇒  $\langle \text{rule } R[\text{elim\_format}] \rangle$ , *assumption*)

**context** *Simple\_Network\_Impl*

**begin**

**Conversion from integers to reals commutes with product construction.** **sublocale**

*conv: Prod\_TA\_Defs*

(*set broadcast*, *map (Simple\_Network\_Language.conv\_A o automaton\_of) automata*, *map\_of bounds'*) .

**lemma** (**in**  $-$ ) *conv\_ac\_inj*:

$ac = ac'$  **if**  $\text{conv\_ac } ac = \text{conv\_ac } ac'$

**using** *that* **by** (*cases ac*; *cases ac'*; *auto*)

**lemma** (**in**  $-$ ) *conv\_cc\_inj*:

$cc = cc'$  **if**  $\text{conv\_cc } cc = \text{conv\_cc } cc'$

**using** *that* **by** (*subst (asm) inj\_map\_eq\_map*) (*auto simp add: conv\_ac\_inj intro: injI*)

**context**

**begin**

**lemma** *conv\_alt\_def*:  
 $\text{conv } (\text{set broadcast}, \text{map automaton\_of automata}, \text{map\_of bounds}') =$   
 $(\text{set broadcast}, \text{map } (\text{Simple\_Network\_Language.conv\_A } o \text{ automaton\_of}) \text{ automata}, \text{map\_of bounds}')$   
**unfolding** *conv\_def* **by** *simp*

**private lemma** 2:  
 $\text{Simple\_Network\_Language.conv\_A } o \text{ automaton\_of} = (\lambda(\text{committed}, \text{urgent}, \text{trans}, \text{inv}).$   
 $(\text{set committed},$   
 $\text{set urgent},$   
 $\text{set } (\text{map Simple\_Network\_Language.conv\_t trans}),$   
 $\text{default\_map\_of } [] (\text{map } (\lambda (l, g). (l, \text{conv\_cc } g)) \text{ inv})))$   
**apply** (*rule ext*)  
**apply** *clarsimp*  
**unfolding** *Simple\_Network\_Language.conv\_A\_def automaton\_of\_def*  
**apply** (*clarsimp split: prod.split*)  
**unfolding** *default\_map\_of\_def*  
**unfolding** *FinFun.map\_default\_def*  
**apply** (*rule ext*)  
**subgoal for** *xs a*  
**by** (*induction xs*) *auto*  
**done**

**lemma** *conv\_n\_ps\_eq*:  
 $\text{conv.n\_ps} = \text{n\_ps}$   
**by** (*simp add: Prod\_TA\_Defs.n\_ps\_def*)

**lemma** *conv\_N\_eq*:  
 $\text{conv.N } i = \text{Simple\_Network\_Language.conv\_A } (N \ i) \text{ if } i < \text{n\_ps}$   
**using** *that* **unfolding** *n\_ps\_def Prod\_TA\_Defs.N\_def fst\_conv snd\_conv* **by** (*subst nth\_map*  
 $| \text{simp})+$

**private lemma** 5:  
 $\text{inv } (\text{conv.N } i) = \text{conv\_cc } o (\text{inv } (N \ i)) \text{ if } i < \text{n\_ps}$   
**unfolding** *conv\_N\_eq*[*OF that*] **unfolding** *Simple\_Network\_Language.conv\_A\_def*  
**by** (*simp split: prod.split add: inv\_def*)

**lemma** *trans\_conv\_N\_eq*:  
 $\text{trans } (\text{conv.N } i) = \text{Simple\_Network\_Language.conv\_t } ' (\text{trans } (N \ i)) \text{ if } i < \text{n\_ps}$   
**unfolding** *conv\_N\_eq*[*OF that*] **unfolding** *Simple\_Network\_Language.conv\_A\_def*  
**by** (*simp split: prod.split add: trans\_def*)

**private lemma** 71:  
 $(l, b, \text{conv\_cc } g, a, r, u, l') \in \text{Simple\_Network\_Language.trans } (\text{conv.N } i)$   
**if**  $(l, b, g, a, r, u, l') \in \text{Simple\_Network\_Language.trans } (N \ i) \ i < \text{n\_ps}$   
**using** *that* **by** (*force simp add: trans\_conv\_N\_eq Simple\_Network\_Language.conv\_t\_def*)

**private lemma** 72:  
 $(l, b, \text{conv\_cc } g, a, r, u, l') \in \text{Simple\_Network\_Language.trans } (\text{conv.N } i)$   
 $\iff (l, b, g, a, r, u, l') \in \text{Simple\_Network\_Language.trans } (N \ i) \text{ if } i < \text{n\_ps}$   
**by** (*auto simp: trans\_conv\_N\_eq*[*OF that*] *Simple\_Network\_Language.conv\_t\_def*  
 $\text{dest: conv\_cc\_inj intro: image_eqI[rotated]})$

**private lemma 73:**

$\exists g'. g = \text{conv\_cc } g' \wedge (l, b, g', a, r, u, l') \in \text{Simple\_Network\_Language.trans } (N \ i)$   
**if**  $(l, b, g, a, r, u, l') \in \text{Simple\_Network\_Language.trans } (\text{conv.N } i) \ i < n\_ps$   
**using that by**  $(\text{force simp: trans\_conv\_N\_eq Simple\_Network\_Language.conv\_t\_def})$

**lemma conv\_bounds[simp]:**

$\text{conv.bounds} = \text{bounds}$   
**unfolding**  $\text{conv.bounds\_def bounds\_def}$  **by**  $\text{simp}$

**lemma conv\_states[simp]:**

$\text{conv.states} = \text{states}$   
**unfolding**  $\text{conv.states\_def states\_def conv\_n\_ps\_eq}$   
**by**  $(\text{auto simp add: trans\_conv\_N\_eq Simple\_Network\_Language.conv\_t\_def})$   $(\text{fastforce, force})$

**private lemma 9:**

$\text{committed } (\text{conv.N } p) = \text{committed } (N \ p) \text{ if } \langle p < n\_ps \rangle$   
**unfolding**  $\text{conv\_N\_eq}$   $[OF \text{ that}]$  **unfolding**  $\text{Simple\_Network\_Language.conv\_A\_def}$   
**by**  $(\text{simp split: prod.split add: committed\_def})$

**private lemma 10:**

$\text{conv.broadcast} = \text{set broadcast}$   
**unfolding**  $\text{conv.broadcast\_def}$  **by**  $\text{simp}$

**lemma conv\_trans\_int:**

$\text{conv.trans\_int} = (\lambda(l, g, a, r, l'). (l, \text{map conv\_ac } g, a, r, l')) \text{ ' trans\_int}$   
**unfolding**  $\text{conv.trans\_int\_def trans\_int\_def}$   
**supply**  $[simp] = L\_len$   
**apply**  $\text{standard}$   
**subgoal**  
**by**  $(\text{clar simp simp: Simple\_Network\_Language.conv\_t\_def conv\_n\_ps\_eq trans\_conv\_N\_eq})$   
 $9)$   
 $(\text{intros add: more\_intros, solve\_triv+})$   
**subgoal**  
**by**  $(\text{rule subsetI, clar simp simp: Simple\_Network\_Language.conv\_t\_def conv\_n\_ps\_eq trans\_conv\_N\_eq})$   
 $9[\text{symmetric}])$   
 $((\text{drule } (1) \ 71)+, \text{intros add: more\_intros, solve\_triv+})$   
**done**

**lemma conv\_trans\_bin:**

$\text{conv.trans\_bin} = (\lambda(l, g, a, r, l'). (l, \text{map conv\_ac } g, a, r, l')) \text{ ' trans\_bin}$   
**unfolding**  $\text{conv.trans\_bin\_def trans\_bin\_def 10 broadcast\_def}$   
**supply**  $[simp] = L\_len$   
**apply**  $\text{standard}$   
**subgoal**  
**by**  $(\text{clar simp simp add: Simple\_Network\_Language.conv\_t\_def conv\_n\_ps\_eq trans\_conv\_N\_eq})$   
 $9)$   
 $(\text{intros add: more\_intros, solve\_triv+})$   
**subgoal**  
**by**  $(\text{rule subsetI, clar simp simp: Simple\_Network\_Language.conv\_t\_def conv\_n\_ps\_eq trans\_conv\_N\_eq})$   
 $9[\text{symmetric}])$   
 $((\text{drule } (1) \ 71)+, \text{intros add: more\_intros, solve\_triv+})$   
**done**

**lemma** *n\_ps\_rangeD*:

$p < n\_ps$  **if**  $p \in \text{set } ps$   $\text{set } ps \subseteq \{0..<n\_ps\}$   
**using** *that* **by** *auto*

**lemma** *maximalD*:

$(\forall g \ f \ r \ l'. \\$   
 $(L \ ! \ q, \ b, \ g, \ In \ a', \ f, \ r, \ l') \\$   
 $\notin (\lambda(l, \ b, \ g, \ a, \ f, \ r, \ l'). \\$   
 $(l, \ b, \ \text{map } \text{conv\_ac } g, \ a, \ f, \ r, \ l')) \ ' \ \text{trans } (N \ q)) \vee \neg \text{check\_bexp } s \ b \ \text{True} \ \text{if}$   
 $\forall q < n\_ps. \ q \notin \text{set } ps \wedge p \neq q \longrightarrow (\forall b. (\forall g \ f \ r \ l'. \\$   
 $(L \ ! \ q, \ b, \ g, \ In \ a', \ f, \ r, \ l') \notin \text{trans } (N \ q)) \vee \neg \text{check\_bexp } s \ b \ \text{True}) \\$   
 $q < n\_ps \ q \notin \text{set } ps \ p \neq q \\$   
**for**  $b \ ps \ p \ q \ L \ a' \ s$  **using** *that* **by** *fastforce*

**lemma** *conv\_trans\_broad*:

$\text{conv.trans\_broad} = (\lambda(l, \ g, \ a, \ r, \ l'). (l, \ \text{map } \text{conv\_ac } g, \ a, \ r, \ l')) \ ' \ \text{trans\_broad}$   
**unfolding**  $\text{conv.trans\_broad\_alt\_def}$   $\text{trans\_broad\_alt\_def}$

**apply** *standard*

**supply** [*simp*] =  $L\_len$

**subgoal**

**proof** –

**have** \*\*:  $\exists \ gs'. \ gs = \text{conv\_cc } o \ gs' \wedge$   
 $(\forall p \in \text{set } ps. (L \ ! \ p, \ bs \ p, \ gs' \ p, \ In \ a, \ fs \ p, \ rs \ p, \ ls' \ p) \in \text{trans } (N \ p))$   
**if** *assms*:  
 $\forall p \in \text{set } ps. (L \ ! \ p, \ bs \ p, \ gs \ p, \ In \ a, \ fs \ p, \ rs \ p, \ ls' \ p) \in \text{trans } (\text{conv}.N \ p)$   
 $\text{set } ps \subseteq \{0..<n\_ps\} \ \forall p. \ p \notin \text{set } ps \longrightarrow gs \ p = []$   
**for**  $L \ ps \ bs \ gs \ a \ fs \ rs \ ls'$

**proof** –

**have** \*:  $gs \ p = \text{conv\_cc } (\text{Hilbert\_Choice.inv } \text{conv\_cc } (gs \ p))$  **if**  $p \in \text{set } ps$  **for**  $p$   
**using** *that* *assms* **by** (*auto* 4 3 *simp*:  $f\_inv\_into\_f \ \text{dest}!$ : 73)

**show** *?thesis*

**apply** (*inst\_existentials*  $\text{Hilbert\_Choice.inv } \text{conv\_cc } o \ gs$ )

**subgoal**

**apply** (*rule ext*)

**subgoal for**  $p$

**apply** (*cases*  $p \in \text{set } ps$ )

**subgoal**

**by** (*simp*, *erule* \*)

**subgoal**

**using** *that*(3) **by** (*auto intro*:  $\text{injI } \text{inv\_f\_eq } \text{conv\_ac\_inj}$ )

**done**

**done**

**subgoal**

**using** *that* \* **by** (*force* *dest*!:  $\text{conv\_cc\_inj}$  73)

**done**

**qed**

**have** \*:  $\text{set } ps \subseteq \{0..<n\_ps\} \longleftrightarrow (\forall p \in \text{set } ps. \ p < n\_ps)$  **for**  $ps$   
**by** *auto*

**have** *maximalI*:

$\forall q < n\_ps. \ q \notin \text{set } ps \wedge p \neq q \longrightarrow (\forall b. (\forall g \ f \ r \ l'. \\$   
 $(L \ ! \ q, \ b, \ g, \ In \ a', \ f, \ r, \ l') \notin \text{trans } (N \ q)) \vee \neg \text{check\_bexp } s \ b \ \text{True}) \ \text{if}$   
 $\forall q < n\_ps. \ q \notin \text{set } ps \wedge p \neq q \longrightarrow (\forall b. (\forall g \ f \ r \ l'. \\$   
 $(L \ ! \ q, \ b, \ g, \ In \ a', \ f, \ r, \ l') \notin \text{trans } (\text{conv}.N \ q)) \vee \neg \text{check\_bexp } s \ b \ \text{True})$

```

for ps p L a' s
  using that by (blast dest!: 71)
show ?thesis
  apply (rule subsetI)
  apply (clarsimp simp add: conv_n_ps_eq 9 10 broadcast_def split: prod.split_asm)
  apply (drule (2) **)
  apply (drule (1) 73)+
  apply elims
  apply (intro image_eqI[rotated] CollectI exI conjI)
    apply solve_triv+
  subgoal premises prems — Committed
    using prems(2) ‹set _  $\subseteq$  {0.. $n_{ps}$ }› by (auto dest: n_ps_rangeD simp: 9)
    apply (erule maximalI; fail) — Maximal set
  by solve_triv+ (simp split: prod.split add: map_concat)
qed
subgoal
  apply (rule subsetI)
  apply (clarsimp simp:
    Simple_Network_Language.conv_t_def
    conv_n_ps_eq trans_conv_N_eq 9[symmetric] 10 broadcast_def map_concat)
  apply (drule (1) 71)
  apply (intros add: more_intros)
    apply solve_triv+
    apply (subst comp_def; rule 71; fast elim: n_ps_rangeD; fail)
  subgoal premises prems for L s s' s'' aj p g f r l' gs fs rs ls' ps
    using prems(3,6) 9 by fastforce
    apply (erule maximalD)
    apply (solve_triv | blast)+
  done
done

lemma conv_prod_ta:
  conv.prod_ta = Normalized_Zone_Semantics_Impl.conv_A prod_ta
unfolding conv.prod_ta_def prod_ta_def
unfolding conv.trans_prod_def trans_prod_def
unfolding conv.prod_inv_def prod_inv_def
unfolding conv.N_def N_def conv_n_ps_eq
unfolding conv_A_def
apply simp
apply (rule conjI)
subgoal
  unfolding image_Un
  by ((fo_rule arg_cong2)+; rule conv_trans_int conv_trans_bin conv_trans_broad)
subgoal — Invariant
  unfolding conv.N_def N_def
  by (rule ext) (auto simp: 5 map_concat intro!: map_cong arg_cong[where f = concat])
done

end

```

**Fundamentals** **definition** *clkp\_set'*  $\equiv$

$$(\bigcup A \in \text{set automata}. \bigcup g \in \text{set} (\text{snd} (\text{snd} (\text{snd} A))). \text{collect\_clock\_pairs} (\text{snd} g)) \\ \cup (\bigcup A \in \text{set automata}. \bigcup (l, b, g, \_) \in \text{set} (\text{fst} (\text{snd} (\text{snd} A))). \text{collect\_clock\_pairs} g)$$

**definition** *clk\_set'* **where**

⟨*clk\_set'* =  
*fst* ' *clkp\_set'* ∪  
(∪ *A* ∈ *set automata*. ∪ (∅, ∅, ∅, ∅, ∅, *r*, ∅) ∈ *set (fst (snd (snd A)))*). *set r*)⟩

**lemma** (**in** −) *default\_map\_of\_in\_listD*:

*x* ∈ ∪ (*set* ' *snd* ' *set invs*) **if** *x* ∈ *set (default\_map\_of [] invs l)*

**proof** −

**have** [] ≠ *default\_map\_of [] invs l*

**using** *that* **by** *force*

**then have** *default\_map\_of [] invs l* ∈ *snd* ' *set invs*

**by** (*metis* (*no\_types*) *UNIV\_I Un\_insert\_right range\_default\_map\_of[of [] invs]*  
*image\_eqI insertE subsetCE sup\_bot.right\_neutral*)

**with that show** *?thesis*

**by** *blast*

**qed**

**lemma** *collect\_clock\_pairs\_invsI*:

(*a*, *b*) ∈ ∪ ((*collect\_clock\_pairs o snd*) ' *set invs*)

**if** (*a*, *b*) ∈ *collect\_clock\_pairs (default\_map\_of [] invs l)*

**using that** **unfolding** *collect\_clock\_pairs\_def* **by** (*auto dest!*: *default\_map\_of\_in\_listD*)

**lemma** *mem\_trans\_N\_iff*:

*t* ∈ *Simple\_Network\_Language.trans (N i)* ⟷ *t* ∈ *set (fst (snd (snd (automata ! i))))*

**if** *i* < *n\_ps*

**unfolding** *N\_eq[OF that]* **by** (*auto split: prod.splits simp: automaton\_of\_def trans\_def*)

**lemma** *length\_automata\_eq\_n\_ps*:

*length automata* = *n\_ps*

**unfolding** *n\_ps\_def* **by** *simp*

**lemma** *clkp\_set'\_subs*:

*Timed\_Automata.clkp\_set prod\_ta* ⊆ *clkp\_set'*

**unfolding** *Timed\_Automata.clkp\_set\_def clkp\_set'\_def*

**proof** (*rule union\_subsetI, goal\_cases*)

**case** 1

**have** \*: *False*

**if** *i* < *n\_ps* *L* ∈ *states*

(*a*, *b*) ∈ *collect\_clock\_pairs (Simple\_Network\_Language.inv (N i) (L ! i))*

∀ *x* ∈ *set automata*. ∀ *x* ∈ *set (snd (snd (snd x)))*. (*a*, *b*) ∉ *collect\_clock\_pairs (snd x)*

**for** *a* :: 'c **and** *b* :: int **and** *L* :: 's list **and** *i* :: nat

**using that**

**apply** (*subst (asm) N\_eq*)

**apply** *assumption*

**apply** (*inst\_existentials automata ! i*)

**unfolding** *automaton\_of\_def*

**by** (*force dest!*: *nth\_mem collect\_clock\_pairs\_invsI*

*split: prod.split\_asm simp: inv\_def Prod\_TA\_Defs.n\_ps\_def*)

**from** 1 **show** *?case*

**unfolding** *Timed\_Automata.collect\_clki\_def inv\_of\_prod prod\_inv\_def*

**by** (*auto simp: collect\_clock\_pairs\_concat intro: \**)

**next**

**case** 2

```

then show ?case
  apply simp
  unfolding trans_prod_def Timed_Automata.collect_clk_def
  apply safe
  subgoal
    unfolding trans_int_def by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)
  subgoal
    unfolding trans_bin_def
    by (fastforce
        simp: length_automata_eq_n_ps mem_trans_N_iff
        dest!: collect_clock_pairs_append_cases)
  subgoal
    unfolding trans_broad_def
    apply (clarsimp simp: length_automata_eq_n_ps mem_trans_N_iff)
    apply (drule collect_clock_pairs_append_cases)
    unfolding collect_clock_pairs_concat
    apply (clarsimp; safe)
    by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)+ — slow: 20s
  done
qed

lemma collect_clkvt_subs:
  collect_clkvt (trans_of prod_ta)  $\subseteq$ 
  ( $\bigcup A \in \text{set automata. } \bigcup (\_, \_, \_, \_, \_, r, \_) \in \text{set (fst (snd (snd A)))}. \text{set } r$ )
  apply simp
  unfolding collect_clkvt_def
  apply safe
  unfolding trans_prod_def
  subgoal
    apply simp
    unfolding trans_prod_def Timed_Automata.collect_clk_def
    apply safe
    subgoal
      unfolding trans_int_def
      by (fastforce
          simp: length_automata_eq_n_ps mem_trans_N_iff
          dest!: collect_clock_pairs_append_cases)
    subgoal
      unfolding trans_bin_def
      by (fastforce
          simp: length_automata_eq_n_ps mem_trans_N_iff
          dest!: collect_clock_pairs_append_cases)
    subgoal
      unfolding trans_broad_def
      apply (clarsimp simp: length_automata_eq_n_ps mem_trans_N_iff)
      unfolding collect_clock_pairs_concat
      apply safe
      by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)+ — slow: 20s
    done
  done
done

lemma clk_set'_subs: clk_set prod_ta  $\subseteq$  clk_set'
  using collect_clkvt_subs clkp_set'_subs unfolding clk_set'_def by auto

```

end

```

lemma (in Prod_TA_Defs) finite_range_invI:
  finite (range prod_inv) if assms:  $\forall i < n\_ps. \text{finite } (\text{range } (\text{inv } (N \ i)))$ 
proof -
  let ?N =  $\bigcup (\text{range } ' \text{inv } ' N ' \{0..<n\_ps\})$ 
  let ?X =  $\{I. \text{set } I \subseteq ?N \wedge \text{length } I \leq n\_ps\}$ 
  have finite ?N
    using assms by auto
  then have finite ?X
    by (rule finite_lists_length_le)
  moreover have range prod_inv  $\subseteq \text{concat } ' ?X \cup \{\}\}$ 
  proof
    fix x assume x  $\in \text{range prod\_inv}$ 
    then consider L where x =  $\text{concat } (\text{map } (\lambda p. (\text{inv } (N \ p)) (L \ ! \ p)) [0..<n\_ps]) \mid x = []$ 
    unfolding prod_inv_def by (auto split: if_split_asm)
    then show x  $\in \text{concat } ' ?X \cup \{\}\}$ 
    by (cases; force)
  qed
  ultimately show ?thesis by - (drule finite_subset; auto)
qed

```

```

definition (in Prod_TA_Defs)
  loc_set =
  ( $\bigcup \{\text{fst } ' \text{trans } (N \ p) \mid p. p < n\_ps\}$   $\cup$ 
    $\bigcup \{(\text{snd } o \text{snd } o \text{snd } o \text{snd } o \text{snd } o \text{snd}) ' \text{trans } (N \ p) \mid p. p < n\_ps\}$ )

```

```

lemma (in Prod_TA_Defs) states_loc_set:
  states  $\subseteq \{L. \text{set } L \subseteq \text{loc\_set} \wedge \text{length } L = n\_ps\}$ 
  unfolding states_def loc_set_def
  apply (intros add: more_intros)
  apply (elims add: more_elims)
  apply (drule mem_nth)
  apply simp
  apply (elims add: allE, assumption)
  apply (simp split: prod.split_asm)
  apply (erule disjE; (intros add: disjI1 disjI2 more_intros, solve_triv+); fail)
  by (elims add: more_elims)

```

```

lemma (in Prod_TA_Defs) finite_states:
  assumes finite_trans:  $\forall p < n\_ps. \text{finite } (\text{Simple\_Network\_Language.trans } (N \ p))$ 
  shows finite states
proof -
  have states  $\subseteq \{L. \text{set } L \subseteq \text{loc\_set} \wedge \text{length } L = n\_ps\}$ 
    by (rule states_loc_set)
  also from finite_trans have finite ...
    unfolding loc_set_def by (intro finite_intros) auto
  finally show ?thesis .
qed

```

```

context Simple_Network_Impl
begin

```

```

lemma trans_N_finite:
  assumes  $p < n\_ps$ 
  shows finite (Simple_Network_Language.trans ( $N\ p$ ))
  using assms by (subst N_eq) (auto simp: automaton_of_def trans_def split: prod.split)

```

```

lemma states_finite:
  finite states
  by (intros add: finite_states trans_N_finite)

```

```

lemma bounded_finite:
  finite {s. bounded bounds s}
proof –
  let  $?l = \text{Min } \{fst\ (the\ (bounds\ x)) \mid x. x \in dom\ bounds\}$ 
  let  $?u = \text{Max } \{snd\ (the\ (bounds\ x)) \mid x. x \in dom\ bounds\}$ 
  have finite (dom bounds)
    unfolding bounds_def by simp
  then have  $\{s. bounded\ bounds\ s\} \subseteq \{s. dom\ s = dom\ bounds \wedge ran\ s \subseteq \{?l..?u\}\}$ 
    unfolding bounded_def
    apply clarsimp
    apply (rule conjI)
    subgoal for  $s\ v$ 
      unfolding ran_is_image
      apply clarsimp
      subgoal for  $x\ l\ u$ 
        by (rule order.trans[where  $b = fst\ (the\ (bounds\ x))$ ]; (rule Min_le)?; force)
      done
    subgoal for  $s\ v$ 
      unfolding ran_is_image
      apply clarsimp
      subgoal for  $x\ l\ u$ 
        by (rule order.trans[where  $b = snd\ (the\ (bounds\ x))$ ]; (rule Max_ge)?; force)
      done
    done
  also from  $\langle finite\ (dom\ bounds) \rangle$  have finite ...
    by (rule finite_set_of_finite_maps) blast
  finally show ?thesis .
qed

```

```

lemma trans_prod_finite:
  finite trans_prod
proof –
  have finite trans_int
  proof –
    have trans_int  $\subseteq$ 
       $\{((L, s), g, Internal\ a, r, (L', s')) \mid L\ s\ p\ l\ b\ g\ a\ f\ r\ l'\ s'\ L'. \\
        L \in states \wedge bounded\ bounds\ s \wedge p < n\_ps \wedge \\
        (l, b, g, Sil\ a, f, r, l') \in trans\ (N\ p) \wedge \\
        bounded\ bounds\ s' \\
        \wedge L' = L[p := l']\}$ 
    unfolding trans_int_def by (force simp: L_len)
  also have finite ...
  proof –
    have finite  $\{(a, b, c, d, e, f, g). (a, b, c, Sil\ d, e, f, g) \in trans\ (N\ p)\}$ 

```

```

    if  $p < n\_ps$  for  $p$ 
    using  $[[simproc\ add:\ finite\_Collect]]$  that
    by (auto intro:  $trans\_N\_finite\ finite\_vimageI\ injI$ )
with  $states\_finite\ bounded\_finite$  show  $?thesis$ 
    by defer_ex
qed
finally show  $?thesis$  .
qed
moreover have  $finite\ trans\_bin$ 
proof -
  have  $trans\_bin \subseteq$ 
    {(( $L, s$ ),  $g1 @ g2, Bin\ a, r1 @ r2, (L', s'')$ ) |
       $L\ s\ p\ q\ l1\ b1\ g1\ a\ f1\ r1\ l1'\ l2\ b2\ g2\ f2\ r2\ l2'\ s''\ L'$ .
       $L \in states \wedge bounded\ bounds\ s \wedge$ 
       $p < n\_ps \wedge q < n\_ps \wedge$ 
       $(l1, b1, g1, In\ a, f1, r1, l1') \in trans\ (N\ p) \wedge$ 
       $(l2, b2, g2, Out\ a, f2, r2, l2') \in trans\ (N\ q) \wedge$ 
       $bounded\ bounds\ s'' \wedge$ 
       $L' = L[p := l1', q := l2']$ 
    }
    unfolding  $trans\_bin\_def$  by (fastforce simp:  $L\_len$ )
  also have  $finite\ \dots$ 
  proof -
    have  $finite\ \{(a, b, c, d, e, f, g). (a, b, c, In\ d, e, f, g) \in trans\ (N\ p)\}$ 
      if  $p < n\_ps$  for  $p$ 
      using  $[[simproc\ add:\ finite\_Collect]]$  that
      by (auto intro:  $trans\_N\_finite\ finite\_vimageI\ injI$ )
    moreover have  $finite\ \{(a, b, c, e, f, g). (a, b, c, Out\ d, e, f, g) \in trans\ (N\ p)\}$ 
      if  $p < n\_ps$  for  $p\ d$ 
      using  $[[simproc\ add:\ finite\_Collect]]$  that
      by (auto intro:  $trans\_N\_finite\ finite\_vimageI\ injI$ )
    ultimately show  $?thesis$ 
      using  $states\_finite\ bounded\_finite$  by defer_ex
  qed
  finally show  $?thesis$  .
qed
moreover have  $finite\ trans\_broad$ 
proof -
  define  $P$  where  $P\ ps \equiv set\ ps \subseteq \{0..<n\_ps\} \wedge distinct\ ps$  for  $ps$ 
  define  $Q$  where  $Q\ a\ n\ bs\ gs\ fs\ rs \equiv$ 
    ( $\forall p < n. \exists q < n\_ps. \exists l\ l'. (l, bs\ !\ p, gs\ !\ p, In\ a, fs\ !\ p, rs\ !\ p, l') \in trans\ (N\ q)) \wedge$ 
     $length\ bs = n \wedge length\ gs = n \wedge length\ fs = n \wedge length\ rs = n$  for  $a\ n\ bs\ gs\ fs\ rs$ 
  define  $tag$  where  $tag\ x = True$  for  $x :: 's$ 
  have  $Q\_I: Q\ a\ (length\ ps)\ (map\ bs\ ps)\ (map\ gs\ ps)\ (map\ fs\ ps)\ (map\ rs\ ps)$ 
    if  $set\ ps \subseteq \{0..<n\_ps\}$ 
     $\forall p \in set\ ps. (L\ !\ p, bs\ p, gs\ p, In\ a, fs\ p, rs\ p, ls'\ p) \in trans\ (N\ p)$ 
    for  $ps :: nat\ list$  and  $L\ a\ bs\ gs\ fs\ rs\ ls'$ 
    using that unfolding  $Q\_def$  by (auto 4 4 dest!:  $nth\_mem$ )
  have  $trans\_broad \subseteq$ 
    {(( $L, s$ ),  $g @ concat\ gs, Broad\ a, r @ concat\ rs, (L', s'')$ ) |
       $L\ s\ a\ p\ l\ b\ g\ f\ r\ l'\ ps\ bs\ gs\ fs\ rs\ L'\ s''$ .
       $L \in states \wedge bounded\ bounds\ s \wedge a \in set\ broadcast \wedge$ 
       $p < n\_ps \wedge$ 
       $(l, b, g, Out\ a, f, r, l') \in trans\ (N\ p) \wedge$ 

```

```

    P ps ∧
    Q a (length ps) bs gs fs rs ∧
    L' ∈ states ∧
    bounded bounds s'' ∧
    tag l'
  }
  unfolding trans_broad_def broadcast_def
  apply (rule subsetI)
  apply (elims add: more_elims)
  apply (intros add: more_intros)
    apply solve_triv+
    apply (simp add: L_len; fail)
    apply assumption
    apply (unfold P_def; intros; assumption)
    apply (rule Q_I; assumption)
  subgoal
    by (blast intro: state_preservation_updI state_preservation_fold_updI)
  apply assumption
  unfolding tag_def ..
  also have finite ...
  proof -
    have finite {(a, b, c, e, f, g). (a, b, c, Out d, e, f, g) ∈ trans (N p)}
    if p < n_ps for p d
    using [[simproc add: finite_Collect]] that
    by (auto intro: trans_N_finite finite_vimageI injI)
  moreover have finite {ps. P ps}
    unfolding P_def by (simp add: finite_intros)
  moreover have finite {(bs, gs, fs, rs). Q a n bs gs fs rs} (is finite ?S) for a n
  proof -
    let ?T = ⋃ (trans ' N ' {0..<n_ps})
    have ?S ⊆ {(bs, gs, fs, rs).
      (set bs ⊆ (λ(.,b,.). b) ' ?T ∧ length bs = n) ∧
      (set gs ⊆ (λ(.,.,g,.). g) ' ?T ∧ length gs = n) ∧
      (set fs ⊆ (λ(.,.,.,f,.). f) ' ?T ∧ length fs = n) ∧
      (set rs ⊆ (λ(.,.,.,.,r,.). r) ' ?T ∧ length rs = n)
    }
    unfolding Q_def
    apply safe
    apply (all ⟨drule mem_nth; elims; drule spec; elims⟩)
    apply force+
    done
  also have finite ...
    using trans_N_finite by (intro finite_intros more_finite_intros) auto
  finally show ?thesis .
qed
ultimately show ?thesis
  using states_finite bounded_finite by defer_ex
qed
finally show ?thesis .
qed
ultimately show ?thesis
  by (simp add: trans_prod_def)
qed

```

```

lemma prod_inv_finite:
  finite (range prod_inv)
apply (intros add: finite_range_invI)
unfolding length_automata_eq_n_ps[symmetric]
unfolding N_def
unfolding automaton_of_def
by (auto intro: finite_range_default_map_of_split: prod.split simp: inv_def)

```

**end**

**Collecting variables from expressions. fun vars\_of\_bexp and vars\_of\_exp where**

```

vars_of_bexp (not e) = vars_of_bexp e
| vars_of_bexp (and e1 e2) = (vars_of_bexp e1 ∪ vars_of_bexp e2)
| vars_of_bexp (bexp.or e1 e2) = (vars_of_bexp e1 ∪ vars_of_bexp e2)
| vars_of_bexp (imply e1 e2) = (vars_of_bexp e1 ∪ vars_of_bexp e2)
| vars_of_bexp (eq i x) = vars_of_exp i ∪ vars_of_exp x
| vars_of_bexp (le i x) = vars_of_exp i ∪ vars_of_exp x
| vars_of_bexp (lt i x) = vars_of_exp i ∪ vars_of_exp x
| vars_of_bexp (ge i x) = vars_of_exp i ∪ vars_of_exp x
| vars_of_bexp (gt i x) = vars_of_exp i ∪ vars_of_exp x
| vars_of_bexp bexp.true = {}
| vars_of_exp (const c) = {}
| vars_of_exp (var x) = {x}
| vars_of_exp (if_then_else b e1 e2) = vars_of_bexp b ∪ vars_of_exp e1 ∪ vars_of_exp e2
| vars_of_exp (binop _ e1 e2) = vars_of_exp e1 ∪ vars_of_exp e2
| vars_of_exp (unop _ e) = vars_of_exp e

```

**definition** (in Prod\_TA\_Defs)

```

var_set =
  (⋃ S ∈ {(fst ∘ snd) ‘ trans (N p) | p. p < n_ps}. ⋃ b ∈ S. vars_of_bexp b) ∪
  (⋃ S ∈ {(fst ∘ snd ∘ snd ∘ snd ∘ snd) ‘ trans (N p) | p. p < n_ps}.
    ⋃ f ∈ S. ⋃ (x, e) ∈ set f. {x} ∪ vars_of_exp e)

```

**locale** Simple\_Network\_Impl\_nat\_defs =

Simple\_Network\_Impl automata

**for** automata ::

(nat list × nat list × (nat act, nat, nat, int, nat, int) transition list  
 × (nat × (nat, int) cconstraint) list) list +

**fixes** m :: nat **and** num\_states :: nat ⇒ nat **and** num\_actions :: nat

**locale** Simple\_Network\_Impl\_nat =

Simple\_Network\_Impl\_nat\_defs +

**assumes** has\_clock: m > 0

**assumes** non\_empty: 0 < length automata

**assumes** trans\_num\_states:

∀ i < n\_ps. let (\_, \_, trans, \_) = (automata ! i) in ∀ (l, \_, \_, \_, \_, l') ∈ set trans.  
 l < num\_states i ∧ l' < num\_states i

**and** inv\_num\_states:

∀ i < n\_ps. let (\_, \_, \_, inv) = (automata ! i) in ∀ (x, \_) ∈ set inv. x < num\_states i

**assumes** var\_set:

∀ (\_, \_, trans, \_) ∈ set automata. ∀ (\_, \_, \_, \_, f, \_, \_) ∈ set trans.

∀ (x, upd) ∈ set f. x < n\_vs ∧ (∀ i ∈ vars\_of\_exp upd. i < n\_vs)

```

 $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, b, \_, \_, \_, \_) \in set\ trans.$ 
 $\forall i \in vars\_of\_bexp\ b. i < n\_vs$ 
assumes bounds:
 $\forall i < n\_vs. fst\ (bounds'!\ i) = i$ 
assumes action_set:
 $\forall a \in set\ broadcast. a < num\_actions$ 
 $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, \_, a, \_, \_) \in set\ trans.$ 
 $pred\_act\ (\lambda a. a < num\_actions)\ a$ 
assumes clock_set:
 $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, g, \_, \_, r, \_) \in set\ trans.$ 
 $(\forall c \in set\ r. 0 < c \wedge c \leq m) \wedge$ 
 $(\forall (c, x) \in collect\_clock\_pairs\ g. 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$ 

 $\forall (\_, \_, \_, inv) \in set\ automata. \forall (l, g) \in set\ inv.$ 
 $(\forall (c, x) \in collect\_clock\_pairs\ g. 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$ 

assumes broadcast_receivers:
 $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, g, a, \_, \_, \_) \in set\ trans.$ 
 $case\ a\ of\ In\ a \Rightarrow a \in set\ broadcast \longrightarrow g = [] \mid \_ \Rightarrow True$ 
begin

lemma broadcast_receivers_unguarded:
 $\forall p < n\_ps. \forall l\ b\ g\ a\ f\ r\ l'.$ 
 $(l, b, g, In\ a, f, r, l') \in Simple\_Network\_Language.trans\ (N\ p) \wedge a \in set\ broadcast \longrightarrow g = []$ 
using broadcast_receivers by (fastforce dest: nth_mem simp: n_ps_def mem_trans_N_iff)

sublocale conv: Prod_TA
 $(set\ broadcast, map\ (Simple\_Network\_Language.conv\_A\ o\ automaton\_of)\ automata, map\_of\ bounds')$ 
using broadcast_receivers_unguarded
by  $— (standard,$ 
 $fastforce\ simp: conv.broadcast\_def\ Simple\_Network\_Language.conv\_t\_def\ conv\_n\_ps\_eq\ trans\_conv\_N\_eq)$ 

sublocale TA_Start_No_Ceiling prod_ta init m
proof standard
show finite  $(trans\_of\ prod\_ta)$ 
using trans_prod_finite by simp
next
show finite  $(range\ (inv\_of\ prod\_ta))$ 
using prod_inv_finite by simp
next
from clk_set'_subs have  $clk\_set\ prod\_ta \subseteq clk\_set'.$ 
also have  $\dots \subseteq \{1..m\}$ 
using clock_set unfolding clk_set'_def clkp_set'_def by force
finally show  $clk\_set\ prod\_ta \subseteq \{1..m\}.$ 
next
from clock_set have  $\forall (\_, d) \in clkp\_set'. d \in \mathbb{N}$ 
unfolding clkp_set'_def by force
then show  $\forall (\_, d) \in Timed\_Automata.clkp\_set\ prod\_ta. d \in \mathbb{N}$ 
by  $(auto\ dest!:\ subsetD[OF\ clk\_set'\_subs])$ 
next
show  $0 < m$ 
by  $(rule\ has\_clock)$ 

```

qed

end

**context** *Simple\_Network\_Impl*  
**begin**

**definition** *sem*  $\equiv$  (set broadcast, map (automaton\_of o conv\_automaton) automata, map\_of bounds')

**sublocale** *sem*: Prod\_TA\_sem *sem* .

**lemma** *sem\_N\_eq*:  
*sem.N p* = automaton\_of (conv\_automaton (automata ! p)) **if**  $\langle p < n\_ps \rangle$   
**using** that **unfolding** *sem.N\_def* *n\_ps\_def* **unfolding** *sem\_def* *fst\_conv* *snd\_conv*  
**by** (subst nth\_map) auto

end

**inductive\_cases** *step\_u\_elim*:  
 $A \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$   
 $A \vdash \langle L, s, u \rangle \rightarrow_{Internal\ a} \langle L', s', u' \rangle$   
 $A \vdash \langle L, s, u \rangle \rightarrow_{Bin\ a} \langle L', s'', u' \rangle$   
 $A \vdash \langle L, s, u \rangle \rightarrow_{Broad\ a} \langle L', s'', u' \rangle$

**inductive\_cases** *step\_u\_elim'*:  
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$   
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Internal\ a} \langle L', s', u' \rangle$   
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Bin\ a} \langle L', s'', u' \rangle$   
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Broad\ a} \langle L', s'', u' \rangle$

**lemma** (in *Prod\_TA\_Defs*) *states\_lengthD*:  
*length L* = *n\_ps* **if**  $L \in \text{states}$   
**using** that **unfolding** *states\_def* **by** *simp*

end

**theory** *Simple\_Network\_Language\_Impl\_Refine*  
**imports** *Simple\_Network\_Language\_Impl*  
**begin**

**unbundle** *no\_library\_syntax*  
**notation** *fun\_rel\_sym* (**infixr**  $\rightarrow 60$ )  
**hide\_const** (open) *upd*

**Expression evaluation** **fun** *bval* ::  $('a \Rightarrow 'b) \Rightarrow ('a, 'b :: \text{linorder}) \text{ bexp} \Rightarrow \text{bool}$  **and** *eval*  
**where**  
*bval* \_ *bexp.true*  $\longleftrightarrow \text{True}$  |  
*bval* *s* (*not* *e*)  $\longleftrightarrow \neg \text{bval } s\ e$  |  
*bval* *s* (*and* *e1* *e2*)  $\longleftrightarrow \text{bval } s\ e1 \wedge \text{bval } s\ e2$  |  
*bval* *s* (*bexp.or* *e1* *e2*)  $\longleftrightarrow \text{bval } s\ e1 \vee \text{bval } s\ e2$  |  
*bval* *s* (*imply* *e1* *e2*)  $\longleftrightarrow \text{bval } s\ e1 \longrightarrow \text{bval } s\ e2$  |  
*bval* *s* (*eq* *i* *x*)  $\longleftrightarrow \text{eval } s\ i = \text{eval } s\ x$  |

```

bval s (le i x)  $\longleftrightarrow$  eval s i  $\leq$  eval s x |
bval s (lt i x)  $\longleftrightarrow$  eval s i  $<$  eval s x |
bval s (ge i x)  $\longleftrightarrow$  eval s i  $\geq$  eval s x |
bval s (gt i x)  $\longleftrightarrow$  eval s i  $>$  eval s x
| eval s (const c) = c
| eval s (var x) = s x
| eval s (if_then_else b e1 e2) = (if bval s b then eval s e1 else eval s e2)
| eval s (binop f e1 e2) = f (eval s e1) (eval s e2)
| eval s (unop f e) = f (eval s e)

```

**lemma** check\_bexp\_determ:

```

check_bexp s b b1  $\implies$  check_bexp s b b2  $\implies$  b1 = b2
and is_val_determ: is_val s e v1  $\implies$  is_val s e v2  $\implies$  v1 = v2
by (induction b and e arbitrary: b1 b2 and v1 v2)
(auto dest: elim!: is_val_elims check_bexp_elims)+

```

**lemma** is\_upd\_determ:

```

s1 = s2 if is_upd s f s1 is_upd s f s2
using that unfolding is_upd_def
by (clarsimp, fo_rule arg_cong)
(smt
case_prodE case_prodE' case_prod_conv is_val_determ list.rel_eq list_all2_swap list_all2_trans
)

```

**lemma** is\_upds\_determ:

```

s1 = s2 if is_upds s fs s1 is_upds s fs s2
using that
by (induction fs arbitrary: s) (auto 4 4 elim: is_upds.cases dest: is_upd_determ)

```

**lemma** check\_bexp\_bval:

```

 $\forall x \in \text{vars\_of\_bexp } b. x \in \text{dom } s \implies \text{check\_bexp } s b (\text{bval } (\text{the } o \text{ } s) b)$ 

```

**and** is\_val\_eval:

```

 $\forall x \in \text{vars\_of\_exp } e. x \in \text{dom } s \implies \text{is\_val } s e (\text{eval } (\text{the } o \text{ } s) e)$ 

```

**apply** (induction b **and** e)

**apply** (simp; (subst eq\_commute)?; rule check\_bexp\_is\_val.intros; simp add:

dom\_def)+

**apply** ((subst eq\_commute eval.simps)?; rule check\_bexp\_is\_val.intros; simp add: dom\_def)+

**done**

**lemma** is\_upd\_dom:

```

dom s' = dom s if is_upd s (x, e) s' x  $\in$  dom s

```

**using** that unfolding is\_upd\_def **by** (auto split: if\_split\_asm)

**lemma** is\_upds\_dom:

```

dom s' = dom s if is_upds s upds s'  $\forall (x, e) \in \text{set upds}. x \in \text{dom } s$ 

```

**using** that

**by** (induction upds arbitrary: s)

(erule is\_upds.cases; auto simp: is\_upd\_dom split: prod.split\_asm)+

**definition**

```

mk_upd  $\equiv \lambda(x, e) s. s(x \mapsto \text{eval } (\text{the } o \text{ } s) e)$ 

```

**definition** mk\_upds ::

```

('a  $\Rightarrow$  ('b :: linorder) option)  $\Rightarrow$  ('a  $\times$  ('a, 'b) exp) list  $\Rightarrow$  ('a  $\Rightarrow$  'b option) where

```

$mk\_upds\ s\ upds = fold\ mk\_upd\ upds\ s$

**lemma** *is\_upd\_make\_updI*:

*is\_upd s upd (mk\_upd upd s) if*  $upd = (x, e) \forall x \in vars\_of\_exp\ e. x \in dom\ s$   
**using** *that(1) is\_val\_eval[OF that(2)] unfolding is\_upd\_def mk\_upd\_def by auto*

**lemma** *dom\_mk\_upd*:

$dom\ s \subseteq dom\ (mk\_upd\ upd\ s)$   
**unfolding** *mk\_upd\_def by (auto split: prod.split)*

**lemma** *is\_upds\_make\_updsI*:

*is\_upds s upds (mk\_upds s upds) if*  $\forall (\_, e) \in set\ upds. \forall x \in vars\_of\_exp\ e. x \in dom\ s$   
**using** *that*

**proof** (*induction upds arbitrary: s*)

**case** *Nil*

**then show** *?case*

**by** (*auto intro!: is\_upds.intros simp: mk\_upds\_def*)

**next**

**case** (*Cons upd upds*)

**let** *?s = mk\_upd upd s*

**from** *Cons(2) have is\_upd s upd ?s*

**by** (*auto simp: dom\_def intro: is\_upd\_make\_updI*)

**moreover have** *is\_upds ?s upds (mk\_upds ?s upds)*

**using** *Cons.prem1 dom\_mk\_upd by (intro Cons.IH) force*

**ultimately show** *?case*

**using** *dom\_mk\_upd by (auto intro!: is\_upds.intros simp: mk\_upds\_def)*

**qed**

**Implementation auxiliaries definition**

*union\_map\_of xs =*

*fold ( $\lambda (x, y) m. case\ m\ x\ of\ None \Rightarrow m(x \mapsto [y]) \mid Some\ ys \Rightarrow m(x \mapsto y \# ys)$ ) xs Map.empty*

**lemma** *union\_map\_of\_alt\_def*:

*union\_map\_of xs x = (let*

*ys = rev (map snd (filter ( $\lambda (x', y). x' = x$ ) xs))*

*in if ys = [] then None else Some ys)*

**proof** –

**have** *fold ( $\lambda (x, y) m. case\ m\ x\ of\ None \Rightarrow m(x \mapsto [y]) \mid Some\ ys \Rightarrow m(x \mapsto y \# ys)$ ) xs m x*  
 $= (let$

*ys = rev (map snd (filter ( $\lambda (x', y). x' = x$ ) xs))*

*in*

*case m x of None  $\Rightarrow$  if ys = [] then None else Some ys  $\mid$  Some zs  $\Rightarrow$  Some (ys @ zs)) for x m*

**by** (*induction xs arbitrary: m; clarsimp split: option.split*)

**then show** *?thesis*

**unfolding** *union\_map\_of\_def by simp*

**qed**

**lemma** *in\_union\_map\_ofI*:

$\exists ys. union\_map\_of\ xs\ x = Some\ ys \wedge y \in set\ ys$  **if**  $(x, y) \in set\ xs$

**unfolding** *union\_map\_of\_alt\_def Let\_def using that set\_filter[of  $\lambda(x', y). x' = x\ xs$ ] by auto+*

**lemma** *in\_union\_map\_ofD*:

$(x, y) \in \text{set } xs$  **if**  $\text{union\_map\_of } xs \ x = \text{Some } ys \ y \in \text{set } ys$   
**using** *that* **unfolding**  $\text{union\_map\_of\_alt\_def}$  **by**  $(\text{auto split: if\_split\_asm})$

**Implementation of state set** **context** *Simple\_Network\_Impl\_nat\_defs*  
**begin**

**definition**

$\text{states\_i } i = (\bigcup (l, e, g, a, r, u, l') \in \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! i))))). \{l, l'\}$

**lemma** *states\_mem\_compute*[code]:

$L \in \text{states} \longleftrightarrow \text{length } L = n\_ps \wedge (\forall i < n\_ps. L ! i \in \text{states\_i } i)$   
**unfolding** *states\_def* *states\_i\_def* **by** *simp*  $(\text{metis mem\_trans\_N\_iff})$

**lemma** *states\_mem\_compute'*:

$L \in \text{states} \longleftrightarrow \text{length } L = n\_ps \wedge (\forall i < n\_ps. L ! i \in \text{map } \text{states\_i } i \ [0..<n\_ps] ! i)$   
**unfolding** *states\_mem\_compute* **by** *simp*

**end**

**context** *Simple\_Network\_Impl\_nat*  
**begin**

**Fundamentals** **lemma** *mem\_trans\_N\_iff*:

$\langle t \in \text{Simple\_Network\_Language.trans } (N \ i) \longleftrightarrow t \in \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! i)))) \rangle$   
**if**  $i < n\_ps$   
**unfolding** *N\_def* *fst\_conv* *snd\_conv*  
**unfolding** *automaton\_of\_def*  
**unfolding** *trans\_def*  
**using** *that* **by**  $(\text{cases } \text{automata } ! i) (\text{auto simp: length\_automata\_eq\_n\_ps})$

**lemma** *L\_i\_len*:

$\langle L ! i < \text{num\_states } i \rangle$  **if**  $i < n\_ps \ L \in \text{states}$   
**using** *trans\_num\_states* *that*  
**unfolding** *states\_def* **by**  $(\text{auto } 4 \ 4 \text{ simp: mem\_trans\_N\_iff})$

**lemma** *L\_i\_simp*:

$\langle [0..<\text{num\_states } i] ! (L ! i) = L ! i \rangle$   
**if**  $i < n\_ps \ L \in \text{states}$   
**using** *L\_i\_len* [*OF that*] **by** *simp*

**lemma** *action\_setD*:

$\langle \text{pred\_act } (\lambda a'. a' < \text{num\_actions}) \ a \rangle$   
**if**  $\langle (l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p) \rangle \ \langle p < n\_ps \rangle$   
**using** *that* *action\_set*  
**by**  $(\text{cases } \text{automata } ! p)$   
 $(\text{subst } (\text{asm}) \text{ mem\_trans\_N\_iff; force dest!: nth\_mem simp flip: length\_automata\_eq\_n\_ps})$

**More precise state sets** **definition**

$\text{states}' \equiv \{(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n\_vs\} \wedge \text{bounded bounds } s\}$

**lemma** *states'\_states*[intro, dest]:

$L \in \text{states}$  **if**  $(L, s) \in \text{states}'$

**using** *that unfolding* *states'\_def* **by** *auto*

**lemma** *states'\_dom*[*intro*, *dest*]:  
 $\text{dom } s = \{0..<n\_vs\}$  **if**  $(L, s) \in \text{states}'$   
**using** *that unfolding* *states'\_def* **by** *auto*

**lemma** *states'\_bounded*[*intro*, *dest*]:  
 $\text{bounded bounds } s$  **if**  $(L, s) \in \text{states}'$   
**using** *that unfolding* *states'\_def* **by** *auto*

**Implementation of invariants** **definition** (in *Simple\_Network\_Impl\_nat\_defs*)

$\text{invs } i \equiv \text{let } m = \text{default\_map\_of } [] (\text{snd } (\text{snd } (\text{snd } (\text{automata } ! i))));$   
 $m' = \text{map } (\lambda j. m j) [0..<\text{num\_states } i]$   
in  $m'$

**definition** (in *Simple\_Network\_Impl\_nat\_defs*)  
 $\text{invs1} \equiv \text{map } (\lambda i. \text{let } m = \text{default\_map\_of } [] (\text{snd } (\text{snd } (\text{snd } (\text{automata } ! i))));$   
 $m' = \text{map } (\lambda j. m j) [0..<\text{num\_states } i]$   
in  $m'$ )  $[0..<n\_ps]$

**definition** (in *Simple\_Network\_Impl\_nat\_defs*)  
 $\text{invs2} \equiv \text{IArray } (\text{map } (\lambda i. \text{let } m = \text{default\_map\_of } [] (\text{snd } (\text{snd } (\text{snd } (\text{automata } ! i))));$   
 $m' = \text{IArray } (\text{map } (\lambda j. m j) [0..<\text{num\_states } i])$   
in  $m'$ )  $[0..<n\_ps]$

**lemma** *refine\_invs2*:  
 $\text{invs2 } !! i !! j = \text{invs1 } ! i ! j$  **if**  $i < n\_ps$   
**using** *that unfolding* *invs2\_def* *invs1\_def* **by** *simp*

**definition** (in *Simple\_Network\_Impl\_nat\_defs*)  
 $\text{inv\_fun} \equiv \lambda(L, \_).$   
 $\text{concat } (\text{map } (\lambda i. \text{invs1 } ! i ! (L ! i)) [0..<n\_ps])$

**lemma** *refine\_invs1*:  
 $\langle \text{invs1 } ! i = \text{invs } i \rangle$  **if**  $\langle i < n\_ps \rangle$   
**using** *that unfolding* *invs\_def* *invs1\_def* **by** *simp*

**lemma** *invs\_simp*:  
 $\text{invs1 } ! i ! (L ! i) = \text{Simple\_Network\_Language.inv } (N i) (L ! i)$   
**if**  $i < n\_ps$   $L \in \text{states}$   
**using** *that unfolding* *refine\_invs1*[*OF*  $\langle i < \_ \rangle$ ] *invs\_def* *N\_def* *fst\_conv* *snd\_conv*  
**unfolding** *inv\_def*  
**by** (*subst nth\_map*;  
*clarsimp split: prod.split simp: automaton\_of\_def length\_automata\_eq\_n\_ps L\_i\_len*)

**lemma** *inv\_fun\_inv\_of'*:  
 $(\text{inv\_fun}, \text{inv\_of prod\_ta}) \in \text{inv\_rel } R \text{ states'}$  **if**  $R \subseteq \text{Id} \times_r S$   
**using** *that*  
**unfolding** *inv\_rel\_def*  
**unfolding** *inv\_fun\_def*  
**unfolding** *inv\_of\_prod prod\_inv\_def*  
**apply** (*clarsimp simp: states'\_def*)  
**apply** (*rule arg\_cong[where f = concat]*)

```

apply (auto simp add: invs_simp cong: map_cong)
done

```

```

lemma inv_fun_alt_def:
  inv_fun = (λ(L, _). concat (map (λi. invs2 !! i !! (L ! i)) [0..unfolding inv_fun_def
apply (intro ext)
apply (clarsimp simp del: IArray.sub_def)
apply (fo_rule arg_cong)
apply (simp add: refine_invs2 del: IArray.sub_def)
done

```

**end**

**Implementation of transitions**   **context** *Simple\_Network\_Impl\_nat\_defs*  
**begin**

**definition**

*bounds\_map*  $\equiv$  the o map\_of bounds'

**definition**

*check\_bounded* *s* =  
 $(\forall x \in \text{dom } s. \text{fst } (\text{bounds\_map } x) \leq \text{the } (s \ x) \wedge \text{the } (s \ x) \leq \text{snd } (\text{bounds\_map } x))$

Compute pairs of processes with committed initial locations from location vector.

**definition**

*get\_committed* *L* =  
*List.map\_filter* (λ*p*.  
 let *l* = *L* ! *p* in  
 if *l* ∈ *set* (fst (automata ! *p*)) then *Some* (*p*, *l*) else *None*) [0..*n\_ps*]

Given a process and a location, return the corresponding transitions.

**definition**

*trans\_map* *i*  $\equiv$   
 let *m* = *union\_map\_of* (fst (snd (snd (automata ! *i*)))) in (λ*j*.  
 case *m* *j* of *None*  $\Rightarrow$  [] | *Some* *xs*  $\Rightarrow$  *xs*)

Filter for internal transitions.

**definition**

*trans\_i\_map* *i* *j*  $\equiv$   
*List.map\_filter*  
 $(\lambda (b, g, a, m, l'). \text{case } a \text{ of } \text{Sil } a \Rightarrow \text{Some } (b, g, a, m, l') \mid \_ \Rightarrow \text{None})$   
*(trans\_map* *i* *j*)

Compute valid internal successors given:

- a process *p*,
- initial location *l*,
- location vector *L*,
- and initial state *s*.

**definition**

```

int_trans_from_loc p l L s ≡
  let trans = trans_i_map p l
  in
  List.map_filter (λ (b, g, a, f, r, l').
    let s' = mk_upds s f in
    if bval (the o s) b ∧ check_bounded s' then Some (g, Internal a, r, (L[p := l'], s'))
    else None
  ) trans

```

**definition**

```

int_trans_from_vec pairs L s ≡
  concat (map (λ(p, l). int_trans_from_loc p l L s) pairs)

```

**definition**

```

int_trans_from_all L s ≡
  concat (map (λp. int_trans_from_loc p (L ! p) L s) [0.. $n\_ps$ ])

```

**definition**

```

int_trans_from ≡ λ (L, s).
  let pairs = get_committed L in
  if pairs = []
  then int_trans_from_all L s
  else int_trans_from_vec pairs L s

```

**definition**

```

actions_by_state i ≡
  fold (λ t acc. acc[fst (snd (snd t)) := (i, t) # (acc ! fst (snd (snd t)))])

```

**definition**

```

all_actions_by_state t L ≡
  fold (λ i. actions_by_state i (t i (L ! i))) [0.. $n\_ps$ ] (repeat [] num_actions)

```

**definition**

```

all_actions_from_vec t vec ≡
  fold (λ(p, l). actions_by_state p (t p l)) vec (repeat [] num_actions)

```

**definition**

```

pairs_by_action L s OUT IN ≡
  concat (
    map (λ (p, b1, g1, a1, f1, r1, l1).
      List.map_filter (λ (q, b2, g2, a2, f2, r2, l2).
        if p = q then None else
        let s' = mk_upds (mk_upds s f1) f2 in
        if bval (the o s) b1 ∧ bval (the o s) b2 ∧ check_bounded s'
        then Some (g1 @ g2, Bin a1, r1 @ r2, (L[p := l1, q := l2], s'))
        else None
      ) OUT) IN)

```

**definition**

```

trans_in_map i j ≡
  List.map_filter
    (λ (b, g, a, m, l'). case a of In a ⇒ Some (b, g, a, m, l') | _ ⇒ None)
    (trans_map i j)

```

**definition**

```

trans_out_map i j ≡
  List.map_filter
    (λ (b, g, a, m, l'). case a of Out a ⇒ Some (b, g, a, m, l') | _ ⇒ None)
    (trans_map i j)

```

**definition**

```

bin_actions = filter (λa. a ∉ set broadcast) [0..<num_actions]

```

**lemma** mem\_bin\_actions\_iff:

```

a ∈ set bin_actions ⟷ a ∉ local.broadcast ∧ a < num_actions

```

**unfolding** bin\_actions\_def broadcast\_def **by** auto

**definition**

```

bin_trans_from ≡ λ(L, s).
  let
    pairs = get_committed L;
    In = all_actions_by_state trans_in_map L;
    Out = all_actions_by_state trans_out_map L
  in
    if pairs = [] then
      concat (map (λa. pairs_by_action L s (Out ! a) (In ! a)) bin_actions)
    else
      let
        In2 = all_actions_from_vec trans_in_map pairs;
        Out2 = all_actions_from_vec trans_out_map pairs
      in
        concat (map (λa. pairs_by_action L s (Out ! a) (In2 ! a)) bin_actions)
        @ concat (map (λa. pairs_by_action L s (Out2 ! a) (In ! a)) bin_actions)

```

**definition**

```

trans_in_broad_map i j ≡
  List.map_filter
    (λ (b, g, a, m, l').
      case a of In a ⇒ if a ∈ set broadcast then Some (b, g, a, m, l') else None | _ ⇒ None)
    (trans_map i j)

```

**definition**

```

trans_out_broad_map i j ≡
  List.map_filter
    (λ (b, g, a, m, l').
      case a of Out a ⇒ if a ∈ set broadcast then Some (b, g, a, m, l') else None | _ ⇒ None)
    (trans_map i j)

```

**definition**

```

actions_by_state' xs ≡
  fold (λ t acc. acc[fst (snd (snd t))] := t # (acc ! fst (snd (snd t))))
    xs (repeat [] num_actions)

```

**definition**

$trans\_out\_broad\_grouped\ i\ j \equiv actions\_by\_state' (trans\_out\_broad\_map\ i\ j)$

**definition**

$trans\_in\_broad\_grouped\ i\ j \equiv actions\_by\_state' (trans\_in\_broad\_map\ i\ j)$

**definition**

$pair\_by\_action\ OUT\ IN \equiv$   
 $concat\ (\$   
 $\quad map\ (\lambda(g1, a, r1, (L, s)).$   
 $\quad\quad List.map\ (\lambda(q, g2, a2, f2, r2, l2).$   
 $\quad\quad\quad (g1\ @\ g2, a, r1\ @\ r2, (L[q := l2], mk\_upds\ s\ f2))$   
 $\quad\quad )\ OUT)\ IN)$

**definition make\_combs where**

$make\_combs\ p\ a\ xs \equiv$   
 $let$   
 $\quad ys = List.map\_filter$   
 $\quad\quad (\lambda i.$   
 $\quad\quad\quad if\ i = p\ then\ None$   
 $\quad\quad\quad else\ if\ xs\ !\ i\ !\ a = []\ then\ None$   
 $\quad\quad\quad else\ Some\ (map\ (\lambda t. (i, t))\ (xs\ !\ i\ !\ a))$   
 $\quad\quad )$   
 $\quad\quad [0..<n\_ps]$   
 $in\ if\ ys = []\ then\ []\ else\ product\_lists\ ys$

**definition make\_combs\_from\_pairs where**

$make\_combs\_from\_pairs\ p\ a\ pairs\ xs \equiv$   
 $let$   
 $\quad ys = List.map\_filter$   
 $\quad\quad (\lambda i.$   
 $\quad\quad\quad if\ i = p\ then\ None$   
 $\quad\quad\quad else\ if\ xs\ !\ i\ !\ a = []\ then\ None$   
 $\quad\quad\quad else\ Some\ (map\ (\lambda t. (i, t))\ (xs\ !\ i\ !\ a))$   
 $\quad\quad )$   
 $\quad\quad [0..<n\_ps]$   
 $in\ if\ ys = []\ then\ []\ else\ product\_lists\ ys$

**definition**

$broad\_trans\_from \equiv \lambda(L, s).$   
 $let$   
 $\quad pairs = get\_committed\ L;$   
 $\quad In = map\ (\lambda p. trans\_in\_broad\_grouped\ p\ (L\ !\ p))\ [0..<n\_ps];$   
 $\quad Out = map\ (\lambda p. trans\_out\_broad\_grouped\ p\ (L\ !\ p))\ [0..<n\_ps];$   
 $\quad In = map\ (map\ (filter\ (\lambda (b, \_). bval\ (the\ o\ s\ b)))\ In);$   
 $\quad Out = map\ (map\ (filter\ (\lambda (b, \_). bval\ (the\ o\ s\ b)))\ Out)$   
 $in$   
 $\quad if\ pairs = []\ then$   
 $\quad\quad concat\ (\$   
 $\quad\quad\quad map\ (\lambda a.$

```

concat (map (λp.
  let
    outs = Out ! p ! a
  in if outs = [] then []
  else
    let
      combs = make_combs p a In;
      outs = map (λt. (p, t)) outs;
      combs = (
        if combs = [] then [[x]. x ← outs]
        else concat (map (λx. map (λxs. x # xs) combs) outs));
      init = ([], Broad a, [], (L, s))
    in
      List.map_filter (λcomb.
        let (g, a, r, L', s) =
          fold
            (λ(q, _, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
              (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
            )
            comb
            init
          in if check_bounded s then Some (g, a, r, L', s) else None
        ) combs
      )
    [0..

```

```

      fold
      (λ(q, __, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
        (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
      )
      comb
      init
      in if check_bounded s then Some (g, a, r, L', s) else None
    ) combs
  )
  [0.. $n\_ps$ ])
)
[0.. $num\_actions$ ])

```

**end**

**lemma** *product\_lists\_empty*:  $product\_lists\ xss = [] \longleftrightarrow (\exists xs \in set\ xss. xs = [])$  **for**  $xss$   
**by** (*induction*  $xss$ ) *auto*

**context** *Simple\_Network\_Impl\_nat*  
**begin**

**lemma** *broad\_trans\_from\_alt\_def*:  
 $broad\_trans\_from \equiv \lambda(L, s).$   
*let*  
 $pairs = get\_committed\ L;$   
 $In = map\ (\lambda p. trans\_in\_broad\_grouped\ p\ (L\ !\ p))\ [0.. $n\_ps$ ];$   
 $Out = map\ (\lambda p. trans\_out\_broad\_grouped\ p\ (L\ !\ p))\ [0.. $n\_ps$ ];$   
 $In = map\ (map\ (filter\ (\lambda\ (b, \_). bval\ (the\ o\ s)\ b)))\ In;$   
 $Out = map\ (map\ (filter\ (\lambda\ (b, \_). bval\ (the\ o\ s)\ b)))\ Out$   
*in*  
 if  $pairs = []$  then  
 concat (  
 map ( $\lambda a.$   
 concat ( $map\ (\lambda p.$   
*let*  
 $outs = Out\ !\ p\ !\ a$   
 in if  $outs = []$  then []  
 else  
*let*  
 $combs = make\_combs\ p\ a\ In;$   
 $outs = map\ (\lambda t. (p, t))\ outs;$   
 $combs =$  (  
 if  $combs = []$  then  $[x]. x \leftarrow outs$   
 else concat ( $map\ (\lambda x. map\ (\lambda xs. x \# xs)\ combs)\ outs$ );  
 $init = ([], Broad\ a, [], (L, s))$   
 in  
 filter ( $\lambda\ (g, a, r, L, s). check\_bounded\ s)$  (  
 map ( $\lambda comb.$   
 fold  
 (λ(q, \_\_, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).  
 (g1 @ g2, a, r1 @ r2, (L[q := l2], mk\_upds s f2))  
 )  
 comb

```

      init
    ) combs)
  )
  [0..n_ps])
)
[0..num_actions])
else
  concat (
    map (λa.
      let
        ins_committed =
          List.map_filter (λ(p, _). if In ! p ! a ≠ [] then Some p else None) pairs
      in
        concat (map (λp.
          let
            outs = Out ! p ! a
            in if outs = [] then []
          else if
            (ins_committed = [p] ∨ ins_committed = []) ∧ ¬ list_ex (λ(q, _). q = p) pairs
          then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λx. map (λxs. x # xs) combs) outs);
              init = ([], Broad a, [], (L, s))
            in
              filter (λ (g, a, r, L, s). check_bounded s) (
                map (λcomb.
                  fold
                    (λ(q, _, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
                      (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
                    )
                  comb
                  init
                ) combs)
              )
            )
          )
        )
      )
    )
  )
  [0..num_actions])

```

```

apply (rule eq_reflection)
unfolding broad_trans_from_def
unfolding filter_map_map_filter
unfolding Let_def
by (fo_rule
  arg_cong2[where f = map] arg_cong2[where f = List.map_filter] arg_cong HOL.refl |
  rule if_cong ext | auto split: if_split_asm)+

```

**Correctness for implementations of primitives** lemma *dom\_bounds\_eq*:  
*dom bounds* = {0..*n\_vs*}

```

using bounds unfolding bounds_def
apply simp
unfolding n_vs_def dom_map_of_conv_image fst
apply safe
  apply (force dest: mem_nth; fail)
  apply (force dest: nth_mem; fail)
done

```

```

lemma check_bounded_iff:
  Simple_Network_Language.bounded bounds s  $\longleftrightarrow$  check_bounded s if dom s = {0.. $n\_vs$ }
  using that
  unfolding dom_bounds_eq[symmetric]
  unfolding check_bounded_def Simple_Network_Language.bounded_def bounds_map_def bounds_def
  by auto

```

```

lemma get_committed_mem_iff:
  (p, l)  $\in$  set (get_committed L)  $\longleftrightarrow$  (l = L ! p  $\wedge$  l  $\in$  committed (N p)  $\wedge$  p < n_ps)
  unfolding get_committed_def
  unfolding set_map_filter Let_def
  apply clarsimp
  unfolding N_def fst_conv snd_conv
  unfolding committed_def
  by safe
  ((subst nth_map | subst (asm) nth_map);
   auto split: prod.splits simp: automaton_of_def length_automata_eq n_ps
  )+

```

```

lemma get_committed_empty_iff:
  ( $\forall$  p < n_ps. L ! p  $\notin$  committed (N p))  $\longleftrightarrow$  get_committed L = []
  apply safe
  subgoal
  proof (rule ccontr)
    assume prems:
       $\forall$  p < n_ps. L ! p  $\notin$  committed (N p) and
      get_committed L  $\neq$  []
    then obtain p l where (p, l)  $\in$  set (get_committed L)
      by (metis length_greater_0_conv nth_mem old.prod.exhaust)
    from this[unfolded get_committed_mem_iff] prems(1)
    show False
      by auto
  qed
  subgoal for p
    using get_committed_mem_iff[of p L ! p L] by auto
  done

```

```

lemma get_committed_distinct: distinct (get_committed L)
  unfolding get_committed_def by (rule distinct_map_filterI) (auto simp: Let_def)

```

```

lemma is_upds_make_updsI2:
  is_upds s upds (mk_upds s upds)
  if (l, b, g, a, upds, r, l')  $\in$  Simple_Network_Language.trans (N p) p < n_ps
  dom s = {0.. $n\_vs$ }
  using that var_set
  by (intro is_upds_make_updsI, subst (asm) mem_trans_N_iff)

```

(auto 4 5 simp flip: length\_automata\_eq\_n\_ps dest!: nth\_mem)

**lemma** var\_setD:

$\forall (x, upd) \in \text{set } f. x < n\_vs \wedge (\forall i \in \text{vars\_of\_exp } upd. i < n\_vs)$   
**if**  $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p) \ p < n\_ps$   
**using** var\_set **that**  
**by** (force dest: nth\_mem simp flip: length\_automata\_eq\_n\_ps simp: mem\_trans\_N\_iff)+

**lemma** var\_setD2:

$\forall i \in \text{vars\_of\_bexp } b. i < n\_vs$   
**if**  $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p) \ p < n\_ps$   
**using** var\_set **that**  
**by** (force dest: nth\_mem simp flip: length\_automata\_eq\_n\_ps simp: mem\_trans\_N\_iff)+

**lemma** is\_upds\_dom2:

$\text{dom } s' = \{0..<n\_vs\}$  **if**  $\text{is\_upds } s \ f \ s'$   
 $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p) \ p < n\_ps$   
 $\text{dom } s = \{0..<n\_vs\}$   
**unfolding** that(4)[symmetric] **using** that **by** – (drule (1) var\_setD, erule is\_upds\_dom, auto)

**lemma** is\_updsD:

$s' = \text{mk\_upds } s \ f$  **if**  $\text{is\_upds } s \ f \ s'$   
 $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p) \ p < n\_ps$   
 $\text{dom } s = \{0..<n\_vs\}$   
**proof** –  
**from** is\_upds\_make\_updsI2[OF that(2–)] **have**  $\text{is\_upds } s \ f \ (\text{mk\_upds } s \ f)$  .  
**from** is\_upds\_determ[OF that(1) this] **show** ?thesis .

**qed**

**lemma** is\_upds\_make\_upds\_concatI2:

$\text{is\_upds } s \ (\text{concat } upds) \ (\text{mk\_upds } s \ (\text{concat } upds))$   
**if**  $\text{dom } s = \{0..<n\_vs\}$   
 $\forall upd \in \text{set } upds. \exists p < n\_ps. \exists l \ b \ g \ a \ r \ l'.$   
 $(l, b, g, a, upd, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   
**using** that var\_set  
**by** (intro is\_upds\_make\_updsI, clarsimp)  
(smt (z3) atLeastLessThan\_iff case\_prod\_conv domD var\_setD zero\_le)

**lemma** is\_upds\_concat\_dom2:

**assumes**  $\text{is\_upds } s \ (\text{concat } upds) \ s'$   
**and**  $\forall f \in \text{set } upds. \exists p < n\_ps. \exists l \ b \ g \ a \ r \ l'.$   
 $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   
**and**  $\text{dom } s = \{0..<n\_vs\}$   
**shows**  $\text{dom } s' = \text{dom } s$   
**using** assms **by** (elim is\_upds\_dom) (auto dest!: var\_setD)

**lemma** is\_upds\_dom3:

**assumes**  $\text{is\_upds } s \ (\text{concat\_map } fs \ ps) \ s'$   
**and**  $\forall p \in \text{set } ps. (L \ ! \ p, bs \ p, gs \ p, a, fs \ p, rs \ p, ls' \ p) \in \text{trans } (N \ p)$   
**and**  $\text{set } ps \subseteq \{0..<n\_ps\}$   
**and**  $\text{dom } s = \{0..<n\_vs\}$   
**shows**  $\text{dom } s' = \text{dom } s$   
**using** assms **by** (elim is\_upds\_concat\_dom2; force)

**lemma** *is\_upds\_make\_updsI3*:  
*is\_upds* *s* (*concat\_map* *fs ps*) (*mk\_upds* *s* (*concat\_map* *fs ps*))  
**if** *dom s* = {0..*n\_vs*}  
**and**  $\forall p \in \text{set } ps. (L \! p, bs \ p, gs \ p, a, fs \ p, rs \ p, ls' \ p) \in \text{trans } (N \ p)$   
**and** *set ps*  $\subseteq$  {0..*n\_ps*}  
**for** *s* :: *nat*  $\Rightarrow$  *int option*  
**using** *that* **by** (*elim is\_upds\_make\_upds\_concatI2*) *force*

**lemma** *is\_upds\_concatD*:  
**assumes**  
*dom s* = {0..*n\_vs*} **and**  
 $\forall p \in \text{set } ps.$   
 (*ls p*, *bs p*, *gs p*, *a*, *fs p*, *rs p*, *ls' p*)  
 $\in \text{Simple\_Network\_Language.trans } (N \ p)$  **and**  
*set ps*  $\subseteq$  {0..*n\_ps*} **and**  
*is\_upds* *s* (*concat\_map* *fs ps*) *s'*  
**shows** *s'* = *mk\_upds* *s* (*concat\_map* *fs ps*)  
**proof** –  
**let** *?upds* = *concat\_map* *fs ps*  
**have** *is\_upds* *s* *?upds* (*mk\_upds* *s* *?upds*)  
**using** *assms*(1–3) **by** (*intro is\_upds\_make\_upds\_concatI2*; *force*)  
**from** *is\_upds\_determ*[*OF* *assms*(4) *this*] **show** *?thesis*  
**by** (*simp add: fold\_map\_comp\_def*)  
**qed**

**context**  
**notes** [*simp*] = *length\_automata\_eq\_n\_ps*  
**begin**

**lemma** *trans\_mapI*:  
**assumes**  
 (*L ! p*, *g*, *a*, *f*, *r*, *l'*)  $\in \text{Simple\_Network\_Language.trans } (N \ p)$   
*p* < *n\_ps*  
**shows**  
 (*g*, *a*, *f*, *r*, *l'*)  $\in \text{set } (\text{trans\_map } p \ (L \! p))$   
**using** *assms* **unfolding** *trans\_map\_def* *N\_def* *fst\_conv* *snd\_conv* *trans\_def*  
**by** (*subst* (*asm*) *nth\_map*) (*auto dest: in\_union\_map\_ofI split: prod.split\_asm simp: automa-*  
*ton\_of\_def*)

**lemma** *trans\_i\_mapI*:  
**assumes**  
 (*L ! p*, *b*, *g*, *Sil a'*, *f*, *r*, *l'*)  $\in \text{Simple\_Network\_Language.trans } (N \ p)$   
*p* < *n\_ps*  
**shows**  
 (*b*, *g*, *a'*, *f*, *r*, *l'*)  $\in \text{set } (\text{trans\_i\_map } p \ (L \! p))$   
**using** *assms* **unfolding** *trans\_i\_map\_def* *set\_map\_filter* **by** (*fastforce dest: trans\_mapI*)

**lemma** *trans\_mapI'*:  
**assumes**  
 (*l*, *b*, *g*, *a*, *f*, *r*, *l'*)  $\in \text{Simple\_Network\_Language.trans } (N \ p)$   
*p* < *n\_ps*  
**shows**

$(b, g, a, f, r, l') \in \text{set } (\text{trans\_map } p \ l)$   
**using** *assms* **unfolding** *trans\_map\_def N\_def fst\_conv snd\_conv trans\_def*  
**by** (*subst (asm) nth\_map*) (*auto dest: in\_union\_map\_ofI split: prod.split\_asm simp: automa-*  
*ton\_of\_def*)

**lemma** *trans\_mapD*:

**assumes**

$(b, g, a, f, r, l') \in \text{set } (\text{trans\_map } p \ l)$   
 $p < n\_ps$

**shows**

$(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   
**using** *assms* **unfolding** *trans\_map\_def N\_def fst\_conv snd\_conv trans\_def*  
**by** (*subst nth\_map*) (*auto split: prod.split elim: in\_union\_map\_ofD[rotated] simp: automa-*  
*ton\_of\_def*)

**lemma** *trans\_map\_iff*:

**assumes**

$p < n\_ps$

**shows**

$(b, g, a, f, r, l') \in \text{set } (\text{trans\_map } p \ l)$   
 $\longleftrightarrow (l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   
**using** *trans\_mapD trans\_mapI' <p < n\_ps>* **by** *auto*

**lemma** *trans\_i\_mapD*:

**assumes**

$(b, g, a', f, r, l') \in \text{set } (\text{trans\_i\_map } p \ (L \ ! \ p))$   
 $p < n\_ps$

**shows**

$(L \ ! \ p, b, g, \text{Sil } a', f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   
**using** *assms* **unfolding** *trans\_i\_map\_def set\_map\_filter*  
**by** (*force split: act.split\_asm intro: trans\_mapD*)

**An additional brute force method for forward-chaining of facts** *method frules\_all =*  
*repeat\_rotate <frules>, dedup\_premis*

**Internal transitions** **lemma** *get\_committed\_memI*:

$(p, L \ ! \ p) \in \text{set } (\text{get\_committed } L)$  **if**  $L \ ! \ p \in \text{committed } (N \ p)$   $p < n\_ps$   
**using** *that* **unfolding** *get\_committed\_mem\_iff* **by** *simp*

**lemma** *check\_bexp\_bvalI*:

$\text{bval } (\text{the } o \ s) \ b$  **if**  $\text{check\_bexp } s \ b \ \text{True}$   
 $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   $p < n\_ps$   
 $\text{dom } s = \{0..<n\_vs\}$

**proof** –

**from** *var\_setD2[OF that(2,3)] <dom s = {0..<n\_vs>}>* **have**  $\text{check\_bexp } s \ b \ (\text{bval } (\text{the } o \ s) \ b)$   
**by** (*intro check\_bexp\_bval, simp*)  
**with** *check\_bexp\_determ that(1)* **show** *?thesis*  
**by** *auto*

**qed**

**lemma** *check\_bexp\_bvalD*:

$\text{check\_bexp } s \ b \ \text{True}$  **if**  $\text{bval } (\text{the } o \ s) \ b$   
 $(l, b, g, a, f, r, l') \in \text{Simple\_Network\_Language.trans } (N \ p)$   $p < n\_ps$

```

    dom s = {0.. $n\_vs$ }
proof -
  from var_setD2[OF that(2,3)] ‹dom s = {0.. $n\_vs$ }› have check_bexp s b (bval (the o s) b)
    by (intro check_bexp_bval, simp)
  with check_bexp_determ that(1) show ?thesis
    by auto
qed

lemmas [forward2] =
  trans_i_mapD
  trans_i_mapI
  get_committed_memI
lemmas [forward3] =
  is_upds_make_updsI2
lemmas [forward4] =
  is_updsD
  is_upds_dom2
  check_bexp_bvalI
  check_bexp_bvalD

lemma int_trans_from_correct:
  (int_trans_from, trans_int) ∈ transition_rel states'
  unfolding transition_rel_def
proof (safe del: iffI)
  note [more_elims] = allE
  fix L s g a r L' s' assume (L, s) ∈ states'
  then have L ∈ states dom s = {0.. $n\_vs$ } bounded bounds s
    by auto
  then have [simp]: length L =  $n\_ps$  check_bounded s
    by (auto simp: check_bounded_iff)
  show (((L, s), g, a, r, L', s') ∈ trans_int)
    ‹ $\longleftrightarrow$  ((g, a, r, L', s') ∈ set (int_trans_from (L, s)))›
proof (cases get_committed L = [])
  case True
  then have *: ((L, s), g, a, r, L', s') ∈ trans_int ‹ $\longleftrightarrow$ 
    ((L, s), g, a, r, L', s') ∈ {((L, s), g, Internal a, r, (L', s')) | L s l b g f p a r l' L' s'.
      (l, b, g, Sil a, f, r, l') ∈ trans (N p) ∧
      (∀ p <  $n\_ps$ . L ! p ∉ committed (N p)) ∧
      check_bexp s b True ∧
      L!p = l ∧ p < length L ∧ L' = L[p := l'] ∧ is_upds s f s' ∧
      L ∈ states ∧ bounded bounds s ∧ bounded bounds s'}
    ›
  unfolding get_committed_empty_iff[symmetric] trans_int_def by blast
from True have **: int_trans_from (L, s) = int_trans_from_all L s
  unfolding int_trans_from_def by simp
from ‹dom s = _› show ?thesis
  unfolding * ** int_trans_from_all_def
  apply clarsimp
  unfolding int_trans_from_loc_def Let_def set_map_filter
  apply clarsimp
  apply safe
  subgoal for b f p a' l'
    apply frules
    unfolding check_bounded_iff by (intros; solve_triv)

```

```

    subgoal for  $p \_ a' \text{ upds } l'$ 
      apply simp
      apply frules
      using  $\langle L \in \text{states} \rangle \langle \text{check\_bounded } s \rangle \text{ True}[\text{folded get\_committed\_empty\_iff}]$ 
      unfolding check_bounded_iff by (intros; solve_triv)
    done
  next
  case False
  then have *:  $((L, s), g, a, r, L', s') \in \text{trans\_int} \longleftrightarrow$ 
     $((L, s), g, a, r, L', s') \in \{((L, s), g, \text{Internal } a, r, (L', s')) \mid L \text{ s } l \text{ b } g \text{ f } p \text{ a } r \text{ l' } L' \text{ s'}. \\ (l, b, g, \text{Sil } a, f, r, l') \in \text{trans } (N \text{ p}) \wedge \\ l \in \text{committed } (N \text{ p}) \wedge \\ L!p = l \wedge p < \text{length } L \wedge \text{check\_bexp } s \text{ b } \text{ True} \wedge L' = L[p := l'] \wedge \text{is\_upds } s \text{ f } s' \wedge \\ L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s'\}$ 
    }
    unfolding get_committed_empty_iff[symmetric] trans_int_def by blast
  from False have **:  $\text{int\_trans\_from } (L, s) = \text{int\_trans\_from\_vec } (\text{get\_committed } L) \text{ L } s$ 
    unfolding int_trans_from_def by simp
  from  $\langle \text{dom } s = \_ \rangle \langle L \in \text{states} \rangle$  show ?thesis
    unfolding * ** int_trans_from_vec_def
    apply clarsimp
    unfolding int_trans_from_loc_def Let_def set_map_filter
    apply clarsimp
    apply safe
    subgoal for  $f \text{ p } a' \text{ l'}$ 
      apply frules
      unfolding check_bounded_iff by (intros; solve_triv)
    subgoal for  $p \_ a' \text{ upds } l'$ 
      unfolding get_committed_mem_iff
      apply (elims; simp)
      apply frules
      unfolding check_bounded_iff by (intros; solve_triv)
    done
  qed
qed

```

Mostly copied lemma in\_actions\_by\_stateI:

```

assumes
   $(b1, g1, a, r1) \in \text{set } xs \text{ a} < \text{length } acc$ 
shows
   $(q, b1, g1, a, r1) \in \text{set } (\text{actions\_by\_state } q \text{ xs } acc ! a)$ 
   $\wedge a < \text{length } (\text{actions\_by\_state } q \text{ xs } acc)$ 
using assms unfolding actions_by_state_def
apply (induction xs arbitrary: acc)
apply (simp; fail)
apply simp
apply (erule disjE)
apply (rule fold_acc_preserv
  [where  $P = \lambda acc. (q, b1, g1, a, r1) \in \text{set } (acc ! a) \wedge a < \text{length } acc]$ 
  )
  apply (subst list_update_nth_split; auto)
by auto

```

```

lemma in_actions_by_state'I:
  assumes
     $(b1, g1, a, r1) \in \text{set } xs \ a < \text{num\_actions}$ 
  shows
     $(b1, g1, a, r1) \in \text{set } (\text{actions\_by\_state}' xs \ ! \ a)$ 
     $\wedge a < \text{length } (\text{actions\_by\_state}' xs)$ 
proof -
  let  $?f = (\lambda t \ \text{acc}. \text{acc}[\text{fst } (\text{snd } (\text{snd } t)) := t \ \# \ \text{acc} \ ! \ \text{fst } (\text{snd } (\text{snd } t))])$ 
  have  $(b1, g1, a, r1) \in \text{set } (\text{fold } ?f \ xs \ \text{acc} \ ! \ a)$ 
     $\wedge a < \text{length } (\text{fold } ?f \ xs \ \text{acc})$  if  $a < \text{length } \text{acc}$  for  $\text{acc}$ 
  using assms(1) that
  apply (induction xs arbitrary: acc)
  apply (simp; fail)
  apply simp
  apply (erule disjE)
  apply (rule fold_acc_preserv
    [where  $P = \lambda \text{acc}. (b1, g1, a, r1) \in \text{set } (\text{acc} \ ! \ a) \wedge a < \text{length } \text{acc}$ ]
  )
  apply (subst list_update_nth_split; auto)
  by auto
  with  $\langle a < \_ \rangle$  show ?thesis
  unfolding actions_by_state'_def by simp
qed

```

```

lemma in_actions_by_state_preserv:
  assumes
     $(q, b1, g1, a, r1) \in \text{set } (\text{acc} \ ! \ a) \ a < \text{length } \text{acc}$ 
  shows
     $(q, b1, g1, a, r1) \in \text{set } (\text{actions\_by\_state } y \ xs \ \text{acc} \ ! \ a)$ 
     $\wedge a < \text{length } (\text{actions\_by\_state } y \ xs \ \text{acc})$ 
  using assms unfolding actions_by_state_def
  apply -
  apply (rule fold_acc_preserv
    [where  $P = \lambda \text{acc}. (q, b1, g1, a, r1) \in \text{set } (\text{acc} \ ! \ a) \wedge a < \text{length } \text{acc}$ ]
  )
  apply (subst list_update_nth_split; auto)
  by auto

```

```

lemma length_actions_by_state_preserv[simp]:
  shows  $\text{length } (\text{actions\_by\_state } y \ xs \ \text{acc}) = \text{length } \text{acc}$ 
  unfolding actions_by_state_def by (auto intro: fold_acc_preserv simp: list_update_nth_split)

```

```

lemma in_all_actions_by_stateI:
  assumes
     $a < \text{num\_actions} \ q < n\_ps \ (b1, g1, a, r1) \in \text{set } (M \ q \ (L \ ! \ q))$ 
  shows
     $(q, b1, g1, a, r1) \in \text{set } (\text{all\_actions\_by\_state } M \ L \ ! \ a)$ 
  unfolding all_actions_by_state_def
  apply (rule fold_acc_ev_preserv
    [where  $P = \lambda \text{acc}. (q, b1, g1, a, r1) \in \text{set } (\text{acc} \ ! \ a)$  and  $Q = \lambda \text{acc}. a < \text{length } \text{acc},$ 
      THEN conjunct1]
  )
  apply (rule in_actions_by_state_preserv[THEN conjunct1])
  using assms by (auto intro: in_actions_by_stateI[THEN conjunct1])

```

**lemma** *in\_all\_actions\_from\_vecI*:  
**assumes**  
 $a < \text{num\_actions } (b1, g1, a, r1) \in \text{set } (M \ q \ l) \ (q, l) \in \text{set pairs}$   
**shows**  
 $(q, b1, g1, a, r1) \in \text{set } (\text{all\_actions\_from\_vec } M \text{ pairs } ! \ a)$   
**unfolding** *all\_actions\_from\_vec\_def* **using** *assms*  
**by** (*intro fold\_acc\_ev\_preserv*  
 $[\text{where } P = \lambda \text{ acc. } (q, b1, g1, a, r1) \in \text{set } (\text{acc } ! \ a) \text{ and } Q = \lambda \text{ acc. } a < \text{length acc},$   
 $\text{THEN conjunct1}]$ )  
 $(\text{auto intro: in\_actions\_by\_stateI}[\text{THEN conjunct1}] \text{ in\_actions\_by\_state\_preserv}[\text{THEN}$   
 $\text{conjunct1}])$

**lemma** *actions\_by\_state\_inj*:  
**assumes**  $j < \text{length acc}$   
**shows**  $\forall (q, a) \in \text{set } (\text{actions\_by\_state } i \ xs \ \text{acc } ! \ j). (q, a) \notin \text{set } (\text{acc } ! \ j) \longrightarrow i = q$   
**unfolding** *actions\_by\_state\_def*  
**apply** (*rule fold\_acc\_preserv*  
 $[\text{where } P =$   
 $\lambda \text{ acc'. } (\forall (q, a) \in \text{set } (\text{acc' } ! \ j). (q, a) \notin \text{set } (\text{acc } ! \ j) \longrightarrow i = q) \wedge j < \text{length acc'},$   
 $\text{THEN conjunct1}]$ )  
**subgoal for**  $x \ \text{acc}$   
**by** (*cases fst (snd (snd x)) = j; simp*)  
**using** *assms* **by** *auto*

**lemma** *actions\_by\_state\_inj'*:  
**assumes**  $j < \text{length acc}$   $(q, a) \notin \text{set } (\text{acc } ! \ j)$   $(q, a) \in \text{set } (\text{actions\_by\_state } i \ xs \ \text{acc } ! \ j)$   
**shows**  $i = q$   
**using** *actions\_by\_state\_inj*[*OF assms(1)*] *assms(2-)* **by** *fast*

**lemma** *in\_actions\_by\_stateD*:  
**assumes**  
 $(q, b, g, a, t) \in \text{set } (\text{actions\_by\_state } i \ xs \ \text{acc } ! \ j) \ (q, b, g, a, t) \notin \text{set } (\text{acc } ! \ j)$   
 $j < \text{length acc}$   
**shows**  
 $(b, g, a, t) \in \text{set } xs \wedge j = a$   
**using** *assms* **unfolding** *actions\_by\_state\_def*  
**apply**  $-$   
**apply** (*drule fold\_evD*  
 $[\text{where } y = (b, g, a, t) \text{ and } Q = \lambda \text{ acc'. } \text{length acc'} = \text{length acc}$   
 $\text{and } R = \lambda (\_, \_, a', t). a' = j]$   
 $)$   
**apply** *assumption*  
**apply** (*subst (asm) list\_update\_nth\_split*[*of j*]; *force*)  
**apply** *simp+*  
**apply** (*subst (asm) list\_update\_nth\_split*[*of j*]; *force*)  
**by** *auto*

**lemma** *in\_actions\_by\_state'D*:  
**assumes**  
 $(b, g, a, r) \in \text{set } (\text{actions\_by\_state' } xs \ ! \ a') \ a' < \text{num\_actions}$   
**shows**  
 $(b, g, a, r) \in \text{set } xs \wedge a' = a$   
**proof**  $-$

```

let ?f = ( $\lambda t$  acc. acc[fst (snd (snd t)) := t # acc ! fst (snd (snd t))])
have (b, g, a, r)  $\in$  set xs  $\wedge$  a' = a
  if (b, g, a, r)  $\in$  set (fold ?f xs acc ! a') (b, g, a, r)  $\notin$  set (acc ! a') a' < length acc
  for acc
  using that
  apply -
  apply (drule fold_evD
    [where y = (b, g, a, r) and Q =  $\lambda$  acc'. length acc' = length acc
      and R =  $\lambda$  (_, _, a, t). a = a']
    )
    apply ((subst (asm) list_update_nth_split[where j = a'])?; solves auto)+
  done
with assms show ?thesis
  unfolding actions_by_state'_def by auto
qed

```

```

lemma in_all_actions_by_stateD:
  assumes
    (q, b1, g1, a, r1)  $\in$  set (all_actions_by_state M L ! a') a' < num_actions
  shows
    (b1, g1, a, r1)  $\in$  set (M q (L ! q))  $\wedge$  q < n_ps  $\wedge$  a' = a
  using assms
  unfolding all_actions_by_state_def
  apply -
  apply (drule fold_evD''[where y = q and Q =  $\lambda$  acc. length acc = num_actions])
    apply (simp; fail)
    apply (drule actions_by_state_inj[rotated])
    apply (simp; fail)+
  apply safe
  apply (drule in_actions_by_stateD)
    apply assumption
    apply (rule fold_acc_preserv)
    apply (simp; fail)+
  subgoal premises prems
  proof -
    from prems(2) have q  $\in$  set [0.. $n\_ps$ ] by auto
    then show ?thesis by simp
  qed
  by (auto intro: fold_acc_preserv dest!: in_actions_by_stateD)

```

```

lemma length_all_actions_by_state_preserv:
  length (all_actions_by_state M L) = num_actions
  unfolding all_actions_by_state_def by (auto intro: fold_acc_preserv)

```

```

lemma length_actions_by_state'_preserv:
  length (actions_by_state' xs) = num_actions
  unfolding actions_by_state'_def by (rule fold_acc_preserv; simp)

```

```

lemma all_actions_from_vecD:
  assumes
    (q, b1, g1, a, r1)  $\in$  set (all_actions_from_vec M pairs ! a') a' < num_actions
    distinct (map fst pairs)
  shows
     $\exists$  l. (q, l)  $\in$  set pairs  $\wedge$  (b1, g1, a, r1)  $\in$  set (M q l)  $\wedge$  a' = a

```

```

using assms(1,2)
unfolding all_actions_from_vec_def
apply -
apply (drule fold_evD2'[where
  y = (q, SOME l. (q, l) ∈ set pairs)
  and Q = λ acc. length acc = num_actions])
  apply (clarsimp; fail)
  apply clarsimp
  apply (drule actions_by_state_inj'[rotated])
    apply assumption
    apply assumption
  apply simp
subgoal
  using assms(3) by (meson distinct_map_fstD someI_ex)
apply (solves auto)+
apply clarsimp
apply (drule in_actions_by_stateD)
apply (simp; fail)
  apply (rule fold_acc_preserv)
  apply (solves auto)+
subgoal premises prems for ys zs
  using prems(4) by (subst ⟨pairs = _⟩) auto
done

```

```

lemma all_actions_from_vecD2:
  assumes
    (q, b1, g1, a, r1) ∈ set (all_actions_from_vec M pairs ! a') a' < num_actions
    (q, l) ∈ set pairs distinct (map fst pairs)
  shows
    (b1, g1, a, r1) ∈ set (M q l) ∧ a' = a
  using assms(1,2,3)
  unfolding all_actions_from_vec_def
  apply -
  apply (drule fold_evD2'[where y = (q, l) and Q = λ acc. length acc = num_actions])
    apply (clarsimp; fail)
    apply clarsimp
    apply (drule actions_by_state_inj'[rotated])
      apply assumption
      apply assumption
    apply simp
  subgoal
    using assms(4) by (meson distinct_map_fstD)
  by (auto intro: fold_acc_preserv dest!: in_actions_by_stateD)

```

**Binary transitions** lemma *bin\_trans\_from\_correct*:

```

(bin_trans_from, trans_bin) ∈ transition_rel states'
unfolding transition_rel_def
proof (safe del: iffI)
  fix L s g a r L' s' assume (L, s) ∈ states'
  then have L ∈ states dom s = {0..n_vs} bounded bounds s
    by auto
  then have [simp]: length L = n_ps
    by auto

```

```

define IN where IN = all_actions_by_state trans_in_map L
define OUT where OUT = all_actions_by_state trans_out_map L
have IN_I:
  (p, b, g, a', f, r, l') ∈ set (IN ! a')
  if (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    p < n_ps a' < num_actions
  for p b g a' f r l'
proof -
  from trans_mapI[OF that(1,2)] have (b, g, In a', f, r, l') ∈ set (trans_map p (L ! p))
    by auto
  then have (b, g, a', f, r, l') ∈ set (trans_in_map p (L ! p))
    unfolding trans_in_map_def set_map_filter by (auto 4 7)
  with ⟨p < _⟩ ⟨a' < _⟩ show ?thesis
    unfolding IN_def by (intro in_all_actions_by_stateI)
qed
have OUT_I:
  (p, b, g, a', f, r, l') ∈ set (OUT ! a')
  if (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    p < n_ps a' < num_actions
  for p b g a' f r l'
proof -
  from trans_mapI[OF that(1,2)] have (b, g, Out a', f, r, l') ∈ set (trans_map p (L ! p))
    by auto
  then have (b, g, a', f, r, l') ∈ set (trans_out_map p (L ! p))
    unfolding trans_out_map_def set_map_filter by (auto 4 7)
  with ⟨p < _⟩ ⟨a' < _⟩ show ?thesis
    unfolding OUT_def by (intro in_all_actions_by_stateI)
qed
have IN_D:
  (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p) ∧ p < n_ps ∧ a' = a1
  if (p, b, g, a', f, r, l') ∈ set (IN ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that
  unfolding IN_def
  apply -
  apply (drule in_all_actions_by_stateD)
  apply assumption
  unfolding trans_in_map_def set_map_filter
  apply (clarsimp split: option.split_asm)
  apply (auto split: act.split_asm dest: trans_mapD)
  done
have OUT_D:
  (L ! p, b, g, Out a1, f, r, l') ∈ Simple_Network_Language.trans (N p) ∧ p < n_ps
  if (p, b, g, a', f, r, l') ∈ set (OUT ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that
  unfolding OUT_def
  apply -
  apply (drule in_all_actions_by_stateD)
  apply assumption
  unfolding trans_out_map_def set_map_filter
  apply (clarsimp split: option.split_asm)
  apply (auto split: act.split_asm dest: trans_mapD)
  done

```

```

have IN_p_num:
  p < n_ps
  if (p, b, g, a', f, r, l') ∈ set (IN ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that unfolding IN_def by (auto dest: in_all_actions_by_stateD split: option.split_asm)
have OUT_p_num:
  p < n_ps
  if (p, b, g, a', f, r, l') ∈ set (OUT ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that unfolding OUT_def by (auto dest: in_all_actions_by_stateD split: option.split_asm)
have actions_unique:
  a' = a1
  if (p, b, g, a', f, r, l') ∈ set (IN ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that unfolding IN_def trans_in_map_def set_map_filter
  by (auto dest: in_all_actions_by_stateD split: option.split_asm)
note [forward3] = OUT_I IN_I
note [forward2] = action_setD IN_D OUT_D
show (((L, s), g, a, r, L', s') ∈ trans_bin) =
  ((g, a, r, L', s') ∈ set (bin_trans_from (L, s)))
proof (cases get_committed L = [])
  case True
  with get_committed_empty_iff[of L] have ∀ p < n_ps. L ! p ∉ committed (N p)
  by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_bin ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
      L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
      a ∉ local.broadcast ∧
      (l1, b1, g1, In a, f1, r1, l1') ∈ trans (N p) ∧
      (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
      L!p = l1 ∧ L!q = l2 ∧ p < length L ∧ q < length L ∧ p ≠ q ∧
      check_bexp s b1 True ∧ check_bexp s b2 True ∧
      L' = L[p := l1', q := l2'] ∧ is_upds s f1 s' ∧ is_upds s' f2 s''
      ∧ L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }
  unfolding trans_bin_def by blast
from True have **:
  bin_trans_from (L, s)
  = concat (map (λa. pairs_by_action L s (OUT ! a) (IN ! a)) bin_actions)
  unfolding bin_trans_from_def IN_def OUT_def by simp
from ⟨dom s = _⟩ ⟨L ∈ _⟩ show ?thesis
  unfolding * **
  apply clarsimp
  unfolding pairs_by_action_def
  apply (clarsimp simp: set_map_filter Let_def)
  apply safe
  subgoal for _ s'' _ a' p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'
  apply clarsimp
  apply (inst_existentials a')
  subgoal
  apply frules
  apply simp
  apply frules_all

```

```

    unfolding check_bounded_iff by (intros; solve_triv)
  subgoal
    by (simp add: mem_bin_actions_iff; frules; simp)
  done
subgoal
  unfolding mem_bin_actions_iff
  apply simp
  apply (erule conjE)
  apply frules
  apply elims
  apply frules_all
  apply frules_all
  apply elims
  apply intros
  using ⟨bounded bounds s⟩ unfolding check_bounded_iff[symmetric] by solve_triv+
done
next
case False
with get_committed_empty_iff[of L] have  $\exists p < n\_ps. L \vdash p \in committed (N p)$ 
  by simp
then have *:  $((L, s), g, a, r, L', s') \in trans\_bin \longleftrightarrow ((L, s), g, a, r, L', s') \in$ 
 $\{((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) \mid$ 
 $L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'. \quad$ 
 $a \notin local.broadcast \wedge$ 
 $(l1, b1, g1, In a, f1, r1, l1') \in trans (N p) \wedge$ 
 $(l2, b2, g2, Out a, f2, r2, l2') \in trans (N q) \wedge$ 
 $(l1 \in committed (N p) \vee l2 \in committed (N q)) \wedge$ 
 $L!p = l1 \wedge L!q = l2 \wedge p < length L \wedge q < length L \wedge p \neq q \wedge$ 
 $check\_bexp s b1 True \wedge check\_bexp s b2 True \wedge$ 
 $L' = L[p := l1', q := l2'] \wedge is\_upds s f1 s' \wedge is\_upds s' f2 s'' \wedge$ 
 $L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s''$ 
 $\}$ 
  unfolding trans_bin_def
  by - (rule iffI; elims add: CollectE; intros add: CollectI; blast)
let ?S1 =
 $\{((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) \mid$ 
 $L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'. \quad$ 
 $a \notin local.broadcast \wedge$ 
 $(l1, b1, g1, In a, f1, r1, l1') \in trans (N p) \wedge$ 
 $(l2, b2, g2, Out a, f2, r2, l2') \in trans (N q) \wedge$ 
 $l1 \in committed (N p) \wedge$ 
 $L!p = l1 \wedge L!q = l2 \wedge p < length L \wedge q < length L \wedge p \neq q \wedge$ 
 $check\_bexp s b1 True \wedge check\_bexp s b2 True \wedge$ 
 $L' = L[p := l1', q := l2'] \wedge is\_upds s f1 s' \wedge is\_upds s' f2 s'' \wedge$ 
 $L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s''$ 
 $\}$ 
let ?S2 =
 $\{((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) \mid$ 
 $L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'. \quad$ 
 $a \notin local.broadcast \wedge$ 
 $(l1, b1, g1, In a, f1, r1, l1') \in trans (N p) \wedge$ 
 $(l2, b2, g2, Out a, f2, r2, l2') \in trans (N q) \wedge$ 
 $l2 \in committed (N q) \wedge$ 
 $L!p = l1 \wedge L!q = l2 \wedge$ 

```

```

    p < length L ∧ q < length L ∧ p ≠ q ∧
    check_bexp s b1 True ∧ check_bexp s b2 True ∧
    L' = L[p := l1', q := l2'] ∧
    is_upds s f1 s' ∧ is_upds s' f2 s'' ∧
    L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
  }
have *: ((L, s), g, a, r, L', s') ∈ trans_bin ⟷
  ((L, s), g, a, r, L', s') ∈ ?S1 ∨ ((L, s), g, a, r, L', s') ∈ ?S2
  unfolding * by clarsimp (rule iffI; elims add: disjE; intros add: disjI1 disjI2 HOL.refl)
define pairs where pairs = get_committed L
define In2 where In2 = all_actions_from_vec trans_in_map pairs
define Out2 where Out2 = all_actions_from_vec trans_out_map pairs
have In2_I:
  (p, b, g, a', f, r, l') ∈ set (In2 ! a')
  if (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
  p < n_ps a' < num_actions L ! p ∈ committed (N p)
  for p b g a' f r l'
proof -
  from ⟨L ! p ∈ committed (N p)⟩ ⟨p < n_ps⟩ have (p, L ! p) ∈ set pairs
  unfolding pairs_def get_committed_mem_iff by blast
  from trans_mapI[OF that(1,2)] have (b, g, In a', f, r, l') ∈ set (trans_map p (L ! p))
  by auto
  then have (b, g, a', f, r, l') ∈ set (trans_in_map p (L ! p))
  unfolding trans_in_map_def set_map_filter by (auto 4 7)
  with ⟨p < _⟩ ⟨a' < _⟩ ⟨_ ∈ set pairs⟩ show ?thesis
  unfolding In2_def by (intro in_all_actions_from_vecI)
qed
have Out2_I:
  (p, b, g, a', f, r, l') ∈ set (Out2 ! a')
  if (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
  p < n_ps a' < num_actions L ! p ∈ committed (N p)
  for p b g a' f r l'
proof -
  from ⟨L ! p ∈ committed (N p)⟩ ⟨p < n_ps⟩ have (p, L ! p) ∈ set pairs
  unfolding pairs_def get_committed_mem_iff by blast
  from trans_mapI[OF that(1,2)] have (b, g, Out a', f, r, l') ∈ set (trans_map p (L ! p))
  by auto
  then have (b, g, a', f, r, l') ∈ set (trans_out_map p (L ! p))
  unfolding trans_out_map_def set_map_filter by (auto 4 7)
  with ⟨p < _⟩ ⟨a' < _⟩ ⟨_ ∈ set pairs⟩ show ?thesis
  unfolding Out2_def by (intro in_all_actions_from_vecI)
qed
have distinct (map fst pairs)
  unfolding pairs_def get_committed_def distinct_map inj_on_def Let_def
  by (auto simp: set_map_filter intro!: distinct_map_filterI split: if_split_asm)
have in_pairsD: p < n_ps l = L ! p L ! p ∈ committed (N p)
  if (p, l) ∈ set pairs for p l
  using that using get_committed_mem_iff pairs_def by auto
have In2_D:
  (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p) ∧
  p < n_ps ∧ a' = a1 ∧ L ! p ∈ committed (N p)
  if (p, b, g, a', f, r, l') ∈ set (In2 ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that

```

```

unfolding In2_def
apply –
apply (drule all_actions_from_vecD)
  apply assumption
  apply (rule <distinct _>)
unfolding trans_in_map_def set_map_filter
apply (clarsimp split: option.split_asm)
  apply (auto dest: in_pairsD trans_mapD split: act.split_asm)
done
have Out2_D:
  (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
  ∧ p < n_ps ∧ a' = a1 ∧ L ! p ∈ committed (N p)
  if (p, b, g, a', f, r, l') ∈ set (Out2 ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that
  unfolding Out2_def
  apply –
  apply (drule all_actions_from_vecD)
    apply assumption
    apply (rule <distinct _>)
  unfolding trans_out_map_def set_map_filter
  apply (clarsimp split: option.split_asm)
    apply (auto dest: in_pairsD trans_mapD split: act.split_asm)
  done
from False have **: bin_trans_from (L, s) =
  concat (map ( $\lambda a. \text{pairs\_by\_action } L \ s \ (OUT ! a) \ (In2 ! a)$ ) bin_actions)
  @ concat (map ( $\lambda a. \text{pairs\_by\_action } L \ s \ (Out2 ! a) \ (IN ! a)$ ) bin_actions)
  unfolding bin_trans_from_def IN_def OUT_def In2_def Out2_def pairs_def
  by (simp add: Let_def)
from <dom s = _> <L ∈ _> have
  ((L, s), g, a, r, L', s') ∈ ?S1  $\longleftrightarrow$  (g, a, r, L', s') ∈
  set (concat (map ( $\lambda a. \text{pairs\_by\_action } L \ s \ (OUT ! a) \ (In2 ! a)$ ) bin_actions))
  apply clarsimp
  unfolding pairs_by_action_def
  apply (clarsimp simp: set_map_filter Let_def)
  apply safe
  subgoal for _ s'' _ a' p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'
    apply clarsimp
    apply (inst_existentials a')
    subgoal
      supply [forward4] = In2_I
      apply frules_all
      apply simp
      apply frules_all
      unfolding check_bounded_iff by (intros; solve_triv)
    subgoal
      by (simp add: mem_bin_actions_iff; frules; simp)
    done
  subgoal
    supply [forward2] = In2_D OUT_D
    apply simp
    unfolding mem_bin_actions_iff
    apply (erule conjE)
    apply frules_all

```

```

    apply elims
    apply frules_all
    apply elims
    apply simp
    apply frules_all
    using ⟨bounded bounds s⟩ unfolding check_bounded_iff[symmetric] by (intros; solve_triv)
done
moreover from ⟨dom s = _⟩ ⟨L ∈ _⟩ have
  ((L, s), g, a, r, L', s') ∈ ?S2 ⟷ (g, a, r, L', s')
  ∈ set (concat (map (λa. pairs_by_action L s (Out2 ! a) (IN ! a)) bin_actions))
supply [forward2] = Out2_D In2_D
supply [forward4] = Out2_I
apply clarsimp
unfolding pairs_by_action_def
apply (clarsimp simp: set_map_filter Let_def)
apply safe
subgoal for _ s'' _ a' p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'
  apply clarsimp
  apply (inst_existentials a')
  subgoal
    apply frules_all
    apply simp
    apply frules_all
    unfolding check_bounded_iff by (intros; solve_triv)
  subgoal
    unfolding mem_bin_actions_iff by frules_all simp
  done
subgoal
  apply simp
  unfolding mem_bin_actions_iff
  apply (erule conjE)
  apply frules_all
  apply elims
  apply frules_all
  apply elims
  apply simp
  apply frules_all
  using ⟨bounded bounds s⟩ unfolding check_bounded_iff[symmetric] by (intros; solve_triv)
done
ultimately show ?thesis
  unfolding * ** by simp
qed
qed

```

**Broadcast transitions** lemma make\_combs\_alt\_def:

```

make_combs p a xs ≡
  let
    ys =
      map (λ i. map (λ t. (i, t)) (xs ! i ! a))
      (filter
        (λ i. xs ! i ! a ≠ [] ∧ i ≠ p)
        [0..<n_ps])
  in if ys = [] then [] else product_lists ys

```

```

apply (rule eq_reflection)
unfolding make_combs_def
apply (simp add: map_filter_def comp_def if_distrib[where f = the])
apply intros
apply (fo_rule arg_cong)
apply auto
done

```

```

lemma list_all2_fst_aux:
  map fst xs = ys if list_all2 ( $\lambda x y. \text{fst } x = y$ ) xs ys
  using that by (induction) auto

```

```

lemma broad_trans_from_correct:
  (broad_trans_from, trans_broad)  $\in$  transition_rel states'
  unfolding transition_rel_def
proof (safe del: iffI)
  fix L s g a r L' s' assume (L, s)  $\in$  states'
  then have L  $\in$  states dom s = {0.. $n\_vs$ } bounded bounds s
    by auto
  then have [simp]: length L =  $n\_ps$ 
    by auto
  define IN where IN = map ( $\lambda p. \text{trans\_in\_broad\_grouped } p (L ! p)$ ) [0.. $n\_ps$ ]
  define OUT where OUT = map ( $\lambda p. \text{trans\_out\_broad\_grouped } p (L ! p)$ ) [0.. $n\_ps$ ]
  define IN' where IN' = map (map (filter ( $\lambda (b, \_). \text{bval } (the \ o \ s) \ b$ ))) IN
  define OUT' where OUT' = map (map (filter ( $\lambda (b, \_). \text{bval } (the \ o \ s) \ b$ ))) OUT
  have IN_I:
    ( $b, g, a', f, r, l'$ )  $\in$  set (IN !  $p ! a'$ )
    if (L !  $p, b, g, In \ a', f, r, l'$ )  $\in$  Simple_Network_Language.trans (N p)
       $p < n\_ps \ a' < num\_actions \ a' \in$  set broadcast
    for  $p \ b \ g \ a' \ f \ r \ l'$ 
  proof –
    from trans_mapI[OF that(1,2)]  $\langle a' \in \_ \rangle$  have
      ( $b, g, a', f, r, l'$ )  $\in$  set (trans_in_broad_map  $p (L ! p)$ )
      unfolding trans_in_broad_map_def set_map_filter by (auto 4 7)
    with  $\langle a' < \_ \rangle$  have ( $b, g, a', f, r, l'$ )  $\in$  set (trans_in_broad_grouped  $p (L ! p) ! a'$ )
      unfolding trans_in_broad_grouped_def by (auto dest: in_actions_by_state'I)
    with  $\langle p < \_ \rangle$  show ?thesis
      unfolding IN_def by auto
  qed
  have OUT_I:
    ( $b, g, a', f, r, l'$ )  $\in$  set (OUT !  $p ! a'$ )
    if (L !  $p, b, g, Out \ a', f, r, l'$ )  $\in$  Simple_Network_Language.trans (N p)
       $p < n\_ps \ a' < num\_actions \ a' \in$  set broadcast
    for  $p \ b \ g \ a' \ f \ r \ l'$ 
  proof –
    from trans_mapI[OF that(1,2)]  $\langle a' \in \_ \rangle$  have
      ( $b, g, a', f, r, l'$ )  $\in$  set (trans_out_broad_map  $p (L ! p)$ )
      unfolding trans_out_broad_map_def set_map_filter by (auto 4 7)
    with  $\langle a' < \_ \rangle$  have ( $b, g, a', f, r, l'$ )  $\in$  set (trans_out_broad_grouped  $p (L ! p) ! a'$ )
      unfolding trans_out_broad_grouped_def by (auto dest: in_actions_by_state'I)
    with  $\langle p < \_ \rangle$  show ?thesis
      unfolding OUT_def by auto
  qed
  have IN_D:

```

```

(L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
  ∧ a' = a1 ∧ a1 ∈ set broadcast
if (b, g, a', f, r, l') ∈ set (IN ! p ! a1) a1 < num_actions p < n_ps
for p b g a' f r l' a1
using that unfolding IN_def trans_in_broad_grouped_def
apply simp
apply (drule in_actions_by_state'D)
unfolding trans_in_broad_map_def set_map_filter
by (auto split: option.split_asm) (auto split: act.split_asm if_split_asm dest: trans_mapD)
have [simp]: length IN = n_ps length OUT = n_ps
  unfolding IN_def OUT_def by simp+
have [simp]: length (IN ! p) = num_actions length (OUT ! p) = num_actions if p < n_ps for
p
  using that by (simp add:
    length_actions_by_state'_preserv trans_in_broad_grouped_def trans_out_broad_grouped_def
    IN_def OUT_def)+
have OUT_D:
  (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    ∧ a' = a1 ∧ a1 ∈ set broadcast
  if (b, g, a', f, r, l') ∈ set (OUT ! p ! a1) a1 < num_actions p < n_ps
  for p b g a' f r l' a1
  using that unfolding OUT_def trans_out_broad_grouped_def
  apply simp
  apply (drule in_actions_by_state'D)
  unfolding trans_out_broad_map_def set_map_filter
  by (auto split: option.split_asm) (auto split: act.split_asm if_split_asm dest: trans_mapD)
have IN'_I:
  (b, g, a', f, r, l') ∈ set (IN' ! p ! a')
  if (b, g, a', f, r, l') ∈ set (IN ! p ! a') p < n_ps a' < num_actions
    check_bexp s b True for p b g a' f r l'
  using that ⟨dom s = {0..nvs}⟩ unfolding IN'_def
  by (auto dest!: IN_D[THEN conjunct1] elim: check_bexp_bvalI)
have IN'_D:
  (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    ∧ a' = a1 ∧ a1 ∈ set broadcast ∧ check_bexp s b True
  if (b, g, a', f, r, l') ∈ set (IN' ! p ! a1) a1 < num_actions p < n_ps
  for p b g a' f r l' a1
  using that ⟨dom s = {0..nvs}⟩ unfolding IN'_def by (auto dest!: IN_D elim: check_bexp_bvalD)
have OUT'_I:
  (b, g, a', f, r, l') ∈ set (OUT' ! p ! a')
  if (b, g, a', f, r, l') ∈ set (OUT ! p ! a') p < n_ps a' < num_actions
    check_bexp s b True for p b g a' f r l'
  using that ⟨dom s = {0..nvs}⟩ unfolding OUT'_def
  by (auto dest!: OUT_D[THEN conjunct1] elim: check_bexp_bvalI)
have OUT'_D:
  (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    ∧ a' = a1 ∧ a1 ∈ set broadcast ∧ check_bexp s b True
  if (b, g, a', f, r, l') ∈ set (OUT' ! p ! a1) a1 < num_actions p < n_ps
  for p b g a' f r l' a1
  using that ⟨dom s = {0..nvs}⟩ unfolding OUT'_def by (auto dest!: OUT_D elim:
check_bexp_bvalD)
define make_trans where make_trans a p ≡
  let
    outs = OUT' ! p ! a

```

```

in if outs = [] then []
else
  let
    combs = make_combs p a IN';
    outs = map (λt. (p, t)) outs;
    combs = (
      if combs = [] then [[x]. x ← outs]
      else concat (map (λx. map (λxs. x # xs) combs) outs));
    init = ([], Broad a, [], (L, s))
  in
    filter (λ (g, a, r, L, s). check_bounded s) (
      map (λcomb.
        fold
          (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
            (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
          )
          comb
          init
        ) combs) for a p
{
  fix p ps bs gs a' fs rs ls'
  assume assms:
    ∀ p ∈ set ps.
      (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)
    ∀ q < n_ps. q ∉ set ps ∧ p ≠ q →
      (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)
    p < n_ps set ps ⊆ {0..<n_ps} p ∉ set ps distinct ps sorted ps
    a' < num_actions a' ∈ set broadcast ∀ p ∈ set ps. check_bexp s (bs p) True
  define ys where ys = List.map_filter
    (λ i.
      if i = p then None
      else if IN' ! i ! a' = [] then None
      else Some (map (λt. (i, t)) (IN' ! i ! a'))
    )
    [0..<n_ps]
  have filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
  apply (rule filter_distinct_eqI)
  subgoal
    using assms(4-) by (simp add: sorted_distinct_subseq_iff)
  subgoal
    using assms(1,3-) by (auto 4 3 dest!: IN_I IN'_I)
  subgoal
    using assms(1,2,4,8-)
    apply -
    apply (rule ccontr)
    apply simp
    apply elims
    apply (drule hd_in_set)
    subgoal for x
      by (cases hd (IN' ! x ! a')) (fastforce dest!: IN'_D)
    done
  by auto
then have length ys = length ps
  unfolding ys_def map_filter_def by simp

```

```

have non_empty:  $ys \neq []$  if  $ps \neq []$ 
  using  $\langle \_ = ps \rangle \langle ps \neq [] \rangle$  unfolding  $ys\_def$   $map\_filter\_def$  by simp
have make_combsD:
  map  $(\lambda p. (p, bs\ p, gs\ p, a', fs\ p, rs\ p, ls'\ p))$   $ps \in set\ (make\_combs\ p\ a'\ IN')$ 
  if  $ps \neq []$ 
proof -
  from  $assms(1,3-)$  have
     $\forall i < length\ ps. let\ p = ps\ !\ i\ in\ (p, bs\ p, gs\ p, a', fs\ p, rs\ p, ls'\ p) \in set\ (ys\ !\ i)$ 
    unfolding  $ys\_def$   $Let\_def$   $map\_filter\_def$ 
    apply (simp add:  $comp\_def$   $if\_distrib$  [where  $f = the$ ])
    apply (subst (2)  $map\_cong$ )
    apply (rule  $HOL.refl$ )
    apply (simp; fail)
    apply (simp add:  $\langle \_ = ps \rangle$ )
    by (intros add:  $image\_eqI$  [OF  $HOL.refl$ ]  $IN\_I\ IN'\_I$ ; simp add:  $subset\_code(1)$ )
  with non_empty [OF that] show ?thesis
    unfolding  $make\_combs\_def\ ys\_def$  [symmetric]  $Let\_def$ 
    by (auto simp:  $\langle length\ ys = \_ \rangle$   $product\_lists\_set$  intro:  $list\_all2\_all\_nthI$ )
qed
have make_combs_empty:
   $make\_combs\ p\ a'\ IN' = [] \longleftrightarrow ps = []$ 
proof (cases  $ps = []$ )
  case True
  then show ?thesis
    using  $\langle length\ ys = length\ ps \rangle$  unfolding  $make\_combs\_def\ ys\_def$  [symmetric]  $Let\_def$  by
auto
  next
  case False
  then show ?thesis
    using make_combsD by auto
qed
note make_combsD make_combs_empty
} note make_combsD = this(1) and make_combs_empty = this(2)
have make_combs_emptyD:  $filter\ (\lambda i. IN' ! i ! a' \neq [] \wedge i \neq p)\ [0..<n\_ps] = []$ 
  if  $make\_combs\ p\ a'\ IN' = []$  for  $p\ a'$ 
  apply (rule  $filter\_distinct\_eqI$ )
  subgoal
    by auto
  subgoal
    by auto
  subgoal
    using that unfolding  $make\_combs\_alt\_def$ 
    by (auto simp:  $filter\_empty\_conv\ product\_lists\_empty\ split: if\_split\_asm$ )
  subgoal
    by simp
  done
have make_combsI:
 $\exists ps\ bs\ gs\ fs\ rs\ ls'.$ 
 $(\forall p \in set\ ps.$ 
 $(L\ !\ p, bs\ p, gs\ p, In\ a', fs\ p, rs\ p, ls'\ p) \in trans\ (N\ p)) \wedge$ 
 $(\forall q < n\_ps. q \notin set\ ps \wedge p \neq q \longrightarrow$ 
 $(\forall b\ g\ f\ r\ l'. (L\ !\ q, b, g, In\ a', f, r, l') \notin trans\ (N\ q) \vee \neg check\_bexp\ s\ b\ True)) \wedge$ 
 $set\ ps \subseteq \{0..<n\_ps\} \wedge p \notin set\ ps \wedge distinct\ ps \wedge sorted\ ps \wedge ps \neq [] \wedge a' \in set\ broadcast$ 
 $\wedge (\forall p \in set\ ps. check\_bexp\ s\ (bs\ p)\ True)$ 

```

```

    ∧ xs = map (λ p. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps
    ∧ filter (λ i. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
  if xs ∈ set (make_combs p a' IN') p < n_ps a' < num_actions
  for xs p a'
proof -
  define ps bs gs fs rs ls' where defs:
    ps = map fst xs
    bs = (λ i. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ b)
    gs = (λ i. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ g)
    fs = (λ i. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ f)
    rs = (λ i. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ r)
    ls' = (λ i. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ l')
  have filter (λ i. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
  apply (rule filter_distinct_eqI)
  subgoal
    using that
    unfolding defs make_combs_alt_def Let_def
    by (auto simp: set_map_filter product_lists_set list_all2_map2 list.rel_eq
        dest: list_all2_map_fst_aux split: if_split_asm)
  subgoal
    using that unfolding defs make_combs_alt_def Let_def
    by (auto simp: set_map_filter product_lists_set dest!: list_all2_set1 split: if_split_asm)
  subgoal
    using that
    unfolding defs make_combs_alt_def Let_def
    by (auto
        simp: set_map_filter product_lists_set list_all2_map2 list.rel_eq
        dest!: map_eq_imageD list_all2_map_fst_aux split: if_split_asm)
  subgoal
    by simp
  done
  then have set ps ⊆ {0..<n_ps} p ∉ set ps distinct ps sorted ps
  by (auto intro: sorted_filter')
  have to_map: a' = a the (map_of xs q) = (b, g, a, r, f, l')
  if (q, b, g, a, r, f, l') ∈ set xs for b q g a r f l'
  using that
  apply -
  subgoal
    using ⟨set ps ⊆ ⟩ ⟨a' < ⟩ ⟨xs ∈ ⟩
    by
      (simp
        add: make_combs_alt_def product_lists_set ⟨_ = ps⟩ list_all2_map2
        split: if_split_asm
        ) (auto 4 3 dest!: IN'_D list_all2_set1)
  subgoal
    using ⟨distinct ps⟩
    by (subst (asm) map_of_eq_Some_iff[of xs q, symmetric]) (auto simp: defs)
  done
  from that have *: ∀ p ∈ set ps.
    (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ Simple_Network_Language.trans (N p)
  unfolding make_combs_alt_def ⟨_ = ps⟩
  apply (clarsimp simp: set_map_filter product_lists_set split: if_split_asm)
  using ⟨set ps ⊆ ⟩ unfolding defs
  by (auto simp: comp_def list_all2_map2 list_all2_same to_map

```

```

    elim!: IN'_D[THEN conjunct1, rotated]
  )
from that have ps ≠ [] a' ∈ set broadcast
apply (simp_all add: make_combs_alt_def set_map_filter product_lists_set split: if_split_asm)
  using ⟨_ = ps⟩ ⟨set ps ⊆ {0..<n_ps}⟩
  by (cases xs; auto dest: IN'_D simp: list_all2_Cons1)+
with that have ∀ q < n_ps. q ∉ set ps ∧ p ≠ q →
  (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)
  unfolding make_combs_alt_def
  by (auto 4 3 dest!: list_all2_set2 IN_I IN'_I
      simp: defs set_map_filter product_lists_set split: if_split_asm)
have xs = map (λ p. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps
  apply (intro nth_equalityI)
  apply (simp add: defs; fail)
  subgoal for i
    by (cases xs ! i) (auto 4 4 simp: defs dest: to_map_nth_mem)
  done
have ∀ p ∈ set ps. check_bexp s (bs p) True
proof
  fix q assume q ∈ set ps
  from ⟨xs ∈ _⟩ have distinct (map fst xs)
    unfolding make_combs_alt_def
    by (auto simp: product_lists_set list_all2_map2 to_map
        dest!: list_all2_fst_aux[OF list_all2_mono]
        split: if_split_asm)
  from ⟨q ∈ set ps⟩ ⟨_ = ps⟩ have IN' ! q ! a' ≠ [] q ≠ p q < n_ps
    by auto
  with that have the (map_of xs q) ∈ set (IN' ! q ! a')
    using set_map_of_compr[OF ⟨distinct (map _ _)⟩] unfolding make_combs_alt_def
    apply (clarsimp simp: set_map_filter product_lists_set split: if_split_asm)
    apply (drule list_all2_set2)
    apply auto
    done
  then obtain a where (bs q, gs q, a, fs q, rs q, ls' q) ∈ set (IN' ! q ! a')
    unfolding defs by atomize_elim (auto split!: prod.splits)
  then show check_bexp s (bs q) True
    using IN'_D ⟨a' < _⟩ ⟨set ps ⊆ _⟩ ⟨q ∈ set ps⟩ by auto
qed
show ?thesis
  by (inst_existentials ps bs gs fs rs ls'; fact)
qed
let ?f = λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, L, s).
  (g1 @ g2, a, r1 @ r2, L[q := l2], mk_upds s f2)
have ***:
  fold ?f (map (λp. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps) (g, a, r, L, s)
  = (g @ concat (map gs ps), a, r @ concat (map rs ps),
    fold (λp L. L[p := ls' p]) ps L, mk_upds s (concat_map fs ps))
  for ps bs gs a' fs rs ls' g a r L s
  by (induction ps arbitrary: g a r L s; simp add: mk_upds_def)
have upd_swap:
  (fold (λp L. L[p := ls' p]) ps L)[p := l'] = fold (λp L. L[p := ls' p]) ps (L[p := l'])
  if p ∉ set ps for ps ls' p l'
  using that by (induction ps arbitrary: L) (auto simp: list_update_swap)
have make_transI:

```

```

a' < num_actions ∧ p < n_ps ∧
  (g1 @ concat (map gs ps), Broad a', r1 @ concat (map rs ps),
   (fold (λp L. L[p := ls' p]) ps L)[p := l'], s') ∈ set (make_trans a' p)
if
  L ∈ states and
  g = g1 @ concat (map gs ps) and
  a = Broad a' and
  r = r1 @ concat (map rs ps) and
  L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] and
  a' ∈ set broadcast and
  (L ! p, b1, g1, Out a', f1, r1, l') ∈ trans (N p) and
  ∀ p ∈ set ps. (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)
and
  ∀ q < n_ps. q ∉ set ps ∧ p ≠ q
    → (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True) and
  p < n_ps and
  set ps ⊆ {0..<n_ps} and
  p ∉ set ps and
  distinct ps and
  sorted ps and
  check_bexp s b1 True and
  ∀ p ∈ set ps. check_bexp s (bs p) True and
  is_upds s f1 s'' and
  is_upds s'' (concat_map fs ps) s' and
  Simple_Network_Language.bounded bounds s'
for a' p b1 g1 bs gs ps r1 rs ls' l' f1 s'' fs
using that ⟨dom s = {0..<n_vs}⟩
unfolding make_trans_def
apply (clarsimp simp: set_map_filter Let_def split: prod.split)
supply [forward2] = action_setD
supply [forward4] = is_upds_dom2 is_upds_dom3 is_upds_concatD[rotated 3] OUT_I
OUT'_I
apply frule2
apply simp
apply (rule conjI, rule impI)
subgoal
  apply (subst (asm) make_combs_empty, (assumption | simp)+)
  apply frules_all
  apply (intro conjI)
  apply (solves auto)
  apply (intros add: more_intros)
  apply (solve_triv | intros add: more_intros UN_I)+
subgoal
  unfolding comp_def by (auto elim!: is_upds.cases)
  by (auto simp: check_bounded_iff[symmetric] elim: is_upds_NilE)
apply (rule impI)
apply (frule make_combsD, simp, assumption+)
subgoal
  by (subst (asm) make_combs_empty) (assumption | simp)+
apply frules_all
apply simp
apply (intro conjI)
  apply (solves auto)
  apply (intros add: more_intros)

```

```

    apply (solve_triv | intros add: more_intros UN_I)+
    apply (simp add: *** upd_swap; fail)
    unfolding check_bounded_iff[symmetric] .
have make_transD:
   $\exists s'' b ga f ra l' bs gs fs rs ls' ps.$ 
     $g = ga @ concat (map gs ps) \wedge$ 
     $a = Broad a' \wedge$ 
     $r = ra @ concat (map rs ps) \wedge$ 
     $L' = (fold (\lambda p L. L[p := ls' p]) ps L)[p := l'] \wedge$ 
     $a' \in set broadcast \wedge$ 
     $(L ! p, b, ga, Out a', f, ra, l') \in trans (N p) \wedge$ 
     $(\forall p \in set ps. (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) \in trans (N p)) \wedge$ 
     $(\forall q < n\_ps.$ 
       $q \notin set ps \wedge p \neq q \longrightarrow$ 
       $(\forall b g f r l'. (L ! q, b, g, In a', f, r, l') \notin trans (N q) \vee \neg check\_bexp s b True)) \wedge$ 
     $p < n\_ps \wedge$ 
     $set ps \subseteq \{0..<n\_ps\} \wedge$ 
     $p \notin set ps \wedge$ 
     $distinct ps \wedge$ 
     $sorted ps \wedge$ 
     $check\_bexp s b True \wedge (\forall p \in set ps. check\_bexp s (bs p) True) \wedge$ 
     $is\_upds s f s'' \wedge is\_upds s'' (concat\_map fs ps) s' \wedge$ 
     $bounded bounds s' \wedge$ 
     $filter (\lambda i. IN' ! i ! a' \neq [] \wedge i \neq p) [0..<n\_ps] = ps$ 
  if
     $L \in states$  and
     $a' < num\_actions$  and
     $p < n\_ps$  and
     $(g, a, r, L', s') \in set (make\_trans a' p)$ 
  for  $a' p$ 
  supply [forward2] = action_setD
  supply [forward3] = is_upds_make_updsI3[rotated] OUT'_D
  supply [forward4] = is_upds_dom2 is_upds_dom3 is_upds_concatD[rotated 3] OUT_I
OUT'_I
  using that  $\langle dom s = \{0..<n\_vs\} \rangle$ 
  unfolding make_trans_def
  apply mini_ex
  apply (clarsimp simp: set_map_filter Let_def split: prod.split if_split_asm)
  subgoal for  $b a1 f l'$ 
    apply (inst_existentials
       $b :: (nat, int) bexp$ 
       $g :: (nat, int) acconstraint list$ 
       $f :: (nat \times (nat, int) exp) list$ 
       $r :: nat list$ 
       $l'$ 
       $undefined :: nat \Rightarrow (nat, int) bexp$ 
       $undefined :: nat \Rightarrow (nat, int) acconstraint list$ 
       $undefined :: nat \Rightarrow (nat \times (nat, int) exp) list$ 
       $undefined :: nat \Rightarrow nat list$ 
       $undefined :: nat \Rightarrow nat$ 
       $[] :: nat list$ )
    apply (solves  $\langle auto dest: OUT'_D \rangle$ ) +
  subgoal
    by (auto 4 3 simp: filter_empty_conv dest: bspec dest!: make_combs_emptyD OUT'_D

```

```

IN_I IN'_I)
  apply (solves ⟨auto dest: OUT'_D⟩)+
subgoal
  apply (inst_existentials s')
  subgoal is_upd
    by (auto intro: is_upds_make_updsI2 dest: OUT'_D)
  subgoal
    by simp (rule is_upds.intros)
  subgoal
    by (subst check_bounded_iff) (metis OUT'_D is_upds_dom2 is_upd)+
  subgoal
    by (rule make_combs_emptyD)
  done
done
subgoal for b1 g1 a1 r1 f1 l1' xs
  apply (drule make_combsI, assumption+)
  apply frules
  apply elims
  apply dedup_premis
  apply frules_all
  apply (simp add: *** )
  apply intros
    apply solve_triv+
    apply (erule upd_swap[symmetric]; fail)
    apply solve_triv+
    apply (erule bspec; assumption)
    apply (elims add: allE; intros?; assumption)
    apply solve_triv+
  subgoal for ps gs fs rs ls'
    apply (subst check_bounded_iff)
    subgoal
      apply (subst is_upds_dom3)
      apply (simp add: fold_map_comp_def; fail)
      apply assumption+
    done
    subgoal
      by simp
    done
  apply solve_triv
done
done
have make_trans_iff:
  (∃ s'' aa p b ga f ra l' bs gs fs rs ls' ps.
    g = ga @ concat (map gs ps) ∧
    a = Broad aa ∧
    r = ra @ concat (map rs ps) ∧
    L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] ∧
    aa ∈ set broadcast ∧
    (L ! p, b, ga, Out aa, f, ra, l') ∈ Simple_Network_Language.trans (N p) ∧
    (∀ p ∈ set ps.
      (L ! p, bs p, gs p, In aa, fs p, rs p, ls' p)
      ∈ Simple_Network_Language.trans (N p)) ∧
    (∀ q < n_ps.
      q ∉ set ps ∧ p ≠ q ⟶

```

```

      (∀ b g f r l'.
        (L ! q, b, g, In aa, f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)) ∧
      p < n_ps ∧
      set ps ⊆ {0..<n_ps} ∧
      p ∉ set ps ∧
      distinct ps ∧
      sorted ps ∧
      check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
      is_upds s f s'' ∧
      is_upds s'' (concat_map fs ps) s' ∧
      bounded bounds s') =
    (∃ a' ∈ {0..<num_actions}.
      ∃ p ∈ {0..<n_ps}. (g, a, r, L', s') ∈ set (make_trans a' p)) (is ?l ⟷ ?r)
  if dom s = {0..<n_vs} L ∈ states
proof (intro iffI)
  assume ?l
  with that show ?r
    by elims (drule make_transI; (elim; intros)?; solve_triv)
next
  assume ?r
  with that show ?l
    apply elims
    subgoal for a' p
      apply simp
      apply (drule make_transD)
      apply assumption+
      apply elims
      apply intros
      apply assumption+
      apply blast+
    done
  done
qed
show (((L, s), g, a, r, L', s') ∈ trans_broad) =
  ((g, a, r, L', s') ∈ set (broad_trans_from (L, s)))
proof (cases get_committed L = [])
  case True
  with get_committed_empty_iff[of L] have ∀ p < n_ps. L ! p ∉ committed (N p)
  by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_broad ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L', s')) |
      L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
      a ∈ set broadcast ∧
      (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
      (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
      (∀ q < n_ps. q ∉ set ps ∧ p ≠ q ⟶
        ¬ (∃ b g f r l'. (L ! q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
      L ! p = l ∧
      p < length L ∧ set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
      check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
      L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] ∧
      is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
      L ∈ states ∧ bounded bounds s ∧ bounded bounds s''}
}

```

```

unfolding trans_broad_def broadcast_def[simplified]
by (intro iffI; elims add: CollectE; intros add: CollectI) blast+
from True have **:
  broad_trans_from (L, s)
  = concat (
    map (λa.
      concat (map (λp. make_trans a p) [0..n_ps])
    )
    [0..num_actions]
  )
unfolding broad_trans_from_alt_def IN_def OUT_def IN'_def OUT'_def Let_def make_trans_def
by simp
from ⟨dom s = ⟩ ⟨L ∈ ⟩ ⟨bounded bounds s⟩ show ?thesis
unfolding * **
apply simp
apply (subst make_trans_iff[symmetric])
apply simp+
apply (intro iffI; elims; intros)
apply (solve_triv | blast)+
done
next
case False
with get_committed_empty_iff[of L] have ¬ (∀ p < n_ps. L ! p ∉ committed (N p))
by simp
then have *: ((L, s), g, a, r, L', s') ∈ trans_broad ⟷ ((L, s), g, a, r, L', s') ∈
  {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L', s')) |
    L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
    a ∈ set broadcast ∧
    (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
    (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
    (l ∈ committed (N p) ∨ (∃ p ∈ set ps. L ! p ∈ committed (N p))) ∧
    (∀ q < n_ps. q ∉ set ps ∧ p ≠ q ⟶
      ¬ (∃ b g f r l'. (L ! q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
    L!p = l ∧
    p < length L ∧ set ps ⊆ {0..n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
    check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
    L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'] ∧
    is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
    L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
  }
unfolding trans_broad_def broadcast_def[simplified]
by (intro iffI; elims add: CollectE; intros add: CollectI) blast+
have committed_iff:
  List.map_filter (λ(p, _). if IN' ! p ! a' = [] then None else Some p) (get_committed L) ≠
  [p] ∧
  List.map_filter (λ(p, _). if IN' ! p ! a' = [] then None else Some p) (get_committed L) ≠ []
  ⟷ (∃ q < n_ps. IN' ! q ! a' ≠ [] ∧ q ≠ p ∧ L ! q ∈ committed (N q))
  for p a'
proof -
have *: xs ≠ [p] ∧ xs ≠ [] ⟷ (∃ x ∈ set xs. x ≠ p) if distinct xs for xs
using that by auto (metis distinct.simps(2) distinct_length_2_or_more revg.elims)
show ?thesis
by (subst *)

```

```

    (auto
      intro: distinct_map_filterI get_committed_distinct
      simp: set_map_filter get_committed_mem_iff split: if_split_asm
    )
qed
from False have **:
  broad_trans_from (L, s)
  = concat (
    map (λa.
      let
        ins_committed =
          List.map_filter (λ(p, _). if IN' ! p ! a ≠ [] then Some p else None) (get_committed L)
      in
      concat (map (λp.
        if
          (ins_committed = [p] ∨ ins_committed = [])
          ∧ ¬ list_ex (λ (q, _). q = p) (get_committed L)
        then []
        else
          make_trans a p
        )
      ) [0..<n_ps])
    ) [0..<num_actions])

unfolding broad_trans_from_alt_def IN_def OUT_def IN'_def OUT'_def make_trans_def
unfolding Let_def if_contract
apply simp
apply (fo_rule if_cong arg_cong2[where f = map] arg_cong[where f = concat] | rule
ext)+
  apply blast+
done
from ⟨dom s = _⟩ ⟨L ∈ _⟩ ⟨bounded bounds s⟩ show ?thesis
unfolding * **
apply (simp add: make_trans_iff)
apply (intro iffI; elims)
subgoal for s'a aa p ga f ra l' gs fs rs ls' ps — ?l → ?r
  apply (frule make_transI)
  apply (assumption | blast)+
apply elims
apply intros
  apply (simp; fail)
apply (simp add: Let_def)
apply (erule disjE)
subgoal — The process with the outgoing action label is committed
  using get_committed_mem_iff[of p L ! p L, simplified, symmetric]
  by (inst_existentials p) (auto simp add: list_ex_iff)
apply (erule bexE)
subgoal for q — One of the processes with an ingoing action label is committed
  apply (inst_existentials p)
  apply assumption
  apply (rule IntI)
  apply (simp; fail)
apply simp

```

```

    unfolding committed_iff
    apply (rule disjI1; inst_existentials q; force dest!: IN_I IN'_I)
  done
done
subgoal for  $a' \text{---} ?r \longrightarrow ?l$ 
  apply (clarsimp split: if_split_asm simp: Let_def)
  apply (drule make_transD[rotated 4])
    apply assumption+
  apply elims
  apply intros
    apply assumption+
    apply (erule bspec; assumption)
  subgoal for  $p \ s'' \ g' \ f \ r' \ l' \ gs \ fs \ rs \ ls' \ ps$ 
    unfolding committed_iff by (auto simp: get_committed_mem_iff list_ex_iff)
  apply blast+
done
done
qed
qed

```

**Refinement of the State Implementation** **definition**  $state\_rel :: (nat \rightarrow int) \Rightarrow int\ list \Rightarrow bool$

**where**  
 $state\_rel\ s\ xs \equiv length\ xs = n\_vs \wedge dom\ s = \{0..<n\_vs\} \wedge (\forall i < n\_vs. xs\ !\ i = the\ (s\ i))$

**definition**  $loc\_rel$  **where**

$loc\_rel \equiv \{((L', s'), (L, s)) \mid L\ s\ L'\ s'. L' = L \wedge length\ L = n\_ps \wedge state\_rel\ s\ s'\}$

**lemma**  $state\_impl\_abstract$ :

$\exists L\ s. ((Li, si), (L, s)) \in loc\_rel$  **if**  $length\ Li = n\_ps\ length\ si = n\_vs$   
**using** *that* **unfolding**  $loc\_rel\_def\ state\_rel\_def$   
**by**  $(inst\_existentials\ Li\ \lambda i. if\ i < n\_vs\ then\ Some\ (si\ !\ i)\ else\ None)(auto\ split: if\_split\_asm)$

**lemma**  $state\_rel\_left\_unique$ :

$l \in states' \implies (li, l) \in loc\_rel \implies (li', l) \in loc\_rel \implies li' = li$   
**unfolding**  $loc\_rel\_def\ state\_rel\_def$  **by**  $(auto\ intro: nth\_equalityI)$

**lemma**  $state\_rel\_right\_unique$ :

$l \in states' \implies l' \in states' \implies (li, l) \in loc\_rel \implies (li, l') \in loc\_rel \implies l' = l$   
**unfolding**  $loc\_rel\_def\ state\_rel\_def$

**apply**  $clarsimp$

**apply**  $(rule\ ext)$

**subgoal** **premises**  $prems$  **for**  $L\ s\ s'1\ s'2\ x$

**proof**  $-$

**show**  $s'1\ x = s\ x$

**proof**  $(cases\ x < n\_vs)$

**case**  $True$

**then** **have**  $x \in dom\ s'1\ x \in dom\ s$

**using**  $prems$  **by**  $auto$

**with**  $\langle x < n\_vs \rangle$  **show**  $?thesis$

**using**  $prems(9)$  **by**  $auto$

**next**

**case**  $False$

```

    then have  $x \notin \text{dom } s'1$   $x \notin \text{dom } s$ 
      using prems by auto
    then show ?thesis
      by (auto simp: dom_def)
  qed
qed
done

end

end

fun bvali ::  $\_ \Rightarrow (\text{nat}, 'b :: \text{linorder}) \text{ bexp} \Rightarrow \text{bool}$  and evali where
  bvali s bexp.true = True |
  bvali s (not e)  $\longleftrightarrow \neg \text{bvali } s \text{ } e$  |
  bvali s (and e1 e2)  $\longleftrightarrow \text{bvali } s \text{ } e1 \wedge \text{bvali } s \text{ } e2$  |
  bvali s (bexp.or e1 e2)  $\longleftrightarrow \text{bvali } s \text{ } e1 \vee \text{bvali } s \text{ } e2$  |
  bvali s (imply e1 e2)  $\longleftrightarrow \text{bvali } s \text{ } e1 \longrightarrow \text{bvali } s \text{ } e2$  |
  bvali s (eq i x)  $\longleftrightarrow \text{evali } s \text{ } i = \text{evali } s \text{ } x$  |
  bvali s (le i x)  $\longleftrightarrow \text{evali } s \text{ } i \leq \text{evali } s \text{ } x$  |
  bvali s (lt i x)  $\longleftrightarrow \text{evali } s \text{ } i < \text{evali } s \text{ } x$  |
  bvali s (ge i x)  $\longleftrightarrow \text{evali } s \text{ } i \geq \text{evali } s \text{ } x$  |
  bvali s (gt i x)  $\longleftrightarrow \text{evali } s \text{ } i > \text{evali } s \text{ } x$ 
| evali s (const c) = c
| evali s (var x) = s ! x
| evali s (if_then_else b e1 e2) = (if bvali s b then evali s e1 else evali s e2)
| evali s (binop f e1 e2) = f (evali s e1) (evali s e2)
| evali s (unop f e) = f (evali s e)

definition mk_updsi ::
   $\text{int list} \Rightarrow (\text{nat} \times (\text{nat}, \text{int}) \text{ exp}) \text{ list} \Rightarrow \text{int list}$  where
  mk_updsi s upds = fold ( $\lambda(x, \text{upd}) \text{ s. } s[x := \text{evali } s \text{ upd}]$ ) upds s

context Simple_Network_Impl_nat_defs
begin

definition
  check_boundedi s =
    ( $\forall x < \text{length } s. \text{fst } (\text{bounds\_map } x) \leq s ! x \wedge s ! x \leq \text{snd } (\text{bounds\_map } x)$ )

definition
  states'_memi  $\equiv \lambda(L, s). L \in \text{states} \wedge \text{length } s = n\_vs \wedge \text{check\_boundedi } s$ 

definition
  int_trans_from_loc_impl p l L s  $\equiv$ 
    let trans = trans_i_map p l
    in
    List.map_filter ( $\lambda(b, g, a, f, r, l').$ 
      let s' = mk_updsi s f in
        if bvali s b  $\wedge \text{check\_boundedi } s'$  then Some (g, Internal a, r, (L[p := l'], s')
        else None
    ) trans

definition

```

```

int_trans_from_vec_impl pairs L s ≡
  concat (map (λ(p, l). int_trans_from_loc_impl p l L s) pairs)

```

**definition**

```

int_trans_from_all_impl L s ≡
  concat (map (λp. int_trans_from_loc_impl p (L ! p) L s) [0..<n_ps])

```

**definition**

```

int_trans_impl ≡ λ (L, s).
  let pairs = get_committed L in
  if pairs = []
  then int_trans_from_all_impl L s
  else int_trans_from_vec_impl pairs L s

```

**definition**

```

pairs_by_action_impl L s OUT IN ≡
  concat (
    map (λ (p, b1, g1, a1, f1, r1, l1).
      List.map_filter (λ (q, b2, g2, a2, f2, r2, l2).
        if p = q then None else
        let s' = mk_updsi (mk_updsi s f1) f2 in
        if bvali s b1 ∧ bvali s b2 ∧ check_boundedi s'
        then Some (g1 @ g2, Bin a1, r1 @ r2, (L[p := l1, q := l2], s'))
        else None
      ) OUT) IN)

```

**definition**

```

bin_trans_from_impl ≡ λ(L, s).
  let
    pairs = get_committed L;
    In = all_actions_by_state trans_in_map L;
    Out = all_actions_by_state trans_out_map L
  in
  if pairs = [] then
    concat (map (λa. pairs_by_action_impl L s (Out ! a) (In ! a)) bin_actions)
  else
    let
      In2 = all_actions_from_vec trans_in_map pairs;
      Out2 = all_actions_from_vec trans_out_map pairs
    in
    concat (map (λa. pairs_by_action_impl L s (Out ! a) (In2 ! a)) bin_actions)
    @ concat (map (λa. pairs_by_action_impl L s (Out2 ! a) (In ! a)) bin_actions)

```

**definition**

```

compute_upds_init ≡ List.map_filter (λcomb.
  let (g, a, r, L', s) =
    fold
      (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
        (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
      )
    comb

```

```

    init
    in if check_bounded s then Some (g, a, r, L', s) else None
)

```

**definition**

```

compute_upds_impl init ≡ List.map_filter (λcomb.
let (g, a, r, L', s) =
  fold
    (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
      (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_updsi s f2))
    )
    comb
    init
  in if check_boundedi s then Some (g, a, r, L', s) else None
)

```

**definition** *trans\_from* **where**

```

trans_from st = int_trans_from st @ bin_trans_from st @ broad_trans_from st

```

**definition**

```

broad_trans_from_impl ≡ λ(L, s).
let
  pairs = get_committed L;
  In = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
  Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps];
  In = map (map (filter (λ(b, _). bvali s b))) In;
  Out = map (map (filter (λ(b, _). bvali s b))) Out
in
  if pairs = [] then
    concat (
      map (λa.
        concat (map (λp.
          let
            outs = Out ! p ! a
            in if outs = [] then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λx. map (λxs. x # xs) combs) outs));
              init = ([], Broad a, [], (L, s))
            in
              compute_upds_impl init combs
          )
        ) [0..<n_ps])
    )
  else
    concat (
      map (λa.
        let
          ins_committed =

```

```

    List.map_filter (λ(p, _). if In ! p ! a ≠ [] then Some p else None) pairs;
    always_committed = (length ins_committed > 1)
  in
  concat (map (λp.
    let
      outs = Out ! p ! a
    in if outs = [] then []
    else if
      ¬ always_committed ∧ (ins_committed = [p] ∨ ins_committed = [])
      ∧ ¬ list_ex (λ (q, _). q = p) pairs
    then []
    else
      let
        combs = make_combs p a In;
        outs = map (λt. (p, t)) outs;
        combs = (
          if combs = [] then [[x]. x ← outs]
          else concat (map (λx. map (λxs. x # xs) combs) outs));
        init = ([], Broad a, [], (L, s))
      in
        compute_upds_impl init combs
    )
  ) [0..<n_ps])
) [0..<num_actions])

```

**definition** *trans\_impl* **where**

*trans\_impl st = int\_trans\_impl st @ bin\_trans\_from\_impl st @ broad\_trans\_from\_impl st*

**end**

**context** *Simple\_Network\_Impl\_nat*

**begin**

**lemma** *bval\_bval*:

*state\_rel s si*  $\impl \forall x \in \text{vars\_of\_bexp } b. x \in \text{dom } s \impl \text{bval (the o s) } b = \text{bvali si } b$

**and** *eval\_evali*:

*state\_rel s si*  $\impl \forall x \in \text{vars\_of\_exp } e. x \in \text{dom } s \impl \text{eval (the o s) } e = \text{evali si } e$   
**by** (*induction b and e*) (*auto simp: state\_rel\_def*)

**lemma** *mk\_upds\_mk\_updsi*:

*state\_rel (mk\_upds s upds) (mk\_updsi si upds)*

**if** *assms*: *state\_rel s si*  $\forall (\_, e) \in \text{set upds}. \forall x \in \text{vars\_of\_exp } e. x < n\_vs$   
 $\forall (x, e) \in \text{set upds}. x < n\_vs$

**proof** –

**have** *upd\_stepI*: *state\_rel (mk\_upd (x, e) s')* (*si'*[*x* := *evali si' e*])

**if** *state\_rel s' si'*  $\forall x \in \text{vars\_of\_exp } e. x < n\_vs$   $x < n\_vs$

**for** *s' si' x e*

**using** *that assms unfolding mk\_upd\_def state\_rel\_def* **by** (*auto simp: state\_rel\_def eval\_evali*)

**from** *assms* **show** *?thesis*

**proof** (*induction upds arbitrary: s si*)

**case** *Nil*

```

    then show ?case
      by (simp add: mk_upds_def mk_updsi_def)
  next
    case (Cons upd upds)
    then show ?case
      by (simp add: mk_upds_def mk_updsi_def split: prod.splits) (rprem, auto intro!: upd_stepI)
  qed
qed

```

```

lemma check_bounded_check_boundedi:
  check_bounded s = check_boundedi si if state_rel s si
  using that unfolding check_bounded_def check_boundedi_def state_rel_def by auto

```

**definition**

```

valid_upd  $\equiv \lambda(x, e). x < n\_vs \wedge (\forall x \in vars\_of\_exp\ e. x < n\_vs)$ 

```

**definition**

```

valid_check b  $\equiv (\forall x \in vars\_of\_bexp\ b. x < n\_vs)$ 

```

**context includes** *lifting\_syntax* **begin**

**notation** *rel\_prod* (**infixr**  $\times_R$  56)

**definition** *is\_at\_least\_equality* **where**

```

is_at_least_equality R  $\equiv \forall x\ y. R\ x\ y \longrightarrow x = y$  for R

```

**named\_\_theorems** *is\_at\_least\_equality*

**lemma** [*is\_at\_least\_equality*]:

```

is_at_least_equality (=)

```

```

by (simp add: is_at_least_equality_def)

```

**lemma** [*is\_at\_least\_equality*]:

```

is_at_least_equality R if is_equality R for R

```

```

using that by (simp add: is_at_least_equality_def is_equality_def)

```

**lemma** [*is\_at\_least\_equality*]:

```

is_at_least_equality (eq_onp P)

```

```

by (simp add: is_at_least_equality_def eq_onp_def)

```

**lemma** *is\_at\_least\_equality\_list\_all2*[*is\_at\_least\_equality*]:

```

is_at_least_equality (list_all2 R) if is_at_least_equality R for R

```

```

using that unfolding is_at_least_equality_def

```

```

by (auto simp: list.rel_eq dest: list_all2_mono[where Q = (=)])

```

**lemma** *is\_at\_least\_equality\_rel\_prod*[*is\_at\_least\_equality*]:

```

is_at_least_equality (R1  $\times_R$  R2)

```

```

if is_at_least_equality R1 is_at_least_equality R2 for R1 R2

```

```

using that unfolding is_at_least_equality_def by auto

```

**lemma** *is\_at\_least\_equality\_cong1*:

```

(S1 ==> (=)) f f if is_at_least_equality S1 is_at_least_equality S2 for S1 f

```

```

using that unfolding is_at_least_equality_def by (intro rel_funI) auto

```

**lemma** *is\_at\_least\_equality\_cong2*:

$(S1 \implies S2 \implies (=)) \text{ } f f$  **if** *is\_at\_least\_equality* *S1* *is\_at\_least\_equality* *S2* **for** *S1* *S2* *f*  
**using** *that* **unfolding** *is\_at\_least\_equality\_def* **by** (*intro rel\_funI*) *auto*

**lemma** *is\_at\_least\_equality\_cong3*:

$(S1 \implies S2 \implies S3 \implies (=)) \text{ } f f$   
**if** *is\_at\_least\_equality* *S1* *is\_at\_least\_equality* *S2* *is\_at\_least\_equality* *S3* **for** *S1* *S2* *S3* *f*  
**using** *that* **unfolding** *is\_at\_least\_equality\_def* **by** (*intro rel\_funI*) *force*

**lemma** *is\_at\_least\_equality\_Let*:

$(S \implies ((=) \implies R) \implies R)$  *Let* *Let* **if** *is\_at\_least\_equality* *S* **for** *R*  
**using** *that* **unfolding** *is\_at\_least\_equality\_def*  
**by** (*intro rel\_funI*) (*erule Let\_transfer[THEN rel\_funD, THEN rel\_funD, rotated]*, *auto*)

**lemma** *map\_transfer\_length*:

**fixes** *R S n*  
**shows**  
 $((R \implies S)$   
 $\implies (\lambda x y. \text{list\_all2 } R \ x \ y \wedge \text{length } x = n)$   
 $\implies (\lambda x y. \text{list\_all2 } S \ x \ y \wedge \text{length } x = n))$   
*map map*  
**apply** (*intro rel\_funI conjI*)  
**apply** (*erule list.map\_transfer[THEN rel\_funD, THEN rel\_funD]*, *erule conjunct1*)  
**apply** *simp*  
**done**

**lemma** *upt\_0\_transfer*:

$(eq\_onp \ (\lambda x. x = 0) \implies eq\_onp \ (\lambda x. x = n) \implies \text{list\_all2} \ (eq\_onp \ (\lambda x. x < n)))$  *upt*  
*upt* **for** *n*  
**apply** (*intro rel\_funI upt\_transfer\_upper\_bound[THEN rel\_funD, THEN rel\_funD]*)  
**apply** (*assumption* | *erule eq\_onp\_to\_eq*)+  
**done**

**lemma** *upt\_length\_transfer*:

$(eq\_onp \ (\lambda x. x = 0) \implies eq\_onp \ (\lambda x. x = n)$   
 $\implies (\lambda x y. \text{list\_all2} \ (eq\_onp \ (\lambda x. x < n)) \ x \ y \wedge \text{length } x = n))$  *upt* *upt* **for** *n*  
**apply** (*intro rel\_funI conjI upt\_0\_transfer[THEN rel\_funD, THEN rel\_funD]*, *assumption*+)  
**apply** (*simp add: eq\_onp\_def*)  
**done**

**lemma** *case\_prod\_transfer\_strong*:

**fixes** *A B C*  
**assumes**  $\bigwedge x y. A \ x \ y \implies A \ x \ y \bigwedge x y. B \ x \ y \implies B \ x \ y$   
**shows**  $((A \implies B \implies C) \implies A1 \times_R B1 \implies C)$  *case\_prod case\_prod*  
**apply** (*intro rel\_funI*)  
**apply** *clarsimp*  
**apply** (*drule assms*)+  
**apply** (*drule* (1) *rel\_funD*)+  
**apply** *assumption*  
**done**

**lemma** *concat\_transfer\_strong*:

**fixes** *A B C*  
**assumes**  $\bigwedge x y. A \ x \ y \implies B \ x \ y \bigwedge x y. C \ x \ y \implies \text{list\_all2} \ (\text{list\_all2 } A) \ x \ y$   
**shows**  $(C \implies \text{list\_all2 } B)$  *concat concat*

```

apply (intro rel_funI concat_transfer[THEN rel_funD])
apply (drule assms)
apply (erule list_all2_mono)
apply (erule list_all2_mono)
apply (erule assms)
done

```

```

lemma map_transfer_strong:
  fixes A B C
  assumes  $\bigwedge xs\ ys. C\ xs\ ys \implies list\_all2\ A\ xs\ ys$ 
  shows ((A ==> B) ==> C ==> list_all2 B) map map
  apply (intro rel_funI)
  apply (erule list.map_transfer[THEN rel_funD, THEN rel_funD])
  apply (erule assms)
  done

```

```

lemma list_update_transfer':
  fixes A :: 'a  $\Rightarrow$  'b  $\Rightarrow$  bool
  shows (list_all2 A ==> eq_onp ( $\lambda i. i < n\_ps$ ) ==> A ==> list_all2 A) list_update
list_update
  apply (intro rel_funI)
  apply (rule list_update_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
  apply (auto simp: eq_onp_def)
  done

```

```

lemma list_update_transfer'':
  fixes A :: 'a  $\Rightarrow$  'b  $\Rightarrow$  bool and n
  shows (( $\lambda x\ y. list\_all2\ A\ x\ y \wedge length\ x = n$ ) ==> eq_onp ( $\lambda i. i < n$ ) ==> A
==> ( $\lambda x\ y. list\_all2\ A\ x\ y \wedge length\ x = n$ )) list_update list_update
  apply (intro rel_funI conjI)
  subgoal
    apply (erule conjE)
    apply (rule List.list_update_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
    apply (assumption | elim eq_onp_to_eq)+
    done
  apply simp
  done

```

```

lemma list_update_transfer''':
  fixes A :: 'a  $\Rightarrow$  'b  $\Rightarrow$  bool and n
  shows (( $\lambda x\ y. list\_all2\ A\ x\ y \wedge length\ x = n$ ) ==> (=) ==> A
==> ( $\lambda x\ y. list\_all2\ A\ x\ y \wedge length\ x = n$ )) list_update list_update
  apply (intro rel_funI conjI)
  subgoal
    apply (erule conjE)
    apply (rule List.list_update_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
    apply (assumption | elim eq_onp_to_eq)+
    done
  apply simp
  done

```

```

lemma fold_transfer_strong:
  fixes A B
  assumes  $\bigwedge x\ y. A1\ x\ y \implies A\ x\ y \wedge x\ y. B1\ x\ y \implies B\ x\ y \wedge x\ y. B\ x\ y \implies B2\ x\ y$ 

```

```

 $\bigwedge x y. B x y \implies B3 x y$ 
shows (( $A \implies B2 \implies B1$ )  $\implies$   $list\_all2 A1 \implies B \implies B3$ ) fold fold
apply (intro rel_funI, rule assms)
apply (rule fold_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
  apply (intro rel_funI)
  apply (drule rel_funD, erule assms)
  apply (drule rel_funD, erule assms, erule assms)
apply assumption+
done

```

```

lemma bval_bval_transfer[transfer_rule]:
  (state_rel  $\implies$  eq_onp valid_check  $\implies$  (=)) ( $\lambda s. bval (the\ o\ s)$ ) bval
by (intro rel_funI) (auto simp: eq_onp_def valid_check_def state_rel_def intro!: bval_bval)

```

```

lemma mk_upds_mk_updsi_transfer[transfer_rule]:
  (state_rel  $\implies$   $list\_all2 (eq\_onp\ valid\_upd) \implies state\_rel$ ) mk_upds mk_updsi
apply (intro rel_funI)
subgoal for  $x\ y\ upds\ upds'$ 
  apply (subgoal_tac upds' = upds)
  apply simp
  apply (rule mk_upds_mk_updsi)
  apply assumption
subgoal
  by (smt case_prodI2 case_prod_conv eq_onp_def list_all2_same valid_upd_def)
subgoal
  by (smt case_prodE case_prod_conv eq_onp_def list_all2_same valid_upd_def)
subgoal
  by (metis eq_onp_to_eq list.rel_eq_onp)
done
done

```

```

lemma check_bounded_transfer[transfer_rule]:
  (state_rel  $\implies$  (=)) check_bounded check_boundedi
by (simp add: check_bounded_check_boundedi rel_funI)

```

```

lemma trans_map_transfer:
  (eq_onp ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$ 
     $list\_all2 (eq\_onp\ valid\_check \times_R (=) \times_R eq\_onp (pred\_act (\lambda x. x < num\_actions)))$ 
     $\times_R list\_all2 (eq\_onp\ valid\_upd) \times_R (=)$ 
  ) trans_map trans_map
apply (intro rel_funI, simp add: eq_onp_def, intro list.rel_refl_strong)
apply clarsimp
apply (auto 4 4 dest!: trans_mapD dest: action_setD var_setD var_setD2 intro: list.rel_refl_strong
  simp: valid_upd_def valid_check_def
)
done

```

```

lemma trans_map_transfer':
  (eq_onp ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$ 
     $list\_all2 (eq\_onp\ valid\_check \times_R (=) \times_R (=) \times_R list\_all2 (eq\_onp\ valid\_upd) \times_R (=))$ 
  ) trans_map trans_map
apply (intro rel_funI, simp add: eq_onp_def, intro list.rel_refl_strong)
apply clarsimp

```

```

apply (intro conjI list.rel_refl_strong)
apply (auto 4 4 dest: var_setD trans_mapD var_setD2 simp: valid_upd_def valid_check_def)
done

lemma map_filter_transfer[transfer_rule]:
  (( $S \implies \text{rel\_option } R$ )  $\implies \text{list\_all2 } S \implies \text{list\_all2 } R$ )  $\text{List.map\_filter } \text{List.map\_filter}$ 
  unfolding map_filter_def
  apply (clarsimp intro!: rel_funI)
  subgoal for  $f\ g\ xs\ ys$ 
  apply (rule list.map_transfer[THEN rel_funD, THEN rel_funD, of  $\lambda x\ y. f\ x \neq \text{None} \wedge S\ x\ y$ ])
  apply (rule rel_funI)
  subgoal for  $a\ b$ 
  apply (cases f a)
  apply (auto simp: option_rel_Some1 option_rel_Some2 dest!: rel_funD)
  done
  subgoal
  apply rotate_tac
  apply (induction rule: list_all2_induct)
  apply (auto dest: rel_funD)
  done
done
done

lemma trans_i_map_transfer[transfer_rule]:
  ( $\text{eq\_onp } (\lambda i. i < n\_ps) \implies (=) \implies$ 
     $\text{list\_all2 } (\text{eq\_onp } \text{valid\_check} \times_R (=) \times_R (=) \times_R \text{list\_all2 } (\text{eq\_onp } \text{valid\_upd}) \times_R (=))$ 
  )  $\text{trans\_i\_map } \text{trans\_i\_map}$ 
  supply [transfer_rule] = trans_map_transfer'
  unfolding trans_i_map_def by transfer_prover

lemma int_trans_from_loc_transfer[transfer_rule]:
  ( $\text{eq\_onp } (\lambda i. i < n\_ps) \implies (=) \implies$ 
     $(\lambda x\ y. \text{list\_all2 } (=) x\ y \wedge \text{length } x = n\_ps) \implies$ 
     $\text{state\_rel} \implies \text{list\_all2}((=) \times_R (=) \times_R (=) \times_R (\lambda x\ y. \text{list\_all2 } (=) x\ y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}))$ 
  )  $\text{int\_trans\_from\_loc } \text{int\_trans\_from\_loc\_impl}$ 
  supply [transfer_rule] = list_update_transfer''
  unfolding int_trans_from_loc_def int_trans_from_loc_impl_def Let_def by transfer_prover

lemma n_ps_transfer:
   $\text{eq\_onp } (\lambda x. x = n\_ps) n\_ps n\_ps$ 
  by (simp add: eq_onp_def)

lemma zero_nat_transfer:
   $(=) 0 (0::\text{nat})$ 
  ..

lemma int_trans_from_all_transfer[transfer_rule]:
  ( $(\lambda x\ y. \text{list\_all2 } (=) x\ y \wedge \text{length } x = n\_ps) \implies \text{state\_rel}$ 
     $\implies \text{list\_all2}((=) \times_R (=) \times_R (=) \times_R (\lambda x\ y. \text{list\_all2 } (=) x\ y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}))$ 
  )  $\text{int\_trans\_from\_all } \text{int\_trans\_from\_all\_impl}$ 
  supply [transfer_rule] = zero_nat_transfer n_ps_transfer

```

```

unfolding int_trans_from_all_def int_trans_from_all_impl_def Let_def
by transfer_prover

lemma int_trans_from_vec_transfer[transfer_rule]:
  (list_all2 (eq_onp ( $\lambda x. x < n\_ps$ )  $\times_R (=)$ )  $\implies$  ( $\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps$ )
     $\implies$  state_rel
     $\implies$  list_all2(( $=$ )  $\times_R (=)$   $\times_R (=)$   $\times_R (\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps)$   $\times_R$ 
state_rel))
  int_trans_from_vec int_trans_from_vec_impl
unfolding int_trans_from_vec_def int_trans_from_vec_impl_def Let_def
by transfer_prover

private definition R where  $R \equiv (\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps)$ 

lemma get_committed_transfer[transfer_rule]:
  (( $\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps$ )  $\implies$  list_all2 (eq_onp ( $\lambda x. x < n\_ps$ )  $\times_R$ 
(=)))
  get_committed get_committed
proof -
  have [transfer_rule]:
    R automata automata
    unfolding R_def by (simp add: n_ps_def list.rel_eq)
  show ?thesis
  supply [transfer_rule] = zero_nat_transfer n_ps_transfer
  unfolding get_committed_def
  unfolding Let_def
  apply transfer_prover_start
  using [[goals_limit=15]]
  prefer 8
  apply transfer_step
    prefer 8
  unfolding R_def
    apply transfer_step+
  apply transfer_prover
  done
qed

lemma eq_transfer:
  (list_all2 (eq_onp ( $\lambda x. x < n\_ps$ )  $\times_R (=)$ )  $\implies$  list_all2 (=)  $\implies$  (=)) (=) (=)
  unfolding eq_onp_def
  apply (intro rel_funI)
  apply (drule list_all2_mono[where  $Q = (=)$ ])
  apply (auto simp add: list.rel_eq rel_prod.simps)
  done

lemma int_trans_from_transfer:
  (( $\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps$ )  $\times_R state\_rel$ 
 $\implies$  list_all2 (( $=$ )  $\times_R (=)$   $\times_R (=)$   $\times_R (\lambda x y. list\_all2 (=) x y \wedge length\ x = n\_ps)$   $\times_R$ 
state_rel)))
  int_trans_from int_trans_impl
  supply [transfer_rule] = eq_transfer
  unfolding int_trans_impl_def int_trans_from_def Let_def
  by transfer_prover

```

```

lemma pairs_by_action_transfer[transfer_rule]:
  (( $\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n$ ) ==> state_rel ==>
    list_all2 (( $=$ )  $\times_R$  eq_onp valid_check  $\times_R$  ( $=$ )  $\times_R$  ( $=$ )  $\times_R$  list_all2 (eq_onp valid_upd)  $\times_R$ 
    ( $=$ )  $\times_R$  ( $=$ ))
    ==>
    list_all2 (( $=$ )  $\times_R$  eq_onp valid_check  $\times_R$  ( $=$ )  $\times_R$  ( $=$ )  $\times_R$  list_all2 (eq_onp valid_upd)  $\times_R$ 
    ( $=$ )  $\times_R$  ( $=$ ))
    ==>
    list_all2 (( $=$ )  $\times_R$  ( $=$ )  $\times_R$  ( $=$ )  $\times_R$  ( $\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n$ )  $\times_R$  state_rel))
  pairs_by_action pairs_by_action_impl
supply [transfer_rule] = list_update_transfer'''
unfolding pairs_by_action_def pairs_by_action_impl_def by transfer_prover

```

```

lemmas rel_elims =
  rel_prod.cases
  rel_funD

```

```

lemmas rel_intros =
  rel_funI

```

```

lemma pairs_by_action_transfer':
  (( $\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n$ ) ==> state_rel ==>
    list_all2 (B  $\times_R$  eq_onp valid_check  $\times_R$  C  $\times_R$  D  $\times_R$  list_all2 (eq_onp valid_upd)  $\times_R$  E
     $\times_R$  F) ==>
    list_all2 (B  $\times_R$  eq_onp valid_check  $\times_R$  C  $\times_R$  D  $\times_R$  list_all2 (eq_onp valid_upd)  $\times_R$  E
     $\times_R$  F) ==>
    list_all2 (( $=$ )  $\times_R$  ( $=$ )  $\times_R$  ( $=$ )  $\times_R$  ( $\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n$ )  $\times_R$  state_rel))
  pairs_by_action pairs_by_action_impl
if  $\bigwedge x y. B x y \implies x = y$ 
   $\bigwedge x y. C x y \implies x = y \bigwedge x y. D x y \implies x = y$ 
   $\bigwedge x y. E x y \implies x = y \bigwedge x y. F x y \implies x = y$ 
for B C D E F
apply (intro rel_funI)
apply (rule pairs_by_action_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD,
THEN rel_funD])
apply (assumption | erule that list_all2_mono prod.rel_mono_strong) +
done

```

```

lemma trans_in_map_transfer[transfer_rule]:
  (eq_onp ( $\lambda i. i < n\_ps$ ) ==> ( $=$ ))
  ==> list_all2 (
    eq_onp valid_check  $\times_R$  ( $=$ )  $\times_R$  eq_onp ( $\lambda a. a < \text{num\_actions}$ )  $\times_R$  list_all2 (eq_onp
    valid_upd)  $\times_R$  ( $=$ ))
  ) trans_in_map trans_in_map
supply [transfer_rule] = trans_map_transfer[folded act.rel_eq_onp]
unfolding trans_in_map_def by transfer_prover

```

```

lemma trans_in_map_transfer[transfer_rule]:
  (eq_onp ( $\lambda i. i < n\_ps$ ) ==> ( $=$ ))
  ==> list_all2 (eq_onp valid_check  $\times_R$  ( $=$ )  $\times_R$  ( $=$ )  $\times_R$  list_all2 (eq_onp valid_upd)  $\times_R$ 
  ( $=$ ))
  ) trans_in_map trans_in_map
unfolding trans_in_map_def oops

```

**lemma** *trans\_out\_map\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  *list\_all2* (  
   *eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda a. a < num\_actions$ )  $\times_R$  *list\_all2* (*eq\_onp*  
*valid\_upd*)  $\times_R$  (=)  
 )) *trans\_out\_map* *trans\_out\_map*  
**supply** [*transfer\_rule*] = *trans\_map\_transfer*[*folded act.rel\_eq\_onp*]  
**unfolding** *trans\_out\_map\_def* **by** *transfer\_prover*

**lemma** *trans\_out\_map\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  *list\_all2* (  
   *eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  (=)  $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  $\times_R$  (=)))  
*trans\_out\_map* *trans\_out\_map*  
**unfolding** *trans\_out\_map\_def* **oops**

**lemma** *actions\_by\_state\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$    
   *list\_all2* ((=)  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda i. i < n$ )  $\times_R$  (=))  $\implies$    
   ( $\lambda x y. list\_all2$  (*list\_all2* (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\times_R$  (=)  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < n$ )  
 $\times_R$  (=)))  $x y \wedge length\ x = n$ )  $\implies$    
   ( $\lambda x y. list\_all2$  (*list\_all2* (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\times_R$  (=)  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < n$ )  
 $\times_R$  (=)))  $x y \wedge length\ x = n$ )  
 )  
*actions\_by\_state* *actions\_by\_state* **for** *n*  
**supply** [*transfer\_rule*] = *list\_update\_transfer''*  
**unfolding** *actions\_by\_state\_def* **by** *transfer\_prover*

**lemma** *actions\_by\_state\_transfer'*[*transfer\_rule*]:  
 (  
   *eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$    
   *list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda i. i < n$ )  $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  
 $\times_R$  (=))  $\implies$    
   ( $\lambda x y. list\_all2$  (*list\_all2* (  
   *eq\_onp* ( $\lambda i. i < n\_ps$ )  $\times_R$  *eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < n$ )  
 $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  $\times_R$  (=)))  $x y \wedge length\ x = n$ )  $\implies$    
   ( $\lambda x y. list\_all2$  (*list\_all2* (  
   *eq\_onp* ( $\lambda i. i < n\_ps$ )  $\times_R$  *eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < n$ )  
 $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  $\times_R$  (=)))  $x y \wedge length\ x = n$ )  
 )  
*actions\_by\_state* *actions\_by\_state*  
**for** *n*  
**supply** [*transfer\_rule*] = *list\_update\_transfer''*  
**unfolding** *actions\_by\_state\_def* **by** *transfer\_prover*

**lemma** *transfer\_consts*:  
 (*eq\_onp* ( $\lambda x. x = num\_actions$ )) *num\_actions num\_actions* (*eq\_onp* ( $\lambda x. x = 0$ )) (*0::nat*) 0  
 (*eq\_onp* ( $\lambda x. x = n\_ps$ )) *n\_ps n\_ps*  
**by** (*auto simp: eq\_onp\_def*)

**lemma** *all\_actions\_by\_state\_transfer*[*transfer\_rule*]:  
 (  
   (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  *list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp*  
 $\lambda i. i < num\_actions$ )  $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  $\times_R$  (=)))  
 )

```

====>
( $\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n\_ps$ )
====>
( $\lambda x y. \text{list\_all2 } (\text{list\_all2 } (\text{eq\_onp } (\lambda i. i < n\_ps) \times_R \text{eq\_onp valid\_check} \times_R (=) \times_R \text{eq\_onp } (\lambda i. i < \text{num\_actions}) \times_R \text{list\_all2 } (\text{eq\_onp valid\_upd}) \times_R (=))) x y \wedge \text{length } x = \text{num\_actions}$ )
)
all_actions_by_state all_actions_by_state
supply [transfer_rule] = map_transfer_length upt_0_transfer upt_length_transfer transfer_consts
n_ps_transfer
unfolding all_actions_by_state_def by transfer_prover

```

**lemma** all\_actions\_from\_vec\_transfer[transfer\_rule]:

```

(
  ( $\text{eq\_onp } (\lambda i. i < n\_ps) \text{====> } (=) \text{====> } \text{list\_all2 } (\text{eq\_onp valid\_check} \times_R (=) \times_R \text{eq\_onp } (\lambda i. i < \text{num\_actions}) \times_R \text{list\_all2 } (\text{eq\_onp valid\_upd}) \times_R (=)))$ 
  ====>
   $\text{list\_all2 } (\text{eq\_onp } (\lambda i. i < n\_ps) \times_R (=))$ 
  ====>
  ( $\lambda x y. \text{list\_all2 } (\text{list\_all2 } (\text{eq\_onp } (\lambda i. i < n\_ps) \times_R \text{eq\_onp valid\_check} \times_R (=) \times_R \text{eq\_onp } (\lambda i. i < \text{num\_actions}) \times_R \text{list\_all2 } (\text{eq\_onp valid\_upd}) \times_R (=))) x y \wedge \text{length } x = \text{num\_actions}$ )
)
all_actions_from_vec all_actions_from_vec
supply [transfer_rule] = map_transfer_length upt_length_transfer transfer_consts
unfolding all_actions_from_vec_def all_actions_from_vec_def by transfer_prover

```

**lemma** bin\_actions\_transfer[transfer\_rule]:

```

( $\text{list\_all2 } (\text{eq\_onp } (\lambda x. x < \text{num\_actions}))$ ) bin_actions bin_actions
proof -
have *:  $\text{list\_all2 } (\text{eq\_onp } (\lambda x. x < \text{num\_actions})) [0..<\text{num\_actions}] [0..<\text{num\_actions}]$ 
by (rule list.rel_refl_strong) (auto simp: eq_onp_def)
show ?thesis
unfolding bin_actions_def
by (rule filter_transfer[THEN rel_funD, THEN rel_funD, OF _ *]) (auto simp: eq_onp_def)
qed

```

**lemma** Let\_transfer\_bin\_aux:

```

(( $\lambda x y. \text{list\_all2 } (\text{list\_all2 } (\text{eq\_onp } (\lambda i. i < n\_ps) \times_R \text{eq\_onp valid\_check} \times_R \text{list\_all2 } (\text{rel\_acconstraint } (=) (=)) \times_R \text{eq\_onp } (\lambda i. i < \text{num\_actions}) \times_R \text{list\_all2 } (\text{eq\_onp valid\_upd}) \times_R \text{list\_all2 } (=) \times_R (=))) x y$ 
   $\wedge \text{length } x = \text{num\_actions}$ )
  ====>
  (( $\lambda x y. \text{list\_all2 } (\text{list\_all2 } ((=) \times_R \text{eq\_onp valid\_check} \times_R \text{list\_all2 } (\text{rel\_acconstraint } (=) (=)) \times_R (=) \times_R \text{list\_all2 } (\text{eq\_onp valid\_upd}) \times_R \text{list\_all2 } (=) \times_R (=))) x y$ 
   $\wedge \text{length } x = \text{num\_actions}$ )
  ====>
   $\text{list\_all2 } (\text{list\_all2 } (\text{rel\_acconstraint } (=) (=)) \times_R \text{rel\_label } (=) \times_R \text{list\_all2 } (=) \times_R (\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}))$ 
  ====>
   $\text{list\_all2 } (\text{list\_all2 } (\text{rel\_acconstraint } (=) (=)) \times_R \text{rel\_label } (=) \times_R \text{list\_all2 } (=) \times_R (\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}))$ 
  Let Let
unfolding Let_def

```

```

by (intro rel_funI, drule rel_funD)
  (auto simp: eq_onp_def elim!: list_all2_mono prod.rel_mono_strong)

```

**lemma** *bin\_trans\_from\_transfer*:

```

((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel
==> list_all2 ((=) ×R (=) ×R (=) ×R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R
state_rel)))
bin_trans_from bin_trans_from_impl
unfolding bin_trans_from_impl_def bin_trans_from_def
supply [transfer_rule] =
  map_transfer_length upt_length_transfer transfer_consts eq_transfer Let_transfer_bin_aux
by transfer_prover

```

**lemma** *trans\_map\_transfer''*:

```

((=) ==> (=) ==>
  list_all2 (eq_onp valid_check ×R (=) ×R eq_onp (pred_act (λx. x < num_actions)) ×R
list_all2 (eq_onp valid_upd) ×R (=)))
trans_map trans_map
apply (intro rel_funI, simp add: eq_onp_def, intro list.rel_refl_strong)
apply clarsimp
apply (auto 4 4 dest!: trans_mapD dest: action_setD var_setD intro: list.rel_refl_strong simp:
valid_upd_def)
oops

```

**lemma** *compute\_upds\_transfer*:

```

(
  (list_all2 (=) ×R (=) ×R list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n) ×R
state_rel)
==> list_all2 (list_all2
  ((=) ×R eq_onp valid_check ×R list_all2 (=) ×R (=) ×R list_all2 (eq_onp valid_upd)
×R list_all2 (=) ×R (=))) ==>
  list_all2 (
    list_all2 (=) ×R (=) ×R list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n) ×R
state_rel
  )) compute_upds compute_upds_impl
supply [transfer_rule] = list_update_transfer'''
unfolding compute_upds_def compute_upds_impl_def by transfer_prover

```

**lemma** *in\_transfer*:

```

(eq_onp (λx. x < num_actions) ==> (=) ==> (=)) (∈) (∈)
by (intro rel_funI, rule member_transfer[of (=), THEN rel_funD, THEN rel_funD])
  (auto simp: eq_onp_def rel_set_eq intro: bi_unique_eq)

```

**lemma** *trans\_in\_broad\_map\_transfer*[transfer\_rule]:

```

(eq_onp (λi. i < n_ps) ==> (=) ==> list_all2 (eq_onp valid_check ×R (=) ×R eq_onp
(λx. x < num_actions) ×R list_all2 (eq_onp valid_upd) ×R (=)))
trans_in_broad_map trans_in_broad_map
supply [transfer_rule] = trans_map_transfer[folded act.rel_eq_onp] in_transfer
unfolding trans_in_broad_map_def by transfer_prover

```

**lemma** *trans\_out\_broad\_map\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  *list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp*  
 ( $\lambda x. x < num\_actions$ )  $\times_R$  *list\_all2* (*eq\_onp valid\_upd*)  $\times_R$  (=)))  
*trans\_out\_broad\_map* *trans\_out\_broad\_map*  
**supply** [*transfer\_rule*] = *trans\_map\_transfer*[*folded act.rel eq\_onp*] *in\_transfer*  
**unfolding** *trans\_out\_broad\_map\_def* **by** *transfer\_prover*

We are using the “equality version” of parametricity for (!) here.

**lemma** *actions\_by\_state'\_transfer*[*transfer\_rule*]:  
 (*list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < num\_actions$ )  $\times_R$  *list\_all2*  
 (*eq\_onp valid\_upd*)  $\times_R$  (=))  
 $\implies$  ( $\lambda x y. list\_all2$  (  
*list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < num\_actions$ )  $\times_R$  *list\_all2*  
 (*eq\_onp valid\_upd*)  $\times_R$  (=))) *x y*  
 $\wedge$  *length x* = *num\_actions*  
 ))  
*actions\_by\_state'* *actions\_by\_state'*  
**supply** [*transfer\_rule*] = *transfer\_consts*  
*upt\_length\_transfer*  
*map\_transfer\_length*  
*list\_update\_transfer''*  
**unfolding** *actions\_by\_state'\_def* **by** *transfer\_prover*

**lemma** *trans\_in\_broad\_grouped\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  ( $\lambda x y. list\_all2$  (  
*list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < num\_actions$ )  $\times_R$  *list\_all2*  
 (*eq\_onp valid\_upd*)  $\times_R$  (=))) *x y*  
 $\wedge$  *length x* = *num\_actions*  
 )) *trans\_in\_broad\_grouped* *trans\_in\_broad\_grouped*  
**unfolding** *trans\_in\_broad\_grouped\_def* **by** *transfer\_prover*

**lemma** *trans\_out\_broad\_grouped\_transfer*[*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda i. i < n\_ps$ )  $\implies$  (=)  $\implies$  ( $\lambda x y. list\_all2$  (  
*list\_all2* (*eq\_onp valid\_check*  $\times_R$  (=)  $\times_R$  *eq\_onp* ( $\lambda x. x < num\_actions$ )  $\times_R$  *list\_all2*  
 (*eq\_onp valid\_upd*)  $\times_R$  (=))) *x y*  
 $\wedge$  *length x* = *num\_actions*  
 )) *trans\_out\_broad\_grouped* *trans\_out\_broad\_grouped*  
**unfolding** *trans\_out\_broad\_grouped\_def* **by** *transfer\_prover*

**lemma** *make\_combs\_transfer*:  
**fixes** *R*  
**assumes**  $\bigwedge x y. R\ x\ y \implies x = y$   
**shows**  
 (*eq\_onp* ( $\lambda x. x < n\_ps$ )  
 $\implies$  *eq\_onp* ( $\lambda x. x < num\_actions$ )  
 $\implies$  ( $\lambda x y. list\_all2$  ( $\lambda x y. list\_all2$  (*list\_all2* *R*) *x y*  $\wedge$  *length x* = *num\_actions*) *x y*  
 $\wedge$  *length x* = *n\_ps*)  
 $\implies$  *list\_all2* (*list\_all2* (*eq\_onp* ( $\lambda x. x < n\_ps$ )  $\times_R$  *R*)))  
*make\_combs* *make\_combs*

**proof** –

**have** [*transfer\_rule*]:  
 (*eq\_onp* ( $\lambda x. x < n\_ps$ )  $\implies$  *eq\_onp* ( $\lambda x. x < n\_ps$ )  $\implies$  (=)) (=) (=)

```

(list_all2 R ==> list_all2 R ==> (=)) (=) (=)
(list_all2 (list_all2 (eq_onp ( $\lambda x. x < n\_ps$ )  $\times_R R$ ))
  ==> list_all2 (list_all2 (eq_onp ( $\lambda x. x < n\_ps$ )  $\times_R R$ )) ==> (=)) (=) (=)
apply (simp_all add: eq_onp_def)
subgoal
  by (smt assms list.rel_eq list_all2_mono rel_funI)
subgoal
  by (smt assms fun.rel_eq list_all2_eq list_all2_mono rel_fun_mono rel_prod.cases)
subgoal
  by (smt assms fun.rel_eq list_all2_eq list_all2_mono rel_fun_mono rel_prod.cases)
done
show ?thesis
supply [transfer_rule] = upt_0_transfer transfer_consts
unfolding make_combs_def by transfer_prover
qed

```

**lemma** broad\_trans\_from\_alt\_def2:

```

broad_trans_from = ( $\lambda(L, s).$ 
  let
    pairs = get_committed L;
    In = map ( $\lambda p. \text{trans\_in\_broad\_grouped } p (L ! p)$ ) [0.. $n\_ps$ ];
    Out = map ( $\lambda p. \text{trans\_out\_broad\_grouped } p (L ! p)$ ) [0.. $n\_ps$ ];
    In = map (map (filter ( $\lambda(b, \_). \text{bval } (the \circ s) \ b$ ))) In;
    Out = map (map (filter ( $\lambda(b, \_). \text{bval } (the \circ s) \ b$ ))) Out
  in
    if pairs = [] then
      concat (
        map ( $\lambda a.$ 
          concat (map ( $\lambda p.$ 
            let
              outs = Out ! p ! a
            in if outs = [] then []
            else
              let
                combs = make_combs p a In;
                outs = map ( $\lambda t. (p, t)$ ) outs;
                combs = (
                  if combs = [] then [[x].  $x \leftarrow outs$ ]
                  else concat (map ( $\lambda x. \text{map } (\lambda xs. x \# xs)$  combs) outs));
                init = ([], Broad a, [], (L, s))
              in
                compute_upds init combs
            )
          ) [0.. $n\_ps$ ])
      ) [0.. $num\_actions$ ])
    else
      concat (
        map ( $\lambda a.$ 
          let
            ins_committed =
              List.map_filter ( $\lambda(p, \_). \text{if } In ! p ! a \neq [] \text{ then } Some \ p \text{ else } None$ ) pairs;
            always_committed = (length ins_committed > 1)
          in

```

```

concat (map (λp.
  let
    outs = Out ! p ! a
  in if outs = [] then []
  else if
    ¬ always_committed ∧ (ins_committed = [p] ∨ ins_committed = [])
    ∧ ¬ list_ex (λ (q, _). q = p) pairs
  then []
  else
    let
      combs = make_combs p a In;
      outs = map (λt. (p, t)) outs;
      combs = (
        if combs = [] then [[x]. x ← outs]
        else concat (map (λx. map (λxs. x # xs) combs) outs));
      init = ([], Broad a, [], (L, s))
    in
      compute_upds init combs
)
)
[0..<n_ps])
)
[0..<num_actions])
)

```

**unfolding** broad\_trans\_from\_def compute\_upds\_def  
**apply** (rule HOL.refl)  
**done**

**lemma** concat\_length\_transfer:

((λ x y. list\_all2 (list\_all2 A) x y ∧ length x = n) ==> list\_all2 A) concat concat **for** A n  
**by** (intro rel\_funI concat\_transfer[THEN rel\_funD], elim conjunct1)

**lemma** broad\_trans\_from\_transfer:

((λx y. list\_all2 (==) x y ∧ length x = n\_ps) ×<sub>R</sub> state\_rel  
==> list\_all2 ((=) ×<sub>R</sub> (=) ×<sub>R</sub> (=) ×<sub>R</sub> ((λx y. list\_all2 (==) x y ∧ length x = n\_ps) ×<sub>R</sub>  
state\_rel)))

broad\_trans\_from broad\_trans\_from\_impl

**proof** –

**have** compute\_upds\_impl\_transfer[transfer\_rule]:

```

(list_all2 (==) ×R
  rel_label (eq_onp (λx. x < num_actions)) ×R
  list_all2 (==) ×R
  (λx y. list_all2 (==) x y ∧ length x = n_ps) ×R state_rel ==>
  list_all2
  (list_all2
    (eq_onp (λx. x < n_ps) ×R
      eq_onp valid_check ×R
      list_all2 (rel_acconstraint (==) (==)) ×R
      eq_onp (λx. x < num_actions) ×R
      list_all2 (eq_onp valid_upd) ×R list_all2 (==) ×R (==))) ==>
  list_all2
  (list_all2 (==) ×R
    (==) ×R

```

```

    list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel))
  compute_upds compute_upds_impl
apply (intro rel_funI)
apply (rule compute_upds_transfer[THEN rel_funD, THEN rel_funD])
apply (elim rel_prod.cases)
apply (simp only:)
apply (intro rel_prod.intros)
apply assumption
subgoal
  by (drule label.rel_mono_strong[of _ _ _ (=)]; simp add: eq_onp_def label.rel_eq)
apply (simp; fail)+
apply (elim list_all2_mono rel_prod.cases)
apply (simp only:)
apply (intro rel_prod.intros)
apply (assumption | simp add: eq_onp_def acconstraint.rel_eq)+
done

have [is_at_least_equality]: is_equality (rel_acconstraint (=) (=))
  by (tactic ‹Transfer.eq_tac @ {context} 1›)

have eq_transfer1:
  (list_all2
    (eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
    list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=)) == =>
  list_all2
    (eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x < num_actions)
×R
    list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))
  == => (=) (=) (=)

  by (intro is_at_least_equality_cong2 is_at_least_equality)

let ?R = (eq_onp (λx. x < n_ps) ×R eq_onp valid_check ×R
  list_all2 (rel_acconstraint (=) (=)) ×R
  eq_onp (λx. x < num_actions) ×R list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R
(=))

have eq_transfer5:
  (list_all2 (list_all2 ?R) == => list_all2 (list_all2 ?R) == => (=) (=) (=))
  by (intro is_at_least_equality_cong2 is_at_least_equality)

have eq_transfer2:
  (list_all2 (eq_onp (λx. x < n_ps)) == => list_all2 (eq_onp (λx. x < n_ps)) == => (=) (=) (=))
  by (intro is_at_least_equality_cong2 is_at_least_equality)

have eq_transfer3:
  (eq_onp (λx. x < n_ps) == => eq_onp (λx. x < n_ps) == => (=) (=) (=))
  by (intro is_at_least_equality_cong2 is_at_least_equality)

have eq_transfer4:
  (list_all2 (eq_onp (λx. x < n_ps) ×R (=)) == =>

```

```

    list_all2 (eq_onp (λx. x < n_ps) ×R (=)) ==> (=) (=) (=)
  by (intro is_at_least_equality_cong2 is_at_least_equality)

have make_combs_transfer:
  (eq_onp (λx. x < n_ps) ==>
    eq_onp (λx. x < num_actions) ==>
    (λx y. list_all2 (λx y. list_all2 (list_all2
      (eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
      list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=)))
      x y ∧ length x = num_actions) x y ∧ length x = n_ps) ==>
    list_all2 (list_all2 (eq_onp (λx. x < n_ps) ×R
      eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
      list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))
    )
  ) make_combs make_combs
apply (intro rel_funI)
apply (rule make_combs_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
subgoal
  apply (simp add: acconstraint.rel_eq list.rel_eq prod.rel_eq)
  apply (drule prod.rel_mono_strong[of _ _ _ (=) (=)], erule eq_onp_to_eq)
  apply (drule prod.rel_mono_strong[of _ _ _ (=) (=)])
  apply assumption
  apply (drule prod.rel_mono_strong[of _ _ _ (=) (=)], erule eq_onp_to_eq)
  apply (drule prod.rel_mono_strong[of _ _ _ (=) (=)])
  apply (drule list_all2_mono[of _ _ _ (=)], erule eq_onp_to_eq, simp only: list.rel_eq)
  apply assumption
  apply (drule (2) prod.rel_mono_strong[of _ _ _ (=) (=)];
    simp add: acconstraint.rel_eq list.rel_eq prod.rel_eq)+
done
apply (simp add: prod.rel_eq)+
done
have
  ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel
    ==> list_all2 (
      (=) ×R (=) ×R (=) ×R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel)))
  (
    λ(L, s).
      let
        pairs = [];
        In = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
        Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps]
      in
        concat (
          map (λa.
            concat (map (λp.
              let
                outs = Out ! p ! a
              in if outs = [] then []
            else
              let
                combs = make_combs p a In;
                outs = map (λt. (p, t)) outs;

```

```

      combs = (
        if combs = [] then [[x]. x ← outs]
        else concat (map (λx. map (λxs. x # xs) combs) outs));
      init = ([], Broad a, [], (L, s))
    in
      compute_upds init combs
  )
  [0.. $n\_ps$ ]
)
[0.. $num\_actions$ ]
) (
λ(L, s).
  let
    pairs = [];
    In = map (λp. trans_in_broad_grouped p (L ! p)) [0.. $n\_ps$ ];
    Out = map (λp. trans_out_broad_grouped p (L ! p)) [0.. $n\_ps$ ]
  in
    concat (
      map (λa.
        concat (map (λp.
          let
            outs = Out ! p ! a
          in if outs = [] then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λxs. x # xs) combs) outs);
              init = ([], Broad a, [], (L, s))
            in
              compute_upds_impl init combs
          )
        ) [0.. $n\_ps$ ]
      )
    ) [0.. $num\_actions$ ]))

```

**supply** [transfer\_rule] =

```

  transfer_consts
  upt_0_transfer
  map_transfer_length
  upt_length_transfer
  make_combs_transfer
  compute_upds_transfer
  concat_length_transfer
  eq_transfer1
  eq_transfer2
  eq_transfer3
  eq_transfer5

```

**unfolding** Let\_def by transfer\_prover

**have**

```

((λx y. list_all2 (=) x y ∧ length x =  $n\_ps$ ) ×R state_rel
==> list_all2 (

```

```

(=) ×R (=) ×R (=) ×R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel)))
(
λ(L, s).
  let
    pairs = get_committed L;
    In = map (λp. trans_in_broad_grouped p (L ! p)) [0..

```

```

      outs = Out ! p ! a
in if outs = [] then []
else if
  ¬ always_committed ∧ (ins_committed = [p] ∨ ins_committed = [])
  ∧ ¬ list_ex (λ (q, _). q = p) pairs
then []
else
  let
    combs = make_combs p a In;
    outs = map (λt. (p, t)) outs;
    combs = (
      if combs = [] then [[x]. x ← outs]
      else concat (map (λx. map (λxs. x # xs) combs) outs));
    init = ([], Broad a, [], (L, s))
  in
    compute_upds_impl init combs
)
[0..supply [transfer_rule] =
  transfer_consts
  upt_0_transfer
  map_transfer_length
  upt_length_transfer
  make_combs_transfer
  compute_upds_transfer
  concat_length_transfer
  eq_transfer1
  eq_transfer2
  eq_transfer3
  eq_transfer5
unfolding Let_def by transfer_prover

show ?thesis
supply [transfer_rule] =
  transfer_consts
  upt_0_transfer
  map_transfer_length
  upt_length_transfer
  make_combs_transfer
  compute_upds_transfer
  concat_length_transfer
  eq_transfer1
  eq_transfer2
  eq_transfer3
  eq_transfer4
  eq_transfer5
unfolding broad_trans_from_alt_def2 broad_trans_from_impl_def Let_def by transfer_prover
qed

lemma trans_from_transfer:
  ((λx y. list_all2 (≡) x y ∧ length x = n_ps) ×R state_rel

```

$\implies \text{list\_all2 } ((=) \times_R (=) \times_R (=) \times_R ((\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}))$   
 $\text{trans\_from trans\_impl}$   
**supply** [transfer\_rule] = int\_trans\_from\_transfer bin\_trans\_from\_transfer broad\_trans\_from\_transfer  
**unfolding** trans\_from\_def trans\_impl\_def **by** transfer\_prover

**lemma** list\_all2\_swap:  
 $\text{list\_all2 } S \text{ ys xs if list\_all2 } R \text{ xs ys } S = (\lambda x y. R \text{ y } x) \text{ for } R \text{ } S$   
**using** list\_all2\_swap **that** **by** blast

**lemma** swap\_rel\_prod:  $(\lambda x y. (R \times_R S) \text{ y } x) = (\lambda x y. R \text{ y } x) \times_R (\lambda x y. S \text{ y } x) \text{ for } R \text{ } S$   
**by** (intro ext) auto

**lemma** swap\_eq:  
 $(\lambda x y. y = x) = (=)$   
**by** auto

**lemma** trans\_from\_refine:  
 $(\text{trans\_impl}, \text{trans\_from}) \in \text{fun\_rel\_syn } \text{loc\_rel } (\text{list\_rel } (\text{Id} \times_r \text{Id} \times_r \text{Id} \times_r \text{loc\_rel}))$   
**proof** –  
**have** [rel2p]:  
 $\text{rel2p } \text{loc\_rel} = (\lambda x y. ((\lambda x y. \text{list\_all2 } (=) x y \wedge \text{length } x = n\_ps) \times_R \text{state\_rel}) \text{ y } x)$   
**unfolding** loc\_rel\_def rel2p\_def **by** (intro ext) (auto simp: list\_rel\_eq)  
**have** rel2p (fun\_rel\_syn  
 $\{(L', s'), (L, s) \mid L \text{ s } L' \text{ s}'. L' = L \wedge \text{length } L = n\_ps \wedge \text{state\_rel } s \text{ s}'\}$   
 $(\langle \text{Id} \times_r \text{Id} \times_r \text{Id} \times_r \text{loc\_rel} \rangle \text{list\_rel}))$   
 $\text{trans\_impl trans\_from}$   
**unfolding** rel2p\_def rel2p\_def state\_rel\_def  
**by** (intro rel\_funI trans\_from\_transfer [THEN rel\_funD, THEN list\_all2\_swap])  
 $(\text{auto simp: list\_rel\_eq state\_rel\_def swap\_rel\_prod swap\_eq})$   
**then show** ?thesis  
**unfolding** rel2p\_def relAPP\_def loc\_rel\_def .  
**qed**

**lemma** trans\_from\_correct:  
 $(\text{trans\_from}, \text{trans\_prod}) \in \text{transition\_rel } \text{states}'$   
**using** int\_trans\_from\_correct bin\_trans\_from\_correct broad\_trans\_from\_correct  
**unfolding** trans\_from\_def trans\_prod\_def transition\_rel\_def **by** auto

**lemma** states'\_alt\_def:  
 $\text{states}' = \{(L, s). L \in \text{states} \wedge \text{bounded\_bounds } s\}$   
**unfolding** states'\_def bounded\_def dom\_bounds\_eq **by** auto

**lemma** states'\_alt\_def2:  
 $\text{states}' = \{(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n\_vs\} \wedge \text{check\_bounded } s\}$   
**proof** –  
**have**  $\text{states}' = \{(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n\_vs\} \wedge \text{bounded\_bounds } s\}$   
**unfolding** states'\_def bounded\_def dom\_bounds\_eq **by** auto  
**then show** ?thesis  
**by** (auto simp: check\_bounded\_iff)  
**qed**

**lemma** trans\_prod\_states'\_inv:  
 $l' \in \text{states}' \text{ if } (l, g, a, r, l') \in \text{trans\_prod } l \in \text{states}'$

```

using that unfolding states'_alt_def
by (cases l') (auto dest: trans_prod_bounded_inv trans_prod_states_inv)

lemma prod_ta_states'_inv:
  l' ∈ states' if prod_ta ⊢ l  $\longrightarrow^{g,a,r}$  l' l ∈ states'
  using that by simp (rule trans_prod_states'_inv)

lemma dom_eq_transfer [transfer_rule]:
  (state_rel  $\implies$  (=)) ( $\lambda s. \text{dom } s = \{0..<n\_vs\}$ ) ( $\lambda s. \text{length } s = n\_vs$ )
  by (rule rel_funI) (auto simp: state_rel_def)

lemma states'_memi_correct:
  (states'_memi, ( $\lambda l. l \in \text{states}'$ ))  $\in \text{loc\_rel} \rightarrow \text{bool\_rel}$ 
proof –
  define t where t s  $\equiv \text{dom } s = \{0..<n\_vs\}$  for s :: nat  $\rightarrow$  int
  define ti where ti s  $\equiv \text{length } s = n\_vs$  for s :: int list
  let ?R =  $\lambda x y. (\text{eq\_onp } (\lambda L. \text{length } L = n\_ps) \times_R \text{state\_rel}) y x$ 
  note [transfer_rule] = dom_eq_transfer[folded t_def ti_def]
  have [p2rel]: p2rel (( $\lambda x y. (\text{eq\_onp } (\lambda L. \text{length } L = n\_ps) \times_R \text{state\_rel}) y x$ )) = loc_rel
    by (auto simp: eq_onp_def p2rel_def loc_rel_def)
  have *: ( $\lambda(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n\_vs\} \wedge \text{check\_bounded } s$ ) = ( $\lambda l. l \in \text{states}'$ )
    by (intro ext) (auto simp: states'_alt_def2)
  have (((=)  $\times_R \text{state\_rel}$ )  $\implies$  (=))
    ( $\lambda(L, s). L \in \text{states} \wedge t s \wedge \text{check\_bounded } s$ ) ( $\lambda(L, s). L \in \text{states} \wedge ti s \wedge \text{check\_bounded } s$ )
  s)
  by transfer_prover
  then have
    (((=)  $\times_R \text{state\_rel}$ )  $\implies$  (=))
    ( $\lambda(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n\_vs\} \wedge \text{check\_bounded } s$ )
    states'_memi
    unfolding t_def ti_def states'_memi_def .
  then have [p2rel]: (?R  $\implies$  (=)) states'_memi ( $\lambda l. l \in \text{states}'$ )
    unfolding * by (intro rel_funI) (auto simp: eq_onp_def elim!: rel_funE)
  then have (states'_memi, ( $\lambda l. l \in \text{states}'$ ))  $\in p2rel$  (?R  $\implies$  (=))
    unfolding p2rel_def by simp
  then show ?thesis
    unfolding p2rel .
qed

end

end

end

theory Simple_Network_Language_Model_Checking
  imports Munta_Base.Temporal_Logics
    Simple_Network_Language_Impl_Refine
    Munta_Model_Checker.UPPAAL_Model_Checking
begin

```

## 7 Product Bisimulation

```

no_notation State_Networks.step_sn ( $\_ \vdash \langle \_, \_, \_ \rangle \rightarrow \_ \langle \_, \_, \_ \rangle$ ) [61,61,61,61,61] 61)

```

We have proved the necessary theorems already but we need to lift it to the case where delay and action transitions are strictly alternating.

**inductive** *step\_u'* ::

```

('a, 's, 'c, 't :: time, 'x, 'v :: linorder) nta ⇒ 's list ⇒ ('x → 'v) ⇒ ('c, 't) cval
⇒ 's list ⇒ ('x → 'v) ⇒ ('c, 't) cval ⇒ bool
(⟦_⟧ ⊢ ⟨_, _, _⟩ → ⟨_, _, _⟩ [61,61,61,61] 61) where
  A ⊢ ⟨L, s, u⟩ → ⟨L'', s'', u''⟩ if
  A ⊢ ⟨L, s, u⟩ →Del ⟨L', s', u'⟩
  a ≠ Del
  A ⊢ ⟨L', s', u'⟩ →a ⟨L'', s'', u''⟩

```

**inductive\_cases** *step'\_elims*:  $A \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$

**inductive\_cases** *step\_u'\_elims*:  $A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$

**theorem** *Bisimulation\_Invariant\_strong\_intro*:

```

fixes A :: 'a ⇒ 'a ⇒ bool
and P :: 'a ⇒ bool
and B :: 'a ⇒ 'a ⇒ bool
assumes ⋀ a b. A a b ⇒ P a ⇒ B a b
and ⋀ a b. B a b ⇒ P a ⇒ A a b
and ⋀ a b. P a ⇒ A a b ⇒ P b
shows Bisimulation_Invariant A B (=) P P
apply standard
subgoal A for a b a'
  by (blast intro: assms)
subgoal B for a b a'
  by (blast intro: assms)
subgoal C for a b
  by (rule assms)
subgoal D for a b
  by (rule C, assumption) (rule assms)
done

```

**context** *Prod\_TA\_Defs*

**begin**

**definition**

*all\_prop L s = (L ∈ states ∧ bounded bounds s)*

**lemma** *all\_prop\_boundedD[dest]*:

*bounded bounds s if all\_prop L s*  
**using that unfolding all\_prop\_def ..**

**lemma** *all\_prop\_statesD[dest]*:

*L ∈ states if all\_prop L s*  
**using that unfolding all\_prop\_def ..**

**end**

**context** *Prod\_TA*

**begin**

```

lemma prod_action_step_not_eq_delay:
   $a \neq \text{Del}$  if  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ 
  using that
  apply cases
  unfolding trans_of_def
  unfolding prod_ta_def trans_prod_def
  by (auto simp: trans_int_def trans_bin_def trans_broad_def)

end

locale Prod_TA_urge =
  Prod_TA + assumes urgency_removed:  $\forall N \in \text{set } (\text{fst } (\text{snd } A)). \text{urgent } N = \{\}$ 
begin

lemma prod_all_prop_inv:
   $\text{all\_prop } L' s' \text{ if } \text{all\_prop } L s \text{ prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  using that unfolding all_prop_def
  by (auto elim: bounded_inv states_inv simp: step_iff[symmetric, OF _ _ urgency_removed])

lemma prod_all_prop_inv':
   $\text{all\_prop } L' s' \text{ if } \text{all\_prop } L s \text{ prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  using that by (auto intro: prod_all_prop_inv elim!: step'_elims)

interpretation prod_bisim:
  Bisimulation_Invariant
   $(\lambda (L, s, u) (L', s', u'). \text{prod\_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$ 
   $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$ 
  (=)
   $(\lambda (L, s, u). \text{all\_prop } L s)$ 
   $(\lambda (L, s, u). \text{all\_prop } L s)$ 
  apply (rule Bisimulation_Invariant_strong_intro; clarsimp)
  subgoal
    by (auto intro: step_u'.intros simp: all_prop_def
      dest: action_sound prod_action_step_not_eq_delay delay_sound[OF _ _ _ urgency_removed]
      elim!: step'_elims)
  subgoal
    by (auto 4 4 dest: prod_all_prop_inv action_complete elim: delay_complete elim!: step_u'_elims)
  subgoal
    by (rule prod_all_prop_inv')
  done

lemmas prod_bisim_intro = prod_bisim.Bisimulation_Invariant_axioms

end

```

## 8 The Final Semantics

State formulas

```

datatype ('n, 's, 'a, 'b) sexp =
  true |

```

— Boolean connectives  
 $\text{not } ('n, 's, 'a, 'b) \text{ sexp} \mid \text{and } ('n, 's, 'a, 'b) \text{ sexp } ('n, 's, 'a, 'b) \text{ sexp} \mid$   
 $\text{or } ('n, 's, 'a, 'b) \text{ sexp } ('n, 's, 'a, 'b) \text{ sexp} \mid \text{imply } ('n, 's, 'a, 'b) \text{ sexp } ('n, 's, 'a, 'b) \text{ sexp} \mid$   
— Does var  $a$  equal  $x$ ?  
 $\text{eq } 'a \ 'b \mid$   
 $\text{le } 'a \ 'b \mid$   
 $\text{lt } 'a \ 'b \mid$   
 $\text{ge } 'a \ 'b \mid$   
 $\text{gt } 'a \ 'b \mid$   
— Is procces  $i$  in location  $l$ ?  
 $\text{loc } 'n \ 's$

**datatype**  $('n, 's, 'a, 'b) \text{ formula} =$   
 $\text{EX } ('n, 's, 'a, 'b) \text{ sexp} \mid \text{EG } ('n, 's, 'a, 'b) \text{ sexp}$   
 $\mid \text{AX } ('n, 's, 'a, 'b) \text{ sexp} \mid \text{AG } ('n, 's, 'a, 'b) \text{ sexp}$   
 $\mid \text{Leadsto } ('n, 's, 'a, 'b) \text{ sexp } ('n, 's, 'a, 'b) \text{ sexp}$

**fun**  $\text{check\_sexp} :: (\text{nat}, 's, 'a, 'b :: \text{linorder}) \text{ sexp} \Rightarrow 's \text{ list} \Rightarrow ('a \Rightarrow 'b) \Rightarrow \text{bool}$  **where**  
 $\text{check\_sexp sexp.true } \_ \_ \longleftrightarrow \text{True} \mid$   
 $\text{check\_sexp } (\text{not } e) \ L \ s \longleftrightarrow \neg \text{check\_sexp } e \ L \ s \mid$   
 $\text{check\_sexp } (\text{and } e1 \ e2) \ L \ s \longleftrightarrow \text{check\_sexp } e1 \ L \ s \wedge \text{check\_sexp } e2 \ L \ s \mid$   
 $\text{check\_sexp } (\text{sexp.or } e1 \ e2) \ L \ s \longleftrightarrow \text{check\_sexp } e1 \ L \ s \vee \text{check\_sexp } e2 \ L \ s \mid$   
 $\text{check\_sexp } (\text{imply } e1 \ e2) \ L \ s \longleftrightarrow \text{check\_sexp } e1 \ L \ s \longrightarrow \text{check\_sexp } e2 \ L \ s \mid$   
 $\text{check\_sexp } (\text{eq } i \ x) \ L \ s \longleftrightarrow s \ i = x \mid$   
 $\text{check\_sexp } (\text{le } i \ x) \ L \ s \longleftrightarrow s \ i \leq x \mid$   
 $\text{check\_sexp } (\text{lt } i \ x) \ L \ s \longleftrightarrow s \ i < x \mid$   
 $\text{check\_sexp } (\text{ge } i \ x) \ L \ s \longleftrightarrow s \ i \geq x \mid$   
 $\text{check\_sexp } (\text{gt } i \ x) \ L \ s \longleftrightarrow s \ i > x \mid$   
 $\text{check\_sexp } (\text{loc } i \ x) \ L \ s \longleftrightarrow L ! i = x$

**fun**  $\text{locs\_of\_sexp} :: ('n, 's, 'a, 'b) \text{ sexp} \Rightarrow 'n \text{ set}$  **where**  
 $\text{locs\_of\_sexp } (\text{not } e) = \text{locs\_of\_sexp } e \mid$   
 $\text{locs\_of\_sexp } (\text{and } e1 \ e2) = \text{locs\_of\_sexp } e1 \cup \text{locs\_of\_sexp } e2 \mid$   
 $\text{locs\_of\_sexp } (\text{sexp.or } e1 \ e2) = \text{locs\_of\_sexp } e1 \cup \text{locs\_of\_sexp } e2 \mid$   
 $\text{locs\_of\_sexp } (\text{imply } e1 \ e2) = \text{locs\_of\_sexp } e1 \cup \text{locs\_of\_sexp } e2 \mid$   
 $\text{locs\_of\_sexp } (\text{loc } i \ x) = \{i\} \mid$   
 $\text{locs\_of\_sexp } \_ = \{\}$

**fun**  $\text{vars\_of\_sexp} :: ('n, 's, 'a, 'b) \text{ sexp} \Rightarrow 'a \text{ set}$  **where**  
 $\text{vars\_of\_sexp } (\text{not } e) = \text{vars\_of\_sexp } e \mid$   
 $\text{vars\_of\_sexp } (\text{and } e1 \ e2) = \text{vars\_of\_sexp } e1 \cup \text{vars\_of\_sexp } e2 \mid$   
 $\text{vars\_of\_sexp } (\text{sexp.or } e1 \ e2) = \text{vars\_of\_sexp } e1 \cup \text{vars\_of\_sexp } e2 \mid$   
 $\text{vars\_of\_sexp } (\text{imply } e1 \ e2) = \text{vars\_of\_sexp } e1 \cup \text{vars\_of\_sexp } e2 \mid$   
 $\text{vars\_of\_sexp } (\text{eq } i \ x) = \{i\} \mid$   
 $\text{vars\_of\_sexp } (\text{lt } i \ x) = \{i\} \mid$   
 $\text{vars\_of\_sexp } (\text{le } i \ x) = \{i\} \mid$   
 $\text{vars\_of\_sexp } (\text{ge } i \ x) = \{i\} \mid$   
 $\text{vars\_of\_sexp } (\text{gt } i \ x) = \{i\} \mid$   
 $\text{vars\_of\_sexp } \_ = \{\}$

**fun**  $\text{locs\_of\_formula} :: ('n, 's, 'a, 'b) \text{ formula} \Rightarrow 'n \text{ set}$  **where**  
 $\text{locs\_of\_formula } (\text{formula.EX } \varphi) = \text{locs\_of\_sexp } \varphi \mid$   
 $\text{locs\_of\_formula } (\text{EG } \varphi) = \text{locs\_of\_sexp } \varphi \mid$   
 $\text{locs\_of\_formula } (\text{AX } \varphi) = \text{locs\_of\_sexp } \varphi \mid$

$locs\_of\_formula (AG \varphi) = locs\_of\_sexp \varphi \mid$   
 $locs\_of\_formula (Leadsto \varphi \psi) = locs\_of\_sexp \varphi \cup locs\_of\_sexp \psi$

**fun** *vars\_of\_formula* :: ('n, 's, 'a, 'b) formula  $\Rightarrow$  'a set **where**  
 $vars\_of\_formula (formula.EX \varphi) = vars\_of\_sexp \varphi \mid$   
 $vars\_of\_formula (EG \varphi) = vars\_of\_sexp \varphi \mid$   
 $vars\_of\_formula (AX \varphi) = vars\_of\_sexp \varphi \mid$   
 $vars\_of\_formula (AG \varphi) = vars\_of\_sexp \varphi \mid$   
 $vars\_of\_formula (Leadsto \varphi \psi) = vars\_of\_sexp \varphi \cup vars\_of\_sexp \psi$

**fun** *hd\_of\_formula* :: (nat, 's, 'a, 'b) formula  $\Rightarrow$  's list  $\Rightarrow$  ('a  $\Rightarrow$  'b :: linorder)  $\Rightarrow$  bool **where**  
 $hd\_of\_formula (formula.EX \varphi) L s = check\_sexp \varphi L s \mid$   
 $hd\_of\_formula (EG \varphi) L s = check\_sexp \varphi L s \mid$   
 $hd\_of\_formula (AX \varphi) L s = Not (check\_sexp \varphi L s) \mid$   
 $hd\_of\_formula (AG \varphi) L s = Not (check\_sexp \varphi L s) \mid$   
 $hd\_of\_formula (Leadsto \varphi \_) L s = check\_sexp \varphi L s$

**definition** *models* ( $\_, \_ \models \_ [61, 61] 61$ ) **where**

$A, a_0 \models \Phi \equiv (case \Phi \text{ of}$   
 $formula.EX \varphi \Rightarrow$   
 $Graph\_Defs.Ex\_ev$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$   
 $\mid formula.EG \varphi \Rightarrow$   
 $Graph\_Defs.Ex\_alw$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$   
 $\mid formula.AX \varphi \Rightarrow$   
 $Graph\_Defs.Alw\_ev$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$   
 $\mid formula.AG \varphi \Rightarrow$   
 $Graph\_Defs.Alw\_alw$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$   
 $\mid formula.Leadsto \varphi \psi \Rightarrow$   
 $Graph\_Defs.leadsto$   
 $(\lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$   
 $(\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$   
 $(\lambda (L, s, \_). check\_sexp \psi L (the\ o\ s))$   
 $) a_0$

**lemmas** *models\_iff* = *models\_def*[unfolded *Graph\_Defs.Ex\_alw\_iff* *Graph\_Defs.Alw\_alw\_iff*]

**definition** *prop\_of* **where**

$prop\_of \varphi = PropC (\lambda (L, s, \_). check\_sexp \varphi L (the\ o\ s))$

**fun** *ctl\_of* **where**

$ctl\_of (formula.EX \varphi) = ctl\_formula.EX (prop\_of \varphi)$   
 $\mid ctl\_of (formula.EG \varphi) = ctl\_formula.EG (prop\_of \varphi)$   
 $\mid ctl\_of (formula.AX \varphi) = ctl\_formula.AX (prop\_of \varphi)$   
 $\mid ctl\_of (formula.AG \varphi) = ctl\_formula.AG (prop\_of \varphi)$   
 $\mid ctl\_of (formula.Leadsto \varphi \psi) =$   
 $ctl\_formula.AG (ctl\_formula.ImpliesC (prop\_of \varphi) (ctl\_formula.AX (prop\_of \psi)))$

**context**

**fixes**  $A :: ('a, 's, 'c, 't :: \text{time}, 'x, 'v :: \text{linorder}) \text{ nta}$

**begin**

**interpretation**  $\text{Graph\_Defs } \lambda (L, s, u) (L', s', u'). A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle .$

**lemma**  $\text{models\_ctl\_iff}:$

$A, a_0 \models \Phi \longleftrightarrow \text{models\_ctl } (\text{ctl\_of } \Phi) a_0$

**by** ( $\text{cases } \Phi$ ) ( $\text{simp\_all add: models\_def prop\_of\_def leadsto\_def}$ )

**end**

**fun**  $\text{check\_sexpi} :: (\text{nat}, 's, \text{nat}, \text{int}) \text{ sexp} \Rightarrow 's \text{ list} \Rightarrow \text{int list} \Rightarrow \text{bool}$  **where**

$\text{check\_sexpi } \text{sexp.true } \_ \_ \longleftrightarrow \text{True} \mid$   
 $\text{check\_sexpi } (\text{not } e) L s \longleftrightarrow \neg \text{check\_sexpi } e L s \mid$   
 $\text{check\_sexpi } (\text{and } e1 e2) L s \longleftrightarrow \text{check\_sexpi } e1 L s \wedge \text{check\_sexpi } e2 L s \mid$   
 $\text{check\_sexpi } (\text{sexp.or } e1 e2) L s \longleftrightarrow \text{check\_sexpi } e1 L s \vee \text{check\_sexpi } e2 L s \mid$   
 $\text{check\_sexpi } (\text{imply } e1 e2) L s \longleftrightarrow \text{check\_sexpi } e1 L s \longrightarrow \text{check\_sexpi } e2 L s \mid$   
 $\text{check\_sexpi } (\text{eq } i x) L s \longleftrightarrow s ! i = x \mid$   
 $\text{check\_sexpi } (\text{le } i x) L s \longleftrightarrow s ! i \leq x \mid$   
 $\text{check\_sexpi } (\text{lt } i x) L s \longleftrightarrow s ! i < x \mid$   
 $\text{check\_sexpi } (\text{ge } i x) L s \longleftrightarrow s ! i \geq x \mid$   
 $\text{check\_sexpi } (\text{gt } i x) L s \longleftrightarrow s ! i > x \mid$   
 $\text{check\_sexpi } (\text{loc } i x) L s \longleftrightarrow L ! i = x$

**fun**  $\text{hd\_of\_formulai} :: (\text{nat}, 's, \text{nat}, \text{int}) \text{ formula} \Rightarrow 's \text{ list} \Rightarrow \text{int list} \Rightarrow \text{bool}$  **where**

$\text{hd\_of\_formulai } (\text{formula.EX } \varphi) L s = \text{check\_sexpi } \varphi L s \mid$   
 $\text{hd\_of\_formulai } (\text{EG } \varphi) L s = \text{check\_sexpi } \varphi L s \mid$   
 $\text{hd\_of\_formulai } (\text{AX } \varphi) L s = \text{Not } (\text{check\_sexpi } \varphi L s) \mid$   
 $\text{hd\_of\_formulai } (\text{AG } \varphi) L s = \text{Not } (\text{check\_sexpi } \varphi L s) \mid$   
 $\text{hd\_of\_formulai } (\text{Leadsto } \varphi \_) L s = \text{check\_sexpi } \varphi L s$

**lemma**  $\text{all\_prop\_weak\_cong}[\text{cong}]:$

$\text{Prod\_TA\_Defs.all\_prop } A = \text{Prod\_TA\_Defs.all\_prop } A$  **for**  $A$

**by** ( $\text{rule HOL.refl}$ )

**lemma**  $\text{equiv'}\_weak\_cong}[\text{cong}]:$

$\text{Bisimulation\_Invariant.equiv'} R P Q = \text{Bisimulation\_Invariant.equiv'} R P Q$  **for**  $R P Q$

**by** ( $\text{rule HOL.refl}$ )

**lemma**  $\text{equiv'}\_weak\_cong2}[\text{cong}]:$

$\text{Simulation\_Invariant.equiv'} R P Q = \text{Simulation\_Invariant.equiv'} R P Q$  **for**  $R P Q$

**by** ( $\text{rule HOL.refl}$ )

**lemma**  $\text{models\_ctl\_weak\_cong}[\text{cong}]:$

$\text{Graph\_Defs.models\_ctl } E = \text{Graph\_Defs.models\_ctl } E$  **for**  $E$

**by** ( $\text{rule HOL.refl}$ )

**lemma**  $\text{models\_path\_weak\_cong}[\text{cong}]:$

$\text{Graph\_Defs.models\_path } E = \text{Graph\_Defs.models\_path } E$  **for**  $E$

**by** ( $\text{rule HOL.refl}$ )

**lemma**  $\text{models\_state\_weak\_cong}[\text{cong}]:$

$\text{Graph\_Defs.models\_state } E = \text{Graph\_Defs.models\_state } E$  **for**  $E$

**by** ( $\text{rule HOL.refl}$ )

```

lemma models_ltl_weak_cong[cong]:
  Graph_Defs.models_ltlc E = Graph_Defs.models_ltlc E for E
  by (rule HOL.refl)
lemma models_weak_cong[cong]:
  models E = models E
  for E by (rule HOL.refl)

```

## 9 Instantiating the Model Checking Locale

```

locale Simple_Network_Impl_nat_urge =
  Simple_Network_Impl_nat + assumes no_urgency:  $\forall (\_, U, \_, \_) \in \text{set automata}. U = []$ 

```

This locale certifies that a given local clock ceiling is correct. Moreover, we certify that the vector of initial locations has outgoing transitions for each automaton, and that all variables of the initial state are in bounds.

```

locale Simple_Network_Impl_nat_ceiling_start_state =
  Simple_Network_Impl_nat_urge +
  fixes k :: nat list list list
  and L0 :: nat list
  and s0 :: (nat × int) list
  and formula :: (nat, nat, nat, int) formula
  and show_clock :: nat ⇒ string
  and show_state :: nat list × int list ⇒ string
assumes k_ceiling:
   $\forall i < n\_ps. \forall (l, g) \in \text{set} ((snd \circ snd \circ snd) (\text{automata} ! i)).$ 
   $\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x)$ 
   $\forall i < n\_ps. \forall (l, \_, g, \_) \in \text{set} ((fst \circ snd \circ snd) (\text{automata} ! i)).$ 
   $(\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x))$ 
and k_resets:
   $\forall i < n\_ps. \forall (l, b, g, a, upd, r, l') \in \text{set} ((fst \circ snd \circ snd) (\text{automata} ! i)).$ 
   $\forall c \in \{0..<m+1\} - \text{set } r. k ! i ! l' ! c \leq k ! i ! l ! c$ 
and k_length:
   $\text{length } k = n\_ps \ \forall i < n\_ps. \text{length } (k ! i) = \text{num\_states } i$ 
   $\forall xs \in \text{set } k. \forall xxs \in \text{set } xs. \text{length } xxs = m + 1$ 
and k_0:
   $\forall i < n\_ps. \forall l < \text{num\_states } i. k ! i ! l ! 0 = 0$ 
and inv_unambiguous:
   $\forall (\_, \_, \_, inv) \in \text{set automata}. \text{distinct } (\text{map } fst \ inv)$ 
and s0_bounded: bounded bounds (map_of s0)
and L0_len: length L0 = nps
and L0_has_trans:  $\forall i < n\_ps. L_0 ! i \in \text{fst } ' \text{set } ((fst \circ snd \circ snd) (\text{automata} ! i))$ 
and vars_of_formula: vars_of_formula formula ⊆ {0..nvs}

```

**begin**

The ceiling *k* is correct for each individual automaton in the network. We now construct a ceiling for the product automaton:

**definition**

```

k_fun l c ≡
  if (c > 0 ∧ c ≤ m) ∧ fst l ∈ states then Max {k ! i ! (fst l ! i) ! c | i . i < nps} else 0

```

**lemma** (*in* −) *default\_map\_of\_distinct*:

```

(k, default_map_of xs k) ∈ set xs ∪ {(k, x)} if distinct (map fst xs)

```

**unfolding** *default\_map\_of\_alt\_def* **by** *clarsimp* (*simp add: map\_of\_eq\_Some\_iff* [OF *that*])

**lemma** *N\_inv*:

$(L ! i, \text{inv } (N i) (L ! i)) \in \text{set } ((\text{snd } o \text{ snd } o \text{ snd}) (\text{automata } ! i)) \cup \{(L ! i, [])\}$

**if**  $i < n\_ps$

**unfolding** *N\_def comp\_def fst\_conv snd\_conv inv\_def*

**using** *that*

**apply** (*subst nth\_map*)

**apply** (*simp add: n\_ps\_def; fail*)

**apply** (*clarsimp split: prod.split simp: automaton\_of\_def*)

**subgoal for**  $\_ \_ \_ xs$

**using** *default\_map\_of\_distinct* [of  $xs$   $L ! i$  []] *inv\_unambiguous* *that*

**by** (*auto dest!: nth\_mem simp: n\_ps\_def*)

**done**

**lemma** (**in**  $\_$ ) *subset\_nat\_0\_atLeastLessThan\_conv*:

$S \subseteq \{0..<n::nat\} \longleftrightarrow (\forall x \in S. x < n)$

**by** *auto*

**lemma** *k\_ceiling\_rule*:

$m \leq \text{int } (k ! i ! l ! x)$

**if**  $i < n\_ps$   $(l, b, g, xx) \in \text{set } ((\text{fst } o \text{ snd } o \text{ snd}) (\text{automata } ! i))$

$(x, m) \in \text{collect\_clock\_pairs } g$

**for**  $i \ l \ x \ g \ xx$

**using** *that* *k\_ceiling*(2) **by** *fastforce*

**lemma** *k\_ceiling\_1*:

$\forall s. \forall L \in \text{states}. \forall (x, m) \in \text{clkp\_set prod\_ta } (L, s). m \leq k\_fun (L, s) x$

**proof** *safe*

**fix**  $L \ s \ c \ x$

**assume**  $\langle L \in \text{states} \rangle \langle (c, x) \in \text{Closure.clkp\_set prod\_ta } (L, s) \rangle$

**have**  $0 < c \leq m$

**proof**  $\_$

**from**  $\langle (c, x) \in \_ \rangle$  **have**  $(c, x) \in \text{Timed\_Automata.clkp\_set prod\_ta}$

**unfolding** *TA\_clkp\_set\_unfold* **by** *auto*

**with** *clock\_range* **show**  $0 < c \leq m$

**by** *auto*

**qed**

**with**  $\langle L \in \_ \rangle$  **have**  $k\_fun (L, s) c = \text{Max } \{k ! i ! (L ! i) ! c \mid i. i < n\_ps\}$

**unfolding** *k\_fun\_def* **by** *auto*

**have** *Max\_aux*:  $x \leq \text{int } (\text{Max } \{k ! i ! (L ! i) ! c \mid i. i < n\_ps\})$

**if**  $x \leq \text{int } (k ! p ! (L ! p) ! c)$   $p < n\_ps$  **for**  $p$

**proof**  $\_$

**from**  $\langle p < n\_ps \rangle$  **have**  $k ! p ! (L ! p) ! c \leq \text{Max } \{k ! i ! (L ! i) ! c \mid i. i < n\_ps\}$

**by** (*intro Max\_ge*) *auto*

**with**  $\langle x \leq \_ \rangle$  **show** *?thesis*

**by** *simp*

**qed**

**from**  $\langle (c, x) \in \_ \rangle$  **show**  $\langle x \leq k\_fun (L, s) c \rangle$

**unfolding** *clkp\_set\_def*

**proof** *safe*

**assume**  $\langle (c, x) \in \text{Closure.collect\_clki } (\text{inv\_of prod\_ta}) (L, s) \rangle$

**then show**  $\langle x \leq k\_fun (L, s) c \rangle$

**using** *k\_ceiling*(1) **unfolding** *collect\_clki\_def*  $\langle k\_fun (L, s) c = \_ \rangle$

```

    by (auto 0 8 dest: N_inv
        intro!: Max_aux simp: prod_inv_def collect_clock_pairs_def k_fun_def)
next
assume  $\langle (c, x) \in \text{Closure.collect\_clkt } (\text{trans\_of prod\_ta}) (L, s) \rangle$ 
then show  $\langle x \leq k\_fun (L, s) \ c \rangle$ 
  unfolding collect_clkt_def  $\langle k\_fun (L, s) \ c = \_ \rangle$ 
  apply (clarsimp simp: trans_prod_def collect_clock_pairs_def k_fun_def)
  apply safe
  subgoal
    using k_ceiling(2) unfolding trans_int_def
    apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
    apply (fastforce intro!: Max_aux simp: collect_clock_pairs_def)
    done
  subgoal
    using k_ceiling(2) unfolding trans_bin_def
    apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
    apply (erule disjE)
    apply (force intro!: Max_aux simp: collect_clock_pairs_def)+
    done
  subgoal
    using k_ceiling(2) unfolding trans_broad_def
    apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
    apply (erule disjE)
    apply (fastforce intro!: Max_aux simp: collect_clock_pairs_def)
    apply (erule bexE)
    apply (force intro!: Max_aux simp: collect_clock_pairs_def) — slow: 60s
    done
  done
qed
qed

lemma k_fun_mono':
   $k\_fun (L, s) \ c \leq k\_fun (L', s') \ c$  if
   $\forall i < n\_ps. k!i! (L!i)! \ c \leq k!i! (L'!i)! \ c$   $L \in \text{states}$   $L' \in \text{states}$ 
  using that unfolding k_fun_def
  apply clarsimp
  apply (cases n_ps = 0)
  apply (simp; fail)
  apply (rule Max.boundedI)
  apply (simp; fail)
  apply blast
  apply safe
  subgoal for  $\_ \ i$ 
    by — (rule order.trans[where  $b = k!i! (L'!i)! \ c$ ], auto intro: Max_ge)
  done

lemma k_fun_mono:
   $\langle \text{Max } \{k!i! (L!i)! \ c \mid i. i < n\_ps\} \leq \text{Max } \{k!i! (L'!i)! \ c \mid i. i < n\_ps\} \rangle$ 
  if  $\langle \forall i < n\_ps. k!i! (L!i)! \ c \leq k!i! (L'!i)! \ c \rangle$ 
  apply (cases n_ps = 0)
  apply (simp; fail)
  apply (rule Max.boundedI)
  apply (simp; fail)
  apply blast

```

**apply** *safe*  
**subgoal for**  $\_ i$   
    **using** *that* **by**  $\text{-- (rule order.trans[where } b = k ! i ! (L' ! i) ! c], \text{ auto intro: Max\_ge})}$   
**done**

**lemma** (**in**  $\text{--}$ ) *fold\_upds\_aux1*:  
    *fold*  $(\lambda p L. L[p := g p]) \text{ ps xs ! } i = \text{xs ! } i$  **if**  $\langle i \notin \text{set ps} \rangle$   
    **using** *that* **by** (*induction ps arbitrary: xs*) *auto*

**lemma** (**in**  $\text{--}$ ) *fold\_upds\_aux2*:  
    *fold*  $(\lambda p L. L[p := g p]) \text{ ps xs ! } i = g i$  **if**  $\langle \text{distinct ps} \rangle \langle i \in \text{set ps} \rangle \langle i < \text{length xs} \rangle$   
    **using** *that* **by** (*induction ps arbitrary: xs*) (*auto simp: fold\_upds\_aux1*)

**lemma** (**in**  $\text{--}$ ) *fold\_upds\_aux\_length*:  
    *length* (*fold*  $(\lambda p L. L[p := g p]) \text{ ps xs}$ ) = *length xs*  
    **by** (*induction ps arbitrary: xs*) *auto*

**lemma** *prod\_ta\_step\_statesD*:  
    **assumes**  $\text{prod\_ta} \vdash (L, s) \longrightarrow^{g,a,r} (L', s')$   
    **shows**  $L \in \text{states } L' \in \text{states}$   
    **using** *assms state\_set\_states* **by** (*fastforce dest: state\_setI1 state\_setI2*) $+$

**lemma** *k\_ceiling\_2*:  
     $\forall l g a r l'. \forall c \leq m. \text{prod\_ta} \vdash l \longrightarrow^{g,a,r} l' \wedge c \notin \text{set } r \longrightarrow k\_fun l' c \leq k\_fun l c$   
**proof** *safe*  
    **fix**  $L s g a r L' s' c$   
    **assume**  $A: \langle c \leq m \rangle \langle \text{prod\_ta} \vdash (L, s) \longrightarrow^{g,a,r} (L', s') \rangle \langle c \notin \text{set } r \rangle$   
    **then have**  $L \in \text{states } L' \in \text{states}$   
    **by**  $\text{-- (rule prod\_ta\_step\_statesD, assumption)}+$   
    **from**  $A$  **have**  $\langle \text{Max } \{k ! i ! (L' ! i) ! c \mid i . i < n\_ps\} \leq \text{Max } \{k ! i ! (L ! i) ! c \mid i . i < n\_ps\} \rangle$   
    **apply** *simp*  
    **unfolding** *trans\_prod\_def*  
    **apply** *safe*  
    **subgoal**  
    **using** *k\_resets*  
    **unfolding** *trans\_int\_def*  
    **apply** *clarsimp*  
    **apply** (*rule k\_fun\_mono*)  
    **apply** (*clarsimp simp: mem\_trans\_N\_iff L\_len subset\_nat\_0\_atLeastLessThan\_conv*)  
    **subgoal for**  $b f p aa l' i$   
    **by** (*cases p = i; force simp add: L\_len*)  
    **done**  
    **subgoal**  
    **using** *k\_resets*  
    **unfolding** *trans\_bin\_def*  
    **apply** *clarsimp*  
    **apply** (*rule k\_fun\_mono*)  
    **apply** (*clarsimp simp: mem\_trans\_N\_iff L\_len subset\_nat\_0\_atLeastLessThan\_conv*)  
    **subgoal for**  $\_ \_ p q b1 g1 f1 r1 l1' b2 g2 f2 r2 l2' i$   
    **by** (*cases p = i; cases q = i; force simp add: L\_len*)  
    **done**  
    **subgoal**  
    **using** *k\_resets*  
    **unfolding** *trans\_broad\_def*

```

apply clarsimp
apply (rule k_fun_mono)
apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
subgoal premises prems for s'a aa p b ga fra l' bs gs fs rs ls' ps i
proof (cases p = i)
  case True
    with  $\langle p \notin \_ \rangle \langle i < \_ \rangle \langle L \in \text{states} \rangle$  have ( $\text{fold } (\lambda p L. L[p := ls' p]) \text{ } ps \text{ } L$ )[ $p := l'$ ] !  $i = l'$ 
      by (simp add: L_len fold_upds_aux_length)
    with prems  $\langle p = i \rangle$  show ?thesis
      by (fastforce simp add: L_len)
  next
    case False
    then have *: ( $\text{fold } (\lambda p L. L[p := ls' p]) \text{ } ps \text{ } L$ )[ $p := l'$ ] !  $i$ 
      =  $\text{fold } (\lambda p L. L[p := ls' p]) \text{ } ps \text{ } L$  !  $i$ 
      by simp
    show ?thesis
    proof (cases i ∈ set ps)
      case True
        then have **:  $\text{fold } (\lambda p L. L[p := ls' p]) \text{ } ps \text{ } L$  !  $i = ls' i$ 
          using  $\langle \text{distinct } ps \rangle \langle i < n\_ps \rangle \langle L \in \text{states} \rangle$  by (auto simp: fold_upds_aux2)
        moreover have
          ( $L ! i, bs i, gs i, In aa, fs i, rs i, ls' i$ )  $\in \text{set } (fst (snd (snd (automata ! i))))$ 
          using  $\langle p \neq i \rangle$  True prems by fast
        moreover have  $c \in \{0..<Suc m\} - \text{set } (rs i)$ 
          using  $\langle p \neq i \rangle$  True prems by force
        ultimately show ?thesis
          using prems(2)  $\langle i < n\_ps \rangle$  by (auto 4 3 simp add: *)
      next
        case False
        with  $\langle p \neq i \rangle$  show ?thesis
          by (simp add: fold_upds_aux1)
    qed
  qed
done
done
with  $\langle L \in \text{states} \rangle \langle L' \in \text{states} \rangle \langle c \leq m \rangle$  show  $k\_fun (L', s') c \leq k\_fun (L, s) c$ 
by (auto simp: k_fun_def)
qed

```

**abbreviation** *F* **where**  $F \equiv \lambda(L, s). \text{hd\_of\_formula } \text{formula } L \text{ (the } o \text{ } s)$

**abbreviation** *Fi*  $\equiv \lambda(L, s). \text{hd\_of\_formula}_i \text{ formula } L \text{ } s$

**lemma** (*in Simple\_Network\_Impl\_nat*) *check\_sexp\_check\_sexp\_i*:

*check\_sexp e L (the o s)  $\longleftrightarrow$  check\_sexp\_i e L s'*

**if** *state\_rel s s' vars\_of\_sexp e*  $\subseteq \{0..<n\_vs\}$

**using** *that* **unfolding** *state\_rel\_def* **by** (*induction e*) *auto*

**lemma** (*in Simple\_Network\_Impl\_nat*) *hd\_of\_formula\_hd\_of\_formula\_i*:

*hd\_of\_formula  $\varphi$  L (the o s)  $\longleftrightarrow$  hd\_of\_formula\_i  $\varphi$  L s'*

**if** *state\_rel s s' vars\_of\_formula  $\varphi$*   $\subseteq \{0..<n\_vs\}$

**using** *that* **by** (*induction  $\varphi$* ) (*auto simp: check\_sexp\_check\_sexp\_i*)

```

lemma  $F\_Fi$ :
   $F\ l \longleftrightarrow Fi\ l'$  if  $(l', l) \in loc\_rel$ 
  using vars_of_formula that unfolding loc_rel_def by clarsimp (erule hd_of_formula_hd_of_formulai)

abbreviation  $l_0 \equiv (L_0, map\_of\ s_0)$ 
abbreviation  $s_0i \equiv map\ (the\ o\ map\_of\ s_0)\ [0..<n\_vs]$ 
abbreviation  $l_0i \equiv (L_0, s_0i)$ 

lemma state_rel_start:
  state_rel (map_of  $s_0$ )  $s_0i$ 
  using s_0_bounded unfolding state_rel_def bounded_def dom_bounds_eq by auto

lemma statesI:
   $L \in states$  if  $length\ L = n\_ps\ \forall i < n\_ps.\ L\ !\ i \in fst\ 'set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$ 
  using that unfolding states_def by (auto 4 3 simp: mem_trans_N_iff[symmetric])

lemma  $L_0\_states[simp, intro]$ :
   $L_0 \in states$ 
  using L_0_has_trans L_0_len by (auto intro: statesI)

lemma  $l_0\_states'[simp, intro]$ :
   $l_0 \in states'$ 
  using state_rel_start s_0_bounded unfolding states'_def state_rel_def by auto

sublocale reach: Reachability_Problem_Defs
  prod_ta
   $l_0$ 
   $m$ 
   $k\_fun$ 
   $F$ 
  .

lemma (in  $-$ ) collect_clk_state_setI:
  assumes  $(x, d) \in Closure.collect\_clk\ (trans\_of\ A)\ l$ 
  shows  $l \in state\_set\ (trans\_of\ A)\ l \in Simulation\_Graphs\_TA.state\_set\ A$ 
  using assms unfolding collect_clk_def by (auto simp: state_set_def)

lemma clkp_set_statesD:
  fixes  $x\ d$ 
  assumes  $(x, d) \in Closure.clkp\_set\ prod\_ta\ (L, s)$ 
  shows  $L \in states$ 
  using assms
  unfolding clkp_set_def collect_clki_def
  apply safe
  subgoal
    apply simp
    unfolding prod_inv_def
    unfolding collect_clock_pairs_def
    apply auto
    done
  apply (drule collect_clk_state_setI)
  using state_set_states by auto

```

```

sublocale reach1: Reachability_Problem
  prod_ta
  l0
  m
  k_fun
  F
apply standard
subgoal
  apply safe
  using k_ceiling_1
  subgoal for L s x m
    apply (cases L ∈ states)
    apply blast
    apply (auto dest: clkp_set_statesD simp: k_fun_def)
    done
  done
subgoal
  apply safe
  subgoal for L s g a r L' s' c
    apply (cases c ≤ m)
    using k_ceiling_2
    apply force
    apply (auto simp: k_fun_def)
    done
  done
subgoal
  by (simp add: k_fun_def)
subgoal
  by (simp add: k_fun_def)
done — slow: 60s

lemma states'_superset:
  {l0} ∪ Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod ⊆ states'
  (is {l0} ∪ ?S ⊆ states')
proof —
  have ?S ⊆ states'
proof safe
  fix L s
  assume (L, s) ∈ ?S
  then have L ∈ states
    using state_set_states[unfolded trans_of_prod state_set_eq] by blast
  moreover have bounded bounds s
    using ⟨(L, s) ∈ _⟩
    unfolding state_set_def
    unfolding trans_prod_def
    unfolding trans_int_def trans_bin_def trans_broad_def
    by auto
  ultimately show (L, s) ∈ states'
    by (auto simp: states'_alt_def)
qed
then show ?thesis
  by simp
qed

```

**definition**  $k\_i \equiv IArray \ (map \ (IArray \ o \ (map \ (IArray \ o \ map \ int))) \ k)$

**definition**

$k\_impl \equiv \lambda(l, \_). IArray \ (map \ (\lambda \ c. Max \ \{k\_i \ !! \ i \ !! \ (l \ ! \ i) \ !! \ c \mid i. \ i < n\_ps\}) \ [0..<m+1])$

**lemma** *Max\_int\_commute:*

$int \ (Max \ S) = Max \ (int \ 'S) \text{ if } finite \ S \ S \neq \{\}$

**apply** *(rule mono\_Max\_commute)*

**apply** *rule*

**using** *that* **by** *auto*

**lemma** *(in Simple\_Network\_Impl\_nat) n\_ps\_gt\_0: n\_ps > 0*

**using** *length\_automata\_eq\_n\_ps non\_empty* **by** *auto*

**lemma** *statesD:*

$L \ ! \ i \in fst \ 'set \ (fst \ (snd \ (snd \ (automata \ ! \ i))))$

$\vee L \ ! \ i \in (snd \ o \ snd \ o \ snd \ o \ snd \ o \ snd \ o \ snd) \ 'set \ (fst \ (snd \ (snd \ (automata \ ! \ i))))$

**if**  $L \in states \ length \ L = n\_ps \ i < n\_ps$

**using** *that* **unfolding** *states\_def*

**apply** *safe*

**apply** *—*

**apply** *(elim allE impE, assumption)*

**apply** *safe*

**apply** *(force simp: mem\_trans\_N\_iff)+*

**done**

**lemma** *k\_impl\_k\_fun:*

$k\_impl \ (L, \ s) = IArray \ (map \ (k\_fun \ (L, \ s)) \ [0..<m+1]) \text{ if } L \in states$

**proof** *—*

**define**  $k\_i2$  **where**  $k\_i2 \ i \ c = k\_i \ !! \ i \ !! \ (L \ ! \ i) \ !! \ c$  **for**  $i \ c$

**have**  $k\_i2\_k: k\_i2 \ i \ c = k \ ! \ i \ ! \ (L \ ! \ i) \ ! \ c$  **if**  $i < n\_ps \ c \leq m$  **for**  $i \ c$

**proof** *—*

**have**  $i < length \ k$

**by** *(simp add: k\_length(1) that(1))*

**moreover** **have**  $L \ ! \ i < length \ (k \ ! \ i)$

**using**  $L\_i\_len[OF \ \langle L \in states \rangle] \ k\_length(2) \ \langle i < n\_ps \rangle$  **by** *auto*

**moreover** **have**  $c < length \ (k \ ! \ i \ ! \ (L \ ! \ i))$

**using**  $k\_length(3) \ \langle c \leq m \rangle \ \langle i < length \ k \rangle \ \langle L \ ! \ i < length \ (k \ ! \ i) \rangle$  **by** *(auto dest: nth\_mem)*

**ultimately** **show** *?thesis*

**unfolding**  $k\_i2\_def \ k\_i\_def$  **by** *simp*

**qed**

**have**  $k\_impl\_alt\_def: k\_impl \ (L, \ s) = IArray \ (map \ (\lambda \ c. Max \ \{k\_i2 \ i \ c \mid i. \ i < n\_ps\}) \ [0..<m+1])$

**unfolding**  $k\_impl\_def \ k\_i2\_def$  **by** *auto*

**have**  $Max\_cong: Max \ \{f \ i \mid i. \ i < n\_ps\} = Max \ \{g \ i \mid i. \ i < n\_ps\}$

**if**  $\bigwedge i. \ i < n\_ps \implies f \ i = g \ i$  **for**  $f \ g :: nat \Rightarrow int$

**by** *(rule arg\_cong[where f = Max]) (force simp: that)*

**from** *that n\_ps\_gt\_0* **show** *?thesis*

**unfolding**  $k\_impl\_alt\_def$

**unfolding**  $k\_i2\_def[symmetric]$

**apply** *(clarsimp simp: k\_fun\_def k\_i2\_k cong: Max\_cong)*

```

apply safe
subgoal
  by (subst Max_int_commute; force simp: setcompr_eq_image image_comp comp_def)
subgoal
  using k_0 L_i_len[OF _ ⟨L ∈ states⟩] by (intro linorder_class.Max_eqI) auto
subgoal
  by (subst Max_int_commute; force simp: setcompr_eq_image image_comp comp_def)
done
qed

```

```

sublocale impl: Reachability_Problem_Impl
  where trans_fun = trans_from
  and inv_fun = inv_fun
  and F_fun = Fi
  and ceiling = k_impl
  and A = prod_ta
  and l0 = l0
  and l0i = l0i
  and F = PR_CONST F
  and n = m
  and k = k_fun
  and trans_impl = trans_impl
  and states' = states'
  and loc_rel = loc_rel
  apply standard

```

```

subgoal
  unfolding trans_of_prod by (rule trans_from_correct)

```

```

subgoal
  apply (rule trans_from_refine)
done

```

```

subgoal
  unfolding trans_of_prod
  by (rule set_mp[OF _ inv_fun_inv_of'[where R = loc_rel and S = {(s, s'). state_rel s' s}]])
  (auto simp: loc_rel_def)

```

```

subgoal
  using states'_superset by simp

```

```

subgoal
  using state_rel_start unfolding loc_rel_def by auto

```

```

subgoal for l li li'
  unfolding trans_of_prod by (rule state_rel_left_unique)

```

```

subgoal for l l' li
  unfolding trans_of_prod by (rule state_rel_right_unique)

subgoal
  unfolding inv_rel_def using L0_states
  by (auto simp: loc_rel_def state_rel_def reach.k'_def k_fun_def k_impl_k_fun)

subgoal
  unfolding inv_rel_def by (clarsimp dest!: F_Fi)

done — slow: 50s

end

no_notation UPPAAL_Model_Checking.models (_, _  $\models$  _ [61,61] 61)

context Reachability_Problem_Impl
begin

lemma F_reachable_correct:
  op.F_reachable
 $\longleftrightarrow (\exists l'. \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l'))$ 
  using E_op''.E_from_op_reachability_check[symmetric] reachability_check_new
  unfolding E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
  unfolding F_rel_def by auto

lemma E_op''.F_reachable_correct:
  E_op''.F_reachable
 $\longleftrightarrow (\exists l'. \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l'))$ 
  using E_op''.E_from_op_reachability_check[symmetric] reachability_check_new
  unfolding E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
  unfolding F_rel_def by auto

lemma Ex_ev_impl_hnr:
  assumes  $\forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow \neg \text{deadlock } (l_0, u_0)$ 
  shows

    (uncurry0
      (pw_impl (return  $\circ$  fst) state_copy_impl tracei subsumes_impl a0_impl F_impl succs_impl
        emptiness_check_impl),
      uncurry0 (SPEC ( $\lambda r. (r \longleftrightarrow (\forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow \text{Ex\_ev } (\lambda(l, \_). F \ l) (l_0, u_0))$ ))))))
     $\in \text{unit\_assn}^k \rightarrow_a \text{bool\_assn}$ 

proof -
  interpret Bisimulation_Invariant
    ( $\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
    ( $\lambda(l, u) (l', u'). \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
    ( $\lambda(l, u) (l', u'). l' = l \wedge (\forall c. c \in \text{clk\_set } (A) \longrightarrow u \ c = u' \ c)$ )
    ( $\lambda \_. \text{True}$ ) ( $\lambda \_. \text{True}$ )

```

```

    apply (rule ta_bisimulation)
  done
define spec where spec = ( $\forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow Ex\_ev (\lambda(l, \_). F \ l) (l_0, u_0)$ )
have *:  $E\_op''.F\_reachable \longleftrightarrow spec$ 
  unfolding spec_def
  unfolding  $E\_op''\_F\_reachable\_correct$ 
proof safe
  fix  $l' :: \langle 's \rangle$  and  $u_0 :: \langle nat \Rightarrow real \rangle$ 
  assume
     $\langle \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. conv\_A \ A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l') \rangle$  and
     $\langle \forall c \in \{1..n\}. u_0 \ c = 0 \rangle$ 
  then show  $\langle Ex\_ev (\lambda(l, \_). F \ l) (l_0, u_0) \rangle$ 
    using assms by (subst  $Ex\_ev$ ; unfold reaches_steps'[symmetric]) blast+
next
  assume  $\langle \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow Ex\_ev (\lambda(l, \_). F \ l) (l_0, u_0) \rangle$ 
  then have  $Ex\_ev (\lambda(l, \_). F \ l) (l_0, \lambda_. 0)$ 
    by auto
  then obtain  $l' \ u'$  where  $conv\_A \ A \vdash' \langle l_0, (\lambda_. 0) \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l'$ 
    apply (subst (asm)  $Ex\_ev$ )
    using assms
    unfolding reaches_steps'[symmetric]
    by auto
  then show  $\langle \exists l'. \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. conv\_A \ A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l') \rangle$ 
proof (inst_existentials  $l'$ , safe, unfold reaches_steps'[symmetric])
  fix  $u_0 :: \langle nat \Rightarrow real \rangle$ 
  assume  $\langle reaches (l_0, \lambda_. 0) (l', u') \rangle$  and  $\langle F \ l' \rangle$  and  $\langle \forall c \in \{1..n\}. u_0 \ c = 0 \rangle$ 
  then have  $equiv' (l_0, \lambda_. 0) (l_0, u_0)$ 
    unfolding equiv'_def using clocks_I[of  $\lambda_. 0 \ u_0$ ] by auto
  with  $\langle reaches \_ \_ \rangle \langle F \ l' \rangle$  show  $\langle \exists u'. reaches (l_0, u_0) (l', u') \wedge F \ l' \rangle$ 
    by - (drule (1) bisim.A_B.simulation_reaches, unfold equiv'_def, auto)
qed
qed
show ?thesis
  unfolding spec_def[symmetric] using pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
  by sepref_to_hoare (sep_auto simp: *)
qed

end

context Simple_Network_Impl_nat_urge
begin

sublocale conv: Prod_TA_urge
  (set broadcast, map (Simple_Network_Language.conv_A o automaton_of) automata, map_of
  bounds')
  by standard
  (use no_urgency in  $\langle auto \ simp: Simple\_Network\_Language.conv\_A\_def \ automaton\_of\_def \ urgent\_def \rangle$ )

abbreviation  $A \equiv (set \ broadcast, \ map \ automaton\_of \ automata, \ map\_of \ bounds')$ 

interpretation conv_eq_bisim:

```

```

Bisimulation_Invariant
(λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
(λ(L, s, u) (L', s', u'). conv.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
(λ((L, s), u) (L', s', u'). L = L' ∧ u = u' ∧ s = s')
(λ((L, s), u). conv.all_prop L s)
(λ(L, s, u). conv.all_prop L s)
proof goal_cases
  case 1
  interpret Bisimulation_Invariant
(λ(L, s, u) (L', s', u'). conv_A prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
(λ(L, s, u) (L', s', u'). conv.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
(=)
(λ(L, s, u). conv.all_prop L s)
(λ(L, s, u). conv.all_prop L s)
by (rule Bisimulation_Invariant_strong_intro)
  (auto simp: conv_prod_ta elim: conv.prod_all_prop_inv')
show ?case
  by standard (auto simp: conv_prod_ta elim: conv.prod_all_prop_inv')
qed

interpretation Bisimulation_Invariant
(λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
(λ(L, s, u) (L', s', u'). Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
(λ((L, s), u) (L', s', u'). L = L' ∧ u = u' ∧ s = s')
(λ((L, s), u). conv.all_prop L s)
(λ(L, s, u). conv.all_prop L s)
unfolding conv_alt_def
apply (rule Bisimulation_Invariant_sim_replace, rule Bisimulation_Invariant_composition)
  apply (rule conv_eq_bisim.Bisimulation_Invariant_axioms conv.prod_bisim_intro)+
apply auto
done

lemmas prod_bisim = Bisimulation_Invariant_axioms

lemmas deadlock_iff = deadlock_iff

lemma conv_all_prop:
  conv.all_prop = all_prop
  unfolding conv.all_prop_def all_prop_def by simp

definition prop_of2 where
  prop_of2 φ = PropC (λ((L, s), _). check_sexp φ L (the o s))

fun ctl_of2 where
  ctl_of2 (formula.EX φ) = ctl_formula.EX (prop_of2 φ)
| ctl_of2 (formula.EG φ) = ctl_formula.EG (prop_of2 φ)
| ctl_of2 (formula.AX φ) = ctl_formula.AX (prop_of2 φ)
| ctl_of2 (formula.AG φ) = ctl_formula.AG (prop_of2 φ)
| ctl_of2 (formula.Leadsto φ ψ) =
  ctl_formula.AG (ctl_formula.ImpliesC (prop_of2 φ) (ctl_formula.AX (prop_of2 ψ)))

lemma models_correct:
  Simple_Network_Language.conv A, (L0, s0, u0) ⊨ Φ = (case Φ of
    formula.EX φ ⇒

```

```

    Graph_Defs.Ex_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
  | formula.EG φ ⇒
    Not o Graph_Defs.Alw_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). ¬ check_sexp φ L (the o s))
  | formula.AX φ ⇒
    Graph_Defs.Alw_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
  | formula.AG φ ⇒
    Not o Graph_Defs.Ex_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). ¬ check_sexp φ L (the o s))
  | formula.Leadsto φ ψ ⇒
    Graph_Defs.leadsto
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
      (λ ((L, s), _). check_sexp ψ L (the o s))
  ) ((L0, s0), u0)
  if L0 ∈ states Simple_Network_Language.bounded bounds s0
proof -
  have *: ((Not ∘ case_prod) (λ(L, s) _. check_sexp φ L (the o s)))
    = (λ((L, s), _). ¬ check_sexp φ L (the o s))
  for φ by (auto simp: comp_def)
  have rel_ctl_formula_compatible (ctl_of2 Φ) (ctl_of Φ)
    by (cases Φ; simp add: prop_of_def prop_of2_def rel_fun_def A_B.equiv'_def)
  moreover have A_B.equiv' ((L0, s0), u0) (L0, s0, u0)
    unfolding A_B.equiv'_def unfolding conv_all_prop all_prop_def using that by simp
  ultimately show ?thesis
    apply (subst models_ctl_iff)
    apply (subst CTL_compatible[THEN rel_funD, symmetric])
    apply assumption+
    apply (cases Φ; simp add:
      Graph_Defs.models_ctl.simps prop_of2_def
      Graph_Defs.Ex_alw_iff Graph_Defs.Alw_alw_iff Graph_Defs.leadsto_def *)
  done
qed
end

context Simple_Network_Impl_nat_ceiling_start_state — slow: 70s
begin

definition Alw_ev_checker where
  Alw_ev_checker = dfs_map_impl'
    (impl.succs_P_impl' Fi) impl.a0_impl impl.subsumes_impl (return ∘ fst)
    impl.state_copy_impl

definition leadsto_checker where
  leadsto_checker ψ = do {
    r ← leadsto_impl
    impl.state_copy_impl (impl.succs_P_impl' (λ (L, s). ¬ check_sexp ψ L s))
  }

```

```

impl.a0_impl impl.subsumes_impl (return o fst)
impl.succs_impl' impl.emptiness_check_impl impl.F_impl
(impl.Q_impl (λ (L, s). ¬ check_sexp_i ψ L s))
impl.trace_i;
return (¬ r)
}

```

**definition**

```

reachability_checker ≡
  pw_impl
  (return o fst) impl.state_copy_impl impl.trace_i impl.subsumes_impl impl.a0_impl impl.F_impl
  impl.succs_impl impl.emptiness_check_impl

```

**definition** *model\_checker* **where**

```

model_checker = (
  case formula of
    formula.EX _ ⇒ reachability_checker |
    formula.AG _ ⇒ do {
      r ← reachability_checker;
      return (¬ r)
    } |
    formula.AX _ ⇒ do {
      r ← if PR_CONST F l0
      then Alw_ev_checker
      else return False;
      return (¬ r)
    } |
    formula.EG _ ⇒
      if PR_CONST F l0
      then Alw_ev_checker
      else return False |
    formula.Leadsto _ ψ ⇒ leadsto_checker ψ
)

```

**abbreviation**  $u_0 \equiv (\lambda \_. 0 :: \text{real})$

**lemma** *all\_prop\_start*:

```

all_prop L0 (map_of s0)
using L0_states s0_bounded unfolding all_prop_def ..

```

**lemma** *deadlock\_start\_iff*:

```

Graph_Defs.deadlock
(λ(L, s, u) (L', s', u'). Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩) (L0,
(map_of s0), u0)
⟷ reach.deadlock (l0, u0)
using all_prop_start by (subst deadlock_iff[symmetric]) (auto simp: conv_all_prop)

```

**lemma** *F\_Fi'*:

```

check_sexp ψ L (the o s) ⟷ check_sexp_i ψ L' s'
if ((L', s'), (L, s)) ∈ loc_rel formula = Leadsto φ ψ
using vars_of_formula that unfolding loc_rel_def by (auto elim!: check_sexp_check_sexp_i)

```

**theorem** *model\_check'*:

```

(uncurry0 model_checker,
  uncurry0 (
    SPEC (λr.
      ¬ Graph_Defs.deadlock
      (λ(L, s, u) (L', s', u').
        Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩) (L₀, (map_of s₀), u₀)
        → r = (Simple_Network_Language.conv A, (L₀, (map_of s₀), u₀) ⊨ formula)
      )
    )
  )
)
∈ unit_assnk →a bool_assn
proof -
define protect where
  protect = ((λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩))

have start: l₀ ∈ Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod
  if ¬ Graph_Defs.deadlock protect (l₀, λ_. 0)
  using that unfolding protect_def by (rule impl.init_state_in_state_set[simplified])

interpret ta_bisim: Bisimulation_Invariant
  (λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(l, u) (l', u'). l' = l ∧ (∀ c. c ∈ clk_set (conv_A prod_ta) → u c = u' c))
  (λ_. True) (λ_. True)
  by (rule ta_bisimulation[of conv_A prod_ta])

let ?φ1 = λφ. λ(p, _). case p of (L, s) ⇒ ¬ check_sexp φ L (the ∘ s)
let ?φ2 = λφ. λ(p, _). case p of (L, s) ⇒ check_sexp φ L (the ∘ s)

have start_sim:
  ta_bisim.A_B.equiv' (l₀, u) (l₀, λ_. 0) ta_bisim.A_B.equiv' (l₀, λ_. 0) (l₀, u)
  if ∀ c ∈ {Suc 0..m}. u c = 0 for u
  using impl.clocks_I[of u λ_. 0] that unfolding ta_bisim.A_B.equiv'_def by auto

have compatibleI: φ a = φ b
  if ta_bisim.A_B.equiv' a b ∧ l u u'. φ (l, u) = φ (l, u') for a b φ
  using that unfolding ta_bisim.A_B.equiv'_def by auto

have bisims:
  (∀ u₀. (∀ c ∈ {Suc 0..m}. u₀ c = 0) → reach.Ex_ev (?φ1 φ) (l₀, u₀))
  ⇔ reach.Ex_ev (?φ1 φ) (l₀, u₀)
  (∀ u₀. (∀ c ∈ {Suc 0..m}. u₀ c = 0) → reach.Ex_ev (?φ2 φ) (l₀, u₀))
  ⇔ reach.Ex_ev (?φ2 φ) (l₀, u₀)
  (∀ u₀. (∀ c ∈ {Suc 0..m}. u₀ c = 0) → reach.Alw_ev (?φ1 φ) (l₀, u₀))
  ⇔ reach.Alw_ev (?φ1 φ) (l₀, u₀)
  (∀ u₀. (∀ c ∈ {Suc 0..m}. u₀ c = 0) → reach.Alw_ev (?φ2 φ) (l₀, u₀))
  ⇔ reach.Alw_ev (?φ2 φ) (l₀, u₀)
  (∀ u₀. (∀ c ∈ {Suc 0..m}. u₀ c = 0) → reach.leadsto (?φ2 φ) (?φ2 ψ) (l₀, u₀))
  ⇔ reach.leadsto (?φ2 φ) (?φ2 ψ) (l₀, u₀)
  for φ ψ
  apply safe
  apply (all ⟨(solves auto)?⟩)
  subgoal for u
    using start_sim

```

```

    by (subst (asm) ta_bisim.Ex_ev_iff[of ? $\varphi$ 1  $\varphi$  ? $\varphi$ 1  $\varphi$ ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Ex_ev_iff[of ? $\varphi$ 2  $\varphi$  ? $\varphi$ 2  $\varphi$ ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Alw_ev_iff[of ? $\varphi$ 1  $\varphi$  ? $\varphi$ 1  $\varphi$ ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Alw_ev_iff[of ? $\varphi$ 2  $\varphi$  ? $\varphi$ 2  $\varphi$ ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Leadsto_iff[of ? $\varphi$ 2  $\varphi$  ? $\varphi$ 2  $\varphi$  ? $\varphi$ 2  $\psi$  ? $\varphi$ 2  $\psi$ ])
      ((erule compatibleI)?; auto)+
done

```

have deadlock\_bisim:

```

  ( $\forall u_0. (\forall c \in \{1..m\}. u_0 \ c = 0) \longrightarrow \neg \text{reach.deadlock } (l_0, u_0)$ )
 $\longleftrightarrow \neg \text{Graph\_Defs.deadlock\_protect } (l_0, \lambda_. 0)$ 
  unfolding protect_def
  apply (safe; clarsimp)
  apply (drule start_sim(2))
  apply (subst (asm) ta_bisim.deadlock_iff)
  unfolding ta_bisim.A_B.equiv'_def
  by auto

```

have \*\*\*\*\*:

```

  return True = (return False  $\gg$  return o Not)
  by auto

```

show ?thesis

```

  unfolding deadlock_start_iff
  using models_correct[OF L0_states s0_bounded]
  unfolding protect_def[symmetric]
  apply (simp split del: if_split)
  apply sepref_to_hoare
  apply sep_auto
  unfolding model_checker_def Alw_ev_checker_def leadsto_checker_def reachability_checker_def
  apply (cases formula; simp)

```

subgoal

```

  apply (cases Graph_Defs.deadlock_protect (l0,  $\lambda_. 0$ ))
  subgoal
    using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
    by (sep_auto elim!: cons_post_rule)
  subgoal
    using impl.Ex_ev_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def]
    by (sep_auto simp: pure_def protect_def bisims)
  done

```

subgoal premises prems for  $\varphi$

proof -

```

  have *: ( $\lambda(l, u). \neg F \ l$ ) = (( $\lambda(p, \_).$  case p of (L, s)  $\Rightarrow \neg \text{check\_sexp } \varphi \ L \ (the \circ s)$ ))
    using prems by auto

```

```

show ?thesis
  using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
  unfolding PR_CONST_def * protect_def
  by (sep_auto elim!: cons_post_rule simp: pure_def bisims)
qed

subgoal premises prems for  $\varphi$ 
proof -
  have *:  $(\lambda(l, u). \neg F l) = ((\lambda(p, \_). \text{case } p \text{ of } (L, s) \Rightarrow \text{check\_sexp } \varphi L (\text{the } \circ s)))$ 
  using prems by auto
show ?thesis
  apply intros
  subgoal
    using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
    unfolding PR_CONST_def * protect_def
    apply simp
    unfolding *****
    apply (erule bind_rule)
    apply (sep_auto simp: pure_def bisims(2-))
    done
  subgoal
    using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
    unfolding PR_CONST_def * protect_def
    apply (sep_auto elim!: cons_post_rule simp: pure_def bisims(2-))
    done
  done
qed

subgoal
  apply (cases Graph_Defs.deadlock protect (l0,  $\lambda\_.$  0))
  subgoal
    using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
    by (sep_auto elim!: cons_post_rule)
  subgoal
    using impl.Ex_ev_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def]
    apply (sep_auto simp: pure_def protect_def bisims)
    done
  done
done

subgoal premises prems for  $\varphi \psi$ 
proof -
  have *:  $(\lambda(l, u). F l) = ((\lambda(p, \_). \text{case } p \text{ of } (L, s) \Rightarrow \text{check\_sexp } \varphi L (\text{the } \circ s)))$ 
  using prems by auto
  have **:  $(\lambda(L, s). \neg \text{check\_sexp } \psi L s, \lambda(L, s). \neg \text{check\_sexp } \psi L (\text{the } \circ s))$ 
     $\in \text{inv\_rel loc\_rel states'}$ 
  unfolding trans_of_prod using prems by (auto simp: F_Fi' inv_rel_def)
  have ***:  $(\lambda(l, u). \neg (\text{case } l \text{ of } (L, s) \Rightarrow \neg \text{check\_sexp } \psi L (\text{the } \circ s)))$ 
     $= (\lambda(l, u). (\text{case } l \text{ of } (L, s) \Rightarrow \text{check\_sexp } \psi L (\text{the } \circ s)))$ 
  by auto
show ?thesis
  apply (cases reach.deadlock (l0,  $\lambda\_.$  0))
  subgoal
    using impl.leadsto_impl_hnr'[OF **, to_hnr, unfolded hn_refine_def]
    unfolding protect_def ⟨formula =  $\_$ ⟩ by (sep_auto simp: pure_def)

```

```

    using impl.leadsto_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def,
OF **]
    prems start
    unfolding PR_CONST_def * protect_def by (sep_auto simp: pure_def bisims ****)
  qed
done
qed

```

## 9.1 Extracting an Efficient Implementation

```

lemma reachability_checker_alt_def':
  reachability_checker  $\equiv$ 
  do {
     $x \leftarrow$  do {
      let key = return  $\circ$  fst;
      let sub = impl.subsumes_impl;
      let copy = impl.state_copy_impl;
      let start = impl.a0_impl;
      let final = impl.F_impl;
      let succs = impl.succs_impl;
      let empty = impl.emptyness_check_impl;
      let trace = impl.tracei;
      pw_impl key copy trace sub start final succs empty
    };
     $\_ \leftarrow$  return ();
    return  $x$ 
  }
unfolding reachability_checker_def by simp

```

```

lemma Alw_ev_checker_alt_def':
  Alw_ev_checker  $\equiv$ 
  do {
     $x \leftarrow$  let
      key = return  $\circ$  fst;
      sub = impl.subsumes_impl;
      copy = impl.state_copy_impl;
      start = impl.a0_impl;
      succs = impl.succs_P_impl' Fi
    in dfs_map_impl' succs start sub key copy;
     $\_ \leftarrow$  return ();
    return  $x$ 
  }
unfolding Alw_ev_checker_def by simp

```

```

lemma leadsto_checker_alt_def':
  leadsto_checker  $\psi \equiv$ 
  do {
     $r \leftarrow$  let
      key = return  $\circ$  fst;
      sub = impl.subsumes_impl;
      copy = impl.state_copy_impl;
      start = impl.a0_impl;
      final = impl.F_impl;
      final' = (impl.Q_impl ( $\lambda(L, s). \neg$  check_sexp  $\psi$   $L$   $s$ ));

```

```

      succs = impl.succs_P_impl' (λ(L, s). ¬ check_sexpi ψ L s);
      succs' = impl.succs_impl';
      empty = impl.emptyness_check_impl;
      trace = impl.tracei
    in
      leadsto_impl copy succs start sub key succs' empty final final' trace;
    return (¬ r)
  }
unfolding leadsto_checker_def by simp

theorem k_impl_alt_def:
  k_impl = (λ(l, s). IArray (map (λc. MAX i ∈ {0..<n_ps}. k_i !! i !! (l ! i) !! c) [0..<m + 1]))
proof -
  have {i. i < p} = {0..<p} for p :: nat
  by auto
  then show ?thesis
  unfolding k_impl_def setcompr_eq_image by simp
qed

definition
  trans_map_inner ≡ map (λi. union_map_of (fst (snd (snd (automata ! i)))) [0..<n_ps])

lemma trans_map_alt_def:
  trans_map i j = (case (IArray trans_map_inner !! i) j of None ⇒ [] | Some xs ⇒ xs)
  if i < n_ps
  using that unfolding trans_map_inner_def trans_map_def by (auto simp: n_ps_def)

schematic_goal succs_impl_alt_def:
  impl.succs_impl ≡ ?impl
  unfolding impl.succs_impl_def
  apply (abstract_let impl.E_op''_impl E_op''_impl)
  unfolding impl.E_op''_impl_def fw_impl'_int
  apply (abstract_let trans_impl trans_impl)
  unfolding trans_impl_def
  apply (abstract_let int_trans_impl int_trans_impl)
  apply (abstract_let bin_trans_from_impl bin_trans_impl)
  apply (abstract_let broad_trans_from_impl broad_trans_impl)
  unfolding int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def
  apply (abstract_let trans_in_broad_grouped trans_in_broad_grouped)
  apply (abstract_let trans_out_broad_grouped trans_out_broad_grouped)
  apply (abstract_let trans_in_map trans_in_map)
  apply (abstract_let trans_out_map trans_out_map)
  apply (abstract_let int_trans_from_all_impl int_trans_from_all_impl)
  unfolding int_trans_from_all_impl_def
  apply (abstract_let int_trans_from_vec_impl int_trans_from_vec_impl)
  unfolding int_trans_from_vec_impl_def
  apply (abstract_let int_trans_from_loc_impl int_trans_from_loc_impl)
  unfolding int_trans_from_loc_impl_def
  apply (abstract_let trans_i_map trans_i_map)
  unfolding trans_out_broad_grouped_def trans_out_broad_map_def
  unfolding trans_in_broad_grouped_def trans_in_broad_map_def
  unfolding trans_in_map_def trans_out_map_def
  unfolding trans_i_map_def
  apply (abstract_let trans_map trans_map)

```

```

apply (abstract_let inv_fun :: nat list  $\times$  int list  $\Rightarrow$  _ inv_fun)
unfolding inv_fun_alt_def
apply (abstract_let invs2 invs)
unfolding invs2_def
unfolding k_impl_alt_def
apply (abstract_let k_i k_i)
apply (abstract_let n_ps n_ps)
by (rule Pure.reflexive)

```

```

schematic_goal succs_P_impl_alt_def:
  impl.succs_P_impl Pi  $\equiv$  ?impl
  if (Pi, P)  $\in$  inv_rel loc_rel states'
  for P Pi
  unfolding impl.succs_P_impl_def[OF that]
  apply (abstract_let impl.E_op''_impl E_op''_impl)
  unfolding impl.E_op''_impl_def fw_impl'_int
  apply (abstract_let trans_impl trans_impl)
  unfolding trans_impl_def
  apply (abstract_let int_trans_impl int_trans_impl)
  apply (abstract_let bin_trans_from_impl bin_trans_impl)
  apply (abstract_let broad_trans_from_impl broad_trans_impl)
  unfolding int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def
  apply (abstract_let trans_in_broad_grouped trans_in_broad_grouped)
  apply (abstract_let trans_out_broad_grouped trans_out_broad_grouped)
  apply (abstract_let trans_in_map trans_in_map)
  apply (abstract_let trans_out_map trans_out_map)
  apply (abstract_let int_trans_from_all_impl int_trans_from_all_impl)
  unfolding int_trans_from_all_impl_def
  apply (abstract_let int_trans_from_vec_impl int_trans_from_vec_impl)
  unfolding int_trans_from_vec_impl_def
  apply (abstract_let int_trans_from_loc_impl int_trans_from_loc_impl)
  unfolding int_trans_from_loc_impl_def
  apply (abstract_let trans_i_map trans_i_map)
  unfolding trans_out_broad_grouped_def trans_out_broad_map_def
  unfolding trans_in_broad_grouped_def trans_in_broad_map_def
  unfolding trans_in_map_def trans_out_map_def
  unfolding trans_i_map_def
  apply (abstract_let trans_map trans_map)
  apply (abstract_let inv_fun :: nat list  $\times$  int list  $\Rightarrow$  _ inv_fun)
  unfolding inv_fun_alt_def
  apply (abstract_let invs2 invs)
  unfolding invs2_def
  unfolding k_impl_alt_def
  apply (abstract_let k_i k_i)
  apply (abstract_let n_ps n_ps)
  by (rule Pure.reflexive)

```

```

lemmas succs_P'_impl_alt_def =
  impl.succs_P_impl'_def[OF impl.F_fun, unfolded succs_P_impl_alt_def[OF impl.F_fun]]

```

```

schematic_goal Alw_ev_checker_alt_def:
  Alw_ev_checker  $\equiv$  ?impl

```

**unfolding** *Alw\_ev\_checker\_alt\_def'*  
**unfolding** *succs\_P'\_impl\_alt\_def*  
**unfolding** *k\_impl\_alt\_def k\_i\_def*

**unfolding** *impl.E\_op''\_impl\_def impl.abstr\_repair\_impl\_def impl.abstra\_repair\_impl\_def*  
**unfolding** *impl.start\_inv\_check\_impl\_def unbounded\_dbm\_impl\_def unbounded\_dbm'\_def*  
**unfolding** *impl.init\_dbm\_impl\_def impl.a0\_impl\_def*  
**unfolding** *impl.F\_impl\_def*  
**unfolding** *impl.subsumes\_impl\_def*  
**unfolding** *impl.emptyness\_check\_impl\_def*  
**unfolding** *impl.state\_copy\_impl\_def*  
**by** (*rule Pure.reflexive*)

**lemma** *ψ\_compatibleI*:

$(\lambda(L, s). \neg \text{check\_sexpi } \psi \ L \ s, \lambda(L, s). \neg \text{check\_sexp } \psi \ L \ (the \circ s)) \in \text{inv\_rel loc\_rel states'}$   
**if** *formula = Leadsto φ ψ*  
**using** *F\_Fi'[OF \_ that]* **unfolding** *inv\_rel\_def* **by** *auto*

**lemma** *Q\_impl\_alt\_def*:

$\text{impl.Q\_impl } (\lambda(L, s). \neg \text{check\_sexpi } \psi \ L \ s) \equiv$   
 $\lambda xi. \text{return (case xi of (a1, a2) } \Rightarrow (\lambda(L, s). \neg \text{check\_sexpi } \psi \ L \ s) \ a1)$   
**if** *formula = Leadsto φ ψ*  
**by** (*intro impl.Q\_impl\_def[where Q = λ(L, s). ¬ check\_sexp ψ L (the o s)] ψ\_compatibleI[OF that]*)

**schematic\_goal** *leadsto\_checker\_alt\_def*:

$\text{leadsto\_checker } \psi \equiv ?\text{impl if formula = Leadsto } \varphi \ \psi$   
**unfolding** *leadsto\_checker\_alt\_def'*  
**unfolding** *Q\_impl\_alt\_def[OF that] impl.F\_impl\_def*  
**unfolding** *impl.succs\_P\_impl'\_def[OF ψ\_compatibleI[OF that]]*  
**unfolding** *succs\_P\_impl\_alt\_def[OF ψ\_compatibleI[OF that]]*  
**unfolding** *impl.succs\_impl'\_def succs\_impl\_alt\_def*  
**unfolding** *k\_impl\_alt\_def k\_i\_def*

**unfolding** *impl.E\_op''\_impl\_def impl.abstr\_repair\_impl\_def impl.abstra\_repair\_impl\_def*  
**unfolding** *impl.start\_inv\_check\_impl\_def unbounded\_dbm\_impl\_def unbounded\_dbm'\_def*  
**unfolding** *impl.init\_dbm\_impl\_def impl.a0\_impl\_def*  
**unfolding** *impl.F\_impl\_def*  
**unfolding** *impl.subsumes\_impl\_def*  
**unfolding** *impl.emptyness\_check\_impl\_def*  
**unfolding** *impl.state\_copy\_impl\_def*  
**by** (*rule Pure.reflexive*)

**schematic\_goal** *reachability\_checker\_alt\_def*:

$\text{reachability\_checker} \equiv ?\text{impl}$   
**unfolding** *reachability\_checker\_alt\_def'*  
**unfolding** *succs\_impl\_alt\_def*  
**unfolding** *k\_impl\_alt\_def k\_i\_def*

**unfolding** *impl.E\_op''\_impl\_def impl.abstr\_repair\_impl\_def impl.abstra\_repair\_impl\_def*  
**unfolding** *impl.start\_inv\_check\_impl\_def unbounded\_dbm\_impl\_def unbounded\_dbm'\_def*  
**unfolding** *impl.init\_dbm\_impl\_def impl.a0\_impl\_def*  
**unfolding** *impl.F\_impl\_def*  
**unfolding** *impl.subsumes\_impl\_def*

```

    unfolding impl.emptiness_check_impl_def
    unfolding impl.state_copy_impl_def
    by (rule Pure.reflexive)

end

concrete_definition reachability_checker
  uses Simple_Network_Impl_nat_ceiling_start_state.reachability_checker_alt_def

concrete_definition Alw_ev_checker
  uses Simple_Network_Impl_nat_ceiling_start_state.Alw_ev_checker_alt_def

concrete_definition leadsto_checker
  uses Simple_Network_Impl_nat_ceiling_start_state.leadsto_checker_alt_def

context Simple_Network_Impl_nat_ceiling_start_state — slow: 100s
begin

lemma model_checker_unfold_leadsto:
  model_checker = (
  case formula of Simple_Network_Language_Model_Checking.formula.EX xa  $\Rightarrow$  reachability_checker
  | Simple_Network_Language_Model_Checking.formula.EG xa  $\Rightarrow$ 
    if PR_CONST F l0 then Alw_ev_checker else return False
  | Simple_Network_Language_Model_Checking.formula.AX xa  $\Rightarrow$ 
    (if PR_CONST F l0 then Alw_ev_checker else return False)  $\gg$  ( $\lambda r$ . return ( $\neg$  r))
  | Simple_Network_Language_Model_Checking.formula.AG xa  $\Rightarrow$ 
    reachability_checker  $\gg$  ( $\lambda r$ . return ( $\neg$  r))
  | Simple_Network_Language_Model_Checking.formula.Leadsto  $\varphi$   $\psi \Rightarrow$ 
    Simple_Network_Language_Model_Checking.leadsto_checker
    broadcast bounds' automata m num_states num_actions k L0 s0 formula  $\psi$  show_clock
  show_state)

  unfolding model_checker_def
  using leadsto_checker.refine[OF Simple_Network_Impl_nat_ceiling_start_state_axioms]
  by (simp split: formula.split)

lemmas model_checker_def_refined = model_checker_unfold_leadsto[unfolded
  reachability_checker.refine[OF Simple_Network_Impl_nat_ceiling_start_state_axioms]
  Alw_ev_checker.refine[OF Simple_Network_Impl_nat_ceiling_start_state_axioms]
]

end

concrete_definition model_checker uses
  Simple_Network_Impl_nat_ceiling_start_state.model_checker_def_refined

definition precondition_mc where
  precondition_mc
  show_clock show_state broadcast bounds' automata m num_states num_actions k L0 s0 formula
 $\equiv$ 
  if Simple_Network_Impl_nat_ceiling_start_state
    broadcast bounds' automata m num_states num_actions k L0 s0 formula
  then
    model_checker

```

```

    broadcast bounds' automata m num_states num_actions k L0 s0 formula show_clock
show_state
  >>= (λ x. return (Some x))
else return None

```

**abbreviation** *N* where

```

N broadcast automata bounds ≡
Simple_Network_Language.conv
(set broadcast, map automaton_of automata, map_of bounds)

```

**definition** *has\_deadlock* where

```

has_deadlock A a0 ≡
Graph_Defs.deadlock (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩) a0

```

**theorem** *model\_check*:

```

<emp> precondition
show_clock show_state broadcast bounds automata m num_states num_actions k L0 s0 formula
<λ Some r ⇒ ↑(
  Simple_Network_Impl_nat_ceiling_start_state
  broadcast bounds automata m num_states num_actions k L0 s0 formula ∧
  (¬ has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0) →
    r = N broadcast automata bounds, (L0, map_of s0, λ_. 0) ⊨ formula
  ))
| None ⇒ ↑(¬ Simple_Network_Impl_nat_ceiling_start_state
  broadcast bounds automata m num_states num_actions k L0 s0 formula)
>t

```

**proof** –

```

define A where A ≡ N broadcast automata bounds
define check where check ≡ A, (L0, map_of s0, λ_. 0) ⊨ formula
note [sep_heap_rules] =
  Simple_Network_Impl_nat_ceiling_start_state.model_check'[
    to_hnr, unfolded hn_refine_def,
    of broadcast bounds automata m num_states num_actions k L0 s0 formula,
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    folded A_def check_def has_deadlock_def,
    simplified
  ]
show ?thesis
  unfolding A_def[symmetric] check_def[symmetric]
  unfolding precondition_def
  by (sep_auto simp: model_checker.refine[symmetric] pure_def)

```

qed

end

## 10 Deadlock Checking

**theory** *Deadlock*

**imports**

```

Timed_Automata.Timed_Automata Timed_Automata.CTL Difference_Bound_Matrices.DBM_Operations
Timed_Automata.Normalized_Zone_Semantics
Munta_Base.Normalized_Zone_Semantics_Impl

```

```

Munta_Base.Normalized_Zone_Semantics_Impl_Refine
Difference_Bound_Matrices.FW_More
Munta_Base.TA_Syntax_Bundles
begin

```

```

unbundle no_library_syntax

```

## 10.1 Notes

If we want to run the deadlock checker together with a reachability check for property  $P$ , we can:

- run a reachability check with  $\neg \text{check\_deadlock}$  first; if we find a deadlock, we are done; else we can check whether any of the reachable states satisfies  $P$ ;
- or run a reachability check with  $P$  first, which might give us earlier termination; if we find that  $P$  is satisfied, can we directly report that the original formula is satisfied?

Generally, it seems advantageous to compute  $\neg \text{check\_deadlock}$  on the final set of states, and not intermediate subsumed ones, as the operation is expensive.

## 10.2 Abstract Reachability Checking

**definition**  $\text{zone\_time\_pre} :: ('c, ('t::\text{time})) \text{ zone} \Rightarrow ('c, 't) \text{ zone}$   
 $(\_ \downarrow [71] \ 71)$

**where**

$$Z^\downarrow = \{u \mid u \text{ d. } (u \oplus d) \in Z \wedge d \geq (0::'t)\}$$

**definition**  $\text{zone\_set\_pre} :: ('c, 't::\text{time}) \text{ zone} \Rightarrow 'c \text{ list} \Rightarrow ('c, 't) \text{ zone}$

**where**

$$\text{zone\_set\_pre } Z \ r = \{u \mid ([r \rightarrow (0::'t)]u) \in Z\}$$

**definition**  $\text{zone\_pre} :: ('c, 't::\text{time}) \text{ zone} \Rightarrow 'c \text{ list} \Rightarrow ('c, 't) \text{ zone}$

**where**

$$\text{zone\_pre } Z \ r = (\text{zone\_set\_pre } Z \ r)^\downarrow$$

**lemma**  $\text{zone\_time\_pre\_mono}$ :

$$A^\downarrow \subseteq B^\downarrow \text{ if } A \subseteq B$$

**using** *that unfolding zone\_time\_pre\_def by auto*

**lemma**  $\text{clock\_set\_split}$ :

$$P \ ([r \rightarrow 0]u \ x) \longleftrightarrow (x \notin \text{set } r \longrightarrow P \ (u \ x)) \wedge (x \in \text{set } r \longrightarrow P \ 0)$$

**by** *(cases x ∈ set r) auto*

**context**  $\text{Regions\_TA}$

**begin**

**definition**

$$\text{check\_deadlock } l \ Z \equiv Z \subseteq$$

$$\bigcup \{(\text{zone\_set\_pre } \{u. u \vdash \text{inv\_of } A \ l'\} \ r \cap \{u. u \vdash g\} \cap \{u. u \vdash \text{inv\_of } A \ l'\})^\downarrow \mid g \ a \ r \ l'\}.$$

$$A \vdash l \longrightarrow^{g,a,r} l'\}$$

**lemma** *V\_zone\_time\_pre*:

$x \in (Z \cap V)^\downarrow$  **if**  $x \in Z^\downarrow$   $x \in V$

**using** *that unfolding zone\_time\_pre\_def* **by** (*auto simp: V\_def cval\_add\_def*)

**lemma** *check\_deadlock\_alt\_def*:

$check\_deadlock\ l\ Z = (Z \subseteq \bigcup \{$   
 $(zone\_set\_pre\ (\{u. u \vdash inv\_of\ A\ l'\} \cap V)\ r \cap \{u. \forall x \in set\ r. u\ x \geq 0\}$   
 $\cap \{u. u \vdash g\} \cap \{u. u \vdash inv\_of\ A\ l'\})^\downarrow \cap V$   
 $\mid g\ a\ r\ l'. A \vdash l \longrightarrow^{g,a,r} l'\})$  **(is**  $\_ = (?L \subseteq ?R)$  **) if**  $Z \subseteq V$

**proof** –

**{ fix**  $g\ a\ r\ l'\ x$

**assume**  $t: A \vdash l \longrightarrow^{g,a,r} l'$

**assume**  $x: x \in (zone\_set\_pre\ \{u. u \vdash inv\_of\ A\ l'\}\ r \cap \{u. u \vdash g\} \cap \{u. u \vdash inv\_of\ A\ l'\})^\downarrow$

**assume**  $x \in V$

**let**  $?A = zone\_set\_pre\ \{u. u \vdash inv\_of\ A\ l'\}\ r \cap \{u. u \vdash g\} \cap \{u. u \vdash inv\_of\ A\ l'\}$

**let**  $?B = zone\_set\_pre\ (\{u. u \vdash inv\_of\ A\ l'\} \cap V)\ r \cap \{u. \forall x \in set\ r. u\ x \geq 0\}$   
 $\cap \{u. u \vdash g\} \cap \{u. u \vdash inv\_of\ A\ l'\}$

**from** *valid\_abstraction* **have**  $collect\_clkvt\ (trans\_of\ A) \subseteq X$

**by** (*auto elim: valid\_abstraction.cases*)

**have**  $*$ :  $0 \leq u\ c$  **if**  $c \in set\ r\ u \in V$  **for**  $c\ u$

**proof** –

**from**  $t\ \langle c \in set\ r \rangle$  **have**  $c \in collect\_clkvt\ (trans\_of\ A)$

**unfolding** *collect\_clkvt\_def* **by** *force*

**with**  $\langle \_ \subseteq X \rangle$  **have**  $c \in X$

**by** *auto*

**with**  $\langle u \in V \rangle$  **show**  $?thesis$

**by** (*auto simp: V\_def*)

**qed**

**have**  $*$ :  $u \in zone\_set\_pre\ (\{u. u \vdash inv\_of\ A\ l'\} \cap V)\ r$

**if**  $u \in zone\_set\_pre\ \{u. u \vdash inv\_of\ A\ l'\}\ r\ u \in V$  **for**  $u$

**using** *that unfolding zone\_set\_pre\_def V\_def* **by** (*auto split: clock\_set\_split*)

**from**  $x\ \langle x \in V \rangle$  **have**  $x \in (?A \cap V)^\downarrow$

**by** (*rule V\_zone\_time\_pre*)

**moreover** **have**  $y \in ?B$  **if**  $y \in ?A \cap V$  **for**  $y$

**using** *that* **by** (*auto intro: \* \*\**)

**ultimately** **have**  $x \in ?B^\downarrow$

**unfolding** *zone\_time\_pre\_def* **by** *auto*

**} note**  $*$  = *this*

**have**  $zone\_set\_pre\ (Z \cap V)\ r \subseteq zone\_set\_pre\ Z\ r$  **for**  $Z\ r$

**unfolding** *zone\_set\_pre\_def* **by** *auto*

**with**  $\langle Z \subseteq V \rangle$  **show**  $?thesis$

**unfolding** *check\_deadlock\_def*

**apply** *safe*

**subgoal** **for**  $x$

**apply** *rotate\_tac*

**apply** (*drule subsetD, assumption*)

**apply** (*drule subsetD, assumption*)

**apply** *clar simp*

**apply** (*frule \*, assumption+*)

**subgoal** **for**  $g\ a\ r\ l'$

**by** (*inst\_existentials*)

$(zone\_set\_pre\ (\{u. u \vdash inv\_of\ A\ l'\} \cap V)\ r \cap \{u. \forall x \in set\ r. 0 \leq u\ x\} \cap \{u. u \vdash g\} \cap$

```

      {u. u ⊢ inv_of A l} )↓ ∩ V) auto
    done
  apply (drule subsetD, assumption)+
  apply safe
  subgoal for x X g a r l'
  by (drule
      subsetD[OF zone_time_pre_mono,
        where B1 = zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l},
        rotated]; force)
  done
qed

```

```

lemma step_trans1:
  assumes A ⊢t ⟨l, u⟩ →(g,a,r) ⟨l', u'⟩
  shows u ∈ zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} A ⊢ l →g,a,r l'
  using assms by (auto elim!: step_trans.cases simp: zone_set_pre_def)

```

```

lemma step_trans2:
  assumes u ∈ zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} A ⊢ l →g,a,r l'
  shows ∃ u'. A ⊢t ⟨l, u⟩ →(g,a,r) ⟨l', u'⟩
  using assms unfolding zone_set_pre_def by auto

```

```

lemma time_pre_zone:
  u ∈ (Z ∩ {u. u ⊢ inv_of A l} )↓ if A ⊢ ⟨l, u⟩ →d ⟨l', u'⟩ u' ∈ Z
  using that by (auto elim!: step_t.cases simp: zone_time_pre_def)

```

```

lemma time_pre_zone':
  ∃ d u'. u' ∈ Z ∧ A ⊢ ⟨l, u⟩ →d ⟨l', u'⟩ if u ∈ (Z ∩ {u. u ⊢ inv_of A l} )↓
  using that unfolding zone_time_pre_def by auto

```

```

lemma step_trans3:
  assumes A ⊢' ⟨l, u⟩ →(g,a,r) ⟨l', u'⟩
  shows u ∈ (zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l} )↓
    A ⊢ l →g,a,r l'
  using assms by (auto dest: step_trans1 time_pre_zone step_delay_loc elim: step_trans'.cases)

```

```

lemma step_trans4:
  assumes u ∈ (zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l} )↓
    A ⊢ l →g,a,r l'
  shows ∃ u'. A ⊢' ⟨l, u⟩ →(g,a,r) ⟨l', u'⟩
  using assms by (fast dest: time_pre_zone' step_trans2[rotated])

```

```

lemma check_deadlock_correct:
  check_deadlock l Z ↔ (∀ u ∈ Z. ∃ l' u' g a r. A ⊢' ⟨l, u⟩ →(g,a,r) ⟨l', u'⟩)
  unfolding check_deadlock_def
  apply safe
  subgoal for x
    using step_trans4 by blast
  subgoal for x
    using step_trans3 by fast
  done

```

```

lemma step'_step_trans'_iff:

```

$A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle \longleftrightarrow (\exists g \ a \ r. A \vdash' \langle l, u \rangle \rightarrow^{(g,a,r)} \langle l', u' \rangle)$   
**by** (*metis prod\_cases3 step'.cases step'.intros step\_a.cases step\_a.simps step\_trans'.cases*  
*step\_trans'.intros step\_trans.cases step\_trans.simps*  
 )

**lemma** *check\_deadlock\_correct\_step'*:  
 $check\_deadlock \ l \ Z \longleftrightarrow (\forall u \in Z. \exists l' \ u'. A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$   
**using** *check\_deadlock\_correct\_step'\_step\_trans'\_iff* **by** *simp*

**Unused lemma** *delay\_step\_zone*:  
 $u' \in Z^\uparrow \cap \{u. u \vdash inv\_of \ A \ l\} \text{ if } A \vdash \langle l, u \rangle \rightarrow^d \langle l, u' \rangle \ u \in Z$   
**using** *that* **by** (*auto elim!:* *step\_t.cases simp: zone\_delay\_def*)

**lemma** *delay\_step\_zone'*:  
 $\exists d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u' \rangle \text{ if } u' \in Z^\uparrow \cap \{u. u \vdash inv\_of \ A \ l\}$   
**using** *that* **by** (*auto simp: zone\_delay\_def*)

**lemma** *delay\_step\_zone''*:  
 $(\exists d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u' \rangle) \longleftrightarrow u' \in Z^\uparrow \cap \{u. u \vdash inv\_of \ A \ l\}$   
**using** *delay\_step\_zone delay\_step\_zone'* **by** *blast*

**lemma** *delay\_step\_zone'''*:  
 $\{u' \mid u' d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u' \rangle\} = Z^\uparrow \cap \{u. u \vdash inv\_of \ A \ l\}$   
**using** *delay\_step\_zone''* **by** *auto*

**end**

**context** *Regions\_TA\_Start\_State*  
**begin**

**lemma** *check\_deadlock\_deadlocked*:  
 $\neg check\_deadlock \ l \ Z \longleftrightarrow (\exists u \in Z. sim.sim.deadlocked \ (l, u))$   
**unfolding** *check\_deadlock\_correct\_step' sim.sim.deadlocked\_def* **by** *simp*

**lemma** *deadlock\_check'*:  
 $(\exists x_0 \in a_0. \exists l \ u. sim.sim.reaches \ x_0 \ (l, u) \wedge sim.sim.deadlocked \ (l, u)) \longleftrightarrow$   
 $(\exists l \ Z. reaches \ (l_0, Z_0) \ (l, Z) \wedge \neg check\_deadlock \ l \ Z)$   
**apply** (*subst ta\_reaches\_ex\_iff*)  
**subgoal for**  $l \ u \ u' \ R$   
**by** (*rule sim\_complete\_bisim'.P1\_deadlocked\_compatible[where a = from\_R l R];*  
*(rule sim\_complete\_bisim'.P1\_P1')?*) (*auto intro: sim\_complete\_bisim'.P1\_P1'*)  
**using** *check\_deadlock\_deadlocked* **by** *auto*

**lemma** *deadlock\_check*:  
 $(\exists x_0 \in a_0. sim.sim.deadlock \ x_0) \longleftrightarrow (\exists l \ Z. reaches \ (l_0, Z_0) \ (l, Z) \wedge \neg check\_deadlock \ l \ Z)$   
**unfolding** *deadlock\_check'[symmetric]* *sim.sim.deadlock\_def* **by** *simp*

**end**

### 10.3 Operations

**Subset inclusion check for federations on DBMs lemma**  
 $S \subseteq R \longleftrightarrow S \cap -R = \{\}$

```

by auto

lemma
   $A \subseteq B \cup C \longleftrightarrow A \cap \neg B \cap \neg C = \{\}$ 
by auto

lemma
   $(A \cup B) \cap (C \cup D) = A \cap C \cup A \cap D \cup B \cap C \cup B \cap D$ 
by auto

lemma Le_le_inf[intro]:
   $Le\ (x :: \_ :: \text{linordered\_cancel\_ab\_monoid\_add}) \preceq \infty$ 
by (auto intro: linordered_monoid.order.strict_implies_order)

lemma dbm_entry_val_mono:
   $dbm\_entry\_val\ u\ a\ b\ e'\ \text{if}\ dbm\_entry\_val\ u\ a\ b\ e\ e \leq e'$ 
using that
  by cases
  (auto simp: DBM.less_eq intro: dbm_entry_val_mono1 dbm_entry_val_mono2 dbm_entry_val_mono3
    | simp add: DBM.less_eq dbm_entry_val.simps dbm_le_def
  )+

definition and_entry ::
   $\text{nat} \Rightarrow \text{nat} \Rightarrow ('t :: \{\text{linordered\_cancel\_ab\_monoid\_add}, \text{uminus}\})\ DBMEntry \Rightarrow 't\ DBM \Rightarrow 't\ DBM$ 
  where
   $\text{and\_entry}\ a\ b\ e\ M = (\lambda i\ j. \text{if } i = a \wedge j = b \text{ then } \min\ (M\ i\ j)\ e \text{ else } M\ i\ j)$ 

lemma and_entry_mono:
   $\text{and\_entry}\ a\ b\ e\ M\ i\ j \leq M\ i\ j$ 
by (auto simp: and_entry_def)

abbreviation clock_to_option  $a \equiv (\text{if } a > 0 \text{ then } \text{Some } a \text{ else } \text{None})$ 

definition
   $dbm\_entry\_val'\ u\ a\ b\ e \equiv dbm\_entry\_val\ u\ (\text{clock\_to\_option } a)\ (\text{clock\_to\_option } b)\ e$ 

definition
   $dbm\_minus\ n\ xs\ m = \text{concat}\ (\text{map}\ (\lambda(i, j). \text{map}\ (\lambda M. \text{and\_entry}\ j\ i\ (\text{neg\_dbm\_entry}\ (m\ i\ j))\ M)\ xs)$ 
     $[(i, j). i \leftarrow [0..<Suc\ n], j \leftarrow [0..<Suc\ n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty])$ 

locale Default_Nat_Clock_Numbering =
  fixes  $n :: \text{nat}$  and  $v :: \text{nat} \Rightarrow \text{nat}$ 
  assumes v_is_id:  $\forall\ c. c > 0 \wedge c \leq n \longrightarrow v\ c = c\ \forall\ c. c > n \longrightarrow v\ c = n + 1\ v\ 0 = n + 1$ 
begin

lemma v_id:
   $v\ c = c\ \text{if } v\ c \leq n$ 
using v_is_id that
  apply (cases  $c = 0$ )
  apply (simp; fail)
  apply (cases  $c \leq n$ ; auto)
done

```

**lemma** *le\_n*:  
 $c \leq n$  **if**  $v\ c \leq n$   
**using** *that v\_is\_id* **by** *auto*

**lemma** *gt\_0*:  
 $c > 0$  **if**  $v\ c \leq n$   
**using** *that v\_is\_id* **by** *auto*

**lemma** *le\_n\_iff*:  
 $v\ c \leq n \longleftrightarrow c \leq n \wedge c > 0$   
**using** *v\_is\_id* **by** *auto*

**lemma** *v\_0*:  
 $v\ c = 0 \longleftrightarrow \text{False}$   
**using** *v\_is\_id* **by** (*cases c > 0; simp; cases c > n; simp*)

**lemma** *surj\_on*:  $\forall\ k \leq n. k > 0 \longrightarrow (\exists\ c. v\ c = k)$   
**using** *v\_is\_id* **by** *blast*

**abbreviation** *zone\_of* ( $\llbracket \_ \rrbracket$ ) **where** *zone\_of*  $M \equiv [M]_{v,n}$

**abbreviation**  
 $\text{dbm\_fed } S \equiv \bigcup\ m \in S. \llbracket m \rrbracket$

**abbreviation**  
 $\text{dbm\_list } xs \equiv \text{dbm\_fed } (\text{set } xs)$

**lemma** *dbm\_fed\_singleton*:  
 $\text{dbm\_fed } \{m\} = [m]_{v,n}$   
**by** *auto*

**lemma** *dbm\_list\_single*:  
 $\text{dbm\_list } xs = [m]_{v,n}$  **if**  $\text{set } xs = \{m\}$   
**using** *that* **by** *auto*

**lemma** *dbm\_fed\_superset\_fold*:  
 $S \subseteq \text{dbm\_list } xs \longleftrightarrow \text{fold } (\lambda m\ S. S \cap - ([m]_{v,n}))\ xs\ S = \{\}$

**proof** (*induction xs arbitrary: S*)

**case** *Nil*  
**then show** *?case*  
**by** *auto*

**next**

**case** (*Cons m xs*)  
**have**  $S \subseteq \text{dbm\_list } (m \# xs) \longleftrightarrow S \cap - ([m]_{v,n}) \subseteq \text{dbm\_list } xs$   
**by** *auto*  
**moreover have**  $\dots \longleftrightarrow \text{fold } (\lambda m\ S. S \cap - ([m]_{v,n}))\ xs\ (S \cap - ([m]_{v,n})) = \{\}$   
**by** *fact*  
**ultimately show** *?case*  
**by** *simp*

**qed**

**lemma** *dbm\_fed\_superset\_fold'*:  
 $\text{dbm\_fed } S \subseteq \text{dbm\_list } xs \longleftrightarrow \text{dbm\_fed } (\text{fold } f\ xs\ S) = \{\}$  **if**  
 $\bigwedge\ m\ S. m \in \text{set } xs \implies \text{dbm\_fed } (f\ m\ S) = \text{dbm\_fed } S \cap - ([m]_{v,n})$

```

proof –
  from that have  $\text{fold } (\lambda m \ S. \ S \cap - ([m]_{v,n})) \ xs \ (\text{dbm\_fed } S) = \text{dbm\_fed } (\text{fold } f \ xs \ S)$ 
  proof (induction xs arbitrary: S)
    case Nil
    then show ?case
      by simp
  next
    case (Cons a xs)
    from Cons.prems have
       $\text{dbm\_fed } (\text{fold } f \ xs \ (f \ a \ S)) = \text{fold } (\lambda m \ S. \ S \cap - ([m]_{v,n})) \ xs \ (\text{dbm\_fed } (f \ a \ S))$ 
      by – (rule sym, rule Cons.IH, auto)
    then show ?case
      by (simp add: Cons.prems)
  qed
  then show ?thesis
    by (simp add: dbm_fed_superset_fold)
qed

```

```

lemma dbm_fed_superset_fold'':
   $\text{dbm\_list } S \subseteq \text{dbm\_list } xs \longleftrightarrow \text{dbm\_list } (\text{fold } f \ xs \ S) = \{\}$  if
   $\bigwedge m \ S. \ m \in \text{set } xs \implies \text{dbm\_list } (f \ m \ S) = \text{dbm\_list } S \cap - ([m]_{v,n})$ 
proof –
  from that have  $\text{fold } (\lambda m \ S. \ S \cap - ([m]_{v,n})) \ xs \ (\text{dbm\_list } S) = \text{dbm\_list } (\text{fold } f \ xs \ S)$ 
  proof (induction xs arbitrary: S)
    case Nil
    then show ?case
      by simp
  next
    case (Cons a xs)
    from Cons.prems have
       $\text{dbm\_list } (\text{fold } f \ xs \ (f \ a \ S)) = \text{fold } (\lambda m \ S. \ S \cap - ([m]_{v,n})) \ xs \ (\text{dbm\_list } (f \ a \ S))$ 
      by – (rule sym, rule Cons.IH, auto)
    then show ?case
      by (simp add: Cons.prems)
  qed
  then show ?thesis
    by (simp add: dbm_fed_superset_fold)
qed

```

```

lemma neg_inf:
   $\{u. \neg \text{dbm\_entry\_val } u \ a \ b \ e\} = \{\}$  if  $e = (\infty :: \_ \text{DBMEntry})$ 
  using that by auto

```

```

lemma dbm_entry_val'_diff_shift:
   $\text{dbm\_entry\_val}' (u \oplus d) \ c1 \ c2 \ (M \ c1 \ c2) \text{ if } \text{dbm\_entry\_val}' u \ c1 \ c2 \ (M \ c1 \ c2) \ 0 < c1 \ 0 < c2$ 
  using that unfolding dbm_entry_val'_def cval_add_def
  by (auto elim!: dbm_entry_val.cases intro!: dbm_entry_val.intros)

```

```

lemma dbm_entry_val_iff_bounded_Le1:
   $\text{dbm\_entry\_val } u \ (\text{Some } c1) \ \text{None } e \longleftrightarrow \text{Le } (u \ c1) \leq e$ 
  by (cases e) (auto simp: any_le_inf)

```

```

lemma dbm_entry_val_iff_bounded_Le2:
   $\text{dbm\_entry\_val } u \ \text{None } (\text{Some } c2) \ e \longleftrightarrow \text{Le } (- \ u \ c2) \leq e$ 

```

```

by (cases e) (auto simp: any_le_inf)

lemma dbm_entry_val_iff_bounded_Le3:
  dbm_entry_val u (Some c1) (Some c2) e  $\longleftrightarrow$  Le (u c1 - u c2)  $\leq$  e
  by (cases e) (auto simp: any_le_inf)

lemma dbm_entry_val'_iff_bounded:
  dbm_entry_val' u c1 c2 e  $\longleftrightarrow$  Le((if c1 > 0 then u c1 else 0) - (if c2 > 0 then u c2 else 0))
 $\leq$  e
  if c1 > 0  $\vee$  c2 > 0
  using that unfolding dbm_entry_val'_def
  by (auto simp:
    dbm_entry_val_iff_bounded_Le1 dbm_entry_val_iff_bounded_Le2 dbm_entry_val_iff_bounded_Le3
  )

context
  notes [simp] = dbm_entry_val'_def
begin

lemma neg_entry':
  {u.  $\neg$  dbm_entry_val' u a b e} = {u. dbm_entry_val' u b a (neg_dbm_entry e)}
  if e  $\neq$  ( $\infty :: \_ DBMEntry$ ) a > 0  $\vee$  b > 0

  using that by (cases e; cases a > 0; cases b > 0; auto 4 3 simp: le_minus_iff less_minus_iff)

lemma neg_unbounded:
  {u.  $\neg$  dbm_entry_val' u i j e} = {} if e = ( $\infty :: \_ DBMEntry$ )
  using that by auto

lemma and_entry_sound:
  u  $\vdash_{v,n}$  and_entry a b e M if dbm_entry_val' u a b e u  $\vdash_{v,n}$  M
  using that unfolding DBM_val_bounded_def
  by (cases a; cases b; auto simp: le_n_iff v_is_id(1) min_def v_0 and_entry_def)

lemma DBM_val_bounded_mono:
  u  $\vdash_{v,n}$  M if u  $\vdash_{v,n}$  M'  $\forall$  i  $\leq$  n.  $\forall$  j  $\leq$  n. M' i j  $\leq$  M i j
  using that unfolding DBM_val_bounded_def
  apply (safe; clarsimp simp: le_n_iff v_is_id(1) DBM.less_eq[symmetric])
  apply force
  apply (blast intro: dbm_entry_val_mono)+
  done

lemma and_entry_entry:
  dbm_entry_val' u a b e if u  $\vdash_{v,n}$  and_entry a b e M a  $\leq$  n b  $\leq$  n a > 0  $\vee$  b > 0
proof -
  from that have dbm_entry_val' u a b (min (M a b) e)
  unfolding DBM_val_bounded_def by (fastforce simp: le_n_iff v_is_id(1) and_entry_def)
  then show ?thesis
  by (auto intro: dbm_entry_val_mono)
qed

lemma and_entry_correct:
  [and_entry a b e M] $_{v,n}$  = [M] $_{v,n}$   $\cap$  {u. dbm_entry_val' u a b e}
  if a  $\leq$  n b  $\leq$  n a > 0  $\vee$  b > 0

```

**unfolding** *DBM\_zone\_repr\_def* **using** *that*  
**by** (*blast intro: and\_entry\_entry and\_entry\_sound DBM\_val\_bounded\_mono and\_entry\_mono*)

**lemma** *dbm\_list\_Int\_entry\_iff\_map*:

*dbm\_list xs*  $\cap \{u. \text{dbm\_entry\_val}' u \ i \ j \ e\} = \text{dbm\_list} (\text{map } (\lambda m. \text{and\_entry } i \ j \ e \ m) \ xs)$

**if**  $i \leq n \ j \leq n \ i > 0 \vee j > 0$

**unfolding** *dbm\_entry\_val'\_def*

**by** (*induction xs*;

*simp add: and\_entry\_correct[OF that, symmetric, unfolded dbm\_entry\_val'\_def] Int\_Un\_distrib2*  
 $\rangle$ )

**context**

**fixes**  $m :: \text{nat} \Rightarrow \text{nat} \Rightarrow ('a :: \{\text{time}\}) \text{DBMEntry}$

**assumes**  $\text{Le } 0 \preceq m \ 0 \ 0$

**begin**

**private lemma** *A*:

$\neg ([m]_{v,n}) =$

$(\bigcup (i, j) \in \{(i, j). i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$

$\{u. \neg \text{dbm\_entry\_val } u \ (\text{Some } i) \ (\text{Some } j) \ (m \ i \ j)\})$

$\cup (\bigcup i \in \{i. i > 0 \wedge i \leq n\}. \{u. \neg \text{dbm\_entry\_val } u \ (\text{Some } i) \ \text{None} \ (m \ i \ 0)\})$

$\cup (\bigcup j \in \{j. i > 0 \wedge i \leq n\}. \{u. \neg \text{dbm\_entry\_val } u \ \text{None} \ (\text{Some } j) \ (m \ 0 \ j)\})$

**unfolding** *DBM\_zone\_repr\_def*

**apply** *safe*

**subgoal for** *u*

**unfolding** *DBM\_val\_bounded\_def*

**apply** (*intro conjI impI allI*)

**subgoal**

**by** (*rule*  $\langle \text{Le } 0 \preceq m \ 0 \ 0 \rangle$ )

**subgoal for** *c*

**by** (*auto simp: le\_n\_iff v\_is\_id(1)*)

**subgoal for** *c*

**by** (*auto simp: le\_n\_iff v\_is\_id(1)*)

**subgoal for** *c1 c2*

**by** (*auto elim: allE[where  $x = c1$ ] simp: le\_n\_iff v\_is\_id(1)*)

**done**

**unfolding** *DBM\_val\_bounded\_def* **by** (*simp add: le\_n\_iff v\_is\_id(1)*) $+$

**private lemma** *B*:

$S \cap \neg ([m]_{v,n}) =$

$(\bigcup (i, j) \in \{(i, j). i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$

$S \cap \{u. \neg \text{dbm\_entry\_val } u \ (\text{Some } i) \ (\text{Some } j) \ (m \ i \ j)\})$

$\cup (\bigcup i \in \{i. i > 0 \wedge i \leq n\}. S \cap \{u. \neg \text{dbm\_entry\_val } u \ (\text{Some } i) \ \text{None} \ (m \ i \ 0)\})$

$\cup (\bigcup j \in \{j. i > 0 \wedge i \leq n\}. S \cap \{u. \neg \text{dbm\_entry\_val } u \ \text{None} \ (\text{Some } j) \ (m \ 0 \ j)\})$

**by** (*subst A*) *auto*

**private lemma** *UNION\_cong*:

$(\bigcup x \in S. f \ x) = (\bigcup x \in T. g \ x) \text{ if } S = T \wedge x. x \in T \implies f \ x = g \ x$

**by** (*simp add: that*)

**private lemma** *1*:

$S \cap \neg ([m]_{v,n}) =$

$(\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}. S \cap \{u. \neg \text{dbm\_entry\_val}' u \ i \ j \ (m \ i \ j)\})$

```

proof –
  have *:  $\{(i, j). (0 < i \vee 0 < j) \wedge i \leq n \wedge j \leq n\}$ 
    =  $\{(i, j). 0 < i \wedge 0 < j \wedge i \leq n \wedge j \leq n\}$ 
     $\cup \{(i, j). 0 < i \wedge 0 = j \wedge i \leq n \wedge j \leq n\}$ 
     $\cup \{(i, j). 0 = i \wedge 0 < j \wedge i \leq n \wedge j \leq n\}$ 
    by auto
  show ?thesis
    by (simp only:  $B \text{ UN\_Un } *$ ) (intro arg_cong2[where  $f = (\cup)$ ] UNION_cong; force)
qed

private lemma UNION_remove:
   $(\bigcup x \in S. f x) = (\bigcup x \in T. g x)$ 
  if  $T \subseteq S \wedge x. x \in T \implies f x = g x \wedge x. x \in S - T \implies f x = \{\}$ 
  using that by fastforce

private lemma 2:
   $(\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}. S \cap \{u. \neg \text{dbm\_entry\_val}' u i j (m i j)\})$ 
  =  $(\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m i j \neq \infty\}. S \cap \{u. \text{dbm\_entry\_val}' u j i (\text{neg\_dbm\_entry } (m i j))\})$ 
  apply (rule UNION_remove)
  apply force
  subgoal for  $x$ 
    by (cases  $x$ ; simp add: neg_entry'[simplified])
  by auto

lemma dbm_list_subtract:
   $\text{dbm\_list } xs \cap - ([m]_{v,n}) = \text{dbm\_list } (\text{dbm\_minus } n \text{ } xs \text{ } m)$ 
proof –
  have *:
     $\text{set } [(i, j). i \leftarrow [0..< \text{Suc } n], j \leftarrow [0..< \text{Suc } n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m i j \neq \infty]$ 
    =  $\{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m i j \neq \infty\}$ 
    by (auto simp del: upt.upt_Suc)
  show ?thesis
    unfolding dbm_minus_def
    apply (subst set_concat)
    apply (subst set_map)
    apply (subst *)
    apply (subst 1, subst 2)
    apply (subst UN_UN_flatten)
    apply (subst UN_simps)
    apply (rule UNION_cong[OF HOL.refl])
    apply (simp split del: if_split split: prod.splits)
    apply (subst dbm_list_Int_entry_iff_map[simplified])
    apply auto
  done
qed

end — Context for fixed DBM

end — Simplifier setup

lemma dbm_list_empty_check:
   $\text{dbm\_list } xs = \{\} \longleftrightarrow \text{list\_all } (\lambda m. [m]_{v,n} = \{\}) \text{ } xs$ 

```

```

unfolding list_all_iff by auto

lemmas dbm_list_superset_op =
  dbm_fed_superset_fold''[OF dbm_list_subtract[symmetric], unfolded dbm_list_empty_check]

end

context TA_Start_No_Ceiling
begin

sublocale dbm: Default_Nat_Clock_Numbering n v
  by unfold_locales (auto simp: v_def)

end

Down

Auxiliary lemma dbm_entry_le_iff:
  Le a ≤ Le b ⟷ a ≤ b
  Le a ≤ Lt b ⟷ a < b
  Lt a ≤ Le b ⟷ a ≤ b
  Lt a ≤ Lt b ⟷ a < b
  0 ≤ Le a ⟷ 0 ≤ a
  0 ≤ Lt a ⟷ 0 < a
  Le a ≤ 0 ⟷ a ≤ 0
  Lt a ≤ 0 ⟷ a ≤ 0
  ∞ ≤ x ⟷ x = ∞
  x ≤ ∞ ⟷ True
proof –
  show ∞ ≤ x ⟷ x = ∞
    by (cases x; auto)
qed (auto simp: any_le_inf DBM.neutral)

lemma dbm_entry_lt_iff:
  Le a < Le b ⟷ a < b
  Le a < Lt b ⟷ a < b
  Lt a < Le b ⟷ a ≤ b
  Lt a < Lt b ⟷ a < b
  0 < Le a ⟷ 0 < a
  0 < Lt a ⟷ 0 < a
  Le a < 0 ⟷ a < 0
  Lt a < 0 ⟷ a ≤ 0
  x < ∞ ⟷ x ≠ ∞
  ∞ < x ⟷ False
  by (auto simp: any_le_inf DBM.neutral DBM.less)

lemmas [dbm_entry_simps] = dbm_entry_le_iff(1–9) dbm_entry_lt_iff(1–9)

lemma Le_le_sum_iff:
  Le (y :: _ :: time) ≤ e ⟷ 0 ≤ e + Le (– y)
  by (cases e) (auto simp: DBM.add dbm_entry_le_iff)

lemma dense':

```

```

 $\exists c \geq a. c \leq b$  if  $a \leq b$  for  $a :: \_ :: time$ 
using dense  $\langle a \leq b \rangle$  by auto

lemma aux1:
   $- c \leq (a :: \_ :: time)$  if  $a \geq 0$   $c \geq 0$ 
  using that using dual_order.trans neg_le_0_iff_le by blast

lemma dbm_entries_dense:
   $\exists d \geq 0. Le (- d) \leq l \wedge Le (d :: \_ :: time) \leq r$  if  $0 \leq l$   $l \leq r$ 
  using that by (cases l; cases r; auto simp: dbm_entry_le_iff intro: aux1)

lemma dbm_entries_dense'_aux:
   $\exists d \geq 0. l + Le d \geq 0 \wedge 0 \leq r + Le (- d :: \_ :: time)$  if  $l \leq 0$   $l + r \geq 0$   $r \geq 0$ 
proof ((cases l; cases r), goal_cases)
  case (2 x1 x2)
  have  $\exists d \geq 0. 0 \leq x + d \wedge d < y$  if  $x \leq 0$   $0 < x + y$   $0 < y$  for  $x y :: 'a$ 
    using that by (metis add.right_inverse add_le_cancel_left leD leI linear)
  with 2 that show ?case
    by (auto simp: dbm_entry_le_iff DBM.add)
next
  case (3 x1)
  have  $\exists d \geq 0. 0 \leq x + d$  if  $x \leq 0$  for  $x :: 'a$ 
    using that by (metis add.right_inverse eq_iff neg_0_le_iff_le)
  with that 3 show ?case
    by (auto simp: dbm_entry_le_iff DBM.add)
next
  case (5 a b)
  have  $\exists d \geq 0. 0 < x + d \wedge d < y$  if  $x \leq 0$   $0 < x + y$  for  $x y :: 'a$ 
    using that by (smt add.right_inverse add_less_cancel_left leD le_less_trans linear neg_0_le_iff_le time_class.dense)
  with 5 that show ?case
    by (auto simp: dbm_entry_le_iff DBM.add)
next
  case (6 x2)
  have  $\exists d \geq 0. 0 < x + d$  if  $x \leq 0$  for  $x :: 'a$ 
    by (metis add.inverse_neutral add_minus_cancel add_strict_increasing2 eq_iff less_le less_minus_iff non_trivial_neg not_less_iff_gr_or_eq)
  with that 6 show ?case
    by (auto simp: dbm_entry_le_iff DBM.add)
qed (use that in  $\langle auto simp: dbm_entry_le_iff DBM.add \rangle$ )

lemma dbm_entries_dense':
   $\exists d \geq 0. l + Le d \geq 0 \wedge 0 \leq r + Le (- d :: \_ :: time)$  if  $l \leq 0$   $l + r \geq 0$ 
proof -
  from that have  $r \geq 0$ 
    by (meson add_decreasing order_refl order_trans)
  with that show ?thesis
    by (rule dbm_entries_dense'_aux)
qed

lemma (in time) non_trivial_pos:  $\exists x. x > 0$ 
  by (meson leD le_less_linear neg_le_0_iff_le non_trivial_neg)

lemma dbm_entries_dense_pos:

```

```

   $\exists d > 0. Le (d :: \_ :: time) \leq e$  if  $e > 0$ 
  supply dbm_entry_simps[simp]
proof (cases e)
case (Le d)
  with that show ?thesis
  by auto
next
case prems: (Lt d)
  with that have  $d > 0$ 
  by auto
  from dense[OF this] obtain  $z$  where  $z > 0$   $z < d$ 
  by auto
  then show ?thesis
  by (auto simp: prems)
next
case prems: INF
  obtain  $d :: 'a$  where  $d > 0$ 
  by atomize_elim (rule non_trivial_pos)
  then show ?thesis
  by (auto simp: prems)
qed

lemma le_minus_iff:
   $-x \leq (y :: \_ :: time) \longleftrightarrow 0 \leq y + x$ 
  by (metis add.commute add.right_inverse add_le_cancel_left)

lemma lt_minus_iff:
   $-x < (y :: \_ :: time) \longleftrightarrow 0 < y + x$ 
  by (metis add.commute add_less_cancel_right neg_eq_iff_add_eq_0)

context Default_Nat_Clock_Numbering
begin

lemma DBM_val_bounded_alt_def1:
   $u \vdash_{v,n} m \equiv$ 
   $Le\ 0 \leq m\ 0\ 0 \wedge$ 
   $(\forall c. c > 0 \wedge c \leq n \longrightarrow$ 
     $dbm\_entry\_val\ u\ None\ (Some\ c)\ (m\ 0\ c) \wedge$ 
     $dbm\_entry\_val\ u\ (Some\ c)\ None\ (m\ c\ 0)) \wedge$ 
   $(\forall c1\ c2. c1 > 0 \wedge c1 \leq n \wedge c2 > 0 \wedge c2 \leq n \longrightarrow dbm\_entry\_val\ u\ (Some\ c1)\ (Some\ c2)$ 
   $(m\ c1\ c2))$ 
  unfolding DBM_val_bounded_def by (rule eq_reflection) (auto simp: v_id le_n_iff)

lemma DBM_val_bounded_alt_def2:
   $u \vdash_{v,n} m \equiv$ 
   $Le\ 0 \leq m\ 0\ 0 \wedge$ 
   $(\forall c1\ c2. (c1 \neq 0 \vee c2 \neq 0) \wedge c1 \leq n \wedge c2 \leq n \longrightarrow dbm\_entry\_val'\ u\ c1\ c2\ (m\ c1\ c2))$ 
  unfolding DBM_val_bounded_alt_def1 dbm_entry_val'_def DBM.less_eq
  by (rule eq_reflection; clarsimp; safe; blast)

lemma DBM_val_bounded_altI:
  assumes
     $Le\ 0 \leq m\ 0\ 0$ 
     $\bigwedge c1\ c2. (c1 \neq 0 \vee c2 \neq 0) \wedge c1 \leq n \wedge c2 \leq n \implies dbm\_entry\_val'\ u\ c1\ c2\ (m\ c1\ c2)$ 

```

```

shows
   $u \in \llbracket m \rrbracket$ 
unfolding DBM_zone_repr_def DBM_val_bounded_alt_def2 using assms by auto

lemma dbm_entry_val'_delay1:
  dbm_entry_val' u c1 c2 (m c1 c2) if dbm_entry_val' (u  $\oplus$  d) c1 c2 (m c1 c2)  $d \geq 0$   $c1 > 0$ 
  using that unfolding dbm_entry_val'_def
  by (cases m c1 c2)
    (auto 0 2
      dest: add_strict_increasing2 add_increasing intro!: dbm_entry_val.intros
      simp: cval_add_def
    )

lemma dbm_entry_val'_delay2:
  dbm_entry_val' u (0 :: nat) c2 (m c1 c2) if
  dbm_entry_val' (u  $\oplus$  d) c1 c2 (m c1 c2)  $d \geq 0$ 
   $c1 > 0$   $c2 > 0$   $c1 \leq n$   $c2 \leq n$ 
   $\forall c \leq n. c > 0 \longrightarrow u c \geq 0$ 
proof -
  have - u c2  $\leq$  da
  if 0  $\leq$  d
    0 < c1
    0 < c2
    c1  $\leq$  n
    c2  $\leq$  n
     $\forall c \leq n. 0 < c \longrightarrow 0 \leq u c$ 
    m c1 c2 = Le da
    u c1 - u c2  $\leq$  da
  for da :: 'a
  using that by (auto simp: algebra_simps le_minus_iff)
  moreover have - u c2 < da
  if 0  $\leq$  d
    0 < c1
    0 < c2
    c1  $\leq$  n
    c2  $\leq$  n
     $\forall c \leq n. 0 < c \longrightarrow 0 \leq u c$ 
    m c1 c2 = Lt da
    u c1 - u c2 < da
  for da :: 'a
  using that by (auto simp: algebra_simps lt_minus_iff intro: dual_order.strict_trans2)
  ultimately show ?thesis
  using that unfolding dbm_entry_val'_def
  by (auto elim!: dbm_entry_val.cases intro!: dbm_entry_val.intros simp: cval_add_def)
qed

lemma dbm_entry_val'_nonneg_bound:
  dbm_entry_val' u (0 :: nat) c (Le 0) if u c  $\geq 0$   $c > 0$ 
  using that unfolding dbm_entry_val'_def by auto

lemma neg_diag_empty_spec:
   $\llbracket M \rrbracket = \{\}$  if  $i \leq n$   $M i i < 0$ 
  using that by (meson neg_diag_empty v_is_id(1))

```

**lemma** *in DBM D*:  
 $\text{dbm\_entry\_val}'\ u\ c1\ c2\ (M\ c1\ c2)$  **if**  $u \in \llbracket M \rrbracket$   $c1 \neq 0 \vee c2 \neq 0$   $c1 \leq n$   $c2 \leq n$   
**using** *that unfolding zone\_time\_pre\_def DBM\_zone\_repr\_def DBM\_val\_bounded\_alt\_def2*  
**by auto**

**context**  
**fixes**  $M :: ('t::time)\ DBM$   
**assumes**  $\llbracket M \rrbracket \neq \{\}$   
**begin**

**lemma** *non\_empty\_diag\_0\_0*:  $M\ 0\ 0 \geq 0$   
**using**  $\langle \llbracket M \rrbracket \neq \{\} \rangle$  *neg\_diag\_empty\_spec[of 0 M]* **leI** **by auto**

**lemma**  $M\ k\ 0$ :  $M\ k\ 0 \geq 0$  **if**  $\forall\ u \in \llbracket M \rrbracket. \forall\ c \leq n. c > 0 \longrightarrow u\ c \geq 0\ k \leq n$   
**proof** (*cases k = 0*)

**case** *True* **with** *non\_empty\_diag\_0\_0* **show** *?thesis*  
**by auto**

**next**

**case** *False*

**from**  $\langle \llbracket M \rrbracket \neq \{\} \rangle$  **obtain**  $u$  **where**  $u \in \llbracket M \rrbracket$

**by auto**

**with** *False that(1)*  $\langle k \leq n \rangle$  **have**  $u\ k \geq 0$

**by auto**

**from**  $\langle u \in \_ \rangle\ \langle k \neq 0 \rangle\ \langle k \leq n \rangle$  **have**  $\text{dbm\_entry\_val}'\ u\ k\ 0\ (M\ k\ 0)$

**unfolding** *DBM\_zone\_repr\_def DBM\_val\_bounded\_alt\_def2* **by auto**

**with**  $\langle k \neq 0 \rangle$  **have**  $Le\ (u\ k) \leq M\ k\ 0$

**by** (*simp add: dbm\_entry\_val'\_iff\_bounded*)

**with**  $\langle u\ k \geq 0 \rangle$  **show**  $M\ k\ 0 \geq 0$

**by** (*cases M k 0*) (*auto simp: dbm\_entry\_le\_iff*)

**qed**

**lemma** *non\_empty\_cycle\_free*:  
*cycle\_free M n*  
**using**  $\langle \llbracket M \rrbracket \neq \{\} \rangle$  *non\_empty\_cycle\_free v\_is\_id(1)* **by blast**

**lemma** *canonical\_saturated\_2*:

**assumes**  $Le\ r \leq M\ 0\ c$

**and**  $Le\ (-\ r) \leq M\ c\ 0$

**and** *cycle\_free M n*

**and** *canonical M n*

**and**  $c \leq n$

**and**  $c > 0$

**obtains**  $u$  **where**  $u \in \llbracket M \rrbracket$   $u\ c = -\ r$

**using** *assms v\_0* **by** (*auto simp: v\_is\_id intro: canonical\_saturated\_2[of r M v c n]*)

**lemma**  $M\ 0\ k$ :  $M\ 0\ k \leq 0$

**if** *canonical M n*  $M\ 0\ 0 \leq 0\ \forall\ u \in \llbracket M \rrbracket. \forall\ c \leq n. c > 0 \longrightarrow u\ c \geq 0\ k \leq n$

**proof** (*cases k = 0*)

**case** *True*

**with**  $\langle M\ 0\ 0 \leq 0 \rangle$  **show** *?thesis*

**by auto**

**next**

**case** *False*

**show** *?thesis*

```

proof (rule ccontr)
  assume  $\neg M\ 0\ k \leq 0$ 
  then have  $M\ 0\ k > 0$ 
    by auto
  from  $that(3)\ \langle k \leq n \rangle$  have  $M\ k\ 0 \geq 0$ 
    by (rule  $M\_k\_0$ )
  from  $\langle M\ 0\ k > 0 \rangle$  obtain  $d$  where
     $Le\ d \leq M\ 0\ k\ d > 0$ 
    by (rule  $dbm\_entries\_dense\_pos[elim\_format]$ ) auto
  with  $\langle M\ k\ 0 \geq 0 \rangle$  have  $Le\ (-d) \leq M\ k\ 0$ 
    by (auto simp:  $dbm\_entry\_le\_iff$  intro:  $order.trans[rotated]$ )
  with  $\langle canonical\ M\ n \rangle\ \langle Le\ d \leq M\ 0\ k \rangle$  obtain  $u$  where
     $u \in \llbracket M \rrbracket\ u\ k = -d$ 
    using  $v\_0\ False\ \langle k \leq n \rangle$ 
    by  $-(rule\ canonical\_saturated\_2[of\ d],\ auto\ simp:\ non\_empty\_cycle\_free)$ 
  with  $\langle d > 0 \rangle\ that(3)\ False\ \langle k \leq n \rangle$  show  $False$ 
    by fastforce
qed
qed

end

end

Definition definition
   $down :: nat \Rightarrow ('t::linordered\_cancel\_ab\_monoid\_add)\ DBM \Rightarrow 't\ DBM$ 
where
   $down\ n\ M \equiv$ 
   $\lambda\ i\ j.\ if\ i = 0 \wedge j > 0\ then\ Min\ (\{Le\ 0\} \cup \{M\ k\ j \mid k.\ 1 \leq k \wedge k \leq n\})\ else\ M\ i\ j$ 

Correctness context  $Default\_Nat\_Clock\_Numbering$ 
begin

  sublocale  $Alpha\_defs\ \{1..n\}.$ 

  context
    fixes  $M :: ('t::time)\ DBM$ 
  begin

  lemma  $down\_complete:$   $u \in \llbracket down\ n\ M \rrbracket\ \text{if}\ u \in \llbracket M \rrbracket^\downarrow\ \forall\ c \leq n.\ c > 0 \longrightarrow u\ c \geq 0$ 
  proof (rule  $DBM\_val\_bounded\_altI,\ goal\_cases$ )
    case 1
    with  $\langle u \in \_ \rangle$  show ?case
      unfolding  $down\_def\ zone\_time\_pre\_def$  by (auto intro:  $non\_empty\_diag\_0\_0\ simp:\ neutral[symmetric]$ )
    next
    case  $prems:\ (2\ c1\ c2)$ 
    then consider  $c1 > 0 \mid c1 = 0\ c2 > 0$ 
      by auto
    then show ?case
  proof cases
    case 1
    with  $prems\ \langle u \in \_ \rangle$  show ?thesis

```

```

    unfolding zone_time_pre_def down_def by (auto intro: dbm_entry_val'_delay1 dest:
in_DBM_D)
  next
  case 2
  from  $\langle u \in \llbracket M \rrbracket^\downarrow \rangle$  obtain  $d$  where  $d: 0 \leq d \wedge u \oplus d \in \llbracket M \rrbracket$ 
    unfolding zone_time_pre_def by auto
  let  $?e = \text{Min} (\{Le\ 0\} \cup \{M\ k\ c2 \mid k. 1 \leq k \wedge k \leq n\})$ 
  have  $?e \in \{Le\ 0\} \cup \{M\ k\ c2 \mid k. 1 \leq k \wedge k \leq n\}$ 
    by (intro Min_in) auto
  then consider  $?e = Le\ 0 \mid k$  where  $?e = M\ k\ c2 \wedge k > 0 \wedge k \leq n$ 
    by auto
  then show  $?thesis$ 
    using prems that(2) d 2 unfolding down_def
  by cases (auto intro: dbm_entry_val'_delay2 dbm_entry_val'_nonneg_bound in_DBM_D)
qed
qed

```

```

lemma down_sound:  $u \in \llbracket M \rrbracket^\downarrow$  if  $u \in \llbracket \text{down } n\ M \rrbracket$  canonical  $M\ n$ 
proof -
  note [simp] = dbm_entry_simps and [intro] = order.trans add_right_mono
  from  $\langle u \in \_ \rangle$  non_empty_diag_0_0[of down n M] have  $Le\ 0 \leq M\ 0\ 0$ 
    by (auto simp: down_def neutral)
  note * = in_DBM_D[OF  $\langle u \in \_ \rangle$ ]
  define  $l$  where  $l = \text{Min} (\{M\ 0\ c + Le\ (u\ c) \mid c. 0 < c \wedge c \leq n\} \cup \{Le\ 0\})$ 
    — maximum current violation of the future bounds
  define  $r$  where  $r = \text{Min} (\{M\ c\ 0 + Le\ (-\ u\ c) \mid c. 0 < c \wedge c \leq n\} \cup \{\infty\})$ 
    — slack for shifting upwards
  have  $0 \leq l + r \wedge l \leq 0$ 
  proof -
    have
       $l \in \{M\ 0\ c + Le\ (u\ c) \mid c. 0 < c \wedge c \leq n\} \cup \{Le\ 0\}$ 
       $r \in \{M\ c\ 0 + Le\ (-\ u\ c) \mid c. 0 < c \wedge c \leq n\} \cup \{\infty\}$ 
      unfolding l_def r_def by (intro Min_in; simp)+
    from  $\langle l \in \_ \rangle$  show  $l \leq 0$ 
      unfolding l_def by (auto intro: Min_le simp: DBM.neutral)
    from  $\langle l \in \_ \rangle \langle r \in \_ \rangle$  show  $0 \leq l + r$ 
  proof (safe, goal_cases)
    case prems: (1 c1 c2)
    with  $\langle u \in \_ \rangle$  have  $Le\ (u\ c2 - u\ c1) \leq M\ c2\ c1$ 
      by (auto 0 2 dest: in_DBM_D simp: dbm_entry_val'_iff_bounded down_def)
    also from prems  $\langle \text{canonical } M\ n \rangle$  have  $M\ c2\ 0 + M\ 0\ c1 \geq M\ c2\ c1$ 
      by auto
    finally have  $0 \leq M\ c2\ 0 + M\ 0\ c1 + (Le\ (u\ c1) + Le\ (-\ u\ c2))$ 
      by (simp add: DBM.add Le_le_sum_iff)
    then show ?case
      by (simp add: algebra_simps)
  next
  case (3 c)
  with  $\langle u \in \_ \rangle$  have  $Le\ (u\ c) \leq M\ c\ 0$ 
    by (auto 0 2 dest: in_DBM_D simp: dbm_entry_val'_iff_bounded down_def)
  then show ?case
    by (auto simp: DBM.add Le_le_sum_iff)
qed auto
qed

```

```

from dbm_entries_dense'[OF this(2,1)] obtain d where
   $d \geq 0 \ 0 \leq l + Le \ d \ 0 \leq r + Le \ (- \ d)$ 
  by auto
have  $u \oplus d \in \llbracket M \rrbracket$ 
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  from  $\langle Le \ 0 \leq M \ 0 \ 0 \rangle$  show ?case .
next
  case (2 c1 c2)
  with * have **: dbm_entry_val' u c1 c2 (down n M c1 c2)
    by auto
  from 2 consider
     $c1 \leq n \ c2 \leq n \ c1 > 0 \ c2 > 0$ 
     $| \ c1 = 0 \ c2 \leq n \ c2 > 0 \ | \ c2 = 0 \ c1 \leq n \ c1 > 0$ 
    by auto
  then show ?case
proof cases
  case 1
  then show ?thesis
    using ** unfolding down_def by (auto intro: dbm_entry_val'_diff_shift)
next
  case 2
  then have  $l \leq (M \ 0 \ c2 + Le \ (u \ c2))$ 
    unfolding l_def by (auto intro: Min_le)
  with  $\langle 0 \leq l + Le \ d \rangle$  have  $0 \leq M \ 0 \ c2 + Le \ (u \ c2) + Le \ d$ 
    by auto
  with 2 show ?thesis
    unfolding down_def dbm_entry_val'_def
    by (cases M 0 c2)
    (auto 4 3 simp: cval_add_def DBM.add algebra_simps lt_minus_iff le_minus_iff)
next
  case 3
  then have  $r \leq M \ c1 \ 0 + Le \ (- \ u \ c1)$ 
    unfolding r_def by (auto intro: Min_le)
  with  $\langle 0 \leq r + Le \ (- \ d) \rangle$  have  $0 \leq M \ c1 \ 0 + Le \ (- \ u \ c1) + Le \ (- \ d)$ 
    by auto
  with 3 ** show ?thesis
    unfolding down_def dbm_entry_val'_def
    by (auto elim!: dbm_entry_val.cases simp: cval_add_def algebra_simps DBM.add)
qed
qed
with  $\langle d \geq 0 \rangle$  show ?thesis
  unfolding zone_time_pre_def cval_add_def by auto
qed

```

```

lemma down_canonical:
  canonical (down n M) n
  if assms: canonical M n  $\llbracket M \rrbracket \neq \{\}$   $\forall \ u \in \llbracket M \rrbracket. \forall \ c \leq n. \ c > 0 \longrightarrow u \ c \geq 0 \ M \ 0 \ 0 \leq 0$ 
proof -
  from non_empty_diag_0_0[OF  $\langle \llbracket M \rrbracket \neq \{\} \rangle$ ] have  $M \ 0 \ 0 \geq 0$  .
  with  $\langle M \ 0 \ 0 \leq 0 \rangle$  have  $M \ 0 \ 0 = 0$ 
    by auto
  note  $M\_0\_k = M\_0\_k$ [OF that(2,1,4,3)] and  $M\_k\_0 = M\_k\_0$ [OF that(2,3)]
  have Suc_0_le_iff:  $Suc \ 0 \leq x \longleftrightarrow 0 < x$  for x

```

```

by auto
define S where  $S\ j = \text{Min} (\{Le\ 0\} \cup \{M\ k\ j \mid k. 1 \leq k \wedge k \leq n\})$  for j
{ fix j :: nat
  consider (0)  $S\ j = 0 \ \forall \ i. 1 \leq i \wedge i \leq n \longrightarrow M\ i\ j \geq 0$ 
    | (entry) i where
       $S\ j = M\ i\ j\ 0 < i \leq n \ M\ i\ j \leq 0 \ \forall \ k. 1 \leq k \wedge k \leq n \longrightarrow M\ i\ j \leq M\ k\ j$ 
    unfolding S_def neutral
    using Min_in[of  $\{Le\ 0\} \cup \{M\ k\ j \mid k. 1 \leq k \wedge k \leq n\}$ ]
    using Min_le[of  $\{Le\ 0\} \cup \{M\ k\ j \mid k. 1 \leq k \wedge k \leq n\}$ ]
    by (simp, safe) auto
  } note S_cases = this
show ?thesis
  apply (intro allI impI; elim conjE)
  unfolding down_def S_def[symmetric]
  apply clarsimp
  apply safe
  subgoal premises prems for i j k
    using  $\langle M\ 0\ 0 \geq 0 \rangle$  by (blast intro: add_increasing)
  subgoal for i j k
    using  $\langle \text{canonical } M\ n \rangle$ 
    by (cases rule: S_cases[of k]; cases rule: S_cases[of j])
      (auto intro: order.trans simp: Suc_0_le_iff)
  subgoal premises prems for i j k
  proof -
    from prems have  $M\ 0\ k \leq S\ k$ 
    apply (cases rule: S_cases[of k])
    subgoal
      using  $M\_0\_k$ [of k] by auto
    subgoal for i'
      using  $M\_0\_k$   $\langle \text{canonical } M\ n \rangle$  by (metis add.left_neutral add_right_mono dual_order.trans)
    done
    from  $\langle \text{canonical } M\ n \rangle$  prems have  $M\ i\ k \leq M\ i\ 0 + M\ 0\ k$ 
    by auto
    also from  $\langle \_ \leq S\ k \rangle$  have  $\dots \leq M\ i\ 0 + S\ k$ 
    by (simp add: add_left_mono)
    finally show ?thesis .
  qed
subgoal for i j k
  using  $\langle \text{canonical } M\ n \rangle \ \langle M\ 0\ 0 \leq 0 \rangle$ 
  by (smt  $M\_k\_0$  S_cases add_increasing Orderings.order.trans)
apply (use  $\langle \text{canonical } M\ n \rangle$  in simp; fail)+
subgoal for i j k
  using  $\langle \text{canonical } M\ n \rangle \ \langle M\ 0\ 0 \leq 0 \rangle$ 
  by (smt  $M\_k\_0$  S_cases add_increasing Orderings.order.trans)
apply (use  $\langle \text{canonical } M\ n \rangle$  in simp_all)
done
qed
end

end

end

Free

```

**Definition definition**

$free :: nat \Rightarrow ('t::linordered\_cancel\_ab\_monoid\_add) DBM \Rightarrow nat \Rightarrow 't DBM$

**where**

$free\ n\ M\ x \equiv$

$\lambda\ i\ j. \text{ if } i = x \wedge j \neq x \text{ then } \infty \text{ else if } i \neq x \wedge j = x \text{ then } M\ i\ 0 \text{ else } M\ i\ j$

**definition repair\_pair where**

$repair\_pair\ n\ M\ a\ b = FWI\ (FWI\ M\ n\ b)\ n\ a$

**definition**

$and\_entry\_repair\ n\ a\ b\ e\ M \equiv repair\_pair\ n\ (and\_entry\ a\ b\ e\ M)\ a\ b$

**definition**

$restrict\_zero\ n\ M\ x \equiv$

$let$

$M1 = and\_entry\ x\ 0\ (Le\ 0)\ M;$

$M2 = and\_entry\ 0\ x\ (Le\ 0)\ M1$

$in\ repair\_pair\ n\ M2\ x\ 0$

**definition**

$pre\_reset\ n\ M\ x \equiv free\ n\ (restrict\_zero\ n\ M\ x)\ x$

**definition**

$pre\_reset\_list\ n\ M\ r \equiv fold\ (\lambda\ x\ M. pre\_reset\ n\ M\ x)\ r\ M$

**Auxiliary lemma repair\_pair\_characteristic:**

**assumes**  $canonical\_subs\ n\ I\ M$

**and**  $I \subseteq \{0..n\}$

**and**  $a \leq n\ b \leq n$

**shows**  $canonical\_subs\ n\ (I \cup \{a,b\})\ (repair\_pair\ n\ M\ a\ b) \vee (\exists\ i \leq n. repair\_pair\ n\ M\ a\ b\ i\ i < 0)$

**proof** –

**from**  $fwi\_characteristic[OF\ assms(1,2,4)]$  **have**

$canonical\_subs\ n\ (I \cup \{b\})\ (FWI\ M\ n\ b) \vee (\exists\ i \leq n. FWI\ M\ n\ b\ i\ i < 0)$

**by** *auto*

**then show** *?thesis*

**proof**

**assume**  $canonical\_subs\ n\ (I \cup \{b\})\ (FWI\ M\ n\ b)$

**from**  $fwi\_characteristic[OF\ this\ \langle a \leq n \rangle]$   $assms(2)\ \langle b \leq n \rangle$  **show** *?thesis*

**unfolding**  $repair\_pair\_def$  **by** *simp*

**next**

**assume**  $\exists\ i \leq n. FWI\ M\ n\ b\ i\ i < 0$

**then have**  $\exists\ i \leq n. repair\_pair\ n\ M\ a\ b\ i\ i < 0$

**unfolding**  $repair\_pair\_def$

**apply** *safe*

**subgoal for** *i*

**apply**  $(inst\_existentials\ i)$

**apply** *assumption*

**apply**  $(frule\ FWI\_mono[where\ M = FWI\ M\ n\ b\ and\ k = a])$

**apply** *auto*

**done**

**done**

**then show** *?thesis ..*

```

qed
qed

lemma repair_pair_mono:
  assumes  $i \leq n$ 
    and  $j \leq n$ 
  shows  $\text{repair\_pair } n \ M \ a \ b \ i \ j \leq M \ i \ j$ 
  unfolding repair_pair_def by (auto intro: FWI_mono assms order.trans)

context Default_Nat_Clock_Numbering
begin

lemmas FWI_zone_equiv = FWI_zone_equiv[OF surj_on, symmetric]

lemma repair_pair_zone_equiv:
   $\llbracket \text{repair\_pair } n \ M \ a \ b \rrbracket = \llbracket M \rrbracket$  if  $a \leq n \ b \leq n$ 
  using that unfolding repair_pair_def by (simp add: FWI_zone_equiv)

context
  fixes  $c1 \ c2 \ c \ x :: \text{nat}$ 
  notes  $[simp] = \text{dbm\_entry\_val'}\_iff\_bounded \ \text{dbm\_entry\_sims} \ DBM.add \ \text{algebra\_sims}$ 
begin

lemma dbm_entry_val'_diag_iff:  $\text{dbm\_entry\_val'} \ u \ c \ c \ e \longleftrightarrow e \geq 0$  if  $c > 0$ 
  using that by (cases e) auto

lemma dbm_entry_val'_inf:  $\text{dbm\_entry\_val'} \ u \ c1 \ c2 \ \infty \longleftrightarrow \text{True}$ 
  unfolding dbm_entry_val'_def by auto

lemma dbm_entry_val'_reset_1:
   $\text{dbm\_entry\_val'} \ (u(x := d)) \ x \ c \ e \longleftrightarrow \text{dbm\_entry\_val'} \ u \ 0 \ c \ (e + Le \ (-d))$ 
  if  $d \geq 0 \ c \neq x \ c > 0 \ x > 0$ 
  using that  $\langle d \geq 0 \rangle$  by (cases e) auto

lemma dbm_entry_val'_reset_2:
   $\text{dbm\_entry\_val'} \ (u(x := d)) \ c \ x \ e \longleftrightarrow \text{dbm\_entry\_val'} \ u \ c \ (0 :: \text{nat}) \ (e + Le \ d)$ 
  if  $d \geq 0 \ c \neq x \ c > 0 \ x > 0$ 
  using that  $\langle d \geq 0 \rangle$  by (cases e) auto

lemma dbm_entry_val'_reset_2':
   $\text{dbm\_entry\_val'} \ (u(x := d)) \ 0 \ x \ e \longleftrightarrow Le \ (-d) \leq e$  if  $d \geq 0 \ x > 0$ 
  using that  $\langle d \geq 0 \rangle$  by (cases e) auto

lemma dbm_entry_val'_reset_3:
   $\text{dbm\_entry\_val'} \ (u(x := d)) \ c1 \ c2 \ e \longleftrightarrow \text{dbm\_entry\_val'} \ u \ c1 \ c2 \ e$  if  $c1 \neq x \ c2 \neq x$  for  $e$ 
  using that unfolding dbm_entry_val'_def by (cases e) auto

end

Correctness context
  fixes  $M :: ('t::\text{time}) \ DBM$ 
begin

```

```

lemma free_complete:  $u(x := d) \in \llbracket \text{free } n \ M \ x \rrbracket$ 
  if assms:  $u \in \llbracket M \rrbracket \ d \geq 0 \ x > 0 \ \forall c \leq n. \ M \ c \ c \geq 0$ 
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  with  $\langle \_ \in \llbracket M \rrbracket \rangle$  show ?case
    unfolding free_def by (auto simp: neutral[symmetric] intro: non_empty_diag_0_0)
next
  case prems: (2 c1 c2)
  then have  $c1 \leq n \ c2 \leq n$ 
    by auto
  note [simp] = dbm_entry_simps
  have *:  $Le \ (u \ c1) \leq M \ c1 \ 0 + Le \ d \text{ if } c1 > 0$ 
  proof -
    from  $\langle \_ \in \llbracket M \rrbracket \rangle \ \langle c1 > 0 \rangle \ \langle c1 \leq n \rangle$  have  $Le \ (u \ c1) \leq M \ c1 \ 0$ 
      by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
    with  $\langle d \geq 0 \rangle$  show ?thesis
      by (simp add: algebra_simps add_increasing)
  qed
  have dbm_entry_val'  $(u(x := d)) \ c1 \ x \ (M \ c1 \ 0)$  if  $c1 \neq x$ 
  proof (cases  $c1 = 0$ )
    case True
    with that show ?thesis
      using assms(4)  $\langle d \geq 0 \rangle$  by (auto intro: order.trans[rotated] simp: dbm_entry_val'_reset_2')
  next
    case False
    with that  $\langle x > 0 \rangle$  show ?thesis
      by (subst dbm_entry_val'_reset_2[OF  $\langle d \geq 0 \rangle$ ]) (auto simp: dbm_entry_val'_iff_bounded)
  *)
  qed
  with prems in_DBM_D[OF  $\langle \_ \in \llbracket M \rrbracket \rangle$ ] that(4) show ?case
    by (auto simp: free_def dbm_entry_val'_diag_iff dbm_entry_val'_inf dbm_entry_val'_reset_3)
  qed

lemma free_sound:  $\exists d \geq 0. \ u(x := d) \in \llbracket M \rrbracket \ u \ x \geq 0$ 
  if assms:  $u \in \llbracket \text{free } n \ M \ x \rrbracket \ x > 0 \ x \leq n \text{ canonical } M \ n \ M \ 0 \ x \leq 0 \ M \ 0 \ 0 \leq 0$ 
proof -
  define l where  $l = \text{Min} \ (\{M \ c \ x + Le \ (- \ u \ c) \mid c. \ 0 < c \wedge c \leq n \wedge c \neq x\} \cup \{M \ 0 \ x\})$ 
  define r where  $r = \text{Min} \ (\{M \ x \ c + Le \ (u \ c) \mid c. \ 0 < c \wedge c \leq n \wedge c \neq x\} \cup \{M \ x \ 0\})$ 
  from non_empty_diag_0_0  $\langle u \in \_ \rangle \ \langle x > 0 \rangle$  have  $0 \leq M \ 0 \ 0$ 
    unfolding free_def by fastforce
  note [simp] = dbm_entry_simps and [intro] = order.trans add_right_mono
  have  $0 \leq l + r \ l \leq 0$ 
  proof -
    have
       $l \in \{M \ c \ x + Le \ (- \ u \ c) \mid c. \ 0 < c \wedge c \leq n \wedge c \neq x\} \cup \{M \ 0 \ x\}$ 
       $r \in \{M \ x \ c + Le \ (u \ c) \mid c. \ 0 < c \wedge c \leq n \wedge c \neq x\} \cup \{M \ x \ 0\}$ 
      unfolding l_def r_def by (intro Min_in; simp)+
    from  $\langle l \in \_ \rangle \ \langle M \ 0 \ x \leq 0 \rangle$  show  $l \leq 0$ 
      unfolding l_def by - (rule order.trans[rotated], auto intro: Min_le simp: DBM.neutral)
    from  $\langle l \in \_ \rangle \ \langle r \in \_ \rangle$  show  $0 \leq l + r$ 
  proof (safe, goal_cases)
    case prems: (1 c1 c2)
    with  $\langle \text{canonical } M \ n \rangle \ \langle x \leq n \rangle$  have  $M \ c1 \ x + M \ x \ c2 \geq M \ c1 \ c2$ 
      by auto
  end
end

```

```

from prems  $\langle u \in \_ \rangle$  have  $Le (u \ c1 - u \ c2) \leq M \ c1 \ c2$ 
  unfolding free_def by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
with  $\langle M \ c1 \ c2 \leq M \ c1 \ x + M \ x \ c2 \rangle$  have  $Le (u \ c1 - u \ c2) \leq M \ c1 \ x + M \ x \ c2$ 
  by auto
then have  $0 \leq M \ c1 \ x + M \ x \ c2 + (Le (u \ c2) + Le (- u \ c1))$ 
  by (simp add: DBM.add Le_le_sum_iff)
then show ?case
  by (simp add: algebra_simps)
next
case prems: (2 c)
from prems  $\langle u \in \_ \rangle$  have  $Le (u \ c) \leq M \ c \ 0$ 
  unfolding free_def by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
also from prems  $\langle \text{canonical } M \ n \rangle \langle x \leq n \rangle$  have  $\dots \leq M \ c \ x + M \ x \ 0$ 
  by auto
finally show ?case
  by (simp add: algebra_simps Le_le_sum_iff)
next
case prems: (3 c)
with  $\langle u \in \_ \rangle \langle x > 0 \rangle$  have  $Le (- u \ c) \leq M \ 0 \ c$ 
  unfolding free_def by (auto simp: dbm_entry_val'_iff_bounded dest: in_DBM_D[of _
- 0 c])
also from prems  $\langle \text{canonical } M \ n \rangle \langle x \leq n \rangle$  have  $\dots \leq M \ 0 \ x + M \ x \ c$ 
  by auto
finally show ?case
  by (simp add: algebra_simps Le_le_sum_iff)
next
case 4
from  $\langle 0 \leq M \ 0 \ 0 \rangle \langle \text{canonical } M \ n \rangle \langle x \leq n \rangle$  show ?case
  by auto
qed
qed
from dbm_entries_dense'[OF this(2,1)] obtain d where
   $d \geq 0 \ 0 \leq l + Le \ d \ 0 \leq r + Le (- d)$ 
  by auto
have  $u(x := d) \in \llbracket M \rrbracket$ 
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  from  $\langle 0 \leq M \ 0 \ 0 \rangle$  show ?case unfolding DBM.neutral .
next
case prems: (2 c1 c2)
then have **:  $dbm\_entry\_val' \ u \ c1 \ c2 \ (\text{free } n \ M \ x \ c1 \ c2)$ 
  by (auto intro: in_DBM_D[OF  $\langle u \in \_ \rangle$ ])
show ?case
proof -
  have  $Le (d - u \ c2) \leq M \ x \ c2$ 
  if  $0 < x \ c1 = x \ c2 \neq x \ x \leq n \ c2 \leq n \ 0 < c2$ 
  proof -
  from that have  $r \leq M \ x \ c2 + Le (u \ c2)$ 
  unfolding r_def by (intro Min_le) auto
  with  $\langle 0 \leq r + \_ \rangle$  have  $0 \leq M \ x \ c2 + Le (u \ c2) + Le (- d)$ 
  by auto
  moreover have  $Le (d - u \ c2) \leq M \ x \ c2 \longleftrightarrow 0 \leq M \ x \ c2 + Le (u \ c2) + Le (- d)$ 
  by (cases  $M \ x \ c2$ ) (auto simp: DBM.add algebra_simps)
  ultimately show  $Le (d - u \ c2) \leq M \ x \ c2$ 

```

```

    by simp
qed
moreover have  $Le\ d \leq M\ x\ 0$ 
  if  $c1 = x\ 0 < x\ x \leq n\ c2 = 0$ 
proof -
  from  $\langle 0 \leq r + \_ \rangle$  have  $Le\ d \leq r$ 
    by (simp add:  $Le\_le\_sum\_iff$ )
  also have  $r \leq M\ x\ 0$ 
    unfolding  $r\_def$  by auto
  finally show  $Le\ d \leq M\ x\ 0$  .
qed
moreover have  $Le\ (u\ c1 - d) \leq M\ c1\ x$ 
  if  $0 < x\ c1 \leq n\ x \leq n\ c1 \neq x\ c2 = x\ 0 < c1\ Le\ (u\ c1 - u\ x) \leq M\ c1\ 0$ 
proof -
  from  $that$  have  $l \leq M\ c1\ x + Le\ (-\ u\ c1)$ 
    unfolding  $l\_def$  by (intro  $Min\_le$ ) auto
  with  $\langle 0 \leq l + Le\ d \rangle$  have  $0 \leq M\ c1\ x + Le\ (-\ u\ c1) + Le\ d$ 
    by auto
  moreover have  $Le\ (u\ c1 - d) \leq M\ c1\ x \longleftrightarrow 0 \leq M\ c1\ x + Le\ (-\ u\ c1) + Le\ d$ 
    by (cases  $M\ c1\ x$ ) (auto simp:  $DBM.add\ algebra\_simps$ )
  ultimately show  $Le\ (u\ c1 - d) \leq M\ c1\ x$ 
    by simp
qed
moreover have  $Le\ (-\ d) \leq M\ 0\ x$ 
  if  $x \leq n\ 0 < x\ c2 = x\ c1 = 0\ Le\ (-\ u\ x) \leq M\ 0\ 0$ 
proof -
  from  $\langle 0 \leq l + Le\ d \rangle$  have  $Le\ (-\ d) \leq l$ 
    by (simp add:  $Le\_le\_sum\_iff$ )
  also have  $l \leq M\ 0\ x$ 
    unfolding  $l\_def$  by auto
  finally show  $Le\ (-\ d) \leq M\ 0\ x$  .
qed
ultimately show ?thesis
  using prems  $\langle x > 0 \rangle **$ 
  by (auto simp:  $dbm\_entry\_val'\_iff\_bounded\ free\_def\ split: if\_split\_asm$ )
qed
qed
with  $\langle d \geq 0 \rangle$  show  $\exists d \geq 0. u(x := d) \in \llbracket M \rrbracket$ 
  by auto
from  $\langle x > 0 \rangle \langle x \leq n \rangle$  have  $dbm\_entry\_val'\ u\ 0\ x\ (free\ n\ M\ x\ 0\ x)$ 
  by (auto intro:  $in\_DBM\_D[OF\ \langle u \in \_ \rangle]$ )
with  $\langle 0 < x \rangle$  have  $Le\ (-\ u\ x) \leq M\ 0\ 0$ 
  by (auto simp:  $free\_def\ dbm\_entry\_val'\_iff\_bounded$ )
with  $\langle M\ 0\ 0 \leq 0 \rangle$  have  $Le\ (-\ u\ x) \leq 0$ 
  by blast
then show  $0 \leq u\ x$ 
  by auto
qed

lemma free_correct:
 $\llbracket free\ n\ M\ x \rrbracket = \{u(x := d) \mid u\ d. u \in \llbracket M \rrbracket \wedge d \geq 0\}$ 
if  $x > 0\ x \leq n\ \forall c \leq n. M\ c\ c \geq 0\ \forall u \in \llbracket M \rrbracket. u\ x \geq 0\ canonical\ M\ n$ 
 $M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$ 
using that

```

```

apply safe
subgoal for  $u'$ 
  apply (frule free_sound, assumption+)
  apply (frule free_sound(2), assumption+)
  apply (erule exE)
  subgoal for  $d$ 
    by (inst_existentials  $u'(x := d)$   $u' x$ ; simp)
  done
subgoal for  $u d$ 
  by (auto intro: free_complete)
done

```

```

lemma pre_reset_correct_aux:
   $\{u. (u(x := (0::'t))) \in \llbracket M \rrbracket\} \cap \{u. u\ x \geq 0\} = \{u(x := d) \mid u\ d. u \in \llbracket M \rrbracket \wedge u\ x = 0 \wedge d \geq 0\}$ 
  apply safe
  subgoal for  $u$ 
    by (inst_existentials  $u(x := (0::'t))$   $u\ x$ ) auto
  apply clarsimp
  subgoal for  $u d$ 
    by (subgoal_tac  $u = u(x := 0)$ ) auto
  by auto

```

```

lemma restrict_zero_correct:
   $\llbracket \text{restrict\_zero } n\ M\ x \rrbracket = \{u. u \in \llbracket M \rrbracket \wedge u\ x = 0\}$  if  $0 < x\ x \leq n$ 
  using that unfolding restrict_zero_def
  by (auto simp: repair_pair_zone_equiv and_entry_correct dbm_entry_val' iff_bounded
    dbm_entry_simps)

```

```

lemma restrict_zero_canonical:
  canonical (restrict_zero  $n\ M\ x$ )  $n \vee \text{check\_diag } n$  (uncurry (restrict_zero  $n\ M\ x$ ))
  if canonical  $M\ n\ x \leq n$ 
proof –
  from  $\langle x \leq n \rangle$  have  $*$ :  $\{0..n\} - \{0, x\} \cup \{x, 0\} = \{0..n\}$ 
  by auto
  define  $M1$  and  $M2$  where  $M1 = \text{and\_entry } x\ 0\ (Le\ 0)\ M\ M2 = \text{and\_entry } 0\ x\ (Le\ 0)\ M1$ 
  from  $\langle \text{canonical } M\ n \rangle$  have canonical_sub  $n\ \{0..n\}\ M$ 
  unfolding canonical_alt_def .
  with  $*$  have canonical_sub  $n\ (\{0..n\} - \{0, x\})\ M2$ 
  unfolding and_entry_def M1_M2_def canonical_sub_def by (auto simp: min.coboundedI1)
  from repair_pair_characteristic[OF this, of x 0]  $\langle x \leq n \rangle$  have
    canonical (repair_pair  $n\ M2\ x\ 0$ )  $n \vee \text{check\_diag } n$  (uncurry (repair_pair  $n\ M2\ x\ 0$ ))
  unfolding canonical_alt_def check_diag_def * neutral by auto
  then show ?thesis
  unfolding restrict_zero_def M1_M2_def Let_def .
qed

```

**end**

## 10.4 Structural Properties

```

lemma free_canonical:
  canonical (free  $n\ M\ x$ )  $n$  if canonical  $M\ n\ M\ x\ x \geq 0$ 
  unfolding free_def using that by (auto simp: add_increasing2 any_le_inf)

```

```

lemma free_diag:
  free n M x i i = M i i
  unfolding free_def by auto

lemma check_diag_free:
  check_diag n (uncurry (free n M x)) if check_diag n (uncurry M)
  using that unfolding check_diag_def by (auto simp: free_diag)

lemma
   $\forall i \leq n. (free\ n\ M\ x)\ i\ i \leq 0$  if  $\forall i \leq n. M\ i\ i \leq 0$ 
  using that by (auto simp: free_diag)

lemma canonical_nonneg_diag_non_empty:
  assumes canonical M n  $\forall i \leq n. 0 \leq M\ i\ i$ 
  shows  $[M]_{v,n} \neq \{\}$ 
  using v_0 by (intro canonical_nonneg_diag_non_empty[OF assms]) force

lemma V_structuralI:
   $\llbracket M \rrbracket \subseteq V$  if  $\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0$ 
  using that
  unfolding V_def
proof safe
  fix u i assume u  $\in \llbracket M \rrbracket$  i  $\in \{1..n\}$ 
  then have Suc 0  $\leq i \leq n$  by simp+
  from in_DBM_D[OF  $\langle u \in \_ \rangle$ , of 0 i]  $\langle \_ \leq i \rangle \langle i \leq n \rangle$  have dbm_entry_val' u 0 i (M 0 i)
    by auto
  with  $\langle \_ \leq i \rangle \langle i \leq n \rangle$  have Le ( $- u\ i$ )  $\leq M\ 0\ i$ 
    by (auto simp: dbm_entry_val'_iff_bounded)
  also from that  $\langle \_ \leq i \rangle \langle i \leq n \rangle$  have  $\dots \leq 0$ 
    by simp
  finally show 0  $\leq u\ i$ 
    by (auto simp: dbm_entry_simps)
qed

lemma canonical_V_non_empty_iff:
  assumes canonical M n M 0 0  $\leq 0$ 
  shows  $\llbracket M \rrbracket \subseteq V \wedge \llbracket M \rrbracket \neq \{\} \longleftrightarrow (\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. M\ i\ i \geq 0)$ 
proof (safe, goal_cases)
  case (1 u i)
  with  $\langle M\ 0\ 0 \leq 0 \rangle$  show ?case
    unfolding V_def by - (rule M_0_k[OF  $\_ \langle canonical\ M\ n \rangle$ ], auto)
next
  case (2 x i)
  then show ?case
    using neg_diag_empty_spec[of i M] by fastforce
next
  case prems: (3 u)
  then show ?case
    by (auto dest: subsetD[OF V_structuralI])
next
  case 4
  with canonical_nonneg_diag_non_empty[OF  $\langle canonical\ M\ n \rangle$ ] show ?case
    by simp
qed

```

**lemma**

**assumes**  $(\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. M\ i\ i \geq 0) \ M\ 0\ 0 \leq 0\ x > 0$   
**shows**  $(\forall i \leq n. i > 0 \longrightarrow \text{free } n\ M\ x\ 0\ i \leq 0) \wedge (\forall i \leq n. \text{free } n\ M\ x\ i\ i \geq 0)$   
**using** *assms* **by** (*auto simp: free\_def*)

**lemma**

**assumes**  $(\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. M\ i\ i \geq 0) \implies$   
 $(\forall i \leq n. i > 0 \longrightarrow f\ M\ 0\ i \leq 0) \wedge (\forall i \leq n. f\ M\ i\ i \geq 0)$   
**assumes** *canonical*  $M\ n$  *canonical*  $(f\ M)\ n$   
**assumes**  $M\ 0\ 0 \leq 0\ f\ M\ 0\ 0 \leq 0$   
**assumes** *check\_diag*: *check\_diag*  $n$   $(\text{uncurry } M) \implies \text{check\_diag } n\ (\text{uncurry } (f\ M))$   
**assumes**  $\llbracket M \rrbracket \subseteq V$   
**shows**  $\llbracket f\ M \rrbracket \subseteq V$   
**proof** (*cases*  $\llbracket M \rrbracket = \{\}$ )  
**case** *True*  
**then have** *check\_diag*  $n$   $(\text{uncurry } M)$   
**using** *canonical\_nonneg\_diag\_non\_empty*[*OF*  $\langle \text{canonical } M\ n \rangle$ ] **by** (*force simp: neutral\_check\_diag\_def*)  
**then have** *check\_diag*  $n$   $(\text{uncurry } (f\ M))$   
**by** (*rule check\_diag*)  
**then have**  $\llbracket f\ M \rrbracket = \{\}$   
**by** (*auto dest: neg\_diag\_empty\_spec simp: check\_diag\_def neutral*)  
**then show** *?thesis*  
**by** *auto*  
**next**  
**case** *False*  
**with**  $\langle \llbracket M \rrbracket \subseteq V \rangle$  *canonical\_V\_non\_empty\_iff*[*OF*  $\langle \text{canonical } M\ n \rangle$   $\langle M\ 0\ 0 \leq 0 \rangle$ ] **have**  
 $(\forall i \leq n. 0 < i \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. 0 \leq M\ i\ i)$   
**by** *auto*  
**then have**  $(\forall i \leq n. i > 0 \longrightarrow f\ M\ 0\ i \leq 0) \wedge (\forall i \leq n. f\ M\ i\ i \geq 0)$   
**by** (*rule assms*(1))  
**with**  $\langle \text{canonical } (f\ M)\ n \rangle$  **have**  $\llbracket f\ M \rrbracket \subseteq V \wedge \llbracket f\ M \rrbracket \neq \{\}$   
**using**  $\langle f\ M\ 0\ 0 \leq 0 \rangle$  **by** (*subst canonical\_V\_non\_empty\_iff*) (*auto simp: free\_diag*)  
**then show** *?thesis* ..  
**qed**

**lemma**

$\llbracket \text{free } n\ M\ x \rrbracket \subseteq V$  **if** *assms*:  $x > 0$  *canonical*  $M\ n$   $M\ 0\ 0 \leq 0\ 0 \leq M\ x\ x$   $\llbracket M \rrbracket \subseteq V$   
**proof** (*cases*  $\llbracket M \rrbracket = \{\}$ )  
**case** *True*  
**then obtain**  $i$  **where**  $M\ i\ i < 0\ i \leq n$   
**using** *canonical\_nonneg\_diag\_non\_empty*[*OF*  $\langle \text{canonical } M\ n \rangle$ ] **by** *atomize\_elim force*  
**then have**  $\text{free } n\ M\ x\ i\ i < 0$   
**by** (*auto simp: free\_diag*)  
**with**  $\langle i \leq n \rangle$  **have**  $\llbracket \text{free } n\ M\ x \rrbracket = \{\}$   
**by** (*intro neg\_diag\_empty\_spec*)  
**then show** *?thesis*  
**by** *auto*  
**next**  
**case** *False*  
**with**  $\langle \llbracket M \rrbracket \subseteq V \rangle$  *canonical\_V\_non\_empty\_iff*[*OF* *that*(2,3)] **have**  
 $(\forall i \leq n. 0 < i \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. 0 \leq M\ i\ i)$   
**by** *auto*  
**with** *that* **have**  $(\forall i \leq n. i > 0 \longrightarrow \text{free } n\ M\ x\ 0\ i \leq 0) \wedge (\forall i \leq n. \text{free } n\ M\ x\ i\ i \geq 0)$

```

    by (auto simp: free_def)
  moreover have canonical (free n M x) n
    apply (rule free_canonical)
    apply fact
    apply fact
    done
  ultimately have  $\llbracket \text{free } n \ M \ x \rrbracket \subseteq V \wedge \llbracket \text{free } n \ M \ x \rrbracket \neq \{\}$ 
    using  $\langle M \ 0 \ 0 \leq 0 \rangle$  by (subst canonical_V_non_empty_iff) (auto simp: free_diag)
  then show ?thesis ..
qed

```

```

lemma
  down n M i i = M i i
  unfolding down_def by auto

```

```

lemma
  assumes  $(\forall i \leq n. i > 0 \longrightarrow M \ 0 \ i \leq 0) \wedge (\forall i \leq n. M \ i \ i \geq 0) \ M \ 0 \ 0 \leq 0 \ x > 0$ 
  shows  $(\forall i \leq n. i > 0 \longrightarrow \text{down } n \ M \ 0 \ i \leq 0) \wedge (\forall i \leq n. \text{down } n \ M \ i \ i \geq 0)$ 
  using assms by (auto simp: down_def neutral)

```

```

lemma check_diag_empty:
   $\llbracket M \rrbracket = \{\}$  if check_diag n (uncurry M)
  using check_diag_empty[of n v uncurry M] that v_is_id by auto

```

```

lemma restrict_zero_mono:
  restrict_zero n M x i j  $\leq$  M i j if  $i \leq n \ j \leq n$ 
  unfolding restrict_zero_def
  by simp (rule  $\langle i \leq n \rangle \ \langle j \leq n \rangle$  repair_pair_mono and_entry_mono order.trans)+

```

```

lemma restrict_zero_diag:
  check_diag n (uncurry (restrict_zero n M x)) if check_diag n (uncurry M)
  using that unfolding check_diag_def neutral[symmetric]
  by (elim exE conjE) (frule restrict_zero_mono[where  $M = M$  and  $x = x$ ], auto)

```

```

lemma pre_reset_correct:
   $\llbracket \text{pre\_reset } n \ M \ x \rrbracket = \{u. (u(x := (0::'t::\text{time}))) \in \llbracket M \rrbracket\} \cap \{u. u \ x \geq 0\}$ 
  if  $x > 0 \ x \leq n$  canonical M n  $\vee$  check_diag n (uncurry M)  $M \ 0 \ x \leq 0 \ M \ 0 \ 0 \leq 0$ 

```

```

proof -
  have check_diag: ?thesis if A: check_diag n (uncurry (restrict_zero n M x))
  proof -
    from A have check_diag n (uncurry (pre_reset n M x))
      unfolding pre_reset_def by (rule check_diag_free)
    then have  $\llbracket \text{pre\_reset } n \ M \ x \rrbracket = \{\}$ 
      by (rule check_diag_empty)
    from A have  $\llbracket \text{restrict\_zero } n \ M \ x \rrbracket = \{\}$ 
      by (rule check_diag_empty)
    then have  $\{u. (u(x := (0::'t::\text{time}))) \in \llbracket M \rrbracket\} \cap \{u. u \ x \geq 0\} = \{\}$ 
      using  $\langle 0 < x \rangle \ \langle x \leq n \rangle$  by (auto simp: restrict_zero_correct)
    with  $\langle \llbracket \text{pre\_reset } n \ M \ x \rrbracket = \{\} \rangle$  show ?thesis
      by simp
  qed
  from that(3) show ?thesis
proof
  assume canonical M n

```

```

from restrict_zero_canonical[OF ‹canonical M n› ‹x ≤ n›] have
  canonical (restrict_zero n M x) n ∨ check_diag n (uncurry (restrict_zero n M x))
  (is ?A ∨ ?B) .
then consider ?A ¬ ?B | ?B
  by blast
then show ?thesis
proof cases
  case 1
  assume ?A ¬ ?B
  moreover from ‹¬ ?B› have ∀ c ≤ n. 0 ≤ restrict_zero n M x c c
    unfolding check_diag_def by (auto simp: DBM.neutral)
  moreover have ∀ u ∈ [[restrict_zero n M x]]. 0 ≤ u x
    by (simp add: restrict_zero_correct that)
  moreover from ‹x ≤ n› ‹M 0 x ≤ 0› have restrict_zero n M x 0 x ≤ 0
    by (blast intro: order.trans restrict_zero_mono)
  moreover from ‹x ≤ n› ‹M 0 0 ≤ 0› have restrict_zero n M x 0 0 ≤ 0
    by (blast intro: order.trans restrict_zero_mono)
  ultimately show ?thesis
    using that
    by (auto simp: pre_reset_correct_aux restrict_zero_correct free_correct pre_reset_def)
next
  assume ?B then show ?thesis
    by (rule check_diag)
qed
next
  assume check_diag n (uncurry M)
  then have check_diag n (uncurry (restrict_zero n M x))
    by (rule restrict_zero_diag)
  then show ?thesis
    by (rule check_diag)
qed
qed

lemma zone_set_pre_Cons:
  zone_set_pre [[M]] (x # r) = zone_set_pre {u. (u(x := (0::'t::time))) ∈ [[M]]} r
  unfolding zone_set_pre_def by auto

lemma pre_reset_list_Cons:
  pre_reset_list n M (x # r) = pre_reset_list n (pre_reset n M x) r
  unfolding pre_reset_list_def by simp

lemma pre_reset_diag:
  check_diag n (uncurry (pre_reset n M x)) if check_diag n (uncurry M)
  using that unfolding pre_reset_def by (intro check_diag_free restrict_zero_diag)

lemma free_canonical':
  canonical (free n (M :: ( _ :: time) DBM) x) n ∨ check_diag n (uncurry (free n M x))
  if canonical M n ∨ check_diag n (uncurry M) x ≤ n
  by (smt check_diag_def check_diag_free dbm_entry_le_iff(5) free_canonical leI
    order_mono_setup.refl order_trans that uncurry_apply
  )

lemma pre_reset_canonical':
  canonical (pre_reset n (M :: ( _ :: time) DBM) x) n ∨ check_diag n (uncurry (pre_reset n M

```

```

x))
  if canonical M n ∨ check_diag n (uncurry M) x ≤ n
  using that(1)
proof standard
  assume canonical M n
  with ⟨x ≤ n⟩ have
    canonical (restrict_zero n M x) n ∨ check_diag n (uncurry (restrict_zero n M x))
    by (intro restrict_zero_canonical)
  with ⟨x ≤ n⟩ show ?thesis
    unfolding pre_reset_def by (intro free_canonical')
next
  assume check_diag n (uncurry M)
  from pre_reset_diag[OF this] show ?thesis ..
qed

```

```

lemma pre_reset_list_correct:
  ⌊pre_reset_list n M r⌋ = zone_set_pre ⌊M⌋ r ∩ {u. ∀ x ∈ set r. u x ≥ 0}
  if ∀ x ∈ set r. x > 0 ∧ x ≤ n
    canonical M n ∨ check_diag n (uncurry M) ∀ x ∈ set r. M 0 x ≤ 0 M 0 0 ≤ 0
  using that
  apply (induction r arbitrary: M)
  apply (simp add: zone_set_pre_def pre_reset_list_def)
  subgoal premises prems for x r M
    apply (subst zone_set_pre_Cons)
    apply (subst pre_reset_list_Cons)
    apply (subst prems)
    prefer 5
    apply (subst pre_reset_correct)
    prefer 6
  subgoal
    unfolding zone_set_pre_def by (cases x ∈ set r) auto
  using prems(2-) apply (solves auto)+
  subgoal
    using prems(2-) by (intro pre_reset_canonical'; auto)
  subgoal
    unfolding pre_reset_def free_def using prems(2-)
    by (auto 4 3 intro: order.trans restrict_zero_mono)
  subgoal
    unfolding pre_reset_def free_def using prems(2-)
    by (auto 4 3 intro: order.trans restrict_zero_mono)
  done
done
end

```

Computes  $\text{dbm\_list } xs - \llbracket M \rrbracket = \text{dbm\_list } xs \cap (- \llbracket M \rrbracket)$  by negating each entry of  $M$  and intersecting it with each member of  $xs$ .

**definition**

```

dbm_minus_canonical n xs M =
  [and_entry_repair n j i (neg_dbm_entry (M i j)) M'
    (i, j) ← [(i, j).
      i ← [0.. $\text{Suc } n$ ], j ← [0.. $\text{Suc } n$ ], (i > 0 ∨ j > 0) ∧ i ≤ n ∧ j ≤ n ∧ M i j ≠ ∞],
    M' ← xs
  ]

```

Same as *dbm\_minus\_canonical* but filters out empty DBMs.

**definition**

```
dbm_minus_canonical_check n xs M =
  filter (λM. ¬ check_diag n (uncurry M)) (dbm_minus_canonical n xs M)
```

Checks whether  $\llbracket M \rrbracket - \text{dbm\_list } xs = \{\}$ .

**definition**

```
dbm_subset_fed n M xs ≡
  let xs = filter (λM. ¬ check_diag n (uncurry M)) xs in
  list_all (λ M. check_diag n (uncurry M)) (fold (λm S. dbm_minus_canonical n S m) xs [M])
```

**definition**

```
dbm_subset_fed_check n M xs ≡
  let
    xs = filter (λM. ¬ check_diag n (uncurry M)) xs;
    is_direct_subset = list_ex (λ M'. dbm_subset' n (uncurry M) (uncurry M')) xs
  in is_direct_subset ∨
    list_all (λM. check_diag n (uncurry M)) (fold (λm S. dbm_minus_canonical_check n S
m) xs [M])
```

**definition** *canonical'* n M ≡ *canonical* M n ∨ *check\_diag* n (uncurry M)

**lemma** *canonical'I*:

```
canonical' n (f M) if
  canonical' n M
  canonical M n ⇒ canonical' n (f M) check_diag n (uncurry M) ⇒ check_diag n (uncurry
(f M))
  using that unfolding canonical'_def by metis
```

**lemma** *check\_diag\_repair\_pair*:

```
assumes check_diag n (uncurry M)
shows check_diag n (uncurry (repair_pair n M i j))
using assms repair_pair_mono[where M = M and a = i and b = j] unfolding check_diag_def
by force
```

**lemma** *check\_diag\_and\_entry*:

```
assumes check_diag n (uncurry M)
shows check_diag n (uncurry (and_entry a b e M))
using assms unfolding check_diag_def
apply (elim exE)
subgoal for i
  using and_entry_mono[where M = M and a = a and b = b and e = e, of i i] by auto
done
```

**lemma** *canonical'\_and\_entry\_repair*:

```
canonical' n (and_entry_repair n i j e M) if canonical' n M i ≤ n j ≤ n
  using that(1)
```

**proof** (rule *canonical'I*)

assume *canonical* M n

from  $\langle i \leq n \rangle \langle j \leq n \rangle$  have  $*$ :  $\{0..n\} - \{i, j\} \cup \{i, j\} = \{0..n\}$

by auto

define M1 where M1 = *and\_entry* i j e M

from  $\langle \text{canonical } M \ n \rangle$  have *canonical\_subs* n  $\{0..n\}$  M

```

  unfolding canonical_alt_def .
with * have canonical_subs n ( $\{0..n\} - \{i, j\}$ ) M1
  unfolding and_entry_def M1_def canonical_subs_def by (auto simp: min.coboundedI1)
from repair_pair_characteristic[OF this, of i j]  $\langle i \leq n \rangle \langle j \leq n \rangle$  have
  canonical' n (repair_pair n M1 i j)
  unfolding canonical'_def
  unfolding canonical_alt_def check_diag_def * neutral by auto
then show ?thesis
  unfolding and_entry_repair_def M1_def Let_def .
next
  assume check_diag n (uncurry M)
  then show check_diag n (uncurry (and_entry_repair n i j e M))
    unfolding and_entry_repair_def by (intro check_diag_repair_pair check_diag_and_entry)
qed

```

**lemma** dbm\_minus\_canonical\_canonical':  
 $\forall M \in \text{set } (\text{dbm\_minus\_canonical } n \text{ } xs \text{ } m). \text{canonical}' n M \text{ if } \forall M \in \text{set } xs. \text{canonical}' n M$   
**using** that **unfolding** dbm\_minus\_canonical\_def  
**by** (auto split: if\_split\_asm intro: canonical'\_and\_entry\_repair)

**lemma** dbm\_minus\_canonical\_check\_canonical':  
 $\forall M \in \text{set } (\text{dbm\_minus\_canonical\_check } n \text{ } xs \text{ } m). \text{canonical}' n M \text{ if } \forall M \in \text{set } xs. \text{canonical}' n M$   
**using** dbm\_minus\_canonical\_canonical'[OF that] **unfolding** dbm\_minus\_canonical\_check\_def  
**by** auto

## 10.5 Correctness of dbm\_subset\_fed

**Misc lemma** list\_all\_iffI:  
**assumes**  $\forall x \in \text{set } xs. \exists y \in \text{set } ys. P x \longleftrightarrow Q y$   
**and**  $\forall y \in \text{set } ys. \exists x \in \text{set } xs. P x \longleftrightarrow Q y$   
**shows**  $\text{list\_all } P \text{ } xs \longleftrightarrow \text{list\_all } Q \text{ } ys$   
**using** assms **unfolding** list\_all\_def **by** blast

**lemma** list\_all\_iff\_list\_all2I:  
**assumes**  $\text{list\_all2 } (\lambda x y. P x \longleftrightarrow Q y) \text{ } xs \text{ } ys$   
**shows**  $\text{list\_all } P \text{ } xs \longleftrightarrow \text{list\_all } Q \text{ } ys$   
**using** assms **by** (intro list\_all\_iffI list\_all2\_set1 list\_all2\_set2)

**lemma** list\_all2\_mapI:  
**assumes**  $\text{list\_all2 } (\lambda x y. P (f x) (g y)) \text{ } xs \text{ } ys$   
**shows**  $\text{list\_all2 } P \text{ } (\text{map } f \text{ } xs) \text{ } (\text{map } g \text{ } ys)$   
**using** assms **by** (simp only: list\_rel\_map)

**context** Default\_Nat\_Clock\_Numbering  
**begin**

**lemma** canonical\_empty\_zone:  
 $[M]_{v,n} = \{\} \longleftrightarrow (\exists i \leq n. M i i < 0) \text{ if } \text{canonical } M n$   
**using** v\_0 that surj\_on **by** (intro canonical\_empty\_zone) auto

**lemma** check\_diag\_iff\_empty:  
 $\text{check\_diag } n \text{ } (\text{uncurry } M) \longleftrightarrow \llbracket M \rrbracket = \{\} \text{ if } \text{canonical}' n M$   
**proof** (safe, goal\_cases)

```

case 2
show ?case
proof -
  from that
  show ?thesis
    unfolding canonical'_def
  proof
    assume canonical M n
    from canonical_empty_zone[OF this] ‹ $\llbracket M \rrbracket = \{\}$ › have
       $\exists i \leq n. M \ i \ i < 0$ 
    by auto
    then show ?thesis unfolding check_diag_def neutral
      by auto
  next
    assume check_diag n (uncurry M)
    then show ?thesis unfolding check_diag_def neutral by auto
  qed
qed
qed (auto dest: check_diag_empty)

lemma and_entry_repair_zone_equiv:
   $\llbracket \text{and\_entry\_repair } n \ a \ b \ e \ M \rrbracket = \llbracket \text{and\_entry } a \ b \ e \ M \rrbracket$  if  $a \leq n \ b \leq n$ 
  unfolding and_entry_repair_def using that by (rule repair_pair_zone_equiv)

lemma dbm_minus_rel:
  assumes list_all2  $(\lambda x \ y. \llbracket x \rrbracket = \llbracket y \rrbracket)$  ms ms'
  shows list_all2  $(\lambda x \ y. \llbracket x \rrbracket = \llbracket y \rrbracket)$  (dbm_minus n ms m) (dbm_minus_canonical n ms' m)
  unfolding dbm_minus_def dbm_minus_canonical_def
  apply (rule concat_transfer[unfolded rel_fun_def, rule_format])
  apply (rule list_all2_mapI)
  apply (rule list.rel_refl_strong)
  apply (auto 4 3
    intro: list_all2_mapI list_all2_mono[OF assms]
    simp: and_entry_repair_zone_equiv and_entry_correct split: if_split_asm
  )
done

lemma dbm_minus_canonical_fold_canonical':
   $\forall M \in \text{set } (\text{fold } (\lambda m \ S. \text{dbm\_minus\_canonical } n \ S \ m) \ xs \ ms). \text{canonical}' \ n \ M$ 
  if  $\forall M \in \text{set } ms. \text{canonical}' \ n \ M$  for ms and xs :: ('t :: time) DBM list
  using that by (induction xs arbitrary: ms) (auto dest: dbm_minus_canonical_canonical')

lemma not_check_diag_nonnegD:
   $M \ i \ i \geq 0$  if  $\neg \text{check\_diag } n \ (\text{uncurry } M) \ i \leq n$ 
  using that unfolding check_diag_def by (auto simp: DBM.less_eq[symmetric] neutral)

theorem dbm_subset_fed_correct:
  fixes xs :: (nat  $\Rightarrow$  nat  $\Rightarrow$  ('t :: time) DBMEntry) list
  and S :: (nat  $\Rightarrow$  nat  $\Rightarrow$  't DBMEntry) list
  assumes canonical' n M
  shows  $\llbracket M \rrbracket \subseteq (\bigcup_{m \in \text{set } xs} \llbracket m \rrbracket) \longleftrightarrow \text{dbm\_subset\_fed } n \ M \ xs$ 
proof -
  have *: list_all2  $(\lambda x \ y. \llbracket x \rrbracket = \llbracket y \rrbracket)$ 
    (fold  $(\lambda m \ S. \text{dbm\_minus } n \ S \ m) \ xs \ ms)$ 

```

```

    (fold (λm S. dbm_minus_canonical n S m) xs ms')
  if list_all2 (λx y. ⌊x⌋ = ⌊y⌋) ms ms' for ms ms' and xs :: 't DBM list
  using that
proof (induction xs arbitrary: ms ms')
  case Nil
  then show ?case
    by simp
next
  case prems: (Cons a xs)
  from this(2) show ?case
    by simp (intro prems(1) dbm_minus_rel)
qed
let ?xs = filter (λM. ¬ check_diag n (uncurry M)) xs
have *: list_all2 (λx y. ⌊x⌋ = ⌊y⌋)
  (fold (λm S. dbm_minus n S m) ?xs [M])
  (fold (λm S. dbm_minus_canonical n S m) ?xs [M])
  by (rule *) simp
have **: (⋃ m ∈ set xs. ⌊m⌋) = (⋃ m ∈ set (filter (λM. ¬ check_diag n (uncurry M)) xs). ⌊m⌋)
  by (auto simp: check_diag_empty)
show ?thesis
  apply (subst **)
  apply (subst dbm_list_superset_op[where S = [M], simplified])
  subgoal
    by (auto dest: not_check_diag_nonnegD simp: neutral[symmetric] DBM.less_eq)
  subgoal
    unfolding dbm_subset_fed_def Let_def
    using dbm_minus_canonical_fold_canonical[of [M]] <canonical' n M>
    by (intro list_all_iff_list_all2I list.rel_mono_strong[OF *])(auto dest: check_diag_iff_empty)
  done
qed

lemma dbm_minus_canonical_check_fed_equiv:
  dbm_list (dbm_minus_canonical_check n S m) = dbm_list (dbm_minus_canonical n S m)
  unfolding dbm_minus_canonical_check_def by (auto simp: check_diag_empty)

lemma dbm_minus_canonical_dbm_minus:
  dbm_list (dbm_minus_canonical n xs m) = dbm_list (dbm_minus n xs m)
  using dbm_minus_rel[of xs xs m] unfolding list_all2_same
  by (force dest: list_all2_set1 list_all2_set2)

lemma dbm_minus_canonical_fed_equiv:
  dbm_list (dbm_minus_canonical n xs m) = dbm_list (dbm_minus_canonical n xs' m)
  if dbm_list xs = dbm_list xs' 0 ≤ m 0 0
  unfolding dbm_minus_canonical_dbm_minus
  using that by (auto simp: neutral dbm_list_subtract[symmetric] DBM.less_eq)

theorem dbm_subset_fed_correct':
  fixes xs :: (nat ⇒ nat ⇒ ('t :: time) DBMEntry) list
  and S :: (nat ⇒ nat ⇒ 't DBMEntry) list
  assumes canonical' n M
  shows ⌊M⌋ ⊆ (⋃ m ∈ set xs. ⌊m⌋) ⟷ (
    let xs = filter (λM. ¬ check_diag n (uncurry M)) xs in
    list_all (λ M. check_diag n (uncurry M)) (fold (λm S. dbm_minus_canonical_check n S m)
    xs [M]))

```

**proof** –

**have** *canonical*:  $\forall M \in \text{set } (\text{fold } (\lambda m S. \text{dbm\_minus\_canonical\_check } n S m) \text{ } xs \text{ } ms). \text{canonical}'$   
 $n M$   
**if**  $\forall M \in \text{set } ms. \text{canonical}' n M$  **for** *ms* **and** *xs* :: 't DBM list  
**using that** **by** (*induction xs arbitrary: ms*) (*auto dest: dbm\_minus\_canonical\_check\_canonical'*)  
**have** \*: *dbm\_list* (*fold* ( $\lambda m S. \text{dbm\_minus\_canonical\_check } n S m$ ) *xs ms*) =  
*dbm\_list* (*fold* ( $\lambda m S. \text{dbm\_minus\_canonical } n S m$ ) *xs ms'*)  
**if** *dbm\_list ms* = *dbm\_list ms'*  $\forall m \in \text{set } xs. m \ 0 \ 0 \geq 0$  **for** *ms ms'* **and** *xs* :: 't DBM list  
**using that**  
**proof** (*induction xs arbitrary: ms ms'*)  
**case** *Nil*  
**then show** ?*case*  
**by** *simp*  
**next**  
**case** (*Cons a xs*)  
**from** *Cons.prem*s **show** ?*case*  
**by** – (*simp, rule Cons.IH,*  
*auto intro!: dbm\_minus\_canonical\_fed\_equiv simp add: dbm\_minus\_canonical\_check\_fed\_equiv*  
*)*  
**qed**  
**define** *xs'* **where** *xs'* = *filter* ( $\lambda M. \neg \text{check\_diag } n (\text{uncurry } M)$ ) *xs*  
**have** \*: *dbm\_list* (*fold* ( $\lambda m S. \text{dbm\_minus\_canonical\_check } n S m$ ) *xs'* [*M*]) =  
*dbm\_list* (*fold* ( $\lambda m S. \text{dbm\_minus\_canonical } n S m$ ) *xs'* [*M*])  
**by** (*auto intro!: \* dest: not\_check\_diag\_nonnegD simp: xs'\_def*)  
**have** \*\*: *list\_all* ( $\lambda M. \text{check\_diag } n (\text{uncurry } M)$ ) *xs*  $\longleftrightarrow \text{dbm\_list } xs = \{\}$   
**if**  $\forall M \in \text{set } xs. \text{canonical}' n M$  **for** *xs* :: 't DBM list  
**using that** **by** (*metis (mono\_tags, lifting) Ball\_set SUP\_bot\_conv(2) check\_diag\_iff\_empty*)  
**show** ?*thesis*  
**unfolding**  
*dbm\_subset\_fed\_correct[OF <canonical' \_ \_>] dbm\_subset\_fed\_def*  
*xs'\_def[symmetric] Let\_def*  
**apply** (*subst \*\**)  
**defer**  
**apply** (*subst \*\**)  
**using** *assms* **by** (*auto intro!: dbm\_minus\_canonical\_fold\_canonical' canonical simp: \**)  
**qed**

**lemma** *subset\_if\_pointwise\_le*:

$\llbracket M \rrbracket \subseteq \llbracket M' \rrbracket$  **if** *pointwise\_cmp* ( $\leq$ ) *n M M'*  
**using that** **by** (*simp add: DBM.less\_eq DBM\_le\_subset pointwise\_cmp\_def subsetI*)

**theorem** *dbm\_subset\_fed\_check\_correct*:

**fixes** *xs* :: (*nat*  $\Rightarrow$  *nat*  $\Rightarrow$  ('t :: time) DBMEntry) *list*  
**and** *S* :: (*nat*  $\Rightarrow$  *nat*  $\Rightarrow$  't DBMEntry) *list*  
**assumes** *canonical' n M*  
**shows**  $\llbracket M \rrbracket \subseteq (\bigcup_{m \in \text{set } xs. \llbracket m \rrbracket}) \longleftrightarrow \text{dbm\_subset\_fed\_check } n M \text{ } xs$

**proof** –

**define** *xs'* **where** *xs'* = *filter* ( $\lambda M. \neg \text{check\_diag } n (\text{uncurry } M)$ ) *xs*  
**define** *is\_direct\_subset* **where**  
*is\_direct\_subset* = *list\_ex* ( $\lambda M'. \text{dbm\_subset}' n (\text{uncurry } M) (\text{uncurry } M')$ ) *xs'*  
**have**  $\llbracket M \rrbracket \subseteq (\bigcup_{m \in \text{set } xs. \llbracket m \rrbracket})$  **if** *is\_direct\_subset*  
**proof** –  
**from that** **have**  $\exists m \in \text{set } xs'. \llbracket M \rrbracket \subseteq \llbracket m \rrbracket$   
**unfolding** *is\_direct\_subset\_def list\_ex\_iff dbm\_subset'\_def*

```

      by (auto intro!: subset_if_pointwise_le)
    then show ?thesis
      unfolding xs'_def by auto
  qed
then show ?thesis
  apply (cases is_direct_subset; simp add:
    dbm_subset_fed_check_def is_direct_subset_def dbm_subset_fed_def xs'_def[symmetric]
  )
  unfolding dbm_subset_fed_correct[
    OF ‹canonical' n M›, of xs, unfolded Let_def dbm_subset_fed_def, folded xs'_def,
    symmetric
  ]
  unfolding dbm_subset_fed_correct'[
    OF ‹canonical' n M›, of xs, folded xs'_def, unfolded Let_def, symmetric
  ]
  ..
qed
end

```

## 10.6 Refined DBM Operations

### definition

$V\_dbm = (\lambda i j. \text{if } i = j \text{ then } 0 \text{ else if } i = 0 \wedge j > 0 \text{ then } 0 \text{ else } \infty)$

### definition and\_entry\_upd ::

$\text{nat} \Rightarrow \text{nat} \Rightarrow \text{int DBMEntry} \Rightarrow \text{int DBM}' \Rightarrow \text{int DBM}'$  **where**  
 $\text{and\_entry\_upd } a \ b \ e \ M = M((a,b) := \min (M \ (a, b)) \ e)$

### definition

$\text{and\_entry\_repair\_upd } n \ a \ b \ e \ M \equiv$   
 $\text{Normalized\_Zone\_Semantics\_Impl\_Semantic\_Refinement.repair\_pair } n \ (\text{and\_entry\_upd } a \ b \ e \ M) \ a \ b$

### definition

$\text{dbm\_minus\_canonical\_upd } n \ xs \ m =$   
 $\text{concat } (\text{map } (\lambda(i, j). \text{map } (\lambda M. \text{and\_entry\_repair\_upd } n \ j \ i \ (\text{neg\_dbm\_entry } (m \ (i, j)))) \ M) \ xs)$   
 $[(i, j). i \leftarrow [0..< \text{Suc } n], j \leftarrow [0..< \text{Suc } n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m \ (i, j) \neq \infty]$

### definition

$\text{dbm\_minus\_canonical\_check\_upd } n \ xs \ M =$   
 $\text{filter } (\lambda M. \neg \text{check\_diag } n \ M) \ (\text{dbm\_minus\_canonical\_upd } n \ xs \ M)$

### definition

$\text{dbm\_subset\_fed\_upd } n \ M \ xs \equiv$   
 $\text{let } xs = \text{filter } (\lambda M. \neg \text{check\_diag } n \ M) \ xs;$   
 $\text{is\_direct\_subset} = \text{list\_ex } (\lambda M'. \text{dbm\_subset}' \ n \ M \ M') \ xs$   
 $\text{in is\_direct\_subset} \vee$   
 $\text{list\_all } (\lambda M. \text{check\_diag } n \ M) \ (\text{fold } (\lambda m \ S. \text{dbm\_minus\_canonical\_check\_upd } n \ S \ m) \ xs \ [M])$

### lemma list\_all\_filter\_neg:

$\text{list\_all } P \ (\text{filter } (\lambda x. \neg P \ x) \ xs) \longleftrightarrow (\text{filter } (\lambda x. \neg P \ x) \ xs) = []$

by (auto simp add: list\_all\_iff filter\_empty\_conv)

**lemma** dbm\_subset\_fed\_upd\_alt\_def:

dbm\_subset\_fed\_upd n M xs  $\equiv$   
 let xs = filter ( $\lambda M. \neg \text{check\_diag } n \ M$ ) xs  
 in if xs = [] then check\_diag n M  
   else if list\_ex ( $\lambda M'. \text{dbm\_subset}' n \ M \ M'$ ) xs then True  
   else fold ( $\lambda m \ S. \text{dbm\_minus\_canonical\_check\_upd } n \ S \ m$ ) xs [M] = []

**unfolding** dbm\_subset\_fed\_upd\_def short\_circuit\_conv **using** list\_last

**by** (intro eq\_reflection; force simp: list\_all\_filter\_neg dbm\_minus\_canonical\_check\_upd\_def Let\_def)

**definition**

$V\_dbm' \ n = (\lambda(i, j). (\text{if } i = j \vee i = 0 \wedge j > 0 \vee i > n \vee j > n \text{ then } 0 \text{ else } \infty))$

**definition**

down\_upd :: nat  $\Rightarrow$   $\_ \text{DBM}' \Rightarrow \_ \text{DBM}'$

**where**

down\_upd n M  $\equiv \lambda(i, j).$

if  $i = 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n$  then  $\text{Min } (\{Le \ 0\} \cup \{M \ (k, j) \mid k. 1 \leq k \wedge k \leq n\})$  else  $M \ (i, j)$

**definition**

restrict\_zero\_upd n M x  $\equiv$

let

M1 = and\_entry\_upd x 0 (Le 0) M;

M2 = and\_entry\_upd 0 x (Le 0) M1

in Normalized\_Zone\_Semantics\_Impl\_Semantic\_Refinement.repair\_pair n M2 x 0

**definition**

free\_upd :: nat  $\Rightarrow \_ \text{DBM}' \Rightarrow \text{nat} \Rightarrow \_ \text{DBM}'$

**where**

free\_upd n M x  $\equiv$

$\lambda \ (i, j). \text{if } i = x \wedge j \neq x \wedge i \leq n \wedge j \leq n$

then  $\infty$  else if  $i \neq x \wedge j = x \wedge i \leq n \wedge j \leq n$  then  $M \ (i, 0)$  else  $M \ (i, j)$

**definition**

pre\_reset\_upd n M x  $\equiv \text{free\_upd } n \ (\text{restrict\_zero\_upd } n \ M \ x) \ x$

**definition**

pre\_reset\_list\_upd n M r  $\equiv \text{fold } (\lambda x \ M. \text{pre\_reset\_upd } n \ M \ x) \ r \ M$

## 10.7 Transferring Properties

**context**

includes lifting\_syntax

**begin**

**lemma** neg\_dbm\_entry\_transfer[transfer\_rule]:

(rel\_DBMEntry ri  $\implies$  rel\_DBMEntry ri) neg\_dbm\_entry neg\_dbm\_entry

**by** (auto elim!: DBMEntry.rel\_cases intro!: rel\_funI)

**lemma** fold\_min\_transfer:

((list\_all2 (rel\_DBMEntry ri))  $\implies$  rel\_DBMEntry ri  $\implies$  rel\_DBMEntry ri) (fold min)

```

(fold min)
  by transfer_prover

context
  fixes n :: nat
begin

definition
  RI2 M D ≡ RI n (uncurry M) D

lemma and_entry_transfer[transfer_rule]:
  ((=) ==> (=) ==> rel_DBMEntry ri ==> RI2 ==> RI2) and_entry and_entry_upd
  unfolding and_entry_def and_entry_upd_def
  unfolding rel_fun_def eq_onp_def RI2_def
  by (auto intro: min_ri_transfer[unfolded rel_fun_def, rule_format])

lemma FWI_transfer[transfer_rule]:
  (RI2 ==> eq_onp (λx. x = n) ==> eq_onp (λx. x < Suc n) ==> RI2) FWI FWI' (is
  ?A)
and FW_transfer[transfer_rule]:
  (RI2 ==> eq_onp (λx. x = n) ==> RI2) FW FW' (is ?B)
proof -
  define RI' where
    RI' = (eq_onp (λ x. x < Suc n) ==> eq_onp (λ x. x < Suc n) ==> rel_DBMEntry ri)
  have RI_iff: RI' M M' ⟷ RI n (uncurry M) (uncurry M') for M M'
    unfolding RI'_def rel_fun_def by auto
  { fix M D k assume RI n (uncurry M) D k < Suc n
    then have RI' M (curry D)
      by (simp add: RI_iff)
    with ⟨k < Suc n⟩ have RI' (FWI M n k) (FWI (curry D) n k)
      unfolding FWI_def
      by (intro fwi_RI_transfer[of n, folded RI'_def, unfolded rel_fun_def, rule_format])
      (auto simp: eq_onp_def)
    then have RI n (uncurry (FWI M n k)) (uncurry (FWI (curry D) n k))
      by (simp add: RI_iff)
  } note * = this
show ?A
  unfolding FWI'_def
  unfolding rel_fun_def
  unfolding RI2_def
  apply clarsimp
  apply (subst (asm) (3) eq_onp_def)
  apply (subst (asm) (3) eq_onp_def)
  apply clarsimp
  by (intro *)
{ fix M D assume RI n (uncurry M) D
  then have RI' M (curry D)
    by (simp add: RI_iff)
  then have RI' (FW M n) (FW (curry D) n)
    by (intro FW_RI_transfer[of n, folded RI'_def, unfolded rel_fun_def, rule_format])
    (auto simp: eq_onp_def)
  then have RI n (uncurry (FW M n)) (FW' D n)
    by (simp add: RI_iff FW'_def)
  } note * = this

```

```

show ?B
  unfolding rel_fun_def
  unfolding RI2_def
  apply clarsimp
  apply (subst (asm) (3) eq_onp_def)
  apply clarsimp
  by (intro *)
qed

lemma repair_pair_transfer[transfer_rule]:
  (eq_onp ( $\lambda x. x = n$ ) ==> RI2 ==> eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> RI2)
  repair_pair Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair
  unfolding repair_pair_def Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair_def
  by transfer_prover

lemma and_entry_transfer_weak:
  (eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> rel_DBMEntry ri ==> RI2 ==> RI2)
  and_entry and_entry_upd
  using and_entry_transfer unfolding rel_fun_def eq_onp_def by auto

lemma and_entry_repair_transfer[transfer_rule]:
  (eq_onp ( $\lambda x. x = n$ ) ==> eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> eq_onp ( $\lambda x. x < \text{Suc } n$ ) ==> rel_DBMEntry ri ==> RI2 ==> RI2) and_entry_repair and_entry_repair_upd

supply [transfer_rule] = and_entry_transfer_weak
unfolding and_entry_repair_def and_entry_repair_upd_def by transfer_prover

lemma dbm_minus_canonical_transfer[transfer_rule]:
  (eq_onp ( $\lambda x. x = n$ ) ==> list_all2 RI2 ==> RI2 ==> list_all2 RI2)
  dbm_minus_canonical dbm_minus_canonical_upd

unfolding dbm_minus_canonical_def dbm_minus_canonical_upd_def
apply (intro rel_funI)
apply (rule concat_transfer[unfolded rel_fun_def, rule_format])
apply (rule list.map_transfer[unfolded rel_fun_def, rule_format,
  where Rb = rel_prod (eq_onp ( $\lambda x. x < \text{Suc } n$ )) (eq_onp ( $\lambda x. x < \text{Suc } n$ )))]
apply clarsimp
subgoal
  apply (rule list_all2_mapI)
  apply (erule list_all2_mono)
  apply (rule and_entry_repair_transfer[unfolded rel_fun_def, rule_format])
  apply assumption+
subgoal
  unfolding RI2_def rel_fun_def
  by (auto intro: neg_dbm_entry_transfer[unfolded rel_fun_def, rule_format])
apply assumption
done
subgoal premises prems for n1 n2 xs ys M D
proof -
  have [simp]:  $D(i, j) \neq \infty \iff M i j \neq \infty$  if  $i < \text{Suc } n$   $j < \text{Suc } n$  for  $i j$ 
  using prems that by (auto 4 3 simp: eq_onp_def rel_fun_def RI2_def elim!: DBMEn-
```

```

try.rel_cases)
  from prems have [simp]: n1 = n n2 = n
  by (auto simp: eq_onp_def)
  let ?a = (concat
    (map (λi. concat
      (map (λj. if (0 < i ∨ 0 < j) ∧
        i ≤ n ∧ j ≤ n ∧ M i j ≠ ∞
        then [(i, j)] else []))
      [0.. $\text{Suc } n$ ]))
    [0.. $\text{Suc } n$ ]))
  let ?b = (concat
    (map (λi. concat
      (map (λj. if (0 < i ∨ 0 < j) ∧
        i ≤ n ∧ j ≤ n ∧ D (i, j) ≠ ∞
        then [(i, j)] else []))
      [0.. $\text{Suc } n$ ]))
    [0.. $\text{Suc } n$ ]))
  have ?b = ?a
  by (auto intro!: arg_cong[where f = concat] simp del: upt_Suc)
  then show ?thesis
  by (simp del: upt_Suc add: list_all2_same eq_onp_def)
qed
done

lemma check_diag_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ==> RI2 ==> (=)) (λn M. check_diag n (uncurry M)) check_diag
  unfolding RI2_def rel_fun_def check_diag_def
  by (auto 0 5 dest: neutral_RI simp: eq_onp_def less_Suc_eq_le neutral[symmetric])

lemma dbm_minus_canonical_check_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ==> list_all2 RI2 ==> RI2 ==> list_all2 RI2)
  dbm_minus_canonical_check dbm_minus_canonical_check_upd

  unfolding dbm_minus_canonical_check_def dbm_minus_canonical_check_upd_def by transfer_prover

lemma le_rel_DBMEntry_iff:
  a ≤ b ↔ x ≤ y if rel_DBMEntry ri a x rel_DBMEntry ri b y
  using that by (auto elim!: DBMEntry.rel_cases simp: dbm_entry_simps ri_def)

lemma dbm_subset'_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ==> RI2 ==> RI2 ==> (=))
  (λ n M M'. dbm_subset' n (uncurry M) (uncurry M')) dbm_subset'
  unfolding RI2_def rel_fun_def dbm_subset'_def
  using le_rel_DBMEntry_iff by (clarsimp simp: eq_onp_def pointwise_cmp_def less_Suc_eq_le)
meson

lemma dbm_subset_fed_transfer:
  (eq_onp (λx. x = n) ==> RI2 ==> list_all2 RI2 ==> (=)) dbm_subset_fed_check
  dbm_subset_fed_upd
  unfolding dbm_subset_fed_check_def dbm_subset_fed_upd_def by transfer_prover

lemma V_dbm_transfer[transfer_rule]:
  RI2 V_dbm (V_dbm' n)

```

**unfolding**  $V\_dbm\_def$   $V\_dbm'\_def$   $RI2\_def$  **by** (*auto simp: rel\_fun\_def neutral eq\_onp\_def zero\_RI*)

**lemma**  $down\_transfer[transfer\_rule]$ :  
 ( $eq\_onp (\lambda x. x = n) ==> RI2 ==> RI2$ )  $down$   $down\_upd$   
**unfolding**  $down\_def$   $down\_upd\_def$   
**apply** (*clarsimp simp: rel\_fun\_def RI2\_def eq\_onp\_def, goal\_cases*)  
**subgoal premises**  $prems$  **for**  $M D x$   
**proof** –  
**have**  $A$ : ( $insert (Le\ 0) \{M\ k\ x\ |\ k. Suc\ 0 \leq k \wedge k \leq n\}$ )  
    $= (set (Le\ 0 \# [M\ k\ x. k \leftarrow [1..<Suc\ n]]))$   
   **by** *auto*  
**have**  $B$ : ( $insert (Le\ 0) \{D\ (k, x) \mid k. Suc\ 0 \leq k \wedge k \leq n\}$ )  
    $= (set (Le\ 0 \# [D\ (k, x). k \leftarrow [1..<Suc\ n]]))$   
   **by** *auto*  
**show** *?thesis*  
   **unfolding**  $A\ B\ Min.set\_eq\_fold$   
   **apply** (*rule fold\_min\_transfer[unfolded rel\_fun\_def, rule\_format]*)  
   **unfolding**  $list.rel\_map\ list\_all2\_same$  **using**  $prems$  **by** (*auto simp: zero\_RI*)  
**qed**  
**done**

**lemma**  $free\_upd[transfer\_rule]$ :  
 ( $eq\_onp (\lambda x. x = n) ==> RI2 ==> (=) ==> RI2$ )  $free$   $free\_upd$   
**unfolding**  $free\_def$   $free\_upd\_def$  **by** (*auto simp: RI2\_def rel\_fun\_def eq\_onp\_def*)

**lemma**  $pre\_reset\_transfer[transfer\_rule]$ :  
 ( $eq\_onp (\lambda x. x = n) ==> RI2 ==> eq\_onp (\lambda x. x < Suc\ n) ==> RI2$ )  $pre\_reset$   
 $pre\_reset\_upd$   
**proof** –  
**have**  $[transfer\_rule]$ :  $eq\_onp (\lambda x. x = n) n\ n$   
   **by** (*simp add: eq\_onp\_def*)  
**note**  $[transfer\_rule] = and\_entry\_transfer\_weak$   
**have**  $[transfer\_rule]$ :  
   ( $eq\_onp (\lambda x. x = n) ==> RI2 ==> eq\_onp (\lambda x. x < Suc\ n) ==> RI2$ )  $free$   $free\_upd$   
   **using**  $free\_upd$  **unfolding**  $rel\_fun\_def\ eq\_onp\_def$  **by** *blast*  
**show** *?thesis*  
   **unfolding**  $pre\_reset\_def\ pre\_reset\_upd\_def$   
   **unfolding**  $restrict\_zero\_def\ restrict\_zero\_upd\_def$   
   **by**  $transfer\_prover$   
**qed**

**lemma**  $pre\_reset\_list\_transfer[transfer\_rule]$ :  
 ( $eq\_onp (\lambda x. x = n) ==> RI2 ==> list\_all2 (eq\_onp (\lambda x. x < Suc\ n)) ==> RI2$ )  
 $pre\_reset\_list\ pre\_reset\_list\_upd$   
**unfolding**  $pre\_reset\_list\_def\ pre\_reset\_list\_upd\_def$  **by**  $transfer\_prover$

**end**  
**end**

**definition**  $unbounded\_dbm$  **where**  
 $unbounded\_dbm \equiv \lambda\ i\ j. \text{if } i = j \text{ then } 0 \text{ else } \infty$

**lemma** *canonical\_unbounded\_dbm*:  
*canonical\_unbounded\_dbm n*  
**by** (*auto simp: unbounded\_dbm\_def any\_le\_inf*)

**lemma** *diag\_unbounded\_dbm*:  
*unbounded\_dbm i i = 0*  
**unfolding** *unbounded\_dbm\_def* **by** *simp*

**lemma** *down\_diag*:  
*down n M i i = M i i*  
**unfolding** *down\_def* **by** *auto*

**definition**  
*abstr\_FW n cc M v*  $\equiv$  *FW (abstr cc M v) n*

**definition**  
*abstr\_FW\_upd n cc M*  $\equiv$  *FW' (abstr\_upd cc M) n*

**lemma** *abstr\_mono*:  
*abstr cc M v i j*  $\leq$  *M i j*  
**by** (*subst abstr.simps, induction cc arbitrary: M*) (*auto intro: order.trans abstr\_mono*)

**lemma** *abstr\_FW\_mono*:  
*abstr\_FW n cc M v i j*  $\leq$  *M i j* **if** *i*  $\leq$  *n* *j*  $\leq$  *n*  
**unfolding** *abstr\_FW\_def* **by** (*blast intro: that abstr\_mono fw\_mono order.trans*)

**lemma** *abstr\_FW\_diag\_preservation*:  
 $\forall k \leq n. \text{abstr\_FW } n \text{ cc } M \text{ v } k \text{ k} \leq 0$  **if**  $\forall k \leq n. M \text{ k } k \leq 0$   
**using** *that* **by** (*blast intro: abstr\_FW\_mono order.trans*)

**lemma** *FW\_canonical*:  
*canonical' n (FW M n)*  
**unfolding** *canonical'\_def* **using** *FW\_canonical[of n M]* **by** (*simp add: check\_diag\_def neutral*)

**lemma** *abstr\_FW\_canonical*:  
*canonical' n (abstr\_FW n cc M v)*  
**unfolding** *abstr\_FW\_def* **by** (*rule FW\_canonical*)

**lemma** *down\_check\_diag*:  
*check\_diag n (uncurry (down n M))* **if** *check\_diag n (uncurry M)*  
**using** *that* **unfolding** *check\_diag\_def down\_def* **by** *force*

**context** *Default\_Nat\_Clock\_Numbering*  
**begin**

**lemma** *clock\_numbering*:  
 $\forall c. v \text{ c} > 0 \wedge (\forall x. \forall y. v \text{ x} \leq n \wedge v \text{ y} \leq n \wedge v \text{ x} = v \text{ y} \longrightarrow x = y)$   
**by** (*metis neq0\_conv v\_0 v\_id*)

**lemma** *V\_dbm\_correct*:  
 $\llbracket V\_dbm \rrbracket = V$   
**unfolding** *V\_def DBM\_zone\_repr\_def DBM\_val\_bounded\_alt\_def2*  
**by** (*auto simp: dbm\_entry\_val'\_iff\_bounded V\_dbm\_def dbm\_entry\_simps*)

**lemma** *abstr\_correct*:

$\llbracket \text{abstr } cc \ M \ v \rrbracket = \llbracket M \rrbracket \cap \{u. u \vdash cc\}$  **if**  $\forall c \in \text{collect\_clks } cc. 0 < c \wedge c \leq n$

**apply** (*rule dbm\_abstr\_zone\_eq2*)

**subgoal**

**by** (*rule clock\_numbering*)

**subgoal**

**using** *that* **by** (*auto simp: v\_is\_id*)

**done**

**lemma** *unbounded\_dbm\_correct*:

$\llbracket \text{unbounded\_dbm} \rrbracket = \text{UNIV}$

**unfolding** *DBM\_zone\_repr\_def DBM\_val\_bounded\_alt\_def2 unbounded\_dbm\_def neutral*

**by** (*simp add: dbm\_entry\_val'\_iff\_bounded any\_le\_inf*)

**lemma** *abstr\_correct'*:

$\llbracket \text{abstr } cc \ \text{unbounded\_dbm } v \rrbracket = \{u. u \vdash cc\}$  **if**  $\forall c \in \text{collect\_clks } cc. 0 < c \wedge c \leq n$

**by** (*simp add: unbounded\_dbm\_correct abstr\_correct[OF that] del: abstr.simps*)

**lemma** *abstr\_FW\_correct*:

$\llbracket \text{abstr\_FW } n \ cc \ M \ v \rrbracket = \llbracket M \rrbracket \cap \{u. u \vdash cc\}$  **if**  $\forall c \in \text{collect\_clks } cc. 0 < c \wedge c \leq n$

**unfolding** *abstr\_FW\_def* **by** (*subst FW\_zone\_equiv[symmetric]; intro surj\_on abstr\_correct that*)

**lemma** *abstr\_FW\_correct'*:

$\llbracket \text{abstr\_FW } n \ cc \ \text{unbounded\_dbm } v \rrbracket = \{u. u \vdash cc\}$  **if**  $\forall c \in \text{collect\_clks } cc. 0 < c \wedge c \leq n$

**by** (*simp add: unbounded\_dbm\_correct abstr\_FW\_correct[OF that]*)

**lemma** *down\_V*:

$\llbracket \text{down } n \ M \rrbracket \subseteq V$

**by** (*rule V\_structuralI*) (*auto simp: down\_def neutral intro: Min\_le*)

**lemma** *down\_correct'*:

$\llbracket \text{down } n \ M \rrbracket = \llbracket M \rrbracket^\downarrow \cap V$  **if** *canonical* *M n*

**apply** *safe*

**subgoal for** *u*

**by** (*erule down\_sound, rule that*)

**subgoal for** *u*

**using** *down\_V* **by** *blast*

**subgoal for** *u*

**unfolding** *V\_def* **by** (*erule down\_complete*) *simp*

**done**

**lemma** *down\_correct*:

$\llbracket \text{down } n \ M \rrbracket = \llbracket M \rrbracket^\downarrow \cap V$  **if** *canonical'* *n M*

**using** *that* **unfolding** *canonical'\_def*

**apply** *standard*

**subgoal**

**by** (*rule down\_correct'*)

**by** (*frule down\_check\_diag*) (*simp add: check\_diag\_empty zone\_time\_pre\_def*)

**lemma** *pre\_reset\_diag\_preservation*:

*pre\_reset* *n M x i i*  $\leq M \ i \ i$  **if** *i*  $\leq n$

```

unfolding pre_reset_def by (auto simp add: free_diag restrict_zero_mono that)

lemma pre_reset_list_diag:
  pre_reset_list n M r i i ≤ M i i if i ≤ n
  apply (induction r arbitrary: M)
  apply (simp add: pre_reset_list_def; fail)
  apply (simp add: pre_reset_list_Cons, blast intro: pre_reset_diag_preservation that order.trans)
  done

context
  includes lifting_syntax
begin

lemma abstra_upd_abstra:
  abstra_upd ac M (i, j) = abstra ac (curry M) v i j if 0 < constraint_clk ac i ≤ n j ≤ n
  using that by (cases ac) (auto simp: le_n_iff v_is_id v_0)

lemma abstra_transfer[transfer_rule]:
  (rel_acconstraint (eq_onp (λ x. 0 < x ∧ x < Suc n)) ri) ==> RI2 n ==> RI2 n
  (λ cc M. abstra cc M v) abstra_upd
  apply (intro rel_funI)
  apply (subst RI2_def)
  apply (intro rel_funI)
  apply (elim rel_prod.cases)
  apply (simp only:)
  apply (subst abstra_upd_abstra)
  by (auto 4 3 simp: eq_onp_def RI2_def rel_fun_def
    intro!: min_ri_transfer[unfolded rel_fun_def, rule_format]
    elim!: acconstraint.rel_cases
  )

lemma abstr_transfer[transfer_rule]:
  (list_all2 (rel_acconstraint (eq_onp (λ x. 0 < x ∧ x < Suc n)) ri) ==> RI2 n ==> RI2
  n)
  (λ cc M. abstr cc M v) abstr_upd
  unfolding abstr.simps abstr_upd_def by transfer_prover

lemma abstr_FW_transfer[transfer_rule]:
  (list_all2 (rel_acconstraint (eq_onp (λ x. 0 < x ∧ x < Suc n)) ri) ==> RI2 n ==> RI2
  n)
  (λ cc M. abstr_FW n cc M v) (abstr_FW_upd n)
proof -
  have [transfer_rule]: eq_onp (λ x. x = n) n n
  by (simp add: eq_onp_def)
  show ?thesis
  unfolding abstr_FW_def abstr_FW_upd_def by transfer_prover
qed

end

end

lemma RI2_trivial_transfer[transfer_rule]: (RI2 n) (curry (conv_M M)) M
  unfolding RI2_def rel_fun_def by (auto simp: eq_onp_def)

```

```

end
theory Deadlock_Impl
  imports Deadlock Munta_Base.Abstract_Term
begin

```

```

Misc lemma constraint_clk_conv_ac:
  constraint_clk (conv_ac ac) = constraint_clk ac
  by (cases ac; auto)

```

```

lemma constraint_clk_conv_cc:
  collect_clks (conv_cc cc) = collect_clks cc
  by (auto simp: collect_clks_def constraint_clk_conv_ac image_def)

```

```

lemma atLeastLessThan_alt_def:
  {a..<b} = {k. a ≤ k ∧ k < b}
  by auto

```

```

lemma atLeastLessThan_Suc_alt_def:
  {a..<Suc b} = {k. a ≤ k ∧ k ≤ b}
  by auto

```

```

lemma (in Graph_Defs) deadlock_if_deadlocked:
  deadlock y if deadlocked y
  using that unfolding deadlock_def by auto

```

## 10.8 Functional Refinement

```

Elementary list operations lemma map_conv_rev_fold:
  map f xs = rev (fold (λ a xs. f a # xs) xs [])
proof –
  have fold (λ a xs. f a # xs) xs ys = rev (map f xs) @ ys for ys
  by (induction xs arbitrary: ys) simp_all
  then show ?thesis
  by simp
qed

```

```

lemma concat_map_conv_rev_fold:
  concat (map f xs) = rev (fold (λ xs ys. rev (f xs) @ ys) xs [])
proof –
  have rev (fold (λ xs ys. rev (f xs) @ ys) xs ys) = rev ys @ List.maps f xs for ys
  by (induction xs arbitrary: ys) simp_all
  then show ?thesis
  by simp
qed

```

```

lemma concat_conv_fold_rev:
  concat xss = fold (@) (rev xss) []
  using fold_append_concat_rev[of rev xss] by simp

```

```

lemma filter_conv_rev_fold:
  filter P xs = rev (fold (λx xs. if P x then x # xs else xs) xs [])
proof –
  have rev ys @ filter P xs = rev (fold (λx xs. if P x then x # xs else xs) xs ys) for ys

```

```

    by (induction xs arbitrary: ys) (auto, metis revg.simps(2) revg_fun)
  from this[symmetric] show ?thesis
    by simp
qed

```

#### DBM operations definition

```

free_upd1 n M c =
  (let
    M1 = fold (λi M. if i ≠ c then (M((i, c) := op_mtx_get M (i, 0))) else M) [0..Suc n] M;
    M2 = fold (λi M. if i ≠ c then M((c, i) := ∞) else M) [0..Suc n] M1
  in
    M2
  )

```

#### definition

```

pre_reset_upd1 n M x ≡ free_upd1 n (restrict_zero_upd n M x) x

```

#### definition

```

pre_reset_list_upd1 n M r ≡ fold (λ x M. pre_reset_upd1 n M x) r M

```

#### definition

```

upd_pairs xs = fold (λ(p,q) f. f(p:=q)) xs

```

#### lemma upd\_pairs\_Nil:

```

  upd_pairs [] f = f
  unfolding upd_pairs_def by simp

```

#### lemma upd\_pairs\_Cons:

```

  upd_pairs ((p, q) # xs) f = upd_pairs xs (f(p:=q))
  unfolding upd_pairs_def by simp

```

#### lemma upd\_pairs\_no\_upd:

```

  assumes p ∉ fst ` set xs
  shows upd_pairs xs f p = f p
  using assms by (induction xs arbitrary: f) (auto simp: upd_pairs_Nil upd_pairs_Cons)

```

#### lemma upd\_pairs\_upd:

```

  assumes (p, y) ∈ set xs distinct (map fst xs)
  shows upd_pairs xs f p = y
  using assms proof (induction xs arbitrary: f)
    case Nil
    then show ?case
      by (simp add: upd_pairs_Nil)
  next
    case (Cons x xs)
    then show ?case
      by (cases x) (auto simp add: upd_pairs_Cons upd_pairs_no_upd)
  qed

```

#### lemma upd\_pairs\_append:

```

  upd_pairs (xs @ ys) = upd_pairs ys o upd_pairs xs
  unfolding upd_pairs_def fold_append ..

```

**lemma** *upd\_pairs\_commute\_single*:  
 $\text{upd\_pairs } xs \ (f(a := b)) = (\text{upd\_pairs } xs \ f)(a := b)$  **if**  $a \notin \text{fst } \text{'set } xs$   
**using that** **by** (*induction xs arbitrary: f*) (*auto simp: upd\_pairs\_Nil upd\_pairs\_Cons fun\_upd\_twist*)

**lemma** *upd\_pairs\_append'*:  
 $\text{upd\_pairs } (xs @ ys) = \text{upd\_pairs } xs \circ \text{upd\_pairs } ys$  **if**  $\text{fst } \text{'set } xs \cap \text{fst } \text{'set } ys = \{\}$   
**using that**  
**proof** (*induction xs*)  
**case** *Nil*  
**then show** ?case  
**by** (*simp add: upd\_pairs\_Nil*)  
**next**  
**case** (*Cons a xs*)  
**then show** ?case  
**by** (*cases a*) (*auto simp add: upd\_pairs\_Nil upd\_pairs\_Cons upd\_pairs\_commute\_single*)  
**qed**

**definition**  
 $\text{upd\_pairs}' \ xs = \text{fold } (\lambda(p,q) \ f. \ f(p:=f \ q)) \ xs$

**lemma** *upd\_pairs'\_Nil*:  
 $\text{upd\_pairs}' [] \ f = f$   
**unfolding** *upd\_pairs'\_def* **by** *simp*

**lemma** *upd\_pairs'\_Cons*:  
 $\text{upd\_pairs}' ((p, q) \# xs) \ f = \text{upd\_pairs}' xs \ (f(p:=f \ q))$   
**unfolding** *upd\_pairs'\_def* **by** *simp*

**lemma** *upd\_pairs'\_upd\_pairs*:  
**assumes**  $\text{fst } \text{'set } xs \cap \text{snd } \text{'set } xs = \{\}$   
**shows**  $\text{upd\_pairs}' xs \ f = \text{upd\_pairs } (\text{map } (\lambda(p, q). \ (p, f \ q)) \ xs) \ f$   
**using** *assms*  
**proof** (*induction xs arbitrary: f*)  
**case** *Nil*  
**then show** ?case  
**by** (*simp add: upd\_pairs\_Nil upd\_pairs'\_Nil*)  
**next**  
**case** (*Cons x xs*)  
**obtain** *a b* **where** [*simp*]:  $x = (a, b)$   
**by** (*cases x*)  
**from** *Cons.prem*s **have** \*:  
 $\text{map } (\lambda(p, q). \ (p, \text{if } q = a \text{ then } f \ b \text{ else } f \ q)) \ xs = \text{map } (\lambda(p, q). \ (p, f \ q)) \ xs$   
**by** *auto*  
**from** *Cons* **show** ?case  
**by** (*auto simp add: upd\_pairs\_Cons upd\_pairs'\_Cons \**)  
**qed**

**lemma** *upd\_pairs\_map*:  
 $\text{upd\_pairs } (\text{map } f \ xs) = \text{fold } (\lambda pq \ g. \ \text{let } (p,q) = f \ pq \ \text{in } g(p:=q)) \ xs$   
**unfolding** *upd\_pairs\_def fold\_map*  
**by** (*intro arg\_cong2[where  $f = \text{fold}$ ] ext*) (*auto simp: comp\_def split: prod.split*)

**lemma** *down\_upd\_alt\_def*:

```

down_upd n M =
  upd_pairs (((0, j), fold min [M (k, j). k ← [1..Suc n]] (Le 0)). j ← [1..Suc n])) M
proof -
  define xs where
    xs = (((0::nat, j), Min ({Le 0} ∪ {M (k, j) | k. 1 ≤ k ∧ k ≤ n})). j ← [1..Suc n])
  have distinct (map fst xs)
    unfolding xs_def by (auto simp add: map_concat comp_def distinct_map)
  have *: fold min [M (k, j). k ← [1..Suc n]] (Le 0) = Min ({Le 0} ∪ {M (k, j) | k. 1 ≤ k ∧ k
≤ n})
    for j
    by (subst Min.set_eq_fold[symmetric])
      (auto simp: atLeastLessThan_Suc_alt_def simp del: upt_Suc intro: arg_cong[where f =
Min])
  show ?thesis
    unfolding * xs_def[symmetric]
  by (intro ext, auto simp add: down_upd_def)
    (solves ⟨subst upd_pairs_upd[OF _ ⟨distinct _⟩] upd_pairs_no_upd,
      auto simp: image_def xs_def⟩)+
qed

```

```

lemma down_upd_alt_def1:
  down_upd n M =
    fold (λj M. let (p,q) = ((0, j), fold min [M (k, j). k ← [1..Suc n]] (Le 0)) in M(p:=q))
      [1..Suc n] M
proof -
  have *:
    fold (λj M. let (p,q) = ((0,j), fold min [M (k, j). k ← [1..Suc n]] (Le 0)) in M(p:=q)) xs
  M
  = fold (λj M'. let (p,q) = ((0,j), fold min [M (k, j). k ← [1..Suc n]] (Le 0)) in M'(p:=q)) xs
  M
    for xs
  proof (induction xs arbitrary: M)
    case Nil
      then show ?case
        by simp
    next
      case (Cons x xs)
        then show ?case
          by (auto 4 7 intro: arg_cong2[where f = min] fold_cong)
  qed
  then show ?thesis
    unfolding down_upd_alt_def upd_pairs_map ..
qed

```

```

lemma
  free_upd n M c =
    upd_pairs (((c,i), ∞). i ← [0..Suc n], i ≠ c] @ (((i,c), M(i,0)). i ← [0..Suc n], i ≠ c]) M
  if c ≤ n
proof -
  let ?xs1 = λn. (((c,i), ∞). i ← [0..Suc n], i ≠ c]
  let ?xs2 = λn. (((i,c), M(i,0)). i ← [0..Suc n], i ≠ c]
  define xs where xs = ?xs1 n @ ?xs2 n
  have distinct (map fst xs)
    unfolding xs_def

```

```

  apply (clarsimp simp del: upt_Suc simp add: map_concat comp_def if_distrib split: if_split)
  apply (auto split: if_split_asm simp: distinct_map inj_on_def intro!: distinct_concat)
done
from ‹c ≤ n› show ?thesis
  unfolding xs_def[symmetric]
by (intro ext, auto simp: free_upd_def)
  (solves ‹subst upd_pairs_upd[OF _ ‹distinct (map fst xs)›] upd_pairs_no_upd,
   auto simp: xs_def›)+
qed

```

```

lemma free_upd_alt_def1:
  free_upd n M c = (let
    M1 = upd_pairs' (((i,c), (i,0)). i ← [0..

```

```

lemma free_upd_free_upd1:
  free_upd n M c = free_upd1 n M c if c > 0 c ≤ n
proof -
  let ?x1 = λxs. upd_pairs' (((i,c), (i,0)). i ← xs, i ≠ c) M
  let ?x2 = λxs. fold (λi M. if i ≠ c then (M((i, c) := M(i, 0))) else M) xs M
  have *: ?x1 xs = ?x2 xs for xs
  by (induction xs arbitrary: M) (auto simp: upd_pairs'_Nil upd_pairs'_Cons)
  let ?y1 = λxs. upd_pairs (((c,i), ∞). i ← xs, i ≠ c)
  let ?y2 = fold (λi M. if i ≠ c then M((c,i) := ∞) else M)

```

```

have **: ?y1 xs M = ?y2 xs M for xs M
  by (induction xs arbitrary: M) (auto simp: upd_pairs_Nil upd_pairs_Cons)
show ?thesis
  unfolding free_upd_alt_def1[OF that] free_upd1_def op_mtx_get_def * ** ..
qed

lemma free_upd_alt_def:
  free_upd n M c =
    fold (λi M. if i ≠ c then (M((c,i) := ∞, (i, c) := M(i, 0))) else M) [0..

```

## 10.9 Imperative Refinement

```

context
  fixes n :: nat
  notes [id_rules] = itypeI[of n TYPE (nat)]
  and [sepref_import_param] = IdI[of n]
begin

```

interpretation DBM\_Impl n .

```

sepref_definition down_impl is
  RETURN o PR_CONST (down_upd n) :: mtx_assnd →a mtx_assn
  unfolding down_upd_alt_def1 upd_pairs_map PR_CONST_def
  unfolding Let_def prod.case
  unfolding fold_map comp_def
  unfolding neutral[symmetric]
  by sepref

```

```

context
  notes [map_type_eqs] = map_type_eqI[of TYPE(nat * nat ⇒ 'e) TYPE('e i_mtx)]

```

**begin**

**sepref\_register**

*abstr\_upd FW'' n Normalized\_Zone\_Semantics\_Impl\_Semantic\_Refinement.repair\_pair n*  
*and\_entry\_upd free\_upd1 n restrict\_zero\_upd n pre\_reset\_upd1 n*

**end**

**lemmas** [sepref\_fr\_rules] = *abstr\_upd\_impl.refine fw\_impl.refine FW''*

**sepref\_definition** *abstr\_FW\_impl* **is**

*uncurry (RETURN oo PR\_CONST (abstr\_FW\_upd n)) ::*  
*(list\_assn (aconstraint\_assn clock\_assn id\_assn))<sup>k</sup> \*<sub>a</sub> mtx\_assn<sup>d</sup> →<sub>a</sub> mtx\_assn*  
**unfolding** *abstr\_FW\_upd\_def FW''\_def[symmetric] PR\_CONST\_def* **by** *sepref*

**sepref\_definition** *free\_impl* **is**

*uncurry (RETURN oo PR\_CONST (free\_upd1 n)) ::*  
*[λ(⟦, i). i ≤ n]ₐ mtx\_assn<sup>d</sup> \*<sub>a</sub> nat\_assn<sup>k</sup> → mtx\_assn*  
**unfolding** *free\_upd1\_def op\_mtx\_set\_def[symmetric] PR\_CONST\_def* **by** *sepref*

**sepref\_definition** *and\_entry\_impl* **is**

*uncurry2 (uncurry (λx. RETURN ooo and\_entry\_upd x)) ::*  
*[λ(((i, j), ⟦), ⟦). i ≤ n ∧ j ≤ n]ₐ nat\_assn<sup>k</sup> \*<sub>a</sub> nat\_assn<sup>k</sup> \*<sub>a</sub> id\_assn<sup>k</sup> \*<sub>a</sub> mtx\_assn<sup>d</sup> → mtx\_assn*  
**unfolding** *and\_entry\_upd\_def* **by** *sepref*

**lemmas** [sepref\_fr\_rules] = *and\_entry\_impl.refine*

**sepref\_definition** *restrict\_zero\_impl* **is**

*uncurry (RETURN oo PR\_CONST (restrict\_zero\_upd n)) ::*  
*[λ(⟦, i). i ≤ n]ₐ mtx\_assn<sup>d</sup> \*<sub>a</sub> nat\_assn<sup>k</sup> → mtx\_assn*  
**unfolding** *restrict\_zero\_upd\_def PR\_CONST\_def* **by** *sepref*

**lemmas** [sepref\_fr\_rules] = *free\_impl.refine restrict\_zero\_impl.refine*

**sepref\_definition** *pre\_reset\_impl* **is**

*uncurry (RETURN oo PR\_CONST (pre\_reset\_upd1 n)) ::*  
*[λ(⟦, i). i ≤ n]ₐ mtx\_assn<sup>d</sup> \*<sub>a</sub> (clock\_assn)<sup>k</sup> → mtx\_assn*  
**unfolding** *pre\_reset\_upd1\_def PR\_CONST\_def* **by** *sepref*

**lemmas** [sepref\_fr\_rules] = *pre\_reset\_impl.refine*

**sepref\_definition** *pre\_reset\_list\_impl* **is**

*uncurry (RETURN oo PR\_CONST (pre\_reset\_list\_upd1 n)) ::*  
*mtx\_assn<sup>d</sup> \*<sub>a</sub> (list\_assn (clock\_assn))<sup>k</sup> →<sub>a</sub> mtx\_assn*  
**unfolding** *pre\_reset\_list\_upd1\_def PR\_CONST\_def* **by** *sepref*

**sepref\_definition** *and\_entry\_repair\_impl* **is**

*uncurry2 (uncurry (λx. RETURN ooo PR\_CONST (and\_entry\_repair\_upd n) x)) ::*  
*[λ(((i, j), ⟦), ⟦). i ≤ n ∧ j ≤ n]ₐ nat\_assn<sup>k</sup> \*<sub>a</sub> nat\_assn<sup>k</sup> \*<sub>a</sub> id\_assn<sup>k</sup> \*<sub>a</sub> mtx\_assn<sup>d</sup> → mtx\_assn*  
**unfolding** *and\_entry\_repair\_upd\_def PR\_CONST\_def* **by** *sepref*

**private definition**

*V\_dbm'' = V\_dbm' n*

We use  $V\_dbm''$  because we cannot register  $V\_dbm'$  twice with the refinement framework: we view it as a function first, and as a DBM later.

**lemma**  $V\_dbm'\_alt\_def$ :  
 $V\_dbm' n = op\_amtx\_new (Suc n) (Suc n) (V\_dbm'')$   
**unfolding**  $V\_dbm''\_def$  **by** *simp*

**notation**  $fun\_rel\_syn$  (**infixr**  $\rightarrow 60$ )

We need the custom rule here because  $V\_dbm'$  is a higher-order constant

**lemma**  $[sepref\_fr\_rules]$ :  
 $(uncurry0 (return V\_dbm''), uncurry0 (RETURN (PR\_CONST (V\_dbm''))))$   
 $\in unit\_assn^k \rightarrow_a pure (nat\_rel \times_r nat\_rel \rightarrow Id)$   
**by** *sepref\_to\_hoare\_sep\_auto*

**sepref\_register**  $V\_dbm'' :: nat \times nat \Rightarrow \_ DBMEntry$

Necessary to solve side conditions of  $op\_amtx\_new$

**lemma**  $V\_dbm''\_bounded$ :  
 $mtx\_nonzero V\_dbm'' \subseteq \{0..<Suc n\} \times \{0..<Suc n\}$   
**unfolding**  $mtx\_nonzero\_def$   $V\_dbm''\_def$   $V\_dbm'\_def$  *neutral* **by** *auto*

We need to pre-process the lemmas due to a failure of *TRADE*

**lemma**  $V\_dbm''\_bounded\_1$ :  
 $(a, b) \in mtx\_nonzero V\_dbm'' \implies a < Suc n$   
**using**  $V\_dbm''\_bounded$  **by** *auto*

**lemma**  $V\_dbm''\_bounded\_2$ :  
 $(a, b) \in mtx\_nonzero V\_dbm'' \implies b < Suc n$   
**using**  $V\_dbm''\_bounded$  **by** *auto*

**lemmas**  $[sepref\_opt\_simps] = V\_dbm''\_def$

**sepref\_definition**  $V\_dbm\_impl$  **is**  
 $uncurry0 (RETURN (PR\_CONST (V\_dbm' n))) :: unit\_assn^k \rightarrow_a mtx\_assn$   
**supply**  $V\_dbm''\_bounded\_1[simp]$   $V\_dbm''\_bounded\_2[simp]$   
**using**  $V\_dbm''\_bounded$   
**apply**  $(subst V\_dbm'\_alt\_def)$   
**unfolding**  $PR\_CONST\_def$  **by** *sepref*

This form of 'type casting' is used to assert a bound on the natural number.

**private definition**  $make\_clock :: nat \Rightarrow nat$  **where**  $[sepref\_opt\_simps]$ :  
 $make\_clock x = x$

**lemma**  $make\_clock[sepref\_import\_param]$ :  
 $(make\_clock, make\_clock) \in [\lambda i. i \leq n]_f nat\_rel \rightarrow nbn\_rel (Suc n)$   
**unfolding**  $make\_clock\_def$  **by**  $(rule frefI)$  *simp*

**private definition**  $get\_entries m =$   
 $[(make\_clock i, make\_clock j).$   
 $i \leftarrow [0..<Suc n], j \leftarrow [0..<Suc n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m (i, j) \neq \infty]$

**private lemma**  $get\_entries\_alt\_def$ :  
 $get\_entries m = [(make\_clock i, make\_clock j).$

$i \leftarrow [0..< \text{Suc } n], j \leftarrow [0..< \text{Suc } n], (i > 0 \vee j > 0) \wedge m(i, j) \neq \infty$   
**unfolding** *get\_entries\_def* **by** (*intro arg\_cong*[**where**  $f = \text{concat}$ ] *map\_cong*) *auto*

**private definition**

*upd\_entry i j M m = and\_entry\_repair\_upd n j i (neg\_dbm\_entry (m (i, j))) (op\_mtx\_copy M)*

**private definition**

*upd\_entries i j m = map ( $\lambda M. \text{upd\_entry } i j M m$ )*

**context**

**notes** [*map\_type\_eqs*] = *map\_type\_eqI*[*of TYPE*( $\text{nat} * \text{nat} \Rightarrow 'e$ ) *TYPE*('e i\_mtx)]

**begin**

**sepref\_register**

*dbm\_minus\_canonical\_check\_upd n*  
*dbm\_subset\_fed\_upd n abstr\_FW\_upd n pre\_reset\_list\_upd1 n V\_dbm' n down\_upd n*  
*upd\_entry upd\_entries get\_entries and\_entry\_repair\_upd n*

**end**

**lemma** [*sepref\_import\_param*]: (*neg\_dbm\_entry*, *neg\_dbm\_entry*)  $\in Id \rightarrow Id$  **by** *simp*

**lemmas** [*sepref\_fr\_rules*] = *and\_entry\_repair\_impl.refine*

**sepref\_definition** *upd\_entry\_impl* **is**

*uncurry2 (uncurry ( $\lambda x. \text{RETURN } \text{ooo } PR\_CONST \text{ upd\_entry } x$ ))) ::*  
 $[\lambda(((i, j), \_), \_). i \leq n \wedge j \leq n]_a$   
 $\text{nat\_assn}^k *_a \text{nat\_assn}^k *_a \text{mtx\_assn}^k *_a \text{mtx\_assn}^k \rightarrow \text{mtx\_assn}$   
**unfolding** *upd\_entry\_def PR\_CONST\_def* **by** *sepref*

This is to ensure that the refinement initially infers the right 'type' for list arguments.

**lemma** [*intf\_of\_assn*]:

*intf\_of\_assn AA TYPE('aa)  $\implies$  intf\_of\_assn (list\_assn(AA)) TYPE('aa list)*  
**by** (*rule intf\_of\_assnI*)

**lemmas** [*sepref\_fr\_rules*] = *upd\_entry\_impl.refine*

**sepref\_definition** *upd\_entries\_impl* **is**

*uncurry2 (uncurry ( $\lambda x. \text{RETURN } \text{ooo } PR\_CONST \text{ upd\_entries } x$ ))) ::*  
 $[\lambda(((i, j), \_), \_). i \leq n \wedge j \leq n]_a$   
 $\text{nat\_assn}^k *_a \text{nat\_assn}^k *_a \text{mtx\_assn}^k *_a (\text{list\_assn } \text{mtx\_assn})^k \rightarrow \text{list\_assn } \text{mtx\_assn}$   
**unfolding** *upd\_entries\_def PR\_CONST\_def*  
**unfolding** *map\_conv\_rev\_fold rev\_conv\_fold*  
**supply** [*sepref\_fr\_rules*] = *HOL\_list\_empty\_hnr\_aux*  
**by** *sepref*

**lemma** [*sepref\_import\_param*]: ( $(=)$ ,  $(=)$ )  $\in Id \rightarrow Id \rightarrow Id$  **by** *simp*

**sepref\_definition** *get\_entries\_impl* **is**

*RETURN o PR\_CONST get\_entries ::*  
 $\text{mtx\_assn}^k \rightarrow_a \text{list\_assn } ((\text{clock\_assn}) \times_a (\text{clock\_assn}))$   
**unfolding** *get\_entries\_alt\_def PR\_CONST\_def*  
**unfolding** *map\_conv\_rev\_fold*

```

unfolding concat_conv_fold_rev
supply [sepref_fr_rules] = HOL_list_empty_hnr_aux
by sepref

lemmas [sepref_fr_rules] = upd_entries_impl.refine get_entries_impl.refine

private lemma dbm_minus_canonical_upd_alt_def:
  dbm_minus_canonical_upd n xs m =
  concat (map (λij. map (λ M.
    and_entry_repair_upd n (snd ij) (fst ij) (neg_dbm_entry (m (fst ij, snd ij))) (op_mtx_copy
M))
  xs) (get_entries m))
unfolding dbm_minus_canonical_upd_def op_mtx_copy_def get_entries_def split_beta make_clock_def
by simp

sepref_definition dbm_minus_canonical_impl is
  uncurry (RETURN oo PR_CONST (dbm_minus_canonical_check_upd n)) ::
  (list_assn mtx_assn)k *a mtx_assnk →a list_assn mtx_assn
unfolding dbm_minus_canonical_check_upd_def dbm_minus_canonical_upd_alt_def PR_CONST_def
unfolding upd_entry_def[symmetric] upd_entries_def[symmetric]
unfolding filter_conv_rev_fold concat_map_conv_rev_fold rev_conv_fold
supply [sepref_fr_rules] = HOL_list_empty_hnr_aux
by sepref

lemmas [sepref_fr_rules] = dbm_minus_canonical_impl.refine

sepref_definition dbm_subset_fed_impl is
  uncurry (RETURN oo PR_CONST (dbm_subset_fed_upd n)) ::
  mtx_assnd *a (list_assn mtx_assn)d →a bool_assn
unfolding dbm_subset_fed_upd_alt_def PR_CONST_def
unfolding list_ex_foldli filter_conv_rev_fold
unfolding short_circuit_conv
supply [sepref_fr_rules] = HOL_list_empty_hnr_aux
by sepref

```

**end**

## 10.10 Implementation for a Concrete Automaton

```

context TA_Impl
begin

sublocale TA_Impl_Op
  where loc_rel = loc_rel and f = PR_CONST E_op'' and op_impl = E_op''_impl
  unfolding PR_CONST_def by standard (rule E_op''_impl.refine)

end

context TA_Impl
begin

definition
  check_deadlock_dbm l M = dbm_subset_fed_upd n M (
    [down_upd n (abstr_FW_upd n (inv_of A l) (abstr_FW_upd n g

```

```

    (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of_A l') (V_dbm' n)) r)
  ). (g, a, r, l') ← trans_fun l
]
)

```

**abbreviation** zone\_of ( $\llbracket \_ \rrbracket$ ) **where** zone\_of  $M \equiv [\text{curry } (\text{conv\_M } M)]_{v,n}$

**theorem** check\_deadlock\_dbm\_correct:

```

  TA.check_deadlock l  $\llbracket M \rrbracket$  = check_deadlock_dbm l M if
   $\llbracket M \rrbracket \subseteq V \mid l \in \text{states}' \text{Deadlock.canonical}' n (\text{curry } (\text{conv\_M } M))$ 

```

**proof** –

0. Setup & auxiliary facts

```

include lifting_syntax
note [simp del] = And.simps abstr.simps
have inv_of_simp: inv_of (conv_A A) l = conv_cc (inv_of A l) for l
  using inv_fun unfolding inv_rel_def b_rel_def fun_rel_def conv_A_def
  by (force split: prod.split simp: inv_of_def)

```

```

have trans_funD: l' ∈ states
  collect_clks (inv_of A l) ⊆ clk_set A collect_clks (inv_of A l') ⊆ clk_set A
  collect_clks g ⊆ clk_set A set r ⊆ clk_set A
  if (g, a, r, l') ∈ set (trans_fun l) for g a r l'
  subgoal
    using ⟨l ∈ states'⟩ that trans_impl_states by auto
  subgoal
    by (metis collect_clks_inv_clk_set)
  subgoal
    by (metis collect_clks_inv_clk_set)
  subgoal
    using that ⟨l ∈  $\_$ ⟩ by (intro collect_clocks_clk_set trans_impl_trans_of)
  subgoal
    using that ⟨l ∈  $\_$ ⟩ by (intro reset_clk_set trans_impl_trans_of)
  done

```

1. Transfer to most abstract DBM operations

```

have [transfer_rule]:
  (eq_onp (λ (g, a, r, l'). (g, a, r, l') ∈ set (trans_fun l)) ==> RI2 n)
  (λ(g, a, r, l'). down n
    (abstr_FW n (conv_cc (inv_of A l))
      (abstr_FW n (conv_cc g)
        (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v))
  (λ(g, a, r, l'). down_upd n
    (abstr_FW_upd n (inv_of A l)
      (abstr_FW_upd n g (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of A l') (V_dbm'
n)) r))
    ))
  (is ( $\_$  ==> RI2 n) (λ (g, a, r, l'). ?f g a r l') (λ (g, a, r, l'). ?g g a r l'))
proof –
  {
  fix g a r l' assume (g, a, r, l') ∈ set (trans_fun l)
  have *: rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri (conv_ac ac) ac
    if constraint_clk ac > 0 constraint_clk ac ≤ n for ac
    using that by (cases ac) (auto simp: eq_onp_def ri_def)
  }

```

```

have *: list_all2 (rel_acconstraint (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) ri) (conv_cc g) g
  if collect_clks g  $\subseteq$  clk_set A for g
  unfolding list_all2_map1 using that_clock_range
  by (auto simp: collect_clks_def intro!: * list.rel_refl_strong)
have [transfer_rule]:
  list_all2 (rel_acconstraint (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) ri) (conv_cc g) g
  list_all2 (rel_acconstraint (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) ri)
    (conv_cc (inv_of A l)) (inv_of A l)
  list_all2 (rel_acconstraint (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) ri)
    (conv_cc (inv_of A l')) (inv_of A l')
  by (intro * trans_funD[OF  $\langle (g, a, r, l') \in \text{set } (\text{trans\_fun } l) \rangle$ ]) +
have [transfer_rule]:
  list_all2 (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) r r
  using trans_funD(5)[OF  $\langle (g, a, r, l') \in \text{set } (\text{trans\_fun } l) \rangle$ ] clock_range
  by (auto 4 3 dest: bspec subsetD simp: eq_onp_def list_all2_same)
have RI2 n (?f g a r l') (?g g a r l')
  by transfer_prover
} then show ?thesis
by (intro rel_funI, clarsimp simp: eq_onp_def)
qed
have [transfer_rule]:
  (eq_onp ( $\lambda l'. l' = l$ ) ==> list_all2 (eq_onp ( $\lambda (g,a,r,l'). (g,a,r,l') \in \text{set } (\text{trans\_fun } l)$ )))
  trans_fun trans_fun

  unfolding rel_fun_def eq_onp_def by (auto intro: list.rel_refl_strong)
note [transfer_rule] = dbm_subset_fed_transfer
have [transfer_rule]: eq_onp ( $\lambda l'. l' = l$ ) l l (eq_onp ( $\lambda x. x < \text{Suc } n$ )) n n
  by (auto simp: eq_onp_def)
have *:
  (dbm_subset_fed_check n (curry (conv_M M))
    (map ( $\lambda (g, a, r, l').$ 
      down n
      (abstr_FW n (conv_cc (inv_of A l))
        (abstr_FW n (conv_cc g)
          (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v))
      (trans_fun l))) =
  (dbm_subset_fed_upd n M
    (map ( $\lambda (g, a, r, l').$ 
      down_upd n
      (abstr_FW_upd n (inv_of A l)
        (abstr_FW_upd n g (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of A l') (V_dbm'
n)) r))
      ))
      (trans_fun l)))

  by transfer_prover

```

## 2. Semantic argument establishing equivalences between zones and DBMs

```

have **:
  [down n (abstr_FW n (conv_cc (inv_of A l)) (abstr_FW n (conv_cc g)
    (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v)]v,n
= (zone_set_pre ( $\{u. u \vdash \text{inv\_of } (\text{conv\_A } A) l'\} \cap V$ ) r  $\cap \{u. \forall x \in \text{set } r. u x \geq 0\}$ 
   $\cap \{u. u \vdash \text{conv\_cc } g\} \cap \{u. u \vdash \text{inv\_of } (\text{conv\_A } A) l'\}^\downarrow \cap V$ 
  if  $(g, a, r, l') \in \text{set}(\text{trans\_fun } l)$  for  $g a r l'$ 

```

**proof** –

```

from trans_funD[OF  $\langle (g, a, r, l') \in \text{set } (\text{trans\_fun } l) \rangle$ ] have clock_conditions:
   $\forall c \in \text{collect\_clks } (\text{conv\_cc } (\text{inv\_of } A \ l)). \ 0 < c \wedge c \leq n$ 
   $\forall c \in \text{collect\_clks } (\text{conv\_cc } (\text{inv\_of } A \ l')). \ 0 < c \wedge c \leq n$ 
   $\forall c \in \text{collect\_clks } (\text{conv\_cc } g). \ 0 < c \wedge c \leq n$ 
   $\forall c \in \text{set } r. \ 0 < c \wedge c \leq n$ 
  using  $\langle l \in \text{states}' \rangle$  clock_range by (auto simp: constraint_clk_conv_cc)
have structural_conditions:
  abstr_FW n (conv_cc (inv_of A l')) V_dbm v 0 0  $\leq 0$ 
   $\forall x \in \text{set } r. \text{abstr\_FW } n \ (\text{map } \text{conv\_ac } (\text{inv\_of } A \ l')) \ V\_dbm \ v \ 0 \ x \leq 0$ 
subgoal
  using abstr_FW_diag_preservation[of n V_dbm conv_cc (inv_of A l') v]
  by (simp add: V_dbm_def)
subgoal
  using  $\langle \forall c \in \text{set } r. \ 0 < c \wedge c \leq n \rangle$ 
  by (auto 4 3 simp: V_dbm_def intro: abstr_FW_mono order.trans)
done
note side_conditions = clock_conditions structural_conditions
  abstr_FW_canonical[unfolded canonical'_def] abstr_FW_canonical
show ?thesis
apply (subst dbm.down_correct, rule side_conditions)
apply (subst dbm.abstr_FW_correct, rule side_conditions)
apply (subst dbm.abstr_FW_correct, rule side_conditions)
apply (subst dbm.pre_reset_list_correct, (rule side_conditions)+)
apply (subst dbm.abstr_FW_correct, (rule side_conditions)+)
apply (simp add: inv_of_simp dbm.V_dbm_correct atLeastLessThanSuc_atLeastAtMost
Int_commute)
done
qed
have **:
   $(\bigcup x \in \text{set } (\text{trans\_fun } l). \text{dbm.zone\_of } ( \text{case } x \text{ of } (g, a, r, l') \Rightarrow \text{down } n \text{ (abstr\_FW } n \text{ (conv\_cc } (\text{inv\_of } A \ l)) \text{ (abstr\_FW } n \text{ (conv\_cc } g) \text{ (pre\_reset\_list } n \text{ (abstr\_FW } n \text{ (conv\_cc } (\text{inv\_of } A \ l')) \ V\_dbm \ v) \ r) \ v) \ v) )$ 
   $= (\bigcup (g,a,r,l') \in \text{set } (\text{trans\_fun } l). ((\text{zone\_set\_pre } (\{u. u \vdash \text{inv\_of } (\text{conv\_A } A) \ l'\} \cap V) \ r \cap \{u. \forall x \in \text{set } r. u \ x \geq 0\} \cap \{u. u \vdash \text{conv\_cc } g\} \cap \{u. u \vdash \text{inv\_of } (\text{conv\_A } A) \ l'\})^\downarrow \cap V)$ 
  )
  by (auto simp: **)

```

3. Putting it all together

```

have transD:  $\exists g'. (g', a, r, l') \in \text{set } (\text{trans\_fun } l) \wedge g = \text{conv\_cc } g'$ 
if conv_A A  $\vdash l \xrightarrow{g,a,r} l'$  for g a r l'
using trans_of_trans_impl[OF  $\langle l \in \text{states}' \rangle$ ] that
unfolding trans_of_def conv_A_def by (auto 5 0 split: prod.split_asm)
have transD2:
  conv_A A  $\vdash l \xrightarrow{\text{conv\_cc } g,a,r} l'$  if  $(g, a, r, l') \in \text{set } (\text{trans\_fun } l)$  for g a r l'
using trans_impl_trans_of[OF that  $\langle l \in \text{states}' \rangle$ ]
unfolding trans_of_def conv_A_def by (auto 4 3 split: prod.split)
show ?thesis

```

```

unfolding TA.check_deadlock_alt_def[OF  $\langle \_ \subseteq V \rangle$ ] check_deadlock_dbm_def inv_of_A_def
*[symmetric]
apply (subst dbm.dbm_subset_fed_check_correct[symmetric, OF that(3)])
apply (simp add: **)
apply safe
subgoal for x
  by (auto 4 5 elim!: transD[elim_format] dest: subsetD)
subgoal for x
  apply (drule subsetD, assumption)
  apply clarsimp
  subgoal for g a r l'
    apply (inst_existentials
      (zone_set_pre ({u. u  $\vdash$  inv_of (conv_A A) l'}  $\cap$  V) r  $\cap$  {u.  $\forall x \in \text{set } r. 0 \leq u x$ }
         $\cap$  {u. u  $\vdash$  conv_cc g}  $\cap$  {u. u  $\vdash$  inv_of (conv_A A) l'}) $^\downarrow$   $\cap$  V conv_cc g a r l')
    apply (auto dest: transD2)
  done
done
done
qed

lemmas [sepref_fr_rules] =
  V_dbm_impl.refine abstr_FW_impl.refine pre_reset_list_impl.refine
  down_impl.refine dbm_subset_fed_impl.refine

sepref_definition check_deadlock_impl is
  uncurry (RETURN oo PR_CONST check_deadlock_dbm) ::
  location_assnk *a (mtx_assn n)d  $\rightarrow_a$  bool_assn
unfolding check_deadlock_dbm_def PR_CONST_def
unfolding case_prod_beta
unfolding map_conv_rev_fold
apply (rewrite HOL_list.fold_custom_empty)
by sepref

end

```

## 10.11 Checking a Property on the Reachable Set

```

locale Worklist_Map2_Impl_check =
  Worklist_Map2_Impl_finite +
  fixes Q :: 'a  $\Rightarrow$  bool
  fixes Qi
  assumes Q_refine: (Qi, RETURN o PR_CONST Q)  $\in A^d \rightarrow_a$  bool_assn
  and F_False: F = ( $\lambda \_.$  False)
  and Q_mono:  $\bigwedge a b. a \preceq b \implies \neg \text{empty } a \implies \text{reachable } a \implies \text{reachable } b \implies Q a \implies Q b$ 
begin

```

**definition** check\_passed :: bool nres **where**

```

check_passed = do {
  (r, passed)  $\leftarrow$  pw_algo_map2;
  ASSERT (finite (dom passed));
  passed  $\leftarrow$  ran_of_map passed;
  r  $\leftarrow$  nfoldli passed ( $\lambda b. \neg b$ )
  ( $\lambda$  passed'  $\_.$ 
    do {

```

```

    passed' ← SPEC (λl. set l = passed');
    nfoldli passed' (λb. ¬b)
      (λv' __.
        if Q v' then RETURN True else RETURN False
      )
    False
  }
)
False;
RETURN r
}

```

**lemma** *check\_passed\_correct*:

*check\_passed* ≤ SPEC (λ r. (r ↔ (∃ a. reachable a ∧ ¬ empty a ∧ Q a)))

**proof** –

**have** [simp]: *F\_reachable* = False

**unfolding** *F\_reachable\_def* *F\_False* *Search\_Space\_Defs.F\_reachable\_def* **by** *simp*

**define** *outer\_inv* **where** *outer\_inv* passed done todo r ≡

*set* done ∪ *set* todo = ran passed ∧

(¬ r → (∀ S ∈ *set* done. ∀ x ∈ S. ¬ Q x)) ∧

(r → (∃ a. reachable a ∧ ¬ empty a ∧ Q a))

**for** passed :: 'c ⇒ 'a set option **and** done todo **and** r :: bool

**define** *inner\_inv* **where** *inner\_inv* passed done todo r ≡

*set* done ∪ *set* todo = passed ∧

(¬ r → (∀ x ∈ *set* done. ¬ Q x)) ∧

(r → (∃ a. reachable a ∧ ¬ empty a ∧ Q a))

**for** passed :: 'a set **and** done todo **and** r :: bool

**show** ?thesis

**supply** [refine\_vcg] = pw\_algo\_map2\_correct

**unfolding** *check\_passed\_def*

**apply** (refine\_rcg refine\_vcg)

**subgoal**

**by** *auto*

**subgoal for** \_ brk\_false passed range\_list

**apply** (rule nfoldli\_rule[**where** I = *outer\_inv* passed])

**subgoal**

**unfolding** *outer\_inv\_def*

**by** *auto*

**apply** (refine\_rcg refine\_vcg)

**subgoal for** current done todo r xs

**apply** *clarsimp*

**apply** (rule nfoldli\_rule[**where** I = *inner\_inv* current])

**subgoal**

**unfolding** *inner\_inv\_def*

**by** *auto*

**subgoal for** p x l1 l2 r'

**by** (*clarsimp simp: inner\_inv\_def outer\_inv\_def map\_set\_rel\_def*)

(metis Sup\_insert Un\_insert\_left insert\_iff subset\_Collect\_conv)

subgoal for  $p \ l1 \ l2 \ r'$   
 unfolding inner\_inv\_def outer\_inv\_def  
 by (metis append.assoc append\_Cons append\_Nil set\_append)

subgoal for  $p \ r'$   
 unfolding inner\_inv\_def outer\_inv\_def  
 by clarsimp  
 done

subgoal for  $l1 \ l2 \ r$   
 unfolding outer\_inv\_def  
 by auto

subgoal for  $r$   
 unfolding outer\_inv\_def  
 apply clarsimp  
 apply (elim allE impE)  
 apply (intro conjI; assumption)  
 apply safe  
 apply (drule Q\_mono, assumption+)  
 unfolding map\_set\_rel\_def  
 by auto  
 done  
 done  
 qed

schematic\_goal pw\_algo\_map2\_refine:  
 ( $?x$ , uncurry0 (PR\_CONST pw\_algo\_map2))  $\in$   
 unit\_assn<sup>k</sup>  $\rightarrow_a$  bool\_assn  $\times_a$  hm.hms\_assn'  $K$  (lso\_assn  $A$ )  
 unfolding PR\_CONST\_def hm.hms\_assn'\_id\_hms\_assn[symmetric] by (rule pw\_algo\_map2\_impl.refine\_raw)

sepref\_register pw\_algo\_map2

sepref\_register PR\_CONST  $Q$

sepref\_thm check\_passed\_impl is  
 uncurry0 check\_passed :: unit\_assn<sup>k</sup>  $\rightarrow_a$  bool\_assn  
 supply [sepref\_fr\_rules] = pw\_algo\_map2\_refine ran\_of\_map\_impl.refine lso\_id\_hnr  $Q$ \_refine  
 using pure\_K left\_unique\_K right\_unique\_K  
 unfolding check\_passed\_def  
 apply (rewrite in  $Q$  PR\_CONST\_def[symmetric])  
 unfolding hm.hms\_fold\_custom\_empty  
 unfolding list\_of\_set\_def[symmetric]  
 by sepref

concrete\_definition (in  $-$ ) check\_passed\_impl  
 uses Worklist\_Map2\_Impl\_check.check\_passed\_impl.refine\_raw is (uncurry0 ?f,\_) $\in$

```

lemma check_passed_impl_hnr:
  (uncurry0 (check_passed_impl succsi a0 i Fi Lei emptyi keyi copyi tracei Qi),
   uncurry0 (RETURN (∃ a. reachable a ∧ ¬ empty a ∧ Q a)))
  ∈ unit_assnk →a bool_assn
using
  check_passed_impl.refine[
    OF Worklist_Map2_Impl_check_axioms,
    FCOMP check_passed_correct[THEN Id_SPEC_refine, THEN nres_rell]
  ]
by (simp add: RETURN_def)

```

**end**

## 10.12 Complete Deadlock Checker

**Miscellaneous Properties** **context** *E\_From\_Op\_Bisim*  
**begin**

More general variant of  $?F\_rel = (\lambda(l, D). ?F\ l \wedge \neg check\_diag\ n\ D) \implies (\exists l'\ D'. E^{**}\ a_0\ (l', D') \wedge ?F\_rel\ (l', D')) = (\exists l'\ D'. E\_from\_op^{**}\ a_0\ (l', D') \wedge ?F\_rel\ (l', D'))^*$

**theorem** *E\_from\_op\_reachability\_check*:

**assumes**  $\bigwedge a\ b. P\ a \implies a \sim b \implies wf\_state\ a \implies wf\_state\ b \implies P\ b$

**shows**

$(\exists l'\ D'. E^{**}\ a_0\ (l', D') \wedge P\ (l', D')) \longleftrightarrow (\exists l'\ D'. E\_from\_op^{**}\ a_0\ (l', D') \wedge P\ (l', D'))$

**by** (rule *E\_E1\_steps\_equiv*[OF *E\_E\_from\_op\_step* *E\_from\_op\_E\_step* *E\_from\_op\_wf\_state*];  
 (assumption | rule *assms*))

**end**

**context** *Regions\_TA*  
**begin**

**lemma** *check\_deadlock\_anti\_mono*:

*check\_deadlock* *l* *Z* **if** *check\_deadlock* *l* *Z'*  $Z \subseteq Z'$

**using** *that* **unfolding** *check\_deadlock\_def* **by** *auto*

**lemma** *global\_clock\_numbering*:

*global\_clock\_numbering* *A* *v* *n*

**using** *valid\_abstraction* *clock\_numbering\_le* *clock\_numbering* **by** *blast*

Variant of  $\llbracket (\forall c. 0 < v\ c \wedge (\forall x\ y. v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)) \wedge (\forall c \in clk\_set\ ?A. v\ c \leq n) \wedge (\forall k \leq n. 0 < k \longrightarrow (\exists c. v\ c = k)) \rrbracket$ ; *Closure.valid\_abstraction*  $?A\ X\ (\lambda x\ xa. real\ (k\ x\ xa)) \rrbracket \implies Bisimulation\_Invariant\ (\lambda(l, Z)\ (l', Z'). ?A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\})\ (\lambda(l, D)\ (l', D'). \exists a. ?A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}_a} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\})\ (\lambda(l, Z)\ (l', D). l = l' \wedge Z = [D]_{v,n})\ (\lambda_. True)\ (\lambda(l, y). local.valid\_dbm\ y)$  without emptiness.

**lemma** *bisim*:

*Bisimulation\_Invariant*

$(\lambda(l, Z)\ (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle)$

$(\lambda(l, D)\ (l', D'). \exists a. A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle)$

$(\lambda(l, Z)\ (l', D). l = l' \wedge Z = [D]_{v,n})$

$(\lambda_. True)\ (\lambda(l, D). valid\_dbm\ D)$

**proof** (*standard*, *goal\_cases*)

—  $\beta \Rightarrow \mathcal{N}$

**case**  $(1\ a\ b\ a')$

```

    then show ?case
      by (blast elim: norm_beta_complete1[OF global_clock_numbering valid_abstraction])
next
  —  $\mathcal{N} \Rightarrow \beta$ 
  case (2 a a' b')
  then show ?case
    by (blast intro: norm_beta_sound''[OF global_clock_numbering valid_abstraction])
next
  —  $\beta$  invariant
  case (3 a b)
  then show ?case
    by simp
next
  —  $\mathcal{N}$  invariant
  case (4 a b)
  then show ?case
    by (auto intro: valid_dbm_step_z_norm''[OF global_clock_numbering valid_abstraction])
qed

```

```

lemma steps_z_beta_reaches:
  reaches (l, Z) (l', Z') if  $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle$   $Z' \neq \{\}$ 
  using that
proof (induction (l', Z') arbitrary: l' Z' rule: rtranclp_induct)
  case base
  then show ?case
    by blast
next
  case (step y l'' Z'')
  obtain l' Z' where [simp]:  $y = (l', Z')$ 
    by force
  from step.prem1 step.hyps(2) have  $Z' \neq \{\}$ 
    by (clarsimp simp: step_z_beta'_empty)
  from step.prem1 step.hyps(3)[OF  $\langle y = \_ \rangle$  this] step.hyps(2) show ?case
    including graph_automation_aggressive by auto
qed

```

```

lemma reaches_steps_z_beta_iff:
  reaches (l, Z) (l', Z')  $\longleftrightarrow A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle \wedge Z' \neq \{\}$  if  $Z \neq \{\}$ 
  apply safe
  subgoal
    including graph_automation_aggressive by (induction rule: rtranclp_induct) auto
  using that by (auto dest: steps_z_beta_reaches elim: rtranclp.cases)

```

end

```

lemma dbm_int_init_dbm:
  dbm_int (curry init_dbm) n
  unfolding init_dbm_def by auto

```

```

context TA_Start
begin

```

```

lemma wf_dbm_canonical'D:
  Deadlock.canonical' n (curry (conv_M D)) if wf_dbm D

```

```

using that unfolding wf_dbm_def Deadlock.canonical'_def check_diag_conv_M_iff by simp

lemma subsumes_dbm_subsetD:
  dbm_subset n M M' if subsumes n (l, M) (l', M')  $\neg$  check_diag n M
  using that by (cases l = l') (auto simp: subsumes_simp_1 subsumes_simp_2)

lemma subsumes_loc_eqD:
  l = l' if subsumes n (l, M) (l', M')  $\neg$  check_diag n M
  using that by (cases l = l') (auto simp: subsumes_simp_1 subsumes_simp_2)

lemma init_dbm_zone:
  [curry (init_dbm :: real DBM')]_{v,n} = {u.  $\forall c \in \{1..n\}. u\ c = 0$ }
  using init_dbm_semantics by blast

lemma not_check_deadlock_mono:
  (case a of (l, M)  $\Rightarrow$   $\neg$  TA.check_deadlock l (dbm.zone_of (curry (conv_M M))))  $\implies$ 
  a  $\sim$  b  $\implies$  wf_state a  $\implies$  wf_state b  $\implies$ 
  case b of (l, M)  $\Rightarrow$   $\neg$  TA.check_deadlock l (dbm.zone_of (curry (conv_M M)))
  unfolding TA.check_deadlock_def state_equiv_def dbm_equiv_def by auto

lemma not_check_deadlock_non_empty:
  Z  $\neq$  {} if  $\neg$  TA.check_deadlock l Z
  using that unfolding TA.check_deadlock_def by auto

end

context TA_Impl
begin

lemma op_E_from_op_iff:
  op.E_from_op = E_op''.E_from_op
  unfolding PR_CONST_def ..

lemmas reachable_states = reachable_states[unfolded op_E_from_op_iff]

lemma E_op''_states:
  l'  $\in$  states if E_op''.E_from_op (l, M) (l', M') l  $\in$  states
  using that by (force simp: a0_def state_set_def E_op''.E_from_op_def)

Instantiating the Checker Algorithm corollary check_deadlock_dbm_correct':
  assumes l  $\in$  states' wf_state (l, M)
  shows TA.check_deadlock l (dbm.zone_of (curry (conv_M M))) = check_deadlock_dbm l M
  apply (rule check_deadlock_dbm_correct)
  using assms
  unfolding wf_state_def Deadlock.canonical'_def prod.case
  apply (blast dest: wf_dbm_altD)
  apply (rule assms)
  using assms
  unfolding wf_state_def Deadlock.canonical'_def prod.case
  apply -
  apply (drule wf_dbm_altD(1))
  unfolding canonical'_conv_M_iff check_diag_conv_M_iff by simp

```

**corollary** *check\_deadlock\_dbm\_correct''*:  
**assumes**  $E\_op''.E\_from\_op^{**} a_0 (l, M)$   
**shows**  $TA.check\_deadlock\ l\ (dbm.zone\_of\ (curry\ (conv\_M\ M))) = check\_deadlock\_dbm\ l\ M$   
**using** *assms*  
**apply** –  
**apply** (*rule check\_deadlock\_dbm\_correct'*)  
**apply** (*drule reachable\_states, use states'\_states in auto; fail*)  
**unfolding** *wf\_state\_def prod.case* **apply** (*erule E\_op''.reachable\_wf\_dbm*)  
**done**

**lemma** *not\_check\_deadlock\_non\_empty*:  
 $Z \neq \{\}$  **if**  $\neg TA.check\_deadlock\ l\ Z$   
**using** *that* **unfolding** *TA.check\_deadlock\_def* **by** *auto*

**sepref\_register** *check\_deadlock\_dbm* :: '*s*  $\Rightarrow$  int DBMEntry *i* *mtx*  $\Rightarrow$  bool

**sepref\_definition** *check\_deadlock\_neg\_impl* **is**  
 $RETURN\ o\ (\lambda(l, M). \neg check\_deadlock\_dbm\ l\ M) :: (location\_assn \times_a (mtx\_assn\ n))^d \rightarrow_a$   
 $bool\_assn$   
**supply** [*sepref\_fr\_rules*] = *check\_deadlock\_impl.refine*  
**by** *sepref*

**lemma** *not\_check\_deadlock\_compatible*:  
**assumes**  
 $(case\ a\ of\ (l, Z) \Rightarrow \lambda(l', D'). l' = l \wedge dbm.zone\_of\ (curry\ (conv\_M\ D')) = Z)\ b$   
 $case\ b\ of\ (l, M) \Rightarrow l \in states' \wedge wf\_state\ (l, M)$   
**shows**  
 $(case\ a\ of\ (l, Z) \Rightarrow \neg TA.check\_deadlock\ l\ Z) = (case\ b\ of\ (l, M) \Rightarrow \neg check\_deadlock\_dbm\ l\ M)$   
**using** *assms* **by** (*auto simp: check\_deadlock\_dbm\_correct'[symmetric]*)

**lemma** *deadlock\_check\_diag*:  
 $\neg check\_diag\ n\ M$  **if**  $\neg check\_deadlock\_dbm\ l\ M\ E\_op''.E\_from\_op^{**} a_0 (l, M)$   
**using** *that(1)*  
**apply** (*subst (asm) check\_deadlock\_dbm\_correct'[symmetric]*)  
**subgoal**  
**using** *reachable\_states that(2) states'\_states* **by** *auto*  
**subgoal**  
**unfolding** *wf\_state\_def* **using** *E\_op''.reachable\_wf\_dbm[OF that(2)]* **by** *simp*  
**using** *canonical\_check\_diag\_empty\_iff* **by** (*blast dest: not\_check\_deadlock\_non\_empty*)

**lemma** *norm\_final\_bisim*:  
*Bisimulation\_Invariant*  
 $(\lambda(l, D)\ (l', D'). \exists a. step\_z\_norm'' (conv\_A\ A)\ l\ D\ a\ l'\ D')$   
 $E\_op''.E\_from\_op$   
 $(\lambda(l, M)\ (l', D'). l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = [M]_{v,n})$   
 $(\lambda(l, y). valid\_dbm\ y)\ wf\_state$   
**by** (*rule*  
*Bisimulation\_Invariant\_sim\_replace[OF*  
*Bisimulation\_Invariant\_composition[OF*  
 $step\_z\_norm''\_step\_impl'\_equiv[unfolded\ step\_impl'\_E]\ E\_op''.E\_from\_op\_bisim$   
 $]$   
 $])$

(auto simp add: state\_equiv\_def dbm\_equiv\_def)

**interpretation** bisim:

*Bisimulation\_Invariant*

$\lambda(l, Z) (l', Z'). \text{step\_z\_beta'} (conv\_A A) l Z l' Z'$

$\lambda a b. \text{op.E\_from\_op } a b$

$\lambda(l, Z) (l', D'). l' = l \wedge [\text{curry } (conv\_M D')]_{v,n} = Z$

$\lambda_. \text{True } \lambda(l, M). l \in \text{states} \wedge \text{wf\_state } (l, M)$

**unfolding** op\_E\_from\_op\_iff

**by** (rule Bisimulation\_Invariant\_strengthen\_post',

(rule Bisimulation\_Invariant\_composition[OF TA.bisim norm\_final\_bisim]

Bisimulation\_Invariant\_sim\_replace

)+) (auto 4 3 dest: E\_op''\_states wf\_dbm\_D(3) simp: wf\_state\_def)

**lemmas** beta\_final\_bisim = bisim.Bisimulation\_Invariant\_axioms

**definition**

is\_start\_in\_states = (trans\_fun l<sub>0</sub> ≠ [])

**lemma** is\_start\_in\_states:

$l_0 \in \text{Simulation\_Graphs\_TA.state\_set } A$  **if** is\_start\_in\_states

**proof** –

**from** that **obtain**  $g \ a \ r \ l'$  **where**  $(g, a, r, l') \in \text{set } (\text{trans\_fun } l_0)$

**by** (cases hd (trans\_fun l<sub>0</sub>) rule: prod\_cases4)

(auto dest: hd\_in\_set simp: is\_start\_in\_states\_def)

**from** trans\_impl\_trans\_of[OF this] states'\_states **have**  $A \vdash l_0 \longrightarrow^{g,a,r} l'$

**by** simp

**then show** ?thesis

**unfolding** Simulation\_Graphs\_TA.state\_set\_def trans\_of\_def **by** auto

**qed**

**lemma** deadlocked\_if\_not\_is\_start\_in\_states:

deadlocked (l<sub>0</sub>, Z<sub>0</sub>) **if**  $\neg \text{is\_start\_in\_states}$

**proof** –

**have** \*: False **if**  $A \vdash l_0 \longrightarrow^{g,a,r} l'$  **for**  $g \ a \ r \ l'$

**using** trans\_of\_trans\_impl[OF that]  $\langle \neg \rangle$  states'\_states **unfolding** is\_start\_in\_states\_def

**by** auto

{ **fix**  $l \ g2 \ a2 \ r2 \ l'$  **assume**  $A: conv\_A A \vdash l \longrightarrow^{g2,a2,r2} l'$

**obtain**  $g1 \ a1 \ r1$  **where** \*:  $A \vdash l \longrightarrow^{g1,a1,r1} l'$

**using** A **unfolding** trans\_of\_def conv\_A\_def **by** (cases A) force

**then have**  $\exists \ g1 \ a1 \ r1. A \vdash l \longrightarrow^{g1,a1,r1} l'$

**by** auto

} **note** \*\* = this

**show** ?thesis

**unfolding** deadlocked\_def

**apply** (rule ccontr)

**apply** clarsimp

**apply** (cases rule: step'.cases, assumption)

**apply** (cases rule: step\_a.cases, assumption)

**apply** (auto 4 3 elim: \* dest: \*\* step\_delay\_loc)

**done**

**qed**

**lemma** deadlock\_if\_not\_is\_start\_in\_states:

```

deadlock (l0, Z0) if ¬ is_start_in_states
unfolding deadlock_def using deadlocked_if_not_is_start_in_states[OF that] by blast

sepref_definition is_start_in_states_impl is
  uncurry0 (RETURN is_start_in_states) :: unit_assnk →a bool_assn
  unfolding is_start_in_states_def by sepref

Turning this into a Hoare triple:

thm is_start_in_states_impl.refine[to_hnr, unfolded hn_refine_def, simplified]
lemma is_start_in_states_impl_hoare:
  <emp> is_start_in_states_impl
  <λr. ↑ ((r → l0 ∈ Simulation_Graphs_TA.state_set A)
    ∧ (¬r → deadlocked (l0, λ_. 0)))
  >t
  by (sep_auto
    intro: deadlocked_if_not_is_start_in_states
    simp: mod_star_conv is_start_in_states[simplified]
    heap: is_start_in_states_impl.refine[to_hnr, unfolded hn_refine_def, simplified]
  )

lemma deadlock_if_deadlocked:
  deadlock y if deadlocked y
  using that unfolding deadlock_def by (inst_existentials y) auto

lemma is_start_in_states_impl_hoare':
  <emp> is_start_in_states_impl
  <λr. ↑ ((r → l0 ∈ Simulation_Graphs_TA.state_set A)
    ∧ (¬r → (∃ u0. (∀ c ∈ {1..n}. u0 c = 0) ∧ deadlock (l0, u0))))
  >t
  by (rule cons_post_rule[OF is_start_in_states_impl_hoare]) (auto intro: deadlock_if_deadlocked)

end

context Reachability_Problem_Impl
begin

context
  assumes l0 in A: l0 ∈ Simulation_Graphs_TA.state_set A
begin

interpretation TA:
  Regions_TA_Start_State v n Suc n {1..<Suc n} k conv_A A l0 {u. ∀ c ∈ {1..n}. u c = 0}
  apply standard
  subgoal
    by (intro l0_state_set l0_in_A)
  subgoal
    unfolding V'_def
    apply safe
    subgoal for x
      unfolding V_def by auto
    apply (inst_existentials curry init_dbm :: real DBM)
    apply (simp add: init_dbm_zone)
    by (rule dbm_int_init_dbm)
  subgoal

```

by auto  
done

**interpretation** *bisim*:

*Bisimulation\_Invariant*  
 $\lambda(l, Z) (l', Z'). \text{step\_z\_beta}' (\text{conv\_A } A) l Z l' Z'$   
 $\lambda a b. \text{op.E\_from\_op } a b$   
 $\lambda(l, Z) (l', D'). l' = l \wedge [\text{curry } (\text{conv\_M } D')]_{v,n} = Z$   
 $\lambda_. \text{True } \lambda(l, M). l \in \text{states} \wedge \text{wf\_state } (l, M)$   
 by (rule *beta\_final\_bisim*)

**lemma** *check\_deadlock*:

$(\exists a. E\_op''.E\_from\_op^{**} a_0 a \wedge$   
 $\neg (\text{case } a \text{ of } (l, M) \Rightarrow \text{check\_diag } n M) \wedge (\text{case } a \text{ of } (l, M) \Rightarrow \neg \text{check\_deadlock\_dbm } l M))$   
 $\longleftrightarrow (\exists u_0. (\forall c \in \{1..n\}. u_0 c = 0) \wedge \text{deadlock } (l_0, u_0))$  (is ?l  $\longleftrightarrow$  ?r)

**proof** –

*TA.reaches* corresponds to non-empty steps of *step\_z\_beta'*

*bisim.A.reaches* corresponds to steps of *step\_z\_beta'*

*E\_op''.reaches* *a<sub>0</sub>* corresponds to steps of *op.E\_from\_op* (*E\_op''*)

**have** ?r  $\longleftrightarrow (\exists l Z. \text{TA.reaches } (l_0, \{u. \forall c \in \{1..n\}. u c = 0\}) (l, Z) \wedge \neg \text{TA.check\_deadlock } l Z)$

**using** *TA.deadlock\_check unfolding TA.a<sub>0</sub>\_def from\_R\_def* **by** *simp*

**also have**

$\dots \longleftrightarrow (\exists l Z. \text{bisim.A.reaches } (l_0, \{u. \forall c \in \{1..n\}. u c = 0\}) (l, Z) \wedge \neg \text{TA.check\_deadlock } l Z)$

**apply** *safe*

**subgoal for** *l Z*

**by** (*subst (asm) TA.reaches\_steps\_z\_beta\_iff*) *auto*

**subgoal for** *l Z*

**by** (*subst TA.reaches\_steps\_z\_beta\_iff*) (*auto dest: not\_check\_deadlock\_non\_empty*)

**done**

**also have**  $\dots \longleftrightarrow (\exists l M. E\_op''.E\_from\_op^{**} a_0 (l, M) \wedge \neg \text{check\_deadlock\_dbm } l M)$

**using** *bisim.reaches\_ex\_iff* **[where**

$\varphi = \lambda (l, Z). \neg \text{TA.check\_deadlock } l Z$  **and**  $\psi = \lambda(l, M). \neg \text{check\_deadlock\_dbm } l M,$   
 $OF \text{not\_check\_deadlock\_compatible, of } (l_0, \{u. \forall c \in \{1..n\}. u c = 0\}) a_0$

**]**

**using** *wf\_state\_init init\_dbm\_zone states'\_states* **unfolding** *a<sub>0</sub>\_def* **by** *force*

**also have**  $\dots \longleftrightarrow ?l$

**by** (*auto 4 4 dest: deadlock\_check\_diag*)

**finally show** *?thesis ..*

**qed**

**lemma** *check\_deadlock'*:

$(\nexists a. E\_op''.E\_from\_op^{**} a_0 a \wedge$   
 $\neg (\text{case } a \text{ of } (l, M) \Rightarrow \text{check\_diag } n M) \wedge (\text{case } a \text{ of } (l, M) \Rightarrow \neg \text{check\_deadlock\_dbm } l M))$   
 $\longleftrightarrow (\forall u_0. (\forall c \in \{1..n\}. u_0 c = 0) \longrightarrow \neg \text{deadlock } (l_0, u_0))$  (is ?l  $\longleftrightarrow$  ?r)  
**using** *check\_deadlock* **by** *auto*

**context**

**assumes** *F = (λ \_ . False)*

**begin**

```

interpretation Worklist_Map2_Impl_check
  op.E_from_op a0 F_rel subsumes n succs λ (l, M). check_diag n M subsumes'
  λ (l, M). F l state_assn'
  succs_impl a0_impl F_impl subsumes_impl emptiness_check_impl fst return o fst state_copy_impl
  tracei location_assn λ(l, M). ¬ check_deadlock_dbm l M check_deadlock_neg_impl
apply standard
subgoal
  using check_deadlock_neg_impl.refine unfolding PR_CONST_def .
subgoal
  unfolding F_rel_def unfolding ⟨F = _⟩ by auto
subgoal for a b
  apply (clarsimp simp: E_op''.reachable_def)
  apply (subst (asm) check_deadlock_dbm_correct''[symmetric], assumption)
  apply (subst (asm) check_deadlock_dbm_correct''[symmetric], assumption)
  apply (frule subsumes_loc_eqD, assumption)
  apply (drule subsumes_dbm_subsetD, assumption)
  apply (drule dbm_subset_conv)
  apply (subst (asm) dbm_subset_correct''[symmetric])
  by (auto dest: TA.check_deadlock_anti_mono E_op''.reachable_wf_dbm simp: E_op''.reachable_def)
done

lemmas check_passed_impl_hnr =
  check_passed_impl_hnr[unfolded op.reachable_def, unfolded op_E_from_op_iff check_deadlock]

end

end

definition
  check_passed_impl_start ≡ do {
    r1 ← is_start_in_states_impl;
    if r1 then do {
      r2 ← check_passed_impl succs_impl a0_impl F_impl subsumes_impl emptiness_check_impl
        (return o fst) state_copy_impl tracei check_deadlock_neg_impl;
      return r2
    }
    else return True
  }

lemma check_passed_impl_start_hnr:
  (uncurry0 check_passed_impl_start,
   uncurry0 (RETURN (∃ u0. (∀ c ∈ {1..n}. u0 c = 0) ∧ deadlock (l0, u0)))
  ) ∈ unit_assnk →a bool_assn if F = (λ_. False)

proof –
  define has_deadlock where has_deadlock ≡ ∃ u0. (∀ c ∈ {1..n}. u0 c = 0) ∧ deadlock (l0, u0)
  note [sep_heap_rules] =
    check_passed_impl_hnr[
      OF _ that, to_hnr, unfolded hn_refine_def, folded has_deadlock_def, simplified
    ]
    is_start_in_states_impl_hoare'[folded has_deadlock_def, simplified]
  show ?thesis
  unfolding has_deadlock_def[symmetric] check_passed_impl_start_def
  by sepref_to_hoare (sep_auto simp: mod_star_conv)

```

qed

end

**context** *Reachability\_Problem\_precompiled*  
**begin**

**lemma** *F\_is\_False\_iff*:  
   $PR\_CONST\ F = (\lambda\_ . False) \longleftrightarrow final = []$   
  **unfolding** *F\_def* **by** (*cases final; simp; metis*)

**lemma** *F\_impl\_False*:  $F\_impl = (\lambda\_ . return\ False)$  **if**  $final = []$   
  **using** *that* **unfolding** *F\_impl\_def* **unfolding** *final\_fun\_def* **by** *auto*

**definition** *deadlock\_checker* **where**

*deadlock\_checker*  $\equiv$   
  let  
    key = return  $\circ$  fst;  
    sub = subsumes\_impl;  
    copy = state\_copy\_impl;  
    start = a0\_impl;  
    final = ( $\lambda\_ . return\ False$ );  
    succs = succs\_impl;  
    empty = emptiness\_check\_impl;  
    P = check\_deadlock\_neg\_impl;  
    trace = tracei  
  in do {  
    r1  $\leftarrow$  is\_start\_in\_states\_impl;  
    if r1 then do {  
      r2  $\leftarrow$  check\_passed\_impl succs start final sub empty key copy trace P;  
      return r2  
    }  
    else return True  
  }

**theorem** *deadlock\_checker\_hnr*:

**assumes**  $final = []$   
  **shows**  
     $(uncurry0\ deadlock\_checker, uncurry0\ (RETURN\ (\exists u_0. (\forall c \in \{1..m\}. u_0\ c = 0) \wedge deadlock\ (0, u_0))))$   
     $\in unit\_assn^k \rightarrow_a bool\_assn$   
  **unfolding** *deadlock\_checker\_def* *Let\_def*  
  **using** *check\_passed\_impl\_start\_hnr* [  
    *unfolded F\_is\_False\_iff F\_impl\_False* [*OF assms*] *check\_passed\_impl\_start\_def*,  
    *OF assms*] .

**schematic\_goal** *deadlock\_checker\_alt\_def*:

$deadlock\_checker \equiv ?impl$   
  **unfolding** *deadlock\_checker\_def*  
  **unfolding** *succs\_impl\_def*  
  **unfolding** *E\_op''\_impl\_def* *abstr\_repair\_impl\_def* *abstra\_repair\_impl\_def*  
  **unfolding** *start\_inv\_check\_impl\_def* *unbounded\_dbm\_impl\_def* *unbounded\_dbm'\_def*  
  **unfolding** *check\_deadlock\_neg\_impl\_def* *check\_deadlock\_impl\_def*

```

unfolding is_start_in_states_impl_def
apply (abstract_let IArray.sub (IArray (map (IArray o map int) k)) k)
apply (abstract_let inv_fun inv_fun)
apply (abstract_let trans_impl trans)
unfolding inv_fun_def[abs_def] trans_impl_def[abs_def]
apply (abstract_let IArray inv inv_ia)
apply (abstract_let IArray trans_map trans_map)
unfolding trans_map_def label_def
unfolding init_dbm_impl_def a0_impl_def
unfolding subsumes_impl_def
unfolding emptiness_check_impl_def
unfolding state_copy_impl_def
by (rule Pure.reflexive)

end

concrete_definition deadlock_checker_impl
uses Reachability_Problem_precompiled.deadlock_checker_alt_def

export_code deadlock_checker_impl in SML_imp module_name TA

definition [code]:
  check_deadlock n m k I T  $\equiv$ 
    if Reachability_Problem_precompiled n m k I T
    then deadlock_checker_impl m k I T  $\gg=$  ( $\lambda x.$  return (Some x))
    else return None

theorem deadlock_check:
  (uncurry0 (check_deadlock n m k I T),
   uncurry0 (
     RETURN (
       if (Reachability_Problem_precompiled n m k I T)
       then Some (
         if
            $\exists u_0. (\forall c \in \{1..m\}. u_0\ c = 0) \wedge$ 
           Graph_Defs.deadlock
           ( $\lambda(l, u) (l', u').$ 
             (conv_A (Reachability_Problem_precompiled_defs.A n I T))  $\vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
           (0, u0)
         then True
         else False
       )
       else None
     )
   )
  )  $\in$  unit_assnk  $\rightarrow_a$  id_assn

proof –
define reach_check where
  reach_check =
    ( $\exists u_0. (\forall c \in \{1..m\}. u_0\ c = 0) \wedge$ 
     Graph_Defs.deadlock
     ( $\lambda(l, u) (l', u').$ 
       (conv_A (Reachability_Problem_precompiled_defs.A n I T))  $\vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
     (0, u0))

```

```

note [sep_heap_rules] = Reachability_Problem_precompiled.deadlock_checker_hnr[OF HOL.refl,
  to_hnr, unfolded hn_refine_def, rule_format,
  of n m k I T,
  unfolded reach_check_def[symmetric]
]
show ?thesis
  unfolding reach_check_def[symmetric]
  by sepref_to_hoare
    (sep_auto simp: deadlock_checker_impl.refine[symmetric] mod_star_conv check_deadlock_def)
qed

end
theory Deadlock_Checking
  imports Deadlock_Impl Munta_Model_Checker.UPPAAL_Model_Checking
begin

```

```

Standard Deadlock Checker Implementation context Reachability_Problem_Impl
begin

```

```

definition deadlock_checker where
  deadlock_checker  $\equiv$ 
    let
      key = return  $\circ$  fst;
      sub = subsumes_impl;
      copy = state_copy_impl;
      start = a0_impl;
      final = ( $\lambda$ _. return False);
      succs = succs_impl;
      empty = emptiness_check_impl;
      P = check_deadlock_neg_impl;
      trace = tracei
    in do {
      r1  $\leftarrow$  is_start_in_states_impl;
      if r1 then do {
        r2  $\leftarrow$  check_passed_impl succs start final sub empty key copy trace P;
        return r2
      }
      else return True
    }

```

```

interpretation ta_bisim: Bisimulation_Invariant

```

```

  ( $\lambda$ (l, u) (l', u'). conv_A A  $\vdash'$   $\langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
  ( $\lambda$ (l, u) (l', u'). conv_A A  $\vdash'$   $\langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
  ( $\lambda$ (l, u) (l', u'). l' = l  $\wedge$  ( $\forall$  c. c  $\in$  clk_set (conv_A A)  $\longrightarrow$  u c = u' c))
  ( $\lambda$ _. True) ( $\lambda$ _. True)
  by (rule ta_bisimulation[of conv_A A])

```

```

lemma deadlock_zero_clock_val_iff:

```

```

  ( $\exists$  u0. ( $\forall$  c  $\in$  {1..n}. u0 c = 0)  $\wedge$  deadlock (l0, u0))  $\longleftrightarrow$  deadlock (l0,  $\lambda$ _. 0)

```

```

apply safe

```

```

subgoal for u

```

```

  using clocks_I[of  $\lambda$ _. 0 u] by (subst ta_bisim.deadlock_iff; simp)

```

```

by auto

context
  assumes F_fun_False:  $\bigwedge x. F\_fun\ x = False$  and F_False:  $F = (\lambda_. False)$ 
begin
lemma F_impl_is_False:
  F_impl = ( $\lambda_. return\ False$ )
  unfolding F_impl_def F_fun_False by simp

lemma deadlock_checker_correct:
  (uncurry0 deadlock_checker, uncurry0 (Refine_Basic.RETURN (deadlock (l0,  $\lambda_. 0$ ))))
   $\in unit\_assn^k \rightarrow_a bool\_assn$ 
  unfolding
    deadlock_checker_def Let_def F_impl_is_False[symmetric]
    check_passed_impl_start_def[symmetric] deadlock_zero_clock_val_iff[symmetric]
  using check_passed_impl_start_hnr[OF F_False] .

lemmas deadlock_checker_hoare = deadlock_checker_correct[to_hnr, unfolded hn_refine_def,
simplified]

end

end

context UPPAAL_Reachability_Problem_precompiled'
begin

  • equiv.defs.states'

  • EA is EA, the state automaton constructed from UPPAAL network (equiv/N)

  • A is A, the product automaton constructed from UPPAAL network (equiv/N)

context
  fixes  $\varphi$ 
  assumes formula_is_false: formula = formula.EX (and  $\varphi$  (not  $\varphi$ ))
begin

lemma F_is_FalseI:
  shows PR_CONST ( $\lambda(x, y). F\ x\ y$ ) = ( $\lambda_. False$ )
  unfolding F_def by (subst formula_is_false) auto

lemma final_fun_is_False:
  final_fun a = False
  unfolding final_fun_def by (subst formula_is_false) auto

lemmas deadlock_checker_hoare = impl.deadlock_checker_hoare[
  OF final_fun_is_False F_is_FalseI, folded deadlock_start_iff has_deadlock_def]

end

schematic_goal deadlock_checker_alt_def:
  impl.deadlock_checker  $\equiv ?impl$ 

```

```

unfolding impl.deadlock_checker_def impl.succs_impl_def
unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
unfolding k_impl_alt_def
unfolding impl.check_deadlock_neg_impl_def impl.check_deadlock_impl_def
unfolding impl.is_start_in_states_impl_def
apply (abstract_let k_i ceiling)
apply (abstract_let inv_fun :: ( $\_ \times \text{int list} \Rightarrow \_$ ) inv)
apply (abstract_let trans_fun trans)
unfolding impl.init_dbm_impl_def impl.a0_impl_def
unfolding impl.F_impl_def
unfolding impl.subsumes_impl_def
unfolding impl.emptiness_check_impl_def
unfolding impl.state_copy_impl_def
by (rule Pure.reflexive)

```

**schematic\_goal** deadlock\_checker\_alt\_def\_refined:

```

impl.deadlock_checker  $\equiv$  ?impl
unfolding deadlock_checker_alt_def
unfolding fw_impl'_int
unfolding inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def
unfolding trans_i_from_impl
unfolding runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl
apply (abstract_let IArray (map IArray inv) inv_array)
apply (abstract_let IArray (map IArray trans_out_map) trans_out)
apply (abstract_let IArray (map IArray trans_in_map) trans_in)
apply (abstract_let IArray (map IArray trans_i_map) trans_internal)
apply (abstract_let IArray bounds bounds_array)
apply (abstract_let PF PF)
apply (abstract_let PT PT)
unfolding PF_alt_def PT_alt_def
apply (abstract_let PROG' PROG')
unfolding PROG'_def
apply (abstract_let length prog len_prog)
apply (abstract_let IArray (map (map_option stripf) prog) prog_f)
apply (abstract_let IArray (map (map_option stript) prog) prog_t)
apply (abstract_let IArray prog prog_array)
unfolding all_actions_by_state_impl
apply (abstract_let [ $0..<p$ ] p_ran)
apply (abstract_let [ $0..<na$ ] num_actions_ran)
apply (abstract_let [ $0..<p$ ] num_processes_ran)
apply (abstract_let [ $0..<m+1$ ] num_clocks_ran)
by (rule Pure.reflexive)

```

**end**

**concrete\_definition** deadlock\_checker uses

UPPAAL\_Reachability\_Problem\_precompiled'.deadlock\_checker\_alt\_def\_refined

**definition**

```

precond_dc
  num_processes num_clocks clock_ceiling max_steps I T prog bounds program s0 num_actions
 $\equiv$ 
  if UPPAAL_Reachability_Problem_precompiled'

```

```

num_processes num_clocks max_steps I T prog bounds program s0 num_actions clock_ceiling
then
  deadlock_checker
num_processes num_clocks max_steps I T prog bounds program s0 num_actions clock_ceiling
  ≫ (λx. return (Some x))
else return None

theorem deadlock_check:
  <emp>
    precondition dc p m k max_steps I T prog bounds P s0 na
  <λ Some r ⇒ ↑(
    UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s0 na k ∧
    r = has_deadlock (conv (N p I P T prog bounds)) max_steps (repeat 0 p, s0, λ_. 0))
  | None ⇒ ↑(¬ UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds
    P s0 na k)
  >t
proof –
  define A where A ≡ conv (N p I P T prog bounds)
  note [sep_heap_rules] = UPPAAL_Reachability_Problem_precompiled'.deadlock_checker_hoare[
    OF HOL.refl,
    of p m max_steps I T prog bounds P s0 na k,
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    folded A_def
  ]
  show ?thesis
  unfolding A_def[symmetric] precondition_dc_def
  by (sep_auto simp:
    deadlock_checker.refine[symmetric] UPPAAL_Reachability_Problem_precompiled_defs.init_def
    mod_star_conv has_deadlock_def)
qed

export_code precondition_dc checking SML

end

```

## 11 Renaming Identifiers in Simple Networks

```

theory Simple_Network_Language_Renaming
imports Simple_Network_Language_Model_Checking
begin

```

The part justifies a “renaming” step to normalize identifiers in the input.

```

unbundle no_library_syntax
notation (input) TAG ( _ [] _ [40, 40] 41 )

```

Helpful methods and theorems to work with tags:

```

lemmas TAG_cong = arg_cong[where f = TAG t for t]

```

```

lemma TAG_mp:
  assumes TAG t x x ⇒ y
  shows TAG t y
  using assms unfolding TAG_def by blast

```

```

lemma TAG_mp':
  assumes TAG t x x  $\implies$  y
  shows y
  using assms unfolding TAG_def by blast

lemmas all_mono_rule = all_mono[THEN mp, OF impI, rotated]

lemma imp_mono_rule:
  assumes P1  $\longrightarrow$  P2
  and Q1  $\implies$  P1
  and Q1  $\implies$  P2  $\implies$  Q2
  shows Q1  $\longrightarrow$  Q2
  using assms by blast

lemma Ball_mono:
  assumes  $\forall x \in S. P\ x \wedge x \in S \implies P\ x \implies Q\ x$ 
  shows  $\forall x \in S. Q\ x$ 
  using assms by blast

method prop_monos =
  erule all_mono_rule
| erule (1) imp_mono_rule
| erule disj_mono[rule_format, rotated 2]

locale Prod_TA_Defs_urge =
  fixes A :: ('a, 's, 'c, 't :: {zero}, 'x, 'v :: linorder) nta and urge :: 'c
begin

definition
  add_reset  $\equiv \lambda(C, U, T, I).$ 
    (C, {}, ( $\lambda(l, b, g, a, f, r, l'). (l, b, g, a, f, urge \# r, l')$ ) ' T, I)

definition
  add_inv  $\equiv \lambda(C, U, T, I).$  (C, U, T,
    ( $\lambda l. \text{if } l \in U \text{ then } \text{acconstraint.LE } urge\ 0 \# I\ l \text{ else } I\ l$ ))

definition A_urge  $\equiv$ 
  (fst A, map add_reset (map add_inv (fst (snd A))), snd (snd A))

definition
  N_urge i  $\equiv \text{map add_reset (map add_inv (fst (snd A))) ! } i$ 

end

locale Prod_TA_sem_urge =
  Prod_TA_Defs_urge A urge + Prod_TA_sem A
  for A :: ('a, 's, 'c, 't :: {zero, time}, 'x, 'v :: linorder) nta and urge :: 'c +
  assumes urge_not_in_invariants:
     $\forall(\_, \_, \_, I) \in \text{set (fst (snd A)). } \forall l. urge \notin \text{constraint\_clk ' set (I } l)$ 
  assumes urge_not_in_guards:
     $\forall(\_, \_, T, \_) \in \text{set (fst (snd A)). } \forall(l, b, g, a, f, r, l') \in T. urge \notin \text{constraint\_clk ' set } g$ 
  assumes urge_not_in_resets:

```

$\forall (\_, \_, T, \_) \in \text{set } (\text{fst } (\text{snd } A)). \forall (l, b, g, a, f, r, l') \in T. \text{urge} \notin \text{set } r$   
**begin**

**lemma** *N\_urge\_simp*:

*N\_urge i = add\_reset (add\_inv (N i)) if i < n\_ps*  
**using** *that* **unfolding** *N\_urge\_def N\_def* **by** (*simp add: n\_ps\_def*)

**lemma** *inv\_add\_reset*:

*inv (add\_reset A') = inv A'*  
**unfolding** *inv\_def add\_reset\_def* **by** (*simp split: prod.splits*)

**lemma** *inv\_add\_inv*:

*inv (add\_inv (C, U, T, I)) = ( $\lambda l. \text{if } l \in U \text{ then } \text{acconstraint.LE } \text{urge } 0 \# I \text{ l else } I \text{ l}$ )*  
**unfolding** *add\_inv\_def inv\_def* **by** *simp*

**definition**

*is\_urgent L  $\equiv \exists p < n\_ps. L ! p \in \text{urgent } (N p)$*

**lemma** *map\_nth'*:

*map (!) xs [0..<n] = xs if n = length xs*  
**using** *that* *List.map\_nth* **by** *simp*

**lemma** *A\_urge\_split*:

*A\_urge = (broadcast, map N\_urge [0..<n\_ps], bounds)*  
**unfolding** *broadcast\_def N\_urge\_def bounds\_def n\_ps\_def A\_urge\_def* **by** (*cases A*)(*simp add: map\_nth'*)

**lemma** *inv\_add\_inv'*:

*inv (add\_inv A') =*  
*( $\lambda l. \text{if } l \in \text{urgent } A' \text{ then } \text{acconstraint.LE } \text{urge } 0 \# \text{inv } A' \text{ l else } \text{inv } A' \text{ l}$ )*  
**apply** (*cases A'*)  
**apply** (*simp add: inv\_add\_inv\_urgent\_def*)  
**unfolding** *inv\_def*  
**apply** *auto*  
**done**

**lemma** *A\_split'*:

*A = (fst A, fst (snd A), snd (snd A))*  
**by** *auto*

**lemma** *is\_urgent\_simp*:

*( $\exists p < n\_ps. L ! p \in \text{urgent } (\text{map } N [0..<n\_ps] ! p)$ )  $\longleftrightarrow$  is\_urgent L*  
**unfolding** *is\_urgent\_def* **by** *auto*

**lemma** *urgent\_N\_urge*:

*urgent (N\_urge p) = {} if p < n\_ps*  
**using** *that* **unfolding** *N\_def N\_urge\_def n\_ps\_def urgent\_def add\_reset\_def add\_inv\_def*  
**by** (*auto split: prod.split\_asm*)

**lemma** *committed\_N\_urge*:

*committed (N\_urge p) = committed (N p) if p < n\_ps*  
**using** *that* **unfolding** *N\_def N\_urge\_def n\_ps\_def committed\_def add\_reset\_def add\_inv\_def*  
**by** (*auto split: prod.split*)

**lemma** *is\_urgent\_simp2*:

$(\exists p < n\_ps. L ! p \in \text{urgent} (\text{map } N\_urge [0..<n\_ps] ! p)) \longleftrightarrow \text{False}$   
**unfolding** *is\_urgent\_def* **by** (*auto simp: urgent\_N\_urge*)

**lemma** *inv\_add\_invI1*:

$u \vdash \text{inv} (\text{add\_inv} (N p)) (L ! p)$  **if**  $u \vdash \text{inv} (N p) (L ! p) \neg \text{is\_urgent } L p < n\_ps$   
**using** *that* **unfolding** *inv\_add\_inv' is\_urgent\_def* **by** *simp*

**lemma** *inv\_add\_invI2*:

$u \vdash \text{inv} (\text{add\_inv} (N p)) (L ! p)$  **if**  $u \vdash \text{inv} (N p) (L ! p) u\_urge \leq 0 p < n\_ps$   
**using** *that* **unfolding** *inv\_add\_inv' is\_urgent\_def* **by** (*auto elim: guard\_consI[rotated]*)

**lemma** *inv\_add\_invD*:

$u \vdash \text{inv } A' l$  **if**  $u \vdash \text{inv} (\text{add\_inv } A') l$   
**using** *that* **unfolding** *inv\_add\_inv'* **by** (*auto split: if\_split\_asm simp: clock\_val\_def*)

**lemma** *inv\_N\_urge*:

$\text{inv} (N\_urge p) = \text{inv} (\text{add\_inv} (N p))$  **if**  $p < n\_ps$   
**using** *that* **by** (*simp add: N\_urge\_simp inv\_add\_reset*)

**lemma** *N\_urge\_trans\_simp*:

$\text{trans} (N\_urge i) = (\lambda(l, b, g, a, f, r, l'). (l, b, g, a, f, \text{urge} \# r, l')) \text{ ' } (\text{trans} (N i))$   
**if**  $i < n\_ps$   
**unfolding** *N\_urge\_simp[OF that] add\_inv\_def add\_reset\_def trans\_def* **by** (*simp split: prod.splits*)

**lemma** *trans\_N\_urgeD*:

$(l, b, g, a, f, \text{urge} \# r, l') \in \text{trans} (N\_urge p)$   
**if**  $(l, b, g, a, f, r, l') \in \text{trans} (N p) p < n\_ps$   
**using** *that* **by** (*force simp add: N\_urge\_trans\_simp*)

**lemma** *trans\_N\_urgeE*:

**assumes**  $(l, b, g, a, f, r, l') \in \text{trans} (N\_urge p) p < n\_ps$   
**obtains**  $(l, b, g, a, f, \text{tl } r, l') \in \text{trans} (N p) r = \text{urge} \# \text{tl } r$   
**using** *assms* **by** (*force simp add: N\_urge\_trans\_simp*)

**lemma** *urge\_not\_in\_inv*:

$\text{urge} \notin \text{constraint\_clk} \text{ ' set } (\text{inv} (N p) l)$  **if**  $p < n\_ps$   
**using** *urge\_not\_in\_invariants that* **unfolding** *N\_def inv\_def n\_ps\_def* **by** (*fastforce dest! :nth\_mem*)

**lemma** *urge\_not\_in\_guards'*:

**assumes**  $(l, b, g, a, f, r, l') \in \text{trans} (N p) p < n\_ps$   
**shows**  $\text{urge} \notin \text{constraint\_clk} \text{ ' set } g$   
**using** *urge\_not\_in\_guards assms* **unfolding** *N\_def trans\_def n\_ps\_def* **by** (*fastforce dest! :nth\_mem*)

**lemma** *urge\_not\_in\_resets'*:

**assumes**  $(l, b, g, a, f, r, l') \in \text{trans} (N p) p < n\_ps$   
**shows**  $\text{urge} \notin \text{set } r$   
**using** *urge\_not\_in\_resets assms* **unfolding** *N\_def trans\_def n\_ps\_def* **by** (*fastforce dest! :nth\_mem*)

**lemma** *clk\_upd\_clock\_val\_a\_simp*:

$u(c := d) \vdash_a ac \longleftrightarrow u \vdash_a ac$  **if**  $c \neq \text{constraint\_clk } ac$

using that by (cases ac) auto

**lemma** *clk\_upd\_clock\_val\_simp*:  
 $u(c := d) \vdash cc \longleftrightarrow u \vdash cc$  **if**  $c \notin \text{constraint\_clk}$  ‘ set cc  
 using that **unfolding** *clock\_val\_def list\_all\_def*  
 by (force *simp: image\_def clk\_upd\_clock\_val\_a\_simp*)

**lemma** *inv\_N\_urgeI*:  
 assumes  $u \vdash \text{inv } (N\ p) \ l \ p < n\_ps$   
 shows  $u(\text{urge} := 0) \vdash \text{inv } (N\_urge\ p) \ l$   
 using *assms urge\_not\_in\_inv[of p]*  
 by (auto *simp: inv\_N\_urge inv\_add\_inv' clk\_upd\_clock\_val\_simp intro!: guard\_consI*)

**lemma** *inv\_N\_urge\_updI*:  
 assumes  $u \vdash \text{inv } (N\ p) \ l \ p < n\_ps$   
 shows  $u(\text{urge} := d) \vdash \text{inv } (N\ p) \ l$   
 using *assms urge\_not\_in\_inv[of p]* **by** (auto *simp: clk\_upd\_clock\_val\_simp*)

**lemma** *inv\_N\_urge\_upd\_simp*:  
 assumes  $p < n\_ps$   
 shows  $u(\text{urge} := d) \vdash \text{inv } (N\ p) \ l \longleftrightarrow u \vdash \text{inv } (N\ p) \ l$   
 using *assms urge\_not\_in\_inv[of p]* **by** (auto *simp: clk\_upd\_clock\_val\_simp*)

**lemma** *inv\_N\_urge\_updD*:  
 assumes  $u(\text{urge} := d) \vdash \text{inv } (N\ p) \ l \ p < n\_ps$   
 shows  $u \vdash \text{inv } (N\ p) \ l$   
 using *assms urge\_not\_in\_inv[of p]* **by** (auto *simp: clk\_upd\_clock\_val\_simp*)

**lemma** *inv\_N\_urgeD*:  
 assumes  $u \vdash \text{inv } (N\_urge\ p) \ l \ p < n\_ps$   
 shows  $u(\text{urge} := d) \vdash \text{inv } (N\ p) \ l$   
 using *assms urge\_not\_in\_inv[of p]*  
**by** (auto *simp: inv\_N\_urge inv\_add\_inv' clk\_upd\_clock\_val\_simp split: if\_split\_asm elim: guard\_consE*)

**lemma** *inv\_N\_urge\_urges*:  
 assumes  $\forall p < n\_ps. u(\text{urge} := 0) \oplus d \vdash \text{inv } (N\_urge\ p) \ (L\ !\ p) \text{ is\_urgent } L$   
 shows  $d \leq 0$

**proof** –

**from**  $\langle \text{is\_urgent } L \rangle$  **obtain**  $p$  **where**  $p < n\_ps \ L\ !\ p \in \text{urgent } (N\ p)$   
**unfolding** *is\_urgent\_def* **by** auto  
**then have** *acconstraint.LE urge 0*  $\in \text{set } (\text{inv } (N\_urge\ p) \ (L\ !\ p))$   
**by** (*simp add: inv\_N\_urge inv\_add\_inv'*)  
**with** *assms*  $\langle p < n\_ps \rangle$  **have**  $u(\text{urge} := 0) \oplus d \vdash_a \text{acconstraint.LE urge } 0$   
**unfolding** *clock\_val\_def list\_all\_iff* **by** auto  
**then show** *?thesis*  
**by** (*auto simp: cval\_add\_def*)

**qed**

**lemma** *trans\_urge\_upd\_iff*:  
 assumes  $(l, b, g, a, f, r, l') \in \text{trans } (N\ p) \ p < n\_ps$   
 shows  $u(\text{urge} := d) \vdash g \longleftrightarrow u \vdash g$   
 using *assms urge\_not\_in\_guards'* **by** (auto *simp: clk\_upd\_clock\_val\_simp*)

```

lemma cval_add_0[simp]:
   $u \oplus (0 :: 'tt :: time) = u$ 
  unfolding cval_add_def by simp

lemma clock_val_reset_delay:
   $u(c := 0) \oplus (d :: 'tt :: time) = (u \oplus d)(c := d)$ 
  unfolding cval_add_def by auto

lemma clock_set_upd_simp:
   $[r \rightarrow d]u(c := d') = [r \rightarrow d]u$  if  $c \in \text{set } r$ 
  using that
  apply (induction r arbitrary: u)
  apply auto
  oops

lemma fun_upd_twist2:
   $f(a := x, b := x) = f(b := x, a := x)$ 
  by auto

lemma clock_set_upd_simp:
   $[c \# r \rightarrow d]u(c := d') = [c \# r \rightarrow d]u$ 
  apply (induction r arbitrary: u)
  apply simp
  apply simp
  apply (subst fun_upd_twist2)
  apply auto
  done

lemma clock_set_commute_single:
   $[r \rightarrow d]u(c := d') = ([r \rightarrow d]u)(c := d')$  if  $c \notin \text{set } r$ 
  using that by (induction r) auto

lemma clock_set_upd_simp2:
   $([xs @ c \# ys \rightarrow d]u)(c := d') = ([xs @ ys \rightarrow d]u)(c := d')$ 
  apply (induction xs)
  apply (solves auto)
  apply (intro ext)
  subgoal for a xs x
    by (drule fun_cong[where  $x = x$ ]) auto
  done

lemma clock_set_upd_simp3:
   $([xs \rightarrow d]u)(c := d') = ([\text{filter } (\lambda x. x \neq c) \text{ } xs \rightarrow d]u)(c := d')$ 
  apply (induction xs)
  apply (solves auto)
  apply (rule ext)
  apply (clarsimp, safe)
  apply clarsimp
  subgoal for xs x
    by (drule fun_cong[where  $x = x$ ]) auto
  subgoal for a xs x
    by (drule fun_cong[where  $x = x$ ]) auto
  done

```

**interpretation** *urge*: *Prod\_TA\_sem A\_urge* .

**lemma** *urge\_n\_ps\_eq*:

*urge.n\_ps = n\_ps*

**unfolding** *urge.n\_ps\_def n\_ps\_def* **unfolding** *A\_urge\_def* **by** *simp*

**lemma** *urge\_N\_simp[simp]*:

*urge.N = N\_urge*

**unfolding** *urge.N\_def N\_urge\_def* **unfolding** *A\_urge\_def* **by** *simp*

**lemma** *urge\_states\_eq*:

*urge.states = states*

**unfolding** *states\_def urge.states\_def urge\_n\_ps\_eq*

**by** (*auto simp add: N\_urge\_trans\_simp split: prod.splits*)

**interpretation** *Bisimulation\_Invariant*

$\lambda(L, s, u) (L', s', u'). \text{step\_}u' A L s u L' s' u'$

$\lambda(L, s, u) (L', s', u'). \text{step\_}u' A\_urge L s u L' s' u'$

$\lambda(L, s, u) (L', s', u'). L' = L \wedge s' = s \wedge u' = u(\text{urge} := 0)$

$\lambda(L, s, u). L \in \text{states} \lambda(L, s, u). \text{True}$

**proof** –

~~*cases* (*A/N/s/u/L/s'/u'*) (*then/stop/case*)~~

**have** *A\_urge*  $\vdash \langle L, s, u(\text{urge} := 0) \rangle \rightarrow \langle L', s', u'(\text{urge} := 0) \rangle$

**if** *steps*: *A*  $\vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$  *L*  $\in \text{states}$

**for** *L* :: '*s* list

**and** *s* :: '*x*  $\Rightarrow$  '*v* option

**and** *u* :: '*c*  $\Rightarrow$  '*t*

**and** *L'* :: '*s* list

**and** *s'* :: '*x*  $\Rightarrow$  '*v* option

**and** *u'* :: '*c*  $\Rightarrow$  '*t*

**using** *that*

**proof** *cases*

**case** *steps*: (*1 L'' s'' u'' a*)

**have** \*: (*u*  $\oplus$  *d*)(*urge* := (*u*  $\oplus$  *d*) *urge* – *u* *urge*) = *u*(*urge* := 0)  $\oplus$  *d* **for** *d*

**by** (*auto simp: cval\_add\_def*)

**from** *steps*(1)  $\langle L \in \text{states} \rangle$  **have** *L''*  $\in \text{states}$

**by** (*rule states\_inv*)

**then** **have** [*simp*]: *length L''* = *n\_ps*

**by** (*rule states\_lengthD*)

**from** *steps*(1) **have**

*A\_urge*  $\vdash \langle L, s, u(\text{urge} := 0) \rangle \rightarrow_{\text{Del}} \langle L'', s'', u''(\text{urge} := u'' \text{urge} – u \text{urge}) \rangle$

**apply** *cases*

**apply** (*subst A\_urge\_split*)

**apply** (*subst (asm) A\_split*)

**apply** (*drule sym*)

**apply** *simp*

**unfolding** *TAG\_def*[*of "urgent"*]

**apply** (*simp add: is\_urgent\_simp* \*)

**apply** (*rule step\_u.intros*)

**apply** *tag*

**apply** *simp*

**subgoal**

**by** (*cases is\_urgent L*)

(*auto 4 3*)

```

    intro: inv_N_urge_updI inv_add_invI1 inv_add_invI2
    simp: inv_N_urge clock_val_reset_delay)
  by (tag, simp add: is_urgent_simp2)+
moreover from steps(3,2) have
  A_urge ⊢ ⟨L'', s'', u''(urge := u'' urge - u urge)⟩ →a ⟨L', s', u'(urge := 0)⟩
  apply (cases; subst A_urge_split; subst (asm) A_split)
    apply (all ⟨drule sym⟩)
subgoal delay
  by simp
subgoal internal premises prems[tagged] for l b g a' f r l' N p B broadcast
  using prems apply tag usingT "range"
  apply simp
  apply (rule step_u.intros)
  preferT ⟨TRANS "silent"⟩
    apply (solves ⟨tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp⟩)
  preferT ⟨"committed"⟩
    apply (solves ⟨tag, subst nth_map, simp, simp add: committed_N_urge⟩)
  preferT ⟨"bexp"⟩
    apply (tag, assumption)
  preferT ⟨"guard"⟩
    apply (solves ⟨tag "guard" TRANS __; auto simp: trans_urge_upd_iff⟩)
  preferT ⟨"new valuation"⟩
    apply (tag, unfold clock_set.simps(2)[symmetric] clock_set_upd_simp, simp; fail)
  by (tag; auto intro: inv_N_urgeI)+
subgoal binary premises prems[tagged]
  for a' broadcast' l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s'' B
  using prems apply tag
  usingT RECV "range" SEND "range"
  apply simp
  apply (rule step_u.intros)
  preferT ⟨"not broadcast"⟩
    apply (tag; simp; fail)
  preferT ⟨TRANS "in"⟩
    apply (solves ⟨tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp⟩)
  preferT ⟨TRANS "out"⟩
    apply (solves ⟨tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp⟩)
  preferT ⟨"committed"⟩
    apply (solves ⟨tag, subst nth_map, simp, simp add: committed_N_urge⟩)
  preferT ⟨"bexp"⟩
    apply (tag, assumption)
  preferT ⟨"bexp"⟩
    apply (tag, assumption)
  preferT ⟨"guard"⟩
    apply (solves ⟨tag "guard" TRANS __; auto simp: trans_urge_upd_iff⟩)
  preferT ⟨"guard"⟩
    apply (solves ⟨tag "guard" TRANS __; auto simp: trans_urge_upd_iff⟩)
  preferT ⟨"new valuation"⟩
    apply (solves ⟨tag, simp, unfold clock_set.simps(2)[symmetric] clock_set_upd_simp,
      simp add: clock_set_upd_simp2⟩)
  by (solves ⟨tag; auto intro: inv_N_urgeI; simp⟩)+
subgoal broadcast premises prems[tagged]
  for a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s'' B
  using prems apply tag
  usingT SEL "range" SEND "range"

```

```

    apply (simp add: subset_nat_0_atLeastLessThan_conv)
    apply (rule step_u.intros)
    preferT <"broadcast">
      apply (tag; simp; fail)
    preferT <TRANS "out">
      apply (solves <tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp>)
    preferT <TRANS "in">
      apply (solves <tag, auto 4 3 dest: trans_N_urgeD>)
    preferT <"committed">
      apply (solves <tag, subst nth_map, simp, simp add: committed_N_urge>)
    preferT <"bexp">
      apply (tag, assumption)
    preferT <"bexp">
      apply (tag, assumption)
    preferT <"guard">
      apply (solves <tag "guard" TRANS __; auto simp: trans_urge_upd_iff>)
    preferT <"guard">
      apply (solves <tag "guard" TRANS __; auto simp: trans_urge_upd_iff>)
    preferT <"maximal">
      apply (all <tag; auto intro: inv_N_urgeI; fail | succeed>)
    preferT <"maximal">
      apply (solves <tag; auto; erule (1) trans_N_urgeE, force simp: trans_urge_upd_iff>)
    preferT <"new valuation"> subgoal
      apply tag
      apply clarsimp
      unfolding clock_set.simps(2)[symmetric] clock_set_upd_simp
      apply simp
      apply (subst (2) clock_set_upd_simp3)
      apply (subst clock_set_upd_simp3)
      apply (simp add: filter_concat comp_def)
      done
    done
  done
ultimately show A_urge ⊢ ⟨L, s, u(urge := 0)⟩ → ⟨L', s', u'(urge := 0)⟩
  using ⟨a ≠ _⟩ by (intro step_u'.intros)
qed
next [[cases (λl u s v u' s' u')]]
moreover have ∃ ba. A ⊢ ⟨L, s, u⟩ → ⟨L', s', ba⟩ ∧ u' = ba(urge := 0)
  if A_urge ⊢ ⟨L, s, u(urge := 0)⟩ → ⟨L', s', u'⟩
  and L ∈ states
  for L :: 's list
  and s :: 'x ⇒ 'v option
  and u :: 'c ⇒ 't
  and L' :: 's list
  and s' :: 'x ⇒ 'v option
  and u' :: 'c ⇒ 't
  using that
proof cases
  case steps: (1 L'' s'' u'' a)
  have *: (u(urge := 0) ⊕ d)(urge := (u(urge := 0) ⊕ d) urge + u urge) = u ⊕ d for d
    unfolding cval_add_def by auto
  from steps(1) ⟨L ∈ states⟩ have L'' ∈ states
    by (rule urge.states_inv[unfolded urge_states_eq])
  then have [simp]: length L'' = n_ps

```

```

    by (rule states_lengthD)
have 1:  $([r \rightarrow 0]u'') \text{ urge} = 0$  if  $r = \text{urge} \# xs$  for  $r \ xs$ 
  by (subst that) simp
have 2: filter  $(\lambda x. x \neq \text{urge}) \ r = \text{filter } (\lambda x. x \neq \text{urge}) \ (tl \ r)$  if  $r = \text{urge} \# tl \ r$  for  $r$ 
  by (subst that) simp
from steps(3,2) have  $u' \text{ urge} = 0$ 
  apply (cases; subst (asm) A_urge_split)
    apply (all <drule sym>)
    apply (simp; fail)
  subgoal delay
    by (tag- "new valuation" TRANS_ "range")
      (solves <auto elim!: trans_N_urgeE simp: 1>)
  subgoal binary for  $a' \text{ broadcast}' \ l1 \ b1 \ g1 \ f1 \ r1 \ l1' \ N \ p \ l2 \ b2 \ g2 \ f2 \ r2 \ l2' \ q \ s'' \ B$ 
    by (tag- "new valuation" TRANS_ RECV "range")
      (auto elim!: trans_N_urgeE simp: 1[of_ tl r1 @ r2])
  subgoal broadcast for  $a' \text{ broadcast}' \ l \ b \ g \ f \ r \ l' \ N \ p \ ps \ bs \ gs \ fs \ rs \ ls' \ s'a \ B$ 
    by (tag- "new valuation" TRANS_ SEND "range")
      (auto elim!: trans_N_urgeE simp: 1[of_ tl r @ concat (map rs ps)])
  done
have **:
   $([r \rightarrow 0]u'')(\text{urge} := u'' \text{ urge} + u \text{ urge}) = ([tl \ r \rightarrow 0]u'')(\text{urge} := u'' \text{ urge} + u \text{ urge})$ 
  if  $r = \text{urge} \# tl \ r$  for  $r$ 
  by (subst that) simp
from steps(1) have  $A \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L'', s'', u''(\text{urge} := u'' \text{ urge} + u \text{ urge}) \rangle$ 
  apply cases
  apply (subst (asm) A_urge_split)
  apply (subst A_split)
  apply (drule sym)
  apply simp
  unfolding TAG_def[of "urgent"]
  apply (simp add: is_urgent_simp *)
  apply (rule step_u.intros)
  subgoal
  by tag (auto dest!: inv_add_invD inv_N_urge_updD simp: inv_N_urge clock_val_reset_delay)
    apply (tag, assumption)
    apply (tag- "target invariant" "nonnegative")
    apply (auto simp add: is_urgent_simp2 is_urgent_simp dest: inv_N_urge_urges)
  done
moreover from steps(3,2) have
   $A \vdash \langle L'', s'', u''(\text{urge} := u'' \text{ urge} + u \text{ urge}) \rangle \rightarrow_a \langle L', s', u'(\text{urge} := u'' \text{ urge} + u \text{ urge}) \rangle$ 
  apply (cases; subst (asm) A_urge_split; subst A_split)
    apply (all <drule sym>)
  subgoal delay
    by simp
  subgoal internal premises prems[tagged] for  $l \ b \ g \ a' \ f \ r \ l' \ N \ p \ B \text{ broadcast}$ 
    using prems apply del_tags
    usingT "range"
    apply simp
    apply (rule step_u.intros)
    preferT <TRANS "silent">
      apply (solves <tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp>)
    preferT <"committed">
      apply (solves <tag, subst nth_map, simp, simp add: committed_N_urge>)
    preferT <"bexp">

```

```

    apply (tag; assumption)
  preferT <"guard">
    apply (solves <tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff>)
  preferT <"target invariant">
    apply (all <tag; auto intro: inv_N_urgeD; fail | succeed>)
  preferT <"new valuation">
    apply (solves <tag TRANS _, simp, erule (1) trans_N_urgeE,
      subst clock_set_commute_single, rule urge_not_in_resets', (simp add: **)>+)
done
subgoal binary premises prems[tagged]
  for a' broadcast' l1 b1 g1 f1 r1 l1' Na p l2 b2 g2 f2 r2 l2' q s'' B
  using prems apply del_tags
  usingT RECV "range" SEND "range"
  apply simp
  apply (rule step_u.intros)
  preferT <"not broadcast"> apply (tag; simp; fail)
  preferT <TRANS "in">
    apply (solves <tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp>)
  preferT <TRANS "out">
    apply (solves <tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp>)
  preferT <"committed">
    apply (solves <tag, subst nth_map, simp, simp add: committed_N_urge>)
  preferT <"bexp">
    apply (tag; assumption)
  preferT <"bexp">
    apply (tag; assumption)
  preferT <"guard">
    apply (solves <tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff>)
  preferT <"guard">
    apply (solves <tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff>)
  preferT <"new valuation"> subgoal premises prems[tagged]
    using prems apply tag
    apply (tag TRANS _, simp, erule (1) trans_N_urgeE, erule (1) trans_N_urgeE)
    apply (subst clock_set_upd_simp3)
    apply (subst clock_set_commute_single)
    apply (simp; rule conjI; erule (1) urge_not_in_resets'; fail)
    apply (rule arg_cong)
    subgoal premises prems
      using urge_not_in_resets'[OF prems(7) <p < n_ps>]
      using urge_not_in_resets'[OF prems(9) <q < n_ps>]
      by (subst <r1 = _>, subst <r2 = _>, simp) (subst filter_True, auto)+
    done
  by (solves <tag; auto intro: inv_N_urgeD>)+
subgoal broadcast premises prems[tagged]
  for a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s2 B
  using prems apply del_tags
  usingT SEL "range" SEND "range"
  apply (simp add: subset_nat_0_atLeastLessThan_conv)
  apply (rule step_u.intros)
  preferT <TRANS "out">
    — instantiates schematics
    apply (solves <tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp>)
  preferT <TRANS "in">
    apply (tag, auto 4 3 elim: trans_N_urgeE; fail) — instantiates schematics

```

```

preferT <"broadcast">
  apply (tag; simp; fail)
preferT <"committed">
  apply (solves <tag, subst nth_map, simp, simp add: committed_N_urge>)
preferT <"guard">
  apply (solves <tag TRANS _; auto elim: trans_N_urgeE simp: trans_urge_upd_iff>)
preferT <"guard"> subgoal guards
  by (tag TRANS _; auto elim!: trans_N_urgeE simp: trans_urge_upd_iff)
preferT <"maximal"> subgoal maximal
  by (tag, force simp: trans_urge_upd_iff dest: trans_N_urgeD)
preferT <"new valuation"> subgoal new_valuation premises prems[tagged]
  apply (tag TRANS _)
  using prems apply tag
  apply (subst [[get_tagged <"new valuation">]])
  apply (subst clock_set_upd_simp3)
  apply (simp add: filter_concat comp_def)
  apply (subst clock_set_commute_single[symmetric], (simp; fail))
  apply (rule arg_cong)
  apply (fo_rule arg_cong2)
  subgoal
    by (auto simp: 2_urge_not_in_resets' filter_id_conv elim!: trans_N_urgeE)
  subgoal
    apply (fo_rule arg_cong)
    apply (fastforce elim: trans_N_urgeE simp: 2_urge_not_in_resets' filter_id_conv)+
    done
  done
  by (solves <tag; auto intro: inv_N_urgeD>)+
done
ultimately show ?thesis
  using <a ≠ _> <u' urge = 0> by (intros add: step_u'.intros) auto
qed
moreover have ab ∈ states
  if a ∈ states
    and A ⊢ <a, aa, b> → <ab, ac, ba>
  for a :: 's list
    and aa :: 'x ⇒ 'v option
    and b :: 'c ⇒ 't
    and ab :: 's list
    and ac :: 'x ⇒ 'v option
    and ba :: 'c ⇒ 't
  using that by (elim step_u'_elims states_inv)
ultimately show
  Bisimulation_Invariant
  (λ(L, s, u) (L', s', u').
    A ⊢ <L, s, u> → <L', s', u'>)
  (λ(L, s, u) (L', s', u').
    A_urge ⊢ <L, s, u> → <L', s', u'>)
  (λ(L, s, u) (L', s', u').
    L' = L ∧ s' = s ∧ u' = u(urge := 0))
  (λ(L, s, u). L ∈ states) (λ(L, s, u). True)
  by - (standard; clarsimp)
qed

```

**lemmas** urge\_bisim = Bisimulation\_Invariant\_axioms

end

**context** *Simple\_Network\_Impl\_Defs*  
**begin**

**lemma** *dom\_bounds*: *dom bounds = fst 'set bounds'*  
**unfolding** *bounds\_def* **by** (*simp add: dom\_map\_of\_conv\_image\_fst*)

**lemma** *mem\_trans\_N\_iff*:  
 $t \in \text{Simple\_Network\_Language.trans } (N \ i) \longleftrightarrow t \in \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! \ i))))$   
**if**  $i < n\_ps$   
**unfolding** *N\_eq*[*OF that*] **by** (*auto split: prod.splits simp: automaton\_of\_def trans\_def*)

**lemma** *length\_automata\_eq\_n\_ps*:  
 $\text{length automata} = n\_ps$   
**unfolding** *n\_ps\_def* **by** *simp*

**lemma** *N\_p\_trans\_eq*:  
 $\text{Simple\_Network\_Language.trans } (N \ p) = \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! \ p))))$  **if**  $p < n\_ps$   
**using** *mem\_trans\_N\_iff*[*OF that*] **by** *auto*

**lemma** *loc\_set\_compute*:  
 $\text{loc\_set} = \bigcup ((\lambda(\_, \_, t, \_). \bigcup ((\lambda(l, \_, \_, \_, \_, \_, l'). \{l, l'\}) \text{ 'set } t)) \text{ 'set automata})$   
**unfolding** *loc\_set\_def setcompr\_eq\_image*  
**apply** (*safe; clarsimp simp: N\_p\_trans\_eq n\_ps\_def*)  
**apply** (*drule nth\_mem, erule bexI[rotated], force*) +  
**apply** (*drule mem\_nth, force*) +  
**done**

**lemma** *var\_set\_compute*:  
 $\text{var\_set} =$   
 $(\bigcup S \in (\lambda t. (\text{fst} \circ \text{snd}) \text{ 'set } t) \text{ 'set automata}). \bigcup b \in S. \text{vars\_of\_bexp } b)$   
 $\bigcup$   
 $(\bigcup S \in (\lambda t. (\text{fst} \circ \text{snd} \circ \text{snd} \circ \text{snd} \circ \text{snd}) \text{ 'set } t) \text{ 'set automata}).$   
 $\bigcup f \in S. \bigcup (x, e) \in \text{set } f. \{x\} \cup \text{vars\_of\_exp } e)$   
**unfolding** *var\_set\_def setcompr\_eq\_image*  
**by** (*rule arg\_cong2[where f = ( $\bigcup$ )]; auto simp: N\_p\_trans\_eq n\_ps\_def,*  
*(drule nth\_mem, erule bexI[rotated],*  
*metis (no\_types, lifting) insert\_iff mem\_case\_prodI prod.collapse) +,*  
*(drule mem\_nth, force) +)*

**lemma** *states\_loc\_setD*:  
 $\text{set } L \subseteq \text{loc\_set}$  **if**  $L \in \text{states}$   
**using** *states\_loc\_set* **that** **by** *auto*

end

**context** *Simple\_Network\_Impl*  
**begin**

**lemma** *sem\_bounds\_eq*: *sem.bounds = bounds*

```

unfolding sem.bounds_def bounds_def unfolding sem_def by simp

lemma n_ps_eq[simp]:
  sem.n_ps = n_ps
  unfolding n_ps_def sem.n_ps_def unfolding sem_def by auto

lemma sem_loc_set_eq:
  sem.loc_set = loc_set
  unfolding sem.loc_set_def loc_set_def n_ps_eq setcompr_eq_image
  apply (simp add: sem_N_eq N_eq)
  apply (rule arg_cong2[where f = (∪)])
  apply (safe; clarsimp;
    force split: prod.splits simp: conv_automaton_def trans_def automaton_of_def n_ps_def) +
  done

lemma sem_states_eq:
  sem.states = states
  unfolding sem.states_def states_def n_ps_eq setcompr_eq_image
  by (clarsimp simp: sem_N_eq N_eq;
    force simp: conv_automaton_def trans_def automaton_of_def n_ps_def split: prod.splits) +

end

locale Simple_Network_Rename_Defs =
  Simple_Network_Impl_Defs automata broadcast bounds' for automata ::
    ('s list × 's list × (('a :: countable) act, 's, 'c, 't, 'x :: countable, int) transition list
      × ('s :: countable) × ('c :: countable, 't) cconstraint) list) list
  and broadcast bounds' +
  fixes renum_acts :: 'a ⇒ nat
  and renum_vars :: 'x ⇒ nat
  and renum_clocks :: 'c ⇒ nat
  and renum_states :: nat ⇒ 's ⇒ nat
begin

definition
  map_cconstraint f g xs = map (map_acconstraint f g) xs

definition
  renum_cconstraint = map_cconstraint renum_clocks id

definition
  renum_act = map_act renum_acts

definition
  renum_bexp = map_bexp renum_vars

definition
  renum_exp = map_exp renum_vars

definition
  renum_upd = (λ(x, upd). (renum_vars x, renum_exp upd))

abbreviation

```

$renum\_upds \equiv map\ renum\_upd$

**definition**

$renum\_reset = map\ renum\_clocks$

**definition** *renum\_automaton* **where**

$renum\_automaton\ i \equiv \lambda(committed, urgent, trans, inv). let$   
 $committed' = map\ (renum\_states\ i)\ committed;$   
 $urgent' = map\ (renum\_states\ i)\ urgent;$   
 $trans' = map\ (\lambda(l, b, g, a, upd, r, l').$   
 $(renum\_states\ i\ l, renum\_bexp\ b, renum\_cconstraint\ g, renum\_act\ a, renum\_upds\ upd,$   
 $renum\_reset\ r, renum\_states\ i\ l')$   
 $)\ trans;$   
 $inv' = map\ (\lambda(l, g). (renum\_states\ i\ l, renum\_cconstraint\ g))\ inv$   
 $in\ (committed', urgent', trans', inv')$

**definition**

$vars\_inv \equiv the\_inv\ renum\_vars$

**definition**

$map\_st \equiv \lambda(L, s). (map\_index\ renum\_states\ L, s\ o\ vars\_inv)$

**definition**

$clocks\_inv \equiv the\_inv\ renum\_clocks$

**definition**

$map\_u\ u = u\ o\ clocks\_inv$

**lemma** *map\_u\_add[simp]*:

$map\_u\ (u \oplus d) = map\_u\ u \oplus d$   
**by** (*auto simp: map\_u\_def cval\_add\_def*)

**definition** *renum\_label* **where**

$renum\_label = map\_label\ renum\_acts$

**sublocale** *renum: Simple\_Network\_Impl\_Defs*

$map\_index\ renum\_automaton\ automata$

$map\ renum\_acts\ broadcast$

$map\ (\lambda(a, p). (renum\_vars\ a, p))\ bounds'$

.

**lemma** *renum\_n\_ps\_simp[simp]*:

$renum.n\_ps = n\_ps$   
**unfolding** *n\_ps\_def* *renum.n\_ps\_def* **by** *simp*

**end**

**locale** *Simple\_Network\_Rename\_Defs\_int* =

*Simple\_Network\_Rename\_Defs* *automata* +

*Simple\_Network\_Impl* *automata*

**for** *automata* ::

$('s\ list \times 's\ list \times (('a :: countable)\ act, 's, 'c, int, 'x :: countable, int)\ transition\ list$

```

      × ((s :: countable) × (c :: countable, int) cconstraint) list) list
begin

sublocale renum: Simple_Network_Impl
  map_index renum_automaton automata
  map renum_acts broadcast
  map (λ(a, p). (renum_vars a, p)) bounds'
.

end

lemma
  fixes f :: 'b :: countable ⇒ nat
  assumes inj_on f S finite S infinite (UNIV :: 'b set)
  shows extend_bij_inj: inj (extend_bij f S) and extend_bij_surj: surj (extend_bij f S)
    and extend_bij_extends: ∀ x ∈ S. extend_bij f S x = f x
proof -
  have infinite (ℕ :: nat set)
  unfolding Nats_def by simp
  from assms this have bij (extend_bij f S)
  by (intro extend_bij_bij) auto
  then show inj (extend_bij f S) and surj (extend_bij f S)
  unfolding bij_def by fast+
  from assms ⟨infinite ℕ⟩ show ∀ x ∈ S. extend_bij f S x = f x
  by (intro extend_bij_extends) auto
qed

lemma default_map_of_map:
  default_map_of y (map (λ(a, b). (f a, g b)) xs) (f a) = g (default_map_of x xs a)
  if inj f y = g x
  using that unfolding default_map_of_alt_def
  by (auto 4 4 dest: injD[OF that(1)] map_of_SomeD simp: map_of_eq_None_iff map_of_mapk_SomeI)

lemma default_map_of_map_2:
  default_map_of y (map (λ(a, b). (a, g b)) xs) a = g (default_map_of x xs a) if y = g x
  unfolding default_map_of_alt_def using that by (auto simp: map_of_map)

lemma map_of_map':
  map_of (map (λ(k, v). (k, f k v)) xs)
  = (λk. case map_of xs k of Some v ⇒ Some (f k v) | _ ⇒ None)
  by (induct xs) (auto simp: fun_eq_iff)

lemma default_map_of_map_3:
  default_map_of y (map (λ(a, b). (a, g a b)) xs) a = g a (default_map_of x xs a)
  if ∧k. y = g k x
  unfolding default_map_of_alt_def using that by (auto simp: map_of_map')

lemma dom_map_of_map:
  dom (map_of (map (λ (a, b). (f a, g b)) xs)) = f 'fst 'set xs
  unfolding dom_map_of_conv_image_fst by (auto 4 3)

lemma inj_image_eqI:
  S = T if inj f f ' S = f ' T
  using that by (meson inj_image_eq_iff)

```

**lemmas**  $[finite\_intros] = finite\_set$

**lemma** *map\_of\_NoneI*:

*map\_of xs x = None* **if**  $x \notin fst \text{ `set } xs$   
**by** (*simp add: map\_of\_eq\_None\_iff* *that*)

**lemma** *bij\_f\_the\_inv\_f*:

*f (the\_inv f x) = x* **if** *bij f*  
**using** *that f\_the\_inv\_f* **unfolding** *bij\_def* **by** (*simp add: f\_the\_inv\_f*)

**fun** *map\_sexp* ::

$(nat \Rightarrow 's \Rightarrow 's1) \Rightarrow ('a \Rightarrow 'a1) \Rightarrow ('b \Rightarrow 'b1) \Rightarrow (nat, 's, 'a, 'b) \text{ sexp}$   
 $\Rightarrow (nat, 's1, 'a1, 'b1) \text{ sexp}$

**where**

*map\_sexp* \_ \_ \_ *sexp.true* = *sexp.true* |  
*map\_sexp f g h* (*not e*) = *not* (*map\_sexp f g h e*) |  
*map\_sexp f g h* (*and e1 e2*) = *and* (*map\_sexp f g h e1*) (*map\_sexp f g h e2*) |  
*map\_sexp f g h* (*sexp.or e1 e2*) = *sexp.or* (*map\_sexp f g h e1*) (*map\_sexp f g h e2*) |  
*map\_sexp f g h* (*imply e1 e2*) = *imply* (*map\_sexp f g h e1*) (*map\_sexp f g h e2*) |  
*map\_sexp f g h* (*eq i x*) = *eq* (*g i*) (*h x*) |  
*map\_sexp f g h* (*lt i x*) = *lt* (*g i*) (*h x*) |  
*map\_sexp f g h* (*le i x*) = *le* (*g i*) (*h x*) |  
*map\_sexp f g h* (*ge i x*) = *ge* (*g i*) (*h x*) |  
*map\_sexp f g h* (*gt i x*) = *gt* (*g i*) (*h x*) |  
*map\_sexp f g h* (*loc i x*) = *loc i* (*f i x*)

**fun** *map\_formula* ::

$(nat \Rightarrow 's \Rightarrow 's1) \Rightarrow ('a \Rightarrow 'a1) \Rightarrow ('b \Rightarrow 'b1)$   
 $\Rightarrow (nat, 's, 'a, 'b) \text{ formula} \Rightarrow (nat, 's1, 'a1, 'b1) \text{ formula}$

**where**

*map\_formula f g h* (*formula.EX*  $\varphi$ ) = *formula.EX* (*map\_sexp f g h*  $\varphi$ ) |  
*map\_formula f g h* (*EG*  $\varphi$ ) = *EG* (*map\_sexp f g h*  $\varphi$ ) |  
*map\_formula f g h* (*AX*  $\varphi$ ) = *AX* (*map\_sexp f g h*  $\varphi$ ) |  
*map\_formula f g h* (*AG*  $\varphi$ ) = *AG* (*map\_sexp f g h*  $\varphi$ ) |  
*map\_formula f g h* (*Leadsto*  $\varphi \psi$ ) = *Leadsto* (*map\_sexp f g h*  $\varphi$ ) (*map\_sexp f g h*  $\psi$ )

**locale** *Simple\_Network\_Impl\_real* =

**fixes** *automata* ::

$('s \text{ list} \times 's \text{ list} \times ('a \text{ act}, 's, 'c, \text{real}, 'x, \text{int}) \text{ transition list}$   
 $\times ('s \times ('c, \text{real}) \text{ cconstraint}) \text{ list}) \text{ list}$

**and** *broadcast* ::  $'a \text{ list}$

**and** *bounds'* ::  $('x \times (\text{int} \times \text{int})) \text{ list}$

**begin**

**sublocale** *Simple\_Network\_Impl\_Defs* *automata broadcast bounds'* .

**abbreviation** *sem*  $\equiv (\text{set broadcast}, \text{map automaton\_of automata}, \text{map\_of bounds'})$

**sublocale** *Prod\_TA\_sem*  $(\text{set broadcast}, \text{map automaton\_of automata}, \text{map\_of bounds'})$  .

**end**

```

context Simple_Network_Impl
begin

lemma sem_state_guard_eq:
  (fst ∘ snd) ‘ trans (sem.N p) = (fst ∘ snd) ‘ trans (N p) if p < n_ps
  unfolding sem_N_eq[OF ⟨p < n_ps⟩] N_eq[OF ⟨p < n_ps⟩]
  unfolding automaton_of_def conv_automaton_def trans_def
  by (force split: prod.splits)

lemma sem_state_update_eq:
  (fst ∘ snd ∘ snd ∘ snd ∘ snd) ‘ trans (sem.N p) = (fst ∘ snd ∘ snd ∘ snd ∘ snd) ‘ trans (N p)
  if p < n_ps
  unfolding sem_N_eq[OF ⟨p < n_ps⟩] N_eq[OF ⟨p < n_ps⟩]
  unfolding automaton_of_def conv_automaton_def trans_def
  by (force split: prod.splits)

lemma sem_var_set_eq:
  sem.var_set = var_set
  unfolding sem.var_set_def var_set_def n_ps_eq using sem_state_guard_eq sem_state_update_eq
  by (simp cong: image_cong_simp add: setcompr_eq_image)

end

locale Simple_Network_Rename_Defs_real =
  Simple_Network_Rename_Defs automata +
  Simple_Network_Impl_real automata
  for automata ::
    ('s list × 's list
     × (('a :: countable) act, 's, 'c, real, 'x :: countable, int) transition list
     × (('s :: countable) × ('c :: countable, real) cconstraint) list) list
begin

sublocale renum: Simple_Network_Impl_real
  map_index renum_automaton automata
  map renum_acts broadcast
  map (λ(a,p). (renum_vars a, p)) bounds'
  .

end

locale Simple_Network_Rename' =
  Simple_Network_Rename_Defs where automata = automata for automata ::
    ('s list × 's list
     × (('a :: countable) act, 's, 'c, 't, 'x :: countable, int) transition list
     × (('s :: countable) × ('c :: countable, 't) cconstraint) list) list +
  assumes bij_renum_clocks: bij renum_clocks
  and renum_states_inj: ∀ p < n_ps. inj (renum_states p)
  and bij_renum_vars: bij renum_vars
  and bounds'_var_set: fst ‘ set bounds' = var_set
  and inj_renum_acts: inj renum_acts

locale Simple_Network_Rename_real =
  Simple_Network_Rename_Defs_real where automata = automata +

```

```

Simple_Network_Rename' where automata = automata
for automata ::
  ('s list × 's list
    × (('a :: countable) act, 's, 'c, real, 'x :: countable, int) transition list
    × (('s :: countable) × ('c :: countable, real) cconstraint) list) list +
  assumes urgency_removed:  $\forall i < n\_ps. \text{urgent } (N\ i) = \{\}$ 
begin

lemma aux_1:
  ( $\lambda x. \text{case case } x \text{ of}$ 
    ( $l, g \Rightarrow (\text{renum\_states } p\ l, \text{renum\_cconstraint } g) \text{ of}$ 
      ( $s, cc \Rightarrow (s, \text{map conv\_ac } cc)$ 
        = ( $\lambda (l, g). (\text{renum\_states } p\ l, \text{map conv\_ac } (\text{renum\_cconstraint } g)))$ )))

by auto

lemma clocks_inv_inv:
  clocks_inv (renum_clocks a) = a
  unfolding clocks_inv_def by (subst the_inv_f_f; rule bij_renum_clocks[THEN bij_is_inj]
    HOL.refl)

lemma map_u_renum_cconstraint_clock_valI:
  map_u u  $\vdash$  renum_cconstraint cc if u  $\vdash$  cc
  using that
  unfolding clock_val_def list_all_length
  unfolding renum_cconstraint_def map_cconstraint_def
  unfolding map_u_def
  apply clarsimp
  subgoal for n
    by (cases cc ! n) (auto 4 4 simp: clocks_inv_inv split: acconstraint.split)
  done

lemma map_u_renum_cconstraint_clock_valD:
  u  $\vdash$  cc if map_u u  $\vdash$  renum_cconstraint cc
  using that
  unfolding clock_val_def list_all_length
  unfolding renum_cconstraint_def map_cconstraint_def
  unfolding map_u_def
  apply clarsimp
  subgoal for n
    by (cases cc ! n) (auto 4 4 simp: clocks_inv_inv split: acconstraint.split)
  done

lemma inj_renum_states: inj (renum_states p) if p < n_ps
  using renum_states_inj ⟨p < n_ps⟩ by blast

lemma inv_renum_sem_I:
  assumes
    u  $\vdash$  Simple_Network_Language.inv (N p) l p < n_ps l  $\in$  loc_set
  shows
    map_u u  $\vdash$  Simple_Network_Language.inv (renum.N p) (renum_states p l)
  using assms
  unfolding inv_def
  apply –

```

```

apply (subst (asm) N_eq, assumption)
apply (subst renum.N_eq, subst renum_n_ps_simp, assumption)
apply (subst nth_map_index, simp add: n_ps_def)
unfolding conv_automaton_def automaton_of_def
apply (clarsimp split: prod.split_asm simp: renum_automaton_def comp_def)
unfolding aux_1
apply (subst default_map_of_map[where x = []])
subgoal
  by (intro inj_renum_states ⟨p < n_ps⟩)
  apply (simp add: renum_cconstraint_def map_cconstraint_def; fail)
apply (erule map_u_renum_cconstraint_clock_valI)
done

```

**lemma** inv\_renum\_sem\_D:

```

assumes
  map_u u ⊢ Simple_Network_Language.inv (renum.N p) (renum_states p l) p < n_ps l ∈
  loc_set
shows
  u ⊢ Simple_Network_Language.inv (N p) l
using assms
unfolding inv_def
apply –
apply (subst N_eq, assumption)
apply (subst (asm) renum.N_eq, subst renum_n_ps_simp, assumption)
apply (subst (asm) nth_map_index, simp add: n_ps_def)
unfolding conv_automaton_def automaton_of_def
apply (clarsimp split: prod.split simp: renum_automaton_def comp_def)
unfolding aux_1
apply (subst (asm) default_map_of_map[where x = []])
subgoal
  by (intro inj_renum_states ⟨p < n_ps⟩)
  apply (simp add: renum_cconstraint_def map_cconstraint_def; fail)
apply (erule map_u_renum_cconstraint_clock_valD)
done

```

**lemma** dom\_the\_inv\_comp:

```

dom (m o the_inv f) = f ‘ dom m if inj f range f = UNIV
unfolding dom_def
apply (clarsimp, safe)
subgoal for x y
  apply (subgoal_tac f (the_inv f x) = x)
  apply force
  using that by (auto intro: f_the_inv_f)
subgoal for x y
  using that by (auto simp: the_inv_f_f)
done

```

**lemma** inj\_renum\_vars:

```

inj renum_vars
using bij_renum_vars unfolding bij_def ..

```

**lemma** surj\_renum\_vars:

```

surj renum_vars
using bij_renum_vars unfolding bij_def ..

```

```

lemma map_of_inv_map:
  map_of xs (the_inv f x) = map_of (map ( $\lambda$  (a, b). (f a, b)) xs) x
  if inj f surj f the_inv f x  $\in$  dom (map_of xs)
  apply (subgoal_tac x = f (the_inv f x))
  subgoal premises prems
    using domD[OF that(3)] <inj f>
    apply (subst (2) prems)
    apply safe
    apply (subst map_of_mapk_SomeI; assumption)
    done
  subgoal
    apply (rule sym, rule f_the_inv_f)
    using that by auto
  done

lemma dom_vars_invD:
  assumes x  $\in$  dom (s o vars_inv)
  shows x  $\in$  renum_vars ' dom s (is ?A) and the_inv renum_vars x  $\in$  dom s (is ?B)
proof -
  show ?A
    using assms unfolding vars_inv_def dom_the_inv_comp[OF inj_renum_vars surj_renum_vars]
  .
  then show ?B
    apply (elim imageE)
    apply simp
    apply (subst the_inv_f_f, rule inj_renum_vars, assumption)
    done
qed

lemma bounded_renumI:
  assumes bounded (map_of bounds') s
  shows bounded (map_of (map ( $\lambda$ (a, y). (renum_vars a, y)) bounds')) (s o vars_inv)
  using assms unfolding bounded_def
  apply elims
  apply intros
  subgoal
    unfolding dom_map_of_conv_image_fst vars_inv_def
    by (auto intro!: image_cong simp: dom_the_inv_comp[OF inj_renum_vars surj_renum_vars]
    image_comp)
  subgoal for x
    apply (frule dom_vars_invD)
    apply (frule dom_vars_invD(2))
    apply (drule bspec, assumption)
    apply (auto simp: map_of_inv_map[OF inj_renum_vars surj_renum_vars] vars_inv_def)
    done
  subgoal for x
    apply (frule dom_vars_invD)
    apply (frule dom_vars_invD(2))
    apply (drule bspec, assumption)
    apply (auto simp: map_of_inv_map[OF inj_renum_vars surj_renum_vars] vars_inv_def)
    done
done

```

```

lemma map_of_renum_vars_simp:
  assumes
    dom (s o vars_inv)
    = dom (map_of (map ( $\lambda(a, y). (renum\_vars\ a, y))\ bounds')$ )
    x  $\in$  dom s dom s  $\subseteq$  var_set
  shows map_of (map ( $\lambda(a, y). (renum\_vars\ a, y))\ bounds')$  (renum_vars x) = map_of bounds'
x
proof -
  have *:
    map ( $\lambda(a, y). (renum\_vars\ a, y))\ bounds' = map (\lambda(a, y). (renum\_vars\ a, y))\ bounds'$ 
  by auto
  have x  $\in$  dom (map_of bounds')
  unfolding dom_map_of_conv_image_fst
  using assms
  unfolding vars_inv_def
  apply -
  apply (subst (asm) dom_the_inv_comp, rule inj_renum_vars, rule surj_renum_vars)
  apply (subst (asm) dom_map_of_map)
  apply clarsimp
  apply (subst (asm) inj_on_image_eq_iff[OF inj_renum_vars])
  by auto
  then obtain y where map_of bounds' x = Some y
  by auto
  then show ?thesis
  by (subst map_of_mapk_SomeI) (auto intro: inj_renum_vars)
qed

lemma bounded_renumD:
  assumes
    Simple_Network_Language.bounded
    (map_of (map ( $\lambda(a, y). (renum\_vars\ a, y))\ bounds')$ ) (s o vars_inv)
    and dom s  $\subseteq$  var_set
  shows Simple_Network_Language.bounded (map_of bounds') s
proof -
  show ?thesis
  using assms(1)
  unfolding bounded_def
  apply elims
  apply intros
  subgoal
    unfolding vars_inv_def
    apply (subst (asm) dom_the_inv_comp[OF inj_renum_vars surj_renum_vars])
    apply (subst (asm) dom_map_of_map)
    apply (rule inj_image_eqI[OF inj_renum_vars], simp add: dom_map_of_conv_image_fst)
  done
  subgoal for x
    using assms(2) unfolding vars_inv_def
    apply (subst (asm) (2) dom_the_inv_comp[OF inj_renum_vars surj_renum_vars])
    apply (drule bspec, erule imageI)
    apply (simp add: the_inv_f_f[OF inj_renum_vars] map_of_renum_vars_simp[unfolded
vars_inv_def])
  done
  subgoal for x
    using assms(2) unfolding vars_inv_def

```

```

    apply (subst (asm) (2) dom_the_inv_comp[OF inj_renum_vars surj_renum_vars])
    apply (drule bspec, erule imageI)
    apply (simp add: the_inv_f_f[OF inj_renum_vars] map_of_renum_vars_simp[unfolded
vars_inv_def])
  done
done
qed

```

```

lemma dom_bounds_var_set: dom bounds = var_set
  unfolding dom_bounds using bounds'_var_set .

```

```

lemma sem_states_loc_setD:  $L \neq \emptyset \implies p \in \text{loc\_set}$  if  $p < \text{length automata}$   $L \in \text{states}$  for  $L$   $p$ 
  using that states_loc_set by (force simp: n_ps_def)

```

```

lemma trans_N_renumD:
  assumes (l, b, g, a, f, r, l') ∈ Simple_Network_Language.trans (N p)  $p < n\_ps$ 
  shows (renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f,
renum_reset r, renum_states p l')
  ∈ Simple_Network_Language.trans (renum.N p)
  using assms
  unfolding mem_trans_N_iff[OF assms(2)] renum.mem_trans_N_iff[unfolded renum_n_ps_simp,
OF assms(2)]
  by (force split: prod.split simp: renum_automaton_def n_ps_def)

```

```

lemma match_assumption2:
  assumes  $P \ a1 \ b1 \ a1 = a \ b1 = b$  shows  $P \ a \ b$ 
  using assms by auto

```

```

lemma inj_pair:
  assumes inj f inj g
  shows inj ( $\lambda(a, b). (f \ a, g \ b)$ )
  using assms unfolding inj_on_def by auto

```

```

lemma injective_functions:
  inj renum_reset inj renum_upd inj renum_act inj renum_cconstraint inj renum_bexp
 $\wedge p. p < \text{length automata} \implies \text{inj (renum\_states } p)$ 
  subgoal
    unfolding renum_reset_def using bij_renum_clocks[THEN bij_is_inj] by simp
  subgoal
    unfolding renum_upd_def renum_exp_def using bij_renum_vars[THEN bij_is_inj]
    by (intro inj_pair exp.inj_map inj_mapI)
  subgoal
    unfolding renum_act_def using inj_renum_acts by (rule act.inj_map)
  subgoal
    unfolding renum_cconstraint_def map_cconstraint_def
    by (intro inj_mapI acconstraint.inj_map bij_renum_clocks bij_is_inj bij_id)
  subgoal
    unfolding renum_bexp_def by (intro bexp.inj_map inj_renum_vars)
  subgoal
    by (rule inj_renum_states, simp add: n_ps_def)
done

```

```

lemma trans_N_renumI:
  assumes (

```

```

    renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f, renum_reset
    r,
    renum_states p l'
  ∈ trans (renum.N p) p < n_ps
shows (l, b, g, a, f, r, l') ∈ trans (N p)
using assms
unfolding mem_trans_N_iff[OF assms(2)] renum.mem_trans_N_iff[unfolded renum_n_ps_simp,
OF assms(2)]
apply (clarsimp split: prod.split_asm simp: renum_automaton_def n_ps_def)
apply (fo_rule match_assumption2, assumption)
apply (auto elim!: injD[rotated, THEN sym] intro: injective_functions)
done

```

```

lemma renum_acconstraint_eq_convD:
  assumes map_acconstraint renum_clocks id g = conv_ac g'
  obtains g1 where g = conv_ac g1 g' = map_acconstraint renum_clocks id g1
  subgoal premises
    using assms
    apply (cases g'; cases g; clarsimp)
    subgoal for m c
      by (rule that[of acconstraint.LT c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.LE c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.EQ c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.GT c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.GE c m]; simp)
    done
  done

```

```

lemma renum_cconstraint_eq_convD:
  assumes renum_cconstraint g = conv_cc g'
  obtains g1 where g = conv_cc g1 g' = renum_cconstraint g1
  proof -
    let ?f = λ(ac, ac'). SOME ac1. ac = conv_ac ac1 ∧ ac' = map_acconstraint renum_clocks id
    ac1
    let ?g = map ?f (zip g g')
    from assms have length g = length g'
    unfolding renum_cconstraint_def map_cconstraint_def by -(drule arg_cong[where f =
length], simp)
    then have g = conv_cc ?g ∧ g' = renum_cconstraint ?g
    using assms
    by (simp add: comp_def renum_cconstraint_def map_cconstraint_def)
    (induction rule: list_induct2; simp; elim conjE renum_acconstraint_eq_convD; smt someI)
    then show ?thesis
    by (blast intro: that)
  qed

```

```

lemma trans_sem_N_renumI:
  assumes (
    renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f, renum_reset
    r,

```

$\text{renum\_states } p \ l'$   
 $\in \text{trans } (\text{renum}.N \ p) \ p < n\_ps$   
**shows**  $(l, b, g, a, f, r, l') \in \text{trans } (N \ p)$   
**using**  $\text{assms}(1) \ \langle p < n\_ps \rangle$  **by**  $(\text{simp add: trans\_N\_renumI})$

**lemma**  $\text{trans\_sem\_N\_renumI}'$ :

**assumes**  $(\text{renum\_states } p \ l, b, g, a, f, r, l') \in \text{trans } (\text{renum}.N \ p) \ p < n\_ps$   
**shows**  $\exists \ b' \ g' \ a' \ f' \ r' \ l1.$   
 $\text{TRANS "orig" } \llbracket (l, b', g', a', f', r', l1) \in \text{Simple\_Network\_Language.trans } (N \ p)$   
 $\wedge \text{"renum bexp" } \llbracket b = \text{renum\_bexp } b' \wedge \text{"renum guard" } \llbracket g = \text{renum\_cconstraint } g'$   
 $\wedge \text{"renum action" } \llbracket a = \text{renum\_act } a' \wedge \text{"renum upds" } \llbracket f = \text{renum\_upds } f'$   
 $\wedge \text{"renum reset" } \llbracket r = \text{renum\_reset } r' \wedge \text{"renum loc" } \llbracket l' = \text{renum\_states } p \ l1$

**proof** –

**obtain**  $b' \ g' \ a' \ f' \ r' \ l1$  **where**  $b = \text{renum\_bexp } b' \ g = \text{renum\_cconstraint } g' \ f = \text{renum\_upds } f'$

$a = \text{renum\_act } a' \ r = \text{renum\_reset } r' \ l' = \text{renum\_states } p \ l1$

**using**  $\text{assms}$

**unfolding**  $N\_eq[OF \ \text{assms}(2)] \ \text{renum}.N\_eq[\text{unfolded } \text{renum\_n\_ps\_simp}, OF \ \text{assms}(2)]$

**apply**  $(\text{subst } (\text{asm}) \ \text{nth\_map\_index})$

**subgoal**

**by**  $(\text{simp add: } n\_ps\_def)$

**unfolding**  $\text{renum\_automaton\_def} \ \text{automaton\_of\_def} \ \text{trans\_def}$  **by**  $(\text{auto split: prod.split\_asm})$

**with**  $\text{assms}$  **show**  $?thesis$

**by**  $\text{untag } (\text{fastforce dest!: trans\_sem\_N\_renumI})$

**qed**

**lemma**  $\text{committed\_renum\_eq}$ :

$\text{committed } (\text{renum}.N \ p) = \text{renum\_states } p \ \text{' committed } (N \ p) \ \text{if } p < n\_ps$

**unfolding**

$\text{committed\_def } N\_eq[OF \ \langle p < n\_ps \rangle] \ \text{renum}.N\_eq[\text{unfolded } \text{renum\_n\_ps\_simp}, OF \ \langle p < n\_ps \rangle]$

**apply**  $(\text{subst } \text{nth\_map\_index})$

**subgoal**

**using**  $\langle p < n\_ps \rangle$  **by**  $(\text{simp add: } n\_ps\_def)$

**unfolding**  $\text{automaton\_of\_def} \ \text{conv\_automaton\_def} \ \text{renum\_automaton\_def}$  **by**  $(\text{simp split: prod.split})$

**lemma**  $\text{urgent\_renum\_eq}$ :

$\text{urgent } (\text{renum}.N \ p) = \text{renum\_states } p \ \text{' urgent } (N \ p) \ \text{if } p < n\_ps$

**unfolding**

$\text{urgent\_def } N\_eq[OF \ \langle p < n\_ps \rangle] \ \text{renum}.N\_eq[\text{unfolded } \text{renum\_n\_ps\_simp}, OF \ \langle p < n\_ps \rangle]$

**apply**  $(\text{subst } \text{nth\_map\_index})$

**subgoal**

**using**  $\langle p < n\_ps \rangle$  **by**  $(\text{simp add: } n\_ps\_def)$

**unfolding**  $\text{automaton\_of\_def} \ \text{conv\_automaton\_def} \ \text{renum\_automaton\_def}$  **by**  $(\text{simp split: prod.split})$

**lemma**  $\text{renum\_urgency\_removed}$ :

$\forall i < n\_ps. \text{urgent } (\text{renum}.N \ i) = \{\}$

**using**  $\text{urgency\_removed}$

**apply**  $\text{intros}$

**apply**  $(\text{simp only: urgent\_renum\_eq})$

**apply**  $\text{simp}$

**done**

```

lemma check_bexp_renumD:
  check_bexp s b bv  $\implies$  check_bexp (s o vars_inv) (renum_bexp b) bv
and is_val_renumD:
  is_val s e v  $\implies$  is_val (s o vars_inv) (renum_exp e) v
  apply (induction rule: check_bexp_is_val.inducts)
  apply (solves ‹
    auto
    intro: check_bexp_is_val.intros
    simp: renum_bexp_def renum_exp_def vars_inv_def the_inv_ff[OF inj_renum_vars]
  ›)+
subgoal
  apply (clarsimp simp: renum_bexp_def renum_exp_def vars_inv_def, safe)
  using is_val.simps apply fastforce+
  done
apply (auto intro: check_bexp_is_val.intros simp: renum_exp_def)
done

method solve_case =
  auto
  intro: check_bexp_is_val.intros
  simp: renum_bexp_def renum_exp_def vars_inv_def the_inv_ff[OF inj_renum_vars];
  fail
| use is_val.simps in fastforce

lemma check_bexp_renumI:
  check_bexp (s o vars_inv) (renum_bexp b) bv  $\implies$  check_bexp s b bv
and is_val_renumI:
  is_val (s o vars_inv) (renum_exp e) v  $\implies$  is_val s e v
  apply (induction
    s o vars_inv renum_bexp b bv and s o vars_inv renum_exp e __
    arbitrary: b and e rule: check_bexp_is_val.inducts)
subgoal for b
  by (cases b; solve_case)
subgoal for e bv b
  by (cases b; solve_case)
subgoal for e1 a e2 bv b
  by (cases b; solve_case)
subgoal for e1 a e2 bv b
  by (cases b; solve_case)
subgoal for e1 a e2 bv b
  by (cases b; solve_case)
subgoal for a v bv x b
  by (cases b; solve_case)
subgoal for a v bv x b
  by (cases b; solve_case)
subgoal for a v bv x b
  by (cases b; solve_case)
subgoal for a v bv x b
  by (cases b; solve_case)
subgoal for a v bv x b
  by (cases b; solve_case)
subgoal for c e
  by (cases e; solve_case)

```

```

subgoal for  $x\ v\ e$ 
  by (cases  $e$ ; solve_case)
subgoal for  $e1\ v1\ e2\ v2\ b\ bv\ e$ 
  by (clarsimp simp: renum_bexp_def renum_exp_def; safe; cases  $e$ ; solve_case)
subgoal for  $e1\ v1\ e2\ v2\ f\ e$ 
  by (clarsimp simp: renum_bexp_def renum_exp_def; cases  $e$ ; solve_case)
subgoal for  $e1\ v\ f\ e$ 
  by (cases  $e$ ; solve_case)
done

```

```

lemma renum_reset_map_u:
  [renum_reset  $r \rightarrow 0$ ]map_u  $u = \text{map\_u } ([r \rightarrow 0]u)$ 
  unfolding map_u_def
  apply (rule ext)
  subgoal for  $x$ 
    apply (cases clocks_inv  $x \in \text{set } r$ ; simp add: clocks_inv_def)
    subgoal
      using bij_renum_clocks[THEN bij_is_surj]
      by (auto
        simp: renum_reset_def f_the_inv_f[OF bij_is_inj, OF bij_renum_clocks]
        dest: imageI[where  $f = \text{renum\_clocks}$ ]
      )
    subgoal
      by (subst clock_set_id)
      (auto simp: renum_reset_def the_inv_f_f[OF bij_is_inj, OF bij_renum_clocks])
    done
  done
done

```

```

lemma bounded_renumI':
  assumes bounded_bounds  $s'$ 
  shows bounded renum.bounds ( $s' \circ \text{vars\_inv}$ )
  using assms unfolding renum.bounds_def bounds_def by (simp add: bounded_renumI)

```

```

lemma bounded_renumD':
  assumes bounded_renum.bounds ( $s' \circ \text{vars\_inv}$ )  $\text{dom } s' \subseteq \text{var\_set}$ 
  shows bounded_bounds  $s'$ 
  using assms unfolding renum.bounds_def bounds_def by (simp add: bounded_renumD)

```

```

lemma is_upd_renumD:
  assumes is_upd  $s\ f\ s'$ 
  shows is_upd ( $s \circ \text{vars\_inv}$ ) ( $\text{renum\_upd } f$ ) ( $s' \circ \text{vars\_inv}$ )
  using assms
  unfolding is_upd_def renum_upd_def
  by (force dest!: is_val_renumD
    simp: bij_f_the_inv_f[OF bij_renum_vars] the_inv_f_f[OF inj_renum_vars]
    vars_inv_def)

```

```

lemma is_upds_renumD:
  assumes is_upds  $s1\ ps\ s'$ 
  shows is_upds ( $s1 \circ \text{vars\_inv}$ ) ( $\text{renum\_upds } ps$ ) ( $s' \circ \text{vars\_inv}$ )
  using assms by induction (auto intro: is_upds.intros simp: comp_def dest!: is_upd_renumD)

```

```

lemma is_upds_concat_renumD:
  assumes is_upds  $s1\ (\text{concat } ps)\ s'$ 

```

**shows**  $is\_upds\ (s1\ o\ vars\_inv)\ (concat\_map\ renum\_upds\ ps)\ (s'\ o\ vars\_inv)$   
**using** *assms* **by** (*induction ps arbitrary: s1*) (*auto simp: comp\_def map\_concat dest!: is\_upds\_renumD*)

**lemma** *Ball\_mono*:  
**assumes**  $\forall x \in S. P\ x \wedge x. x \in S \implies P\ x \implies Q\ x$   
**shows**  $\forall x \in S. Q\ x$   
**using** *assms* **by** *blast*

**lemma** *atLeastLessThan\_upperD*:  
**assumes**  $S \subseteq \{l..<u\}\ x \in S$   
**shows**  $x < u$   
**using** *assms* **by** *auto*

**lemma** *imp\_mono\_rule*:  
**assumes**  $P1 \longrightarrow P2$   
**and**  $Q1 \implies P1$   
**and**  $Q1 \implies P2 \implies Q2$   
**shows**  $Q1 \longrightarrow Q2$   
**using** *assms* **by** *blast*

**lemma** *inj\_id*:  
*inj id*  
**by** *auto*

**lemma** *step\_single\_renumD*:  
**assumes**  $step\_u\ sem\ L\ s\ u\ a\ L'\ s'\ u'\ L \in states\ dom\ s \subseteq var\_set$   
**shows**  $step\_u\ renum.sem$   
 $(map\_index\ renum\_states\ L)\ (s\ o\ vars\_inv)\ (map\_u\ u)$   
 $(renum\_label\ a)$   
 $(map\_index\ renum\_states\ L')\ (s'\ o\ vars\_inv)\ (map\_u\ u')$   
**using** *assms*(1-3)  
**apply** (*cases a*)

**supply** [*simp del*] = *map\_map\_index set\_map*  
— To keep the simplifier from breaking through locale abstractions

— Delay

**subgoal**

**supply** [*simp*] = *length\_automata\_eq\_n\_ps L\_len*  
**apply** (*subst (asm) A\_split*)  
**apply** (*subst renum.A\_split*)  
**apply** (*simp only: renum\_label\_def label.map renum\_n\_ps\_simp*)  
**apply** (*erule step\_u\_elims*)  
**apply** *simp*  
**apply** (*rule step\_u.intros*)  
**preferT** <"target invariant"> **subgoal**  
**by** *tag (auto 4 3 simp: dest: inv\_renum\_sem\_I[OF \_\_ sem\_states\_loc\_setD])*  
**preferT** <"nonnegative"> **subgoal**  
**by** *assumption*  
**preferT** <"urgent"> **subgoal**  
**using** *renum\_urgency\_removed* **by** — (*tag, auto*)  
**preferT** <"bounded"> **subgoal**  
**by** *tag (rule bounded\_renumI')*  
**done**

— Internal

```

subgoal for a'
  supply [simp] = length_automata_eq_n_ps L_len
  apply (subst (asm) A_split)
  apply (subst renum.A_split)
  apply (simp only: renum_label_def label.map renum_n_ps_simp)
  apply (erule step_u_elims')
  unfolding TAG_def[of "range"]
  apply simp
  apply (rule step_u.intros)
  preferT <TRANS "silent">
  apply (solves <tag, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+>)
  preferT <"committed"> subgoal
    by tag (auto simp: committed_renum_eq dest!: inj_renum_states[THEN injD, rotated])
  preferT <"bexp"> subgoal
    by tag (erule check_bexp_renumD)
  preferT <"guard"> subgoal
    by tag (rule map_u_renum_cconstraint_clock_valI)
  preferT <"target invariant"> subgoal
    apply (tag- "new loc" TRANS "silent")
    apply clarsimp
  apply (rule inv_renum_sem_I[OF __ sem_states_loc_setD[OF state_preservation_updI]])
    apply fastforce+
  done
  preferT <"loc">
    apply (tag, simp; fail)
  preferT <"range">
    apply (tag, simp; fail)
  preferT <"new loc">
    apply (tag, simp only: map_index_update; fail)
  preferT <"new valuation">
    apply (tag, simp only: renum_reset_map_u; fail)
  preferT <"is_upd"> subgoal
    by (tag, erule is_upds_renumD)
  preferT <"bounded">
    apply (tag, erule bounded_renumI'; fail)
done

```

— Binary

```

subgoal for a'
  supply [simp] = length_automata_eq_n_ps L_len
  apply (subst (asm) A_split)
  apply (subst renum.A_split)
  apply (simp only: renum_label_def label.map renum_n_ps_simp)
  apply (erule step_u_elims')
  unfolding TAG_def[of RECV "range"] TAG_def[of SEND "range"]
  apply simp
  apply (rule step_u.intros)
  preferT <"not broadcast"> subgoal
    apply tag
    unfolding renum.broadcast_def
    apply (clarsimp simp: set_map)
    apply (drule injD[OF inj_renum_acts])

```

```

    apply (subst (asm) broadcast_def, simp)
  done
preferT <TRANS "in">
apply (solves <tag, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+>)
preferT <TRANS "out">
apply (solves <tag, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+>)
preferT <"committed"> subgoal
  by tag (auto simp: committed_renum_eq dest!: inj_renum_states[THEN injD, rotated])

preferT <"bexp"> subgoal
  by tag (erule check_bexp_renumD)

preferT <"bexp"> subgoal
  by tag (erule check_bexp_renumD)

preferT <"guard"> subgoal
  by tag (rule map_u_renum_cconstraint_clock_valI)

preferT <"guard"> subgoal
  by tag (rule map_u_renum_cconstraint_clock_valI)

preferT <"target invariant"> subgoal
  apply (tag- "new loc" TRANS _)
  apply clarsimp
  apply (rule inv_renum_sem_I[OF __ sem_states_loc_setD[OF state_preservation_updI]])
  apply (fastforce intro!: state_preservation_updI)+
  done
preferT <"new loc">
  apply (tag, simp add: map_index_update; fail)
preferT <"new valuation">
  apply (tag, simp add: renum_reset_map_u[symmetric] renum_reset_def; fail)
preferT <"upd">
  apply (tag, erule is_upds_renumD; fail)
preferT <"upd">
  apply (tag, erule is_upds_renumD; fail)
preferT <"bounded">
  apply (tag, erule bounded_renumI'; fail)
apply (tag, simp; fail)+
done

```

— Broadcast

subgoal for  $a'$

```

supply [simp] = length_automata_eq_n_ps L_len
apply (subst (asm) A_split)
apply (subst renum.A_split)
apply (simp only: renum_label_def label.map renum_n_ps_simp)
apply (erule step_u_elims')
apply simp
unfolding TAG_def[of SEND "range"]

```

```

apply (rule step_u.intros)

```

```

preferT <"broadcast"> subgoal

```

```

  unfolding renum.broadcast_def by (tag, subst (asm) broadcast_def, simp add: set_map)

```

```

preferT <TRANS "out">
apply (solves<tag, simp, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+>)

preferT <TRANS "in">
  apply (tag- SEL _)
apply (solves<auto dest!: trans_N_renumD simp add: renum_act_def atLeastLessThan_upperD>)

preferT <"committed"> subgoal
  apply (tag- SEL _)
  apply (simp add: committed_renum_eq)
  apply (erule disj_mono[rule_format, rotated 2], (simp; fail))
  apply (erule disj_mono[rule_format, rotated 2])
  subgoal
    apply clarify
    apply (rule rev_bexI, assumption)
    apply (auto simp: committed_renum_eq)
    done
  apply (auto simp: committed_renum_eq dest!: inj_renum_states[THEN injD, rotated]; fail)
  done

preferT <"maximal"> subgoal
  apply tag
  apply simp
  apply (erule all_mono[THEN mp, OF impI, rotated])
  apply (erule (1) imp_mono_rule)
  apply (erule (1) imp_mono_rule)
  apply clarify
  apply (drule trans_sem_N_renumI', assumption)
  apply (untag, clarsimp simp: renum_act_def)
  apply (drule check_bexp_renumI)
  apply (drule InD2[OF inj_renumActs])
  apply (fastforce dest!: map_u_renum_cconstraint_clock_valD)
  done

preferT <"target invariant"> subgoal for l b g f r l' p ps bs gs fs rs ls' s'
  apply (tag- SEL _ "new loc" TRANS _)
  apply (subgoal_tac fold (λp L. L[p := ls' p]) ps L ∈ states)
  subgoal
    apply simp
    apply (erule all_mono[THEN mp, OF impI, rotated], erule (1) imp_mono_rule,
      drule (1) inv_renum_sem_I)
    subgoal
      apply (rule sem_states_loc_setD, simp)
      apply (rule state_preservation_updI)
      subgoal
        by blast
      .
    subgoal
      apply (fo_rule match_assumption2[where P = clock_val], assumption, rule HOL.refl)
      apply (drule states_lengthD, simp)
      done
    done
  subgoal

```

```

    apply (rule state_preservation_fold_updI)
    apply (erule Ball_mono)
    apply (simp add: atLeastLessThan_upperD; blast)
  by simp
done

preferT <"bexp">
  apply (tag, erule check_bexp_renumD; fail)
preferT <"bexp">
  apply (solves <tag, auto elim: check_bexp_renumD>)
preferT <"guard">
  apply (tag, erule map_u_renum_cconstraint_clock_valI; fail)
preferT <"guard">
  apply (solves <tag, auto elim: map_u_renum_cconstraint_clock_valI>)
preferT <"new loc">
  apply (tag, simp add: map_trans_broad_aux1[symmetric] map_index_update; fail)
preferT <"new valuation">
  apply (tag, simp add: renum_reset_map_u[symmetric] renum_reset_def map_concat
comp_def; fail)
preferT <"upd">
  apply (tag, erule is_upds_renumD; fail)
preferT <"upds">
  apply (tag, drule is_upds_concat_renumD, simp add: comp_def; fail)
preferT <"bounded">
  apply (tag, erule bounded_renumI'; fail)
  apply (tag, simp; fail)+
done
done

lemma inj_the_inv:
  inj (the_inv f) if bij f
  by (auto simp: bij_f_the_inv_f[OF that] dest: arg_cong[where f = f] intro!: injI)

lemma inj_vars_inv:
  inj vars_inv
  using bij_renum_vars unfolding vars_inv_def by (rule inj_the_inv)

lemma comp_vars_inv_upd_commute:
  (s o vars_inv)(x  $\mapsto$  y) = s(vars_inv x  $\mapsto$  y) o vars_inv
  by (intro ext) (auto dest: injD[OF inj_vars_inv])

lemma is_upd_renumI:
  assumes is_upd (s o vars_inv) (renum_upd f) s'
  shows is_upd s f (s' o renum_vars)
  using assms
  unfolding is_upd_def renum_upd_def
  by (clarsimp dest!: is_val_renumI split: prod.split_asm simp: comp_vars_inv_upd_commute)
  (force simp: the_inv_f_f[OF inj_renum_vars] vars_inv_def)

lemma is_upd_renumI':
  assumes is_upd (s o vars_inv) (renum_upd f) s'
  obtains s1 where is_upd s f s1 s1 = s' o renum_vars
  by (simp add: assms is_upd_renumI)

```

```

lemma is_upd_renumI'':
  assumes is_upd s (renum_upd f) s'
  shows is_upd (s o renum_vars) f (s' o renum_vars)
proof -
  have s = (s o renum_vars) o vars_inv
  by (intro ext) (auto simp: bij_f_the_inv_f[OF bij_renum_vars] vars_inv_def)
  with assms show ?thesis
  by (intro is_upd_renumI) simp
qed

lemma is_upds_renumI:
  assumes is_upds (s o vars_inv) (renum_upds upds) s'
  shows  $\exists s1. is\_upds\ s\ upds\ s1 \wedge s1 = s' o renum\_vars$ 
  using assms apply (induction s o vars_inv renum_upds upds s' arbitrary: upds s)
  subgoal for upds s
  apply simp
  apply (subgoal_tac s o vars_inv o renum_vars = s)
  apply (solves (auto intro: is_upds.intros simp: comp_def))
  apply (rule ext)
  apply (simp add: vars_inv_def the_inv_f_f[OF inj_renum_vars])
  done

  apply clarsimp
  apply (erule is_upd_renumI')
  apply (auto simp: vars_inv_def bij_f_the_inv_f[OF bij_renum_vars] comp_def intro!: is_upds.intros)
  done

lemma is_upds_renumI':
  assumes is_upds (s o vars_inv) (renum_upds f) s'
  shows is_upds s f (s' o renum_vars)
  using is_upds_renumI[OF assms] by simp

lemma is_upds_renumI'':
  assumes is_upds s (renum_upds ps) s'
  shows is_upds (s o renum_vars) ps (s' o renum_vars)
  using assms
  by (induction renum_upds ps s' arbitrary: ps) (auto 4 3 intro: is_upd_renumI'' is_upds.intros)

lemma is_upds_concat_renumI'':
  assumes is_upds s (concat_map renum_upds ps) s'
  shows is_upds (s o renum_vars) (concat ps) (s' o renum_vars)
  apply (rule is_upds_renumI'')
  using assms by (simp add: map_concat)

lemma bounded_renumD1:
  assumes bounded renum.bounds s' dom (s' o renum_vars)  $\subseteq$  var_set
  shows bounded bounds (s' o renum_vars)
  using assms
  by (intro bounded_renumD') (auto simp: vars_inv_def bij_f_the_inv_f[OF bij_renum_vars] comp_def)

lemma renum_reset_append:
  renum_reset xs @ renum_reset ys = renum_reset (xs @ ys)
  unfolding renum_reset_def by simp

```

```

lemma if_eq_distrib:
  (if i = j then f i a else f j b) = (f j (if i = j then a else b))
  by auto

lemma dom_comp_eq_vimage:
  dom (s o f) = f -' dom s
  unfolding dom_def by auto

lemma dom_comp_vars_inv_eqD:
  assumes dom s' = dom (s o vars_inv)
  shows dom (s' o renum_vars) = dom s
  using assms inj_renum_vars surj_renum_vars unfolding vars_inv_def
  by (subst (asm) dom_the_inv_comp) (auto simp: dom_comp_eq_vimage dest: injD)

lemma sem_trans_upd_domD:
  assumes (L ! p, b, g', a, f, r, l1) ∈ trans (N p) p < n_ps
  shows fst ' set f ⊆ var_set
proof -
  from assms have fst ' set f ⊆ var_set
  unfolding var_set_def
  apply -
  apply (rule semilattice_sup_class.sup.coboundedI2)
  apply clarsimp
  apply (inst_existentials (fst o snd o snd o snd o snd) ' trans (N p) p)
  apply force+
  done
  then show ?thesis .
qed

lemma SilD:
  fixes map_action
  assumes Sil a = map_act map_action a1
  obtains a' where a1 = Sil a' a = map_action a'
  using assms by (cases a1) auto

lemma InD:
  fixes map_action
  assumes In a = map_act map_action a1
  obtains a' where a1 = In a' a = map_action a'
  using assms by (cases a1) auto

lemma OutD:
  fixes map_action
  assumes Out a = map_act map_action a1
  obtains a' where a1 = Out a' a = map_action a'
  using assms by (cases a1) auto

lemma In_OutD:
  assumes In a = renum_act a1 Out a = renum_act a2
  obtains a' where a = renum_acts a' a1 = In a' a2 = Out a'
  using assms unfolding renum_act_def by (elim InD OutD) (auto simp: injD[OF inj_renum_acts])

lemma renum_sem_broadcast_eq:

```

*renum.broadcast* = *renum\_acts* ‘ *set broadcast*  
**unfolding** *renum.broadcast\_def* **by** *simp*

**lemma** *inj\_eq\_iff*:  
 $f\ x = f\ y \longleftrightarrow x = y$  **if** *inj f*  
**using** *that* **unfolding** *inj\_def* **by** *auto*

**lemma** *trans\_sem\_N\_renum\_broadcastI*:

**assumes**

$\forall p \in \text{set } ps. (\text{renum\_states } p\ (L\ !\ p),\ bs\ p,\ gs\ p,\ In\ a,\ fs\ p,\ rs\ p,\ ls\ p) \in \text{trans } (\text{renum}.N\ p)$   
 $\text{set } ps \subseteq \{0..<n\_ps\}$

**obtains** *bs' gs' fs' rs' ls' a'* **where**

$\forall p \in \text{set } ps. (L\ !\ p,\ bs'\ p,\ gs'\ p,\ In\ a',\ fs'\ p,\ rs'\ p,\ ls'\ p) \in \text{trans } (N\ p)$   
 $\forall p \in \text{set } ps. bs\ p = \text{renum\_bexp } (bs'\ p)$   
 $\forall p \in \text{set } ps. gs\ p = \text{renum\_cconstraint } (gs'\ p)$   
 $ps \neq [] \longrightarrow a = \text{renum\_acts } a'$   
 $\forall p \in \text{set } ps. fs\ p = \text{renum\_upds } (fs'\ p)$   
 $\forall p \in \text{set } ps. rs\ p = \text{renum\_reset } (rs'\ p)$   
 $\forall p \in \text{set } ps. ls\ p = \text{renum\_states } p\ (ls'\ p)$

**proof** –

**define** *t* **where**

$t\ p \equiv \text{SOME } (b',\ g',\ a',\ f',\ r',\ l1). (L\ !\ p,\ b',\ g',\ a',\ f',\ r',\ l1) \in \text{trans } (N\ p)$

$\wedge bs\ p = \text{renum\_bexp } b' \wedge gs\ p = \text{renum\_cconstraint } g' \wedge In\ a = \text{renum\_act } a' \wedge fs\ p =$   
 $\text{renum\_upds } f'$

$\wedge rs\ p = \text{renum\_reset } r' \wedge ls\ p = \text{renum\_states } p\ l1$  **for** *p*

**define** *bs'* **where**  $bs'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow b'$  **for** *p*

**define** *gs'* **where**  $gs'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow g'$  **for** *p*

**define** *as'* **where**  $as'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow a'$  **for** *p*

**define** *fs'* **where**  $fs'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow f'$  **for** *p*

**define** *rs'* **where**  $rs'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow r'$  **for** *p*

**define** *ls'* **where**  $ls'\ p \equiv \text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow l1$  **for** *p*

**have** \*:  $\text{case } t\ p\ \text{of } (b',\ g',\ a',\ f',\ r',\ l1) \Rightarrow$

$(L\ !\ p,\ b',\ g',\ a',\ f',\ r',\ l1) \in \text{trans } (N\ p)$

$\wedge bs\ p = \text{renum\_bexp } b' \wedge gs\ p = \text{renum\_cconstraint } g' \wedge In\ a = \text{renum\_act } a' \wedge fs\ p =$   
 $\text{renum\_upds } f'$

$\wedge rs\ p = \text{renum\_reset } r' \wedge ls\ p = \text{renum\_states } p\ l1$  **if**  $p \in \text{set } ps$  **for** *p*

**using** *assms(1)*

**apply** –

**apply** (*drule* *bspec[OF \_ that]*)

**apply** (*drule* *trans\_sem\_N\_renumI'*)

**subgoal**

**using** *assms(2)* **that** **by** *auto*

**unfolding** *t\_def* **by** (*untag, elims, fo\_rule someI, auto*)

**show** *?thesis*

**proof** (*cases ps*)

**case** *Nil*

**then show** *?thesis*

**by** (*auto intro: that*)

**next**

**case** (*Cons p ps'*)

**then have**  $p \in \text{set } ps$

**by** *auto*

```

define  $a'$  where  $a' = as' p$ 
have  $1: as' q = a'$  and  $In a = renum\_act a'$  if  $q \in set ps$  for  $q$ 
  unfolding  $as'\_def a'\_def$  using  $*[OF that] *[OF \langle p \in set ps \rangle]$ 
  by  $(auto dest: injD[OF injective\_functions(3)])$ 
then obtain  $a1$  where  $a' = In a1 a = renum\_acts a1$ 
  using  $\langle p \in set ps \rangle$  unfolding  $renum\_act\_def$  by  $(auto elim!: InD)$ 
then show  $?thesis$ 
  apply –
  apply  $(rule that[of bs' gs' a1 fs' rs' ls'])$ 
  unfolding  $bs'\_def gs'\_def fs'\_def rs'\_def ls'\_def$ 
  by  $(solves \langle (intros, frule *) \rangle?, auto simp: as'\_def dest: 1 \rangle) +$ 
qed
qed

```

```

lemmas  $all\_mono\_rule = all\_mono[THEN mp, OF impI, rotated]$ 

```

```

method  $prop\_monos =$ 
   $erule all\_mono\_rule$ 
|  $erule (1) imp\_mono\_rule$ 
|  $erule disj\_mono[rule\_format, rotated 2]$ 

```

```

lemma  $inv\_sem\_N\_renum\_broadcastI:$ 

```

```

  assumes
   $\forall pa < n\_ps.$ 
   $[renum\_reset r @ concat (map rs ps) \rightarrow 0] map\_u u$ 
   $\vdash inv (renum.N pa)$ 
   $((fold (\lambda p L. L[p := ls p]) ps (map\_index renum\_states L)) [p := renum\_states p l1] ! pa)$ 
   $\forall p \in set ps. rs p = renum\_reset (rs' p)$ 
   $\forall p \in set ps. ls p = renum\_states p (ls' p)$ 
   $\forall p \in set ps. ls' p \in (\bigcup (l, b, g, a, r, u, l') \in trans (N p). \{l, l'\})$ 
   $l1 \in (\bigcup (l, b, g, a, r, u, l') \in trans (N p). \{l, l'\})$ 
   $L \in states$ 
shows
   $\forall pa < n\_ps.$ 
   $[r @ concat (map rs' ps) \rightarrow 0] u \vdash inv (N pa) ((fold (\lambda p L. L[p := ls' p]) ps L) [p := l1] ! pa)$ 
proof –
  have  $[simp]: renum\_reset r @ concat (map rs ps) = renum\_reset (r @ concat (map rs' ps))$ 
  using  $assms(2)$  by  $(simp cong: map\_cong add: map\_concat renum\_reset\_def)$ 
  have  $[simp]: length L = n\_ps$ 
  using  $\langle L \in states \rangle$  by  $auto$ 
  have  $[simp]:$ 
   $((fold (\lambda p L. L[p := ls p]) ps (map\_index renum\_states L)) [p := renum\_states p l1] ! q)$ 
   $= renum\_states q ((fold (\lambda p L. L[p := ls' p]) ps L) [p := l1] ! q)$ 
  if  $q < n\_ps$  for  $q$ 
  using  $assms(4-)$  that
  apply  $(cases p = q)$ 
  apply  $(simp add: fold\_ups\_aux\_length)$ 
  apply  $(simp$ 
     $add: assms(3) map\_trans\_broad\_aux1[symmetric] fold\_ups\_aux\_length$ 
     $cong: fold\_cong)$ 
  done
  have  $(fold (\lambda p L. L[p := ls' p]) ps L)[p := l1] ! q \in loc\_set$  if  $q < n\_ps$  for  $q$ 
  using  $assms(4-)$ 
  apply  $(intro sem\_states\_loc\_setD)$ 

```

```

subgoal
  using that by (simp add: n_ps_def)
  apply (erule state_preservation_updI)
  by (rule state_preservation_fold_updI)
then show ?thesis
  using assms
  by (auto dest: inv_renum_sem_D simp: renum_reset_map_u simp del: map_map_index
set_map)
qed

method solve_triv =
  assumption
| erule (1) bounded_renumD'; fail
| rule inv_renum_sem_D[OF __ sem_states_loc_setD]; simp; fail
| rule check_bexp_renumI; simp; fail
| rule map_u_renum_constraint_clock_valD; simp; fail
| rule is_upd_renumI is_upd_renumI'' is_upds_renumI' is_upds_renumI'' is_upds_concat_renumI'',
  simp; fail
| simp; fail
| simp add:
  vars_inv_def bij_f_the_inv_f[OF bij_renum_vars] renum_reset_map_u[symmetric] map_index_update
  renum_reset_append;
  fail
| subst nth_map, (simp; fail)+; fail
| (rule state_preservation_updI, blast)+, simp; fail

lemma trans_sem_upd_domI:
  assumes (L ! p, b', g', a, f', r', l1) ∈ trans (N p) dom s = var_set p < n_ps
  shows ∀ (x, _) ∈ set (renum_upds f'). x ∈ dom (s o vars_inv)
  unfolding renum_upd_def using assms
  by (auto simp: dom_comp_eq vimage vars_inv_def the_inv_f_f[OF inj_renum_vars]
    dest!: sem_trans_upd_domD)

lemma step_single_renumI:
  assumes
    step_u renum.sem
    (map_index renum_states L) (s o vars_inv) (map_u u) a L' s' u'
    L ∈ states dom s ⊆ var_set dom s = var_set
  shows ∃ a1 L1 s1 u1. step_u sem L s u a1 L1 s1 u1 ∧ renum_label a1 = a ∧
    L' = map_index renum_states L1 ∧ s' = s1 o vars_inv ∧ u' = map_u u1
  using assms(1-3)
  supply [simp] = length_automata_eq_n_ps L_len
  supply [simp del] = map_map_index set_map
  apply (subst A_split)
  apply (subst (asm) renum.A_split)
  apply (simp only: renum_label_def label.map renum_n_ps_simp)

  using [[goals_limit=2]]

  apply (cases a)

  — Delay
subgoal
  apply (simp only:)

```

```

apply (erule step_u_elims')
apply intros
apply (rule step_u.intros)
preferT <"nonnegative">
  apply assumption
preferT <"urgent"> subgoal
  using urgency_removed by - (tag, auto)
preferT <"target invariant">
  apply (solves <tag, simp, prop_monos+, solve_triv+>)
apply (tag, solve_triv)+
done

```

— Internal

```

subgoal for a'
apply (simp only:)
apply (erule step_u_elims')
subgoal premises prems[tagged] for l b g f r l' p
using prems apply tag
using sym[OF [[get_tagged <"loc">]]]
usingT TRANS _ <"range">
apply simp
apply (drule (1) trans_sem_N_renumI', untag)
apply elims
unfolding renum_act_def
apply (rule SilD, assumption)

```

```

apply intros
apply (rule step_u.intros(2))

```

```

preferT <TRANS "silent"> apply (tag, solve_triv)

```

```

preferT <"committed">
apply (solves <tag, prop_monos; auto simp: committed_renum_eq dest: injD[OF inj_renum_states]>)

```

```

preferT <"bexp"> apply (tag, solve_triv)
preferT <"guard"> apply (tag, solve_triv)

```

```

preferT <"target invariant">
apply (solves <tag "new valuation" "new loc",
  simp add: map_index_update[symmetric] renum_reset_map_u, prop_monos+,
  rule inv_renum_sem_D[OF _ _ sem_states_loc_setD],
  solve_triv+>)

```

```

preferT <"is_upd">
apply (tag, solve_triv)

```

```

preferT <"bounded"> subgoal
apply (tag <"is_upd">, erule bounded_renumD1)
apply (drule is_upds_dom)
subgoal
apply (simp add: dom_comp_eq_vimage)
unfolding renum_upd_def
apply (clarsimp simp: vars_inv_def the_inv_f_f[OF inj_renum_vars] set_map)
apply (drule (1) sem_trans_upd_domD)

```

```

    using assms(4)
    apply auto
    done
  apply (drule dom_comp_vars_inv_eqD, simp)
  done

  apply (tag, solve_triv)+
  usingT <"new loc"> apply solve_triv

  apply (rule ext)
  usingT <"new valuation"> apply solve_triv+
  done
done

— Binary
subgoal for a'
  apply (simp only;)
  apply (erule step_u_elims')
  subgoal premises prems[tagged] for l1 b1 g1 f1 r1 l1' p l2 b2 g2 f2 r2 l2' q s'a
  using prems apply tag
  apply (tag RECV "loc" SEND "loc")
  apply (drule sym[of map_index renum_states L ! _])+
  apply (tag TRANS _ RECV "range" SEND "range")
  apply simp
  apply (drule (1) trans_sem_N_renumI')
  apply (drule (1) trans_sem_N_renumI', untag)
  apply elims
  apply (rule In_OutD, assumption, assumption)

  apply intros
    apply (rule step_u.intros(3))
      preferT <"not broadcast">
      apply (tag, simp add: renum_sem_broadcast_eq broadcast_def)
  using inj_renum_acts
  apply auto[1]
  preferT <TRANS "in"> apply (tag, solve_triv)
  preferT <TRANS "out"> apply (tag, solve_triv)
  preferT <"committed"> subgoal
    by (tag, auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states])
  preferT <"target invariant">
  apply (
    solves <tag "new valuation" "new loc",
    simp add: map_index_update[symmetric] renum_reset_map_u renum_reset_append,
    prop_monos+,
    rule inv_renum_sem_D[OF _ _ sem_states_loc_setD],
    solve_triv+>)

  preferT <"upd"> apply (tag, solve_triv)
  preferT <"upd"> apply (tag, solve_triv)

  preferT <"bounded"> subgoal
    usingT— <"upd">
    apply —

```

```

apply (tag, erule bounded_renumD1)
apply (drule is_upds_dom)
subgoal
  using assms(4) by – (drule trans_sem_upd_domI; simp split: prod.splits)
apply (drule is_upds_dom)
subgoal
  using assms(4) by – (drule trans_sem_upd_domI; simp split: prod.splits)
apply (simp, drule dom_comp_vars_inv_eqD, simp)
done
apply (tag, solve_triv)+
apply (tag "new loc", solve_triv)
apply (rule ext; solve_triv)
apply (tag "new valuation", solve_triv)
done
done

```

— Broadcast

```

subgoal for a'
  apply (simp only:)
  apply (erule step_u_elims')
  subgoal premises prems[tagged] for l b g f r l' p ps bs gs fs rs ls' s'a
  using prems apply (tag SEND "range" TRANS "out")
  using sym[OF [[get_tagged <SEND "loc">]]]
  apply simp
  apply (frule (1) trans_sem_N_renumI')
  apply elims
  subgoal premises prems[tagged] for b' g' a'a f' r' l1
  using prems(1–6) using T– TRANS "in" SEL _ <"renum action">
  unfolding renum_act_def
  apply –
  apply (rule OutD, assumption)
  apply (simp add: atLeastLessThan_upperD)
  apply (erule(1) trans_sem_N_renum_broadcastI)
  apply (subgoal_tac map fs ps = map renum_upds (map fs' ps))
  defer
  apply (simp; fail)

```

**subgoal for** a1 bs' gs' fs' rs' ls1 a2

```

  apply intros
  apply (rule step_u.intros(4)[where ps = ps])

  prefer T <TRANS "out"> apply (tag <TRANS "orig">, solve_triv)

  prefer T <"broadcast">
  apply (tag,
    clarsimp simp: renum_sem_broadcast_eq inj_eq_iff[OF inj_renum_acts] broadcast_def;
    fail)

  prefer T <TRANS "in">
  apply (tag,
    cases ps = []; simp add: atLeastLessThan_upperD inj_eq_iff[OF inj_renum_acts]; fail)

  prefer T <"committed"> subgoal

```

```

    apply (tag, prop_monos)
    apply (solves ⟨auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states]⟩)
    apply prop_monos
    subgoal
      by (auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states] atLeast-
LessThan_upperD)
    subgoal premises prems
      using ⟨L ∈ states⟩ prems(21)
      by (auto simp: inj_eq_iff[OF inj_renum_states] committed_renum_eq)
    done

preferT ⟨"bexp"⟩ apply (tag ⟨"renum bexp"⟩, solve_triv)

preferT ⟨"bexp"⟩ apply (tag, erule Ball_mono, solve_triv; fail)

preferT ⟨"guard"⟩ apply (tag ⟨"renum guard"⟩, solve_triv)

preferT ⟨"guard"⟩ apply (tag, erule Ball_mono, solve_triv)

preferT ⟨"maximal"⟩ subgoal
  apply tag
  apply simp
  apply prop_monos+
  apply clarify
  apply (drule (1) trans_N_renumD)+
  apply (simp add: renum_act_def)
  apply (drule check_bexp_renumD)
  apply (drule map_u_renum_cconstraint_clock_valI)
  apply blast
done

preferT ⟨"target invariant"⟩
  apply (tag "new loc" "new valuation" "renum reset" "renum loc" TRANS _)
  apply (simp add: atLeastLessThan_upperD)
  apply (solves ⟨erule (2) inv_sem_N_renum_broadcastI, blast, fast, solve_triv⟩)

preferT ⟨"upds"⟩
  apply (tag "renum upds", simp only:, solve_triv)

preferT ⟨"bounded"⟩ subgoal
  apply (tag "upd" "upds" "renum upds", simp only:)
  apply (erule bounded_renumD1)
  apply (drule is_upds_dom)
  subgoal
    using assms(4) usingT ⟨TRANS "orig"⟩ by (intro trans_sem_upd_domI)

  apply (drule is_upds_dom)
  subgoal
    using assms(4) usingT ⟨"upd"⟩ ⟨"upds"⟩
    apply (simp add: set_map)
    apply (erule Ball_mono, drule trans_sem_upd_domI, assumption)
    apply (simp add: atLeastLessThan_upperD set_map; fail)+
  done
  apply (simp, drule dom_comp_vars_inv_eqD, simp)

```

```

done

preferT ‹"upd"› apply (tag "renum upds", solve_triv)

apply (tag, solve_triv)+

apply (tag "new loc" "renum loc",
  simp add: map_trans_broad_aux1 map_index_update cong: fold_cong; fail)

apply (solves ‹auto simp: vars_inv_def bij_f_the_inv_f[OF bij_renum_vars]›)
apply (tag "new valuation" "renum reset",
  simp add: renum_reset_map_u[symmetric] renum_reset_def map_concat comp_def
cong: map_cong;
  fail)
done
done
done
done
done
done

lemma step_u_var_set_invariant:
  assumes step_u sem L s u a L' s' u' dom s = var_set
  shows dom s' = var_set
  using assms dom_bounds_var_set by (auto 4 4 dest!: bounded_inv simp: bounded_def)

lemmas step_u_invariants = states_inv step_u_var_set_invariant

interpretation single: Bisimulation_Invariant
  λ(L, s, u) (L', s', u'). ∃ a. step_u sem L s u a L' s' u'
  λ(L, s, u) (L', s', u'). ∃ a. step_u renum.sem L s u a L' s' u'
  λ(L, s, u) (L', s', u'). L' = map_index renum_states L ∧ s' = s o vars_inv ∧ u' = map_u u
  λ(L, s, u). L ∈ states ∧ dom s = var_set λ_. True
  apply standard
  supply [simp del] = map_map_index set_map
  apply clarsimp
  subgoal for L s u L' s' u' a
    by (drule (1) step_single_renumD, auto)
  subgoal
    by clarsimp (drule step_single_renumI[rotated]; blast)
  subgoal
    by clarsimp (intro conjI; elim step_u_invariants)
  subgoal
    .
done

interpretation Bisimulation_Invariant
  λ(L, s, u) (L', s', u'). step_u' sem L s u L' s' u'
  λ(L, s, u) (L', s', u'). step_u' renum.sem L s u L' s' u'
  λ(L, s, u) (L', s', u'). L' = map_index renum_states L ∧ s' = s o vars_inv ∧ u' = map_u u
  λ(L, s, u). L ∈ states ∧ dom s = var_set λ_. True
proof -
  define rsem where rsem = renum.sem
  note step_single_renumD = step_single_renumD[folded rsem_def]
  have rsem ⊢ ‹map_index renum_states L, (s o Simple_Network_Rename_Defs.vars_inv)

```

```

renum_vars, map_u u) → ⟨map_index renum_states L', (s' ∘ Simple_Network_Rename_Defs.vars_inv)
renum_vars, map_u u⟩
  if sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩
    and L ∈ states
    and dom s = var_set
  for L :: 's list
    and s :: 'x ⇒ int option
    and u :: 'c ⇒ real
    and L' :: 's list
    and s' :: 'x ⇒ int option
    and u' :: 'c ⇒ real
  proof -
    from that(1) obtain L1 s1 u1 a where
      sem ⊢ ⟨L, s, u⟩ →Del ⟨L1, s1, u1⟩ a ≠ Del sem ⊢ ⟨L1, s1, u1⟩ →a ⟨L', s', u'⟩
    by (elim step_u'_elims)
  with that(2-) show ?thesis
    apply -
    apply (rule step_u'.intros[where a = renum_label a])
    apply (erule (1) step_single_renumD[where a = Del, unfolded renum_label_def, simpli-
fied], blast)
    apply (cases a; (fast | simp add: renum_label_def))
    apply (erule step_single_renumD)
    apply (blast dest: step_u_invariants)+
  done
qed
moreover have ∃ a aa b. sem ⊢ ⟨L, s, u⟩ → ⟨a, aa, b⟩ ∧ L' = map_index renum_states a ∧
s' = (aa ∘ Simple_Network_Rename_Defs.vars_inv) renum_vars ∧ u' = map_u b
  if L ∈ states
    and dom s = var_set
    and rsem ⊢ ⟨map_index renum_states L, (s ∘ Simple_Network_Rename_Defs.vars_inv)
renum_vars, map_u u⟩ → ⟨L', s', u'⟩
  for L :: 's list
    and s :: 'x ⇒ int option
    and u :: 'c ⇒ real
    and L' :: nat list
    and s' :: nat ⇒ int option
    and u' :: nat ⇒ real
  proof -
    from that(3) obtain L1 s1 u1 a where
      rsem ⊢ ⟨map_index renum_states L, (s ∘ vars_inv), map_u u⟩ →Del ⟨L1, s1, u1⟩
      a ≠ Del rsem ⊢ ⟨L1, s1, u1⟩ →a ⟨L', s', u'⟩
    by (elim step_u'_elims)
  with that(1,2) show ?thesis
    apply -
    apply (drule (1) step_single_renumI[folded rsem_def], blast)
    apply safe
    subgoal premises prems for a1 L1a s1a u1a
      proof -
        { fix a1a L1b s1b u1b
          have
            L ∈ states ⇒
            dom s = var_set ⇒
            renum_label a1a ≠
            Simple_Network_Language.label.Del ⇒

```

```

sem ⊢ ⟨L, s, u⟩ →a1 ⟨L1a, s1a, u1a⟩ ⇒
renum_label a1 =
Simple_Network_Language.label.Del ⇒
L1 = map_index renum_states L1a ⇒
u1 = map_u u1a ⇒
s1 =
(s1a ∘ Simple_Network_Rename_Defs.vars_inv)
renum_vars ⇒
sem ⊢ ⟨L1a, s1a, u1a⟩ →a1a ⟨L1b, s1b, u1b⟩ ⇒
a = renum_label a1a ⇒
L' = map_index renum_states L1b ⇒
s' =
(s1b ∘ Simple_Network_Rename_Defs.vars_inv)
renum_vars ⇒
u' = map_u u1b ⇒
a1 = Simple_Network_Language.label.Del
by (cases a1; simp add: renum_label_def)
} note * = this
from prems show ?thesis
  apply -
  apply (drule step_single_renumI[folded rsem_def])
  apply (blast dest: step_u_invariants)+
  apply (subgoal_tac a1 = Del)
  apply clarsimp
  apply intros
  apply (erule step_u'.intros)
  apply (auto intro: *)
done
qed
done
qed
moreover have x1b ∈ states ∧ dom x1c = var_set
  if x1 ∈ states
    and dom x1a = var_set
    and sem ⊢ ⟨x1, x1a, x2a⟩ → ⟨x1b, x1c, x2c⟩
  for x1 :: 's list
    and x1a :: 'x ⇒ int option
    and x2a :: 'c ⇒ real
    and x1b :: 's list
    and x1c :: 'x ⇒ int option
    and x2c :: 'c ⇒ real
  using that by (elim step_u'_elims) (auto 4 4 dest: step_u_invariants)
moreover note * = calculation
show Bisimulation_Invariant
  (λ(L, s, u) (L', s', u'). sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
  (λ(L, s, u) (L', s', u'). renum.sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
  (λ(L, s, u) (L', s', u').
    L' = map_index renum_states L ∧
    s' = (s o vars_inv) ∧
    u' = map_u u)
  (λ(L, s, u). L ∈ states ∧ dom s = var_set) (λ_. True)
unfolding rsem_def[symmetric]
apply (standard; clarsimp split: prod.splits)
by (rule *; assumption)+

```

qed

**interpretation** *Bisimulation\_Invariant*

$\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ sem } L \ s \ u \ L' \ s' \ u'$   
 $\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ renum.sem } L \ s \ u \ L' \ s' \ u'$   
 $\lambda(L, s, u) (L', s', u'). L' = \text{map\_index renum\_states } L \wedge s' = s \circ \text{vars\_inv} \wedge u' = \text{map\_}u \ u$   
 $\lambda(L, s, u). L \in \text{states} \wedge \text{dom } s = \text{var\_set } \lambda\_ . \text{True}$

**proof** –

**define** *rsem* **where** *rsem* = *renum.sem*

**note** *step\_single\_renumD* = *step\_single\_renumD*[folded *rsem\_def*]

**have** *rsem*  $\vdash \langle \text{map\_index renum\_states } L, (s \circ \text{Simple\_Network\_Rename\_Defs.vars\_inv})$   
 $\text{renum\_vars}, \text{map\_}u \ u \rangle \rightarrow \langle \text{map\_index renum\_states } L', (s' \circ \text{Simple\_Network\_Rename\_Defs.vars\_inv})$   
 $\text{renum\_vars}, \text{map\_}u \ u' \rangle$

**if** *sem*  $\vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$

**and**  $L \in \text{states}$

**and**  $\text{dom } s = \text{var\_set}$

**for**  $L :: 's \text{ list}$

**and**  $s :: 'x \Rightarrow \text{int option}$

**and**  $u :: 'c \Rightarrow \text{real}$

**and**  $L' :: 's \text{ list}$

**and**  $s' :: 'x \Rightarrow \text{int option}$

**and**  $u' :: 'c \Rightarrow \text{real}$

**proof** –

**from** *that*(1) **obtain**  $L1 \ s1 \ u1 \ a$  **where**

*sem*  $\vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L1, s1, u1 \rangle \ a \neq \text{Del}$  *sem*  $\vdash \langle L1, s1, u1 \rangle \rightarrow_a \langle L', s', u' \rangle$

**by** (*elim step\_u'\_elims*)

**with** *that*(2–) **show** *?thesis*

**apply** –

**apply** (*rule step\_u'.intros*[**where**  $a = \text{renum\_label } a$ ])

**apply** (*erule* (1) *step\_single\_renumD*[**where**  $a = \text{Del}$ , *unfolded renum\_label\_def*, *simplified*], *blast*)

**apply** (*cases*  $a$ ; (*fast* | *simp add: renum\_label\_def*))

**apply** (*erule step\_single\_renumD*)

**apply** (*blast dest: step\_u\_invariants*) +

**done**

qed

**moreover** **have**  $\exists a \ aa \ b. \text{sem} \vdash \langle L, s, u \rangle \rightarrow \langle a, aa, b \rangle \wedge L' = \text{map\_index renum\_states } a \wedge$   
 $s' = (aa \circ \text{Simple\_Network\_Rename\_Defs.vars\_inv}) \text{ renum\_vars} \wedge u' = \text{map\_}u \ b$

**if**  $L \in \text{states}$

**and**  $\text{dom } s = \text{var\_set}$

**and** *rsem*  $\vdash \langle \text{map\_index renum\_states } L, (s \circ \text{Simple\_Network\_Rename\_Defs.vars\_inv})$   
 $\text{renum\_vars}, \text{map\_}u \ u \rangle \rightarrow \langle L', s', u' \rangle$

**for**  $L :: 's \text{ list}$

**and**  $s :: 'x \Rightarrow \text{int option}$

**and**  $u :: 'c \Rightarrow \text{real}$

**and**  $L' :: \text{nat list}$

**and**  $s' :: \text{nat} \Rightarrow \text{int option}$

**and**  $u' :: \text{nat} \Rightarrow \text{real}$

**proof** –

**from** *that*(3) **obtain**  $L1 \ s1 \ u1 \ a$  **where**

*rsem*  $\vdash \langle \text{map\_index renum\_states } L, (s \circ \text{vars\_inv}), \text{map\_}u \ u \rangle \rightarrow_{\text{Del}} \langle L1, s1, u1 \rangle$

$a \neq \text{Del}$  *rsem*  $\vdash \langle L1, s1, u1 \rangle \rightarrow_a \langle L', s', u' \rangle$

**by** (*elim step\_u'\_elims*)

**with** *that*(1,2) **show** *?thesis*

```

apply -
apply (drule (1) step_single_renumI[folded rsem_def], blast)
apply safe
subgoal premises prems for a1 L1a s1a u1a
proof -
  {fix a1a L1b s1b u1b
    have
      L ∈ states ⇒
      dom s = var_set ⇒
      renum_label a1a ≠
      Simple_Network_Language.label.Del ⇒
      sem ⊢ ⟨L, s, u⟩ →a1 ⟨L1a, s1a, u1a⟩ ⇒
      renum_label a1 =
      Simple_Network_Language.label.Del ⇒
      L1 = map_index renum_states L1a ⇒
      u1 = map_u u1a ⇒
      s1 =
      (s1a ∘ Simple_Network_Rename_Defs.vars_inv)
      renum_vars ⇒
      sem ⊢ ⟨L1a, s1a, u1a⟩ →a1a ⟨L1b, s1b, u1b⟩ ⇒
      a = renum_label a1a ⇒
      L' = map_index renum_states L1b ⇒
      s' =
      (s1b ∘ Simple_Network_Rename_Defs.vars_inv)
      renum_vars ⇒
      u' = map_u u1b ⇒
      a1 = Simple_Network_Language.label.Del
      by (cases a1; simp add: renum_label_def)
    } note ** = this
from prems show ?thesis
  apply -
  apply (drule step_single_renumI[folded rsem_def])
  apply (blast dest: step_u_invariants)+
  apply (subgoal_tac a1 = Del)
  apply clarsimp
  apply intros
  apply (erule step_u'.intros)
  apply (auto intro: **)
  done
qed
done
qed
moreover have x1b ∈ states ∧ dom x1c = var_set
if x1 ∈ states
  and dom x1a = var_set
  and sem ⊢ ⟨x1, x1a, x2a⟩ → ⟨x1b, x1c, x2c⟩
for x1 :: 's list
  and x1a :: 'x ⇒ int option
  and x2a :: 'c ⇒ real
  and x1b :: 's list
  and x1c :: 'x ⇒ int option
  and x2c :: 'c ⇒ real
using that by (elim step_u'_elims) (auto 4 4 dest: step_u_invariants)
moreover note * = calculation

```

```

show Bisimulation_Invariant
  ( $\lambda(L, s, u) (L', s', u'). \text{sem} \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ )
  ( $\lambda(L, s, u) (L', s', u'). \text{renum.sem} \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ )
  ( $\lambda(L, s, u) (L', s', u').$ 
     $L' = \text{map\_index renum\_states } L \wedge$ 
     $s' = (s \text{ o vars\_inv}) \wedge$ 
     $u' = \text{map\_u } u$ )
  ( $\lambda(L, s, u). L \in \text{states} \wedge \text{dom } s = \text{var\_set} \wedge \lambda\_ . \text{True}$ )
unfolding rsem_def[symmetric]
apply (standard; clarsimp split: prod.splits)
by (rule *; assumption)+
qed

lemmas renum_bisim = Bisimulation_Invariant_axioms

end

locale Simple_Network_Impl_Formula_real =
  Simple_Network_Rename_real where automata = automata
for automata ::
  ('s list  $\times$  's list
     $\times$  (('a :: countable) act, 's, 'c, real, 'x :: countable, int) transition list
     $\times$  (('s :: countable)  $\times$  ('c :: countable, real) cconstraint) list) list +
fixes  $\Phi :: (\text{nat}, 's, 'x, \text{int}) \text{ formula}$ 
and  $s_0 :: ('x \times \text{int}) \text{ list}$ 
and  $L_0 :: 's \text{ list}$ 
begin

definition  $\Phi'$  where
   $\Phi' = \text{map\_formula renum\_states renum\_vars id } \Phi$ 

definition  $a_0$  where
   $a_0 = (L_0, \text{map\_of } s_0, \lambda\_ . 0)$ 

definition  $a_0'$  where
   $a_0' = (\text{map\_index renum\_states } L_0, \text{map\_of } (\lambda(x, v). (\text{renum\_vars } x, v)) s_0), \lambda\_ . 0)$ 

context
assumes  $L_0\_states: L_0 \in \text{states}$ 
assumes  $s_0\_dom: \text{fst } s_0 = \text{var\_set}$  and  $s_0\_distinct: \text{distinct } (\text{map fst } s_0)$ 
begin

lemma state_eq_aux:
assumes  $x \notin \text{renum\_vars} \wedge \text{var\_set}$ 
shows  $\text{vars\_inv } x \notin \text{var\_set}$ 
unfolding vars_inv_def
proof safe
assume  $\text{the\_inv renum\_vars } x \in \text{var\_set}$ 
then have  $\text{renum\_vars } (\text{the\_inv renum\_vars } x) = x$ 
by (intro f_the_inv_f inj_renum_vars) (simp add: surj_renum_vars)
with assms  $\langle \_ \in \text{var\_set} \rangle$  show False
by force
qed

```

```

lemma state_eq:
  assumes fst ‘ set  $s_0 = \text{var\_set } \text{distinct } (\text{map } \text{fst } s_0)$ 
  shows  $\text{map\_of } (\text{map } (\lambda(x, y). (\text{renum\_vars } x, y)) s_0) = (\text{map\_of } s_0 \circ \circ \circ \text{the\_inv\_into}) \text{ UNIV }$ 
  renum\_vars
  (is  $?l = ?r$ )
proof (rule ext)
  fix  $x$ 
  show  $?l \ x = ?r \ x$ 
  proof (cases  $x \in \text{renum\_vars}$  ‘ fst ‘ set  $s_0$ )
    case True
    then show ?thesis
    apply clarsimp
    apply (subst map_of_mapk_SomeI)
    subgoal
      using inj_renum_vars by (auto intro: inj_on_subset)
      apply (rule map_of_is_SomeI, rule assms, assumption)
      apply (simp add: the_inv_ff[OF inj_renum_vars] s0_distinct)
      done
    next
    case False
    then have  $\text{vars\_inv } x \notin \text{fst } \text{‘ set } s_0$ 
      using state_eq_aux assms(1) unfolding vars_inv_def by auto
    with False show ?thesis
    apply –
    apply (frule map_of_NoneI)
    apply (simp add: vars_inv_def)
    apply (auto simp: map_of_eq_None_iff)
    done
  qed
qed

```

**interpretation** *Bisimulation\_Invariant*

```

 $\lambda(L, s, u) (L', s', u'). \text{step\_u' sem } L \ s \ u \ L' \ s' \ u'$ 
 $\lambda(L, s, u) (L', s', u'). \text{step\_u' renum.sem } L \ s \ u \ L' \ s' \ u'$ 
 $\lambda(L, s, u) (L', s', u'). L' = \text{map\_index renum\_states } L \wedge s' = s \circ \text{vars\_inv} \wedge u' = \text{map\_u } u$ 
 $\lambda(L, s, u). L \in \text{states} \wedge \text{dom } s = \text{var\_set } \lambda\_.$  True
by (rule renum_bisim)

```

**lemma** *start\_equiv*:

```

 $A\_B.\text{equiv}' a_0 \ a_0'$ 
unfolding  $A\_B.\text{equiv}'\_def a_0\_def a_0'\_def$ 
apply (clarsimp simp: vars_inv_def, intro conjI)
subgoal
  by (intro state_eq s0_dom s0_distinct)
subgoal
  unfolding  $\text{map\_u\_def}$  by auto
subgoal
  by (rule L0_states)
subgoal
  using  $s_0\_dom \text{ dom\_map\_of\_conv\_image\_fst}[of s_0]$  by fastforce
done

```

**lemma** *check\_serp\_equiv*:

```

assumes  $A\_B.equiv' (L, s, u) (L', s', u') \text{ locs\_of\_sexp } e \subseteq \{0..<n\_ps\}$ 
shows
   $check\_sexp\ e\ L\ (the \circ s) \longleftrightarrow$ 
   $check\_sexp\ (map\_sexp\ renum\_states\ renum\_vars\ id\ e)\ L'\ (the \circ s')$ 
using assms unfolding  $A\_B.equiv'\_def$ 
by (induction  $e$ )
  (simp add:  $inj\_eq\ states\_lengthD\ renum\_states\_inj\ vars\_inv\_def\ the\_inv\_f\_f[OF\ inj\_renum\_vars]$ ) +

lemma models_iff:
   $sem, a_0 \models \Phi = renum.sem, a_0' \models \Phi'$  if  $locs\_of\_formula\ \Phi \subseteq \{0..<n\_ps\}$ 
proof –
  have  $rel\_ctl\_formula\ compatible\ (ctl\_of\ \Phi)\ (ctl\_of\ \Phi')$ 
    using that unfolding  $\Phi'\_def$ 
    by (cases  $\Phi$ ; auto simp:  $check\_sexp\_equiv\ prop\_of\_def\ rel\_fun\_def$ )
  with start_equiv show ?thesis
    by (simp add:  $models\_ctl\_iff\ CTL\_compatible[THEN\ rel\_funD,\ symmetric]$ )
qed

lemma has_deadlock_iff:
   $has\_deadlock\ sem\ a_0 \longleftrightarrow has\_deadlock\ renum.sem\ a_0'$ 
  unfolding  $has\_deadlock\_def$  using start_equiv by (intro deadlock_iff, unfold  $A\_B.equiv'\_def$ )
auto

end

end

lemma Bisimulation_Invariants_Bisimulation_Invariant:
  assumes  $Bisimulation\_Invariants\ A\ B\ sim\ PA\ PA\ PB\ PB$ 
  shows  $Bisimulation\_Invariant\ A\ B\ sim\ PA\ PB$ 
proof –
  interpret  $Bisimulation\_Invariants\ A\ B\ sim\ PA\ PA\ PB\ PB$ 
    by (rule assms)
  show ?thesis
    by (standard; blast intro:  $A\_B\_step\ B\_A\_step$ )
qed

lemma Bisimulation_Invariants_Bisimulation_Invariant_iff:
   $Bisimulation\_Invariants\ A\ B\ sim\ PA\ PA\ PB\ PB \longleftrightarrow Bisimulation\_Invariant\ A\ B\ sim\ PA\ PB$ 
  using
     $Bisimulation\_Invariants\_Bisimulation\_Invariant\ Bisimulation\_Invariant\_Bisimulation\_Invariants$ 
  by blast

lemmas  $Bisimulation\_Invariant\_composition =$ 
   $Bisimulation\_Invariant\_Invariants\_composition[$ 
     $THEN\ Bisimulation\_Invariants\_Bisimulation\_Invariant,$ 
     $OF\ \_Bisimulation\_Invariant\_Bisimulation\_Invariants]$ 

lemma Bisimulation_Invariant_refl:
   $Bisimulation\_Invariant\ A\ A\ (=)\ P\ P$  if  $\bigwedge a\ b. P\ a \implies A\ a\ b \implies P\ b$ 
  by (rule  $Bisimulation\_Invariant.intro$ ) (auto intro: that)

```

```

locale Simple_Network_Rename_int =
  Simple_Network_Rename_Defs_int where automata = automata +
  Simple_Network_Rename' where automata = automata
for automata ::
  ('s list × 's list
   × (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list
   × (('s :: countable) × ('c :: countable, int) cconstraint) list) list +
  assumes urgency_removed:  $\forall (\_, U, \_, \_) \in \text{set } \textit{automata}. U = []$ 
begin

lemma n_ps_eq1:
  Prod_TA_Defs.n_ps
    (set broadcast, map automaton_of (map conv_automaton automata),
     map_of bounds') = n_ps
  unfolding n_ps_def Prod_TA_Defs.n_ps_def by simp

lemma var_set_eq1:
  Prod_TA_Defs.var_set
    (set broadcast, map automaton_of (map conv_automaton automata),
     map_of bounds') = var_set
  unfolding sem_def sem_var_set_eq[symmetric] by simp

lemma urgency_removed':
   $\forall i < n\_ps. \text{urgent}$ 
  (Prod_TA_Defs.N (set broadcast, map automaton_of (map conv_automaton automata), map_of
  bounds') i)
  = {}
  unfolding urgent_def n_ps_def Prod_TA_Defs.n_ps_def Prod_TA_Defs.N_def
  unfolding conv_automaton_def automaton_of_def
  using urgency_removed by (fastforce dest: nth_mem split: prod.split)

sublocale real: Simple_Network_Rename_real where automata = map conv_automaton au-
  tomata
  apply standard
  unfolding n_ps_eq1 var_set_eq1
  by (rule bij_renum_clocks renum_states_inj bij_renum_vars bounds'_var_set inj_renumActs
    urgency_removed') +

lemma sem_unfold1:
  real.sem = sem
  unfolding sem_def by simp

lemma var_set_eq:
  real.var_set = sem.var_set
  unfolding sem_unfold1[symmetric] ..

lemma map_acconstraint_conv_ac_commute:
  map_acconstraint renum_clocks id (conv_ac ac) = conv_ac (map_acconstraint renum_clocks
  id ac)
  by (cases ac; simp)

lemma map_ccconstraint_conv_cc_commute:
  renum_ccconstraint (conv_cc g) = conv_cc (renum_ccconstraint g)
  unfolding renum_ccconstraint_def map_ccconstraint_def by (simp add: map_acconstraint_conv_ac_commute)

```

**lemma** *rename\_conv\_automaton\_commute*:  
*real.renum\_automaton n (conv\_automaton x) = conv\_automaton (real.renum\_automaton n x)*  
**unfolding** *real.renum\_automaton\_def conv\_automaton\_def*  
**by** (*clarsimp split: prod.split simp: map\_ccconstraint\_conv\_cc\_commute*)

**lemma** *sem\_unfold2*:  
*real.renum.sem = renum.sem*  
**by** (*simp add: Simple\_Network\_Impl.sem\_def rename\_conv\_automaton\_commute*)

**sublocale** *renum\_bisim: Bisimulation\_Invariant*  
 $\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ sem } L \text{ s } u \text{ } L' \text{ s' } u'$   
 $\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ renum.sem } L \text{ s } u \text{ } L' \text{ s' } u'$   
 $\lambda(L, s, u) (L', s', u'). L' = \text{map\_index renum\_states } L \wedge s' = s \text{ o vars\_inv} \wedge u' = \text{map\_}u \text{ } u$   
 $\lambda(L, s, u). L \in \text{sem.states} \wedge \text{dom } s = \text{var\_set } \lambda\_ \text{. True}$   
**apply** (*rule Bisimulation\_Invariant\_sim\_replace*)  
**apply** (*rule Bisimulation\_Invariant\_composition*)  
**apply** (*rule real.renum\_bisim[unfolded sem\_unfold1 sem\_unfold2 sem\_var\_set\_eq]*)  
**apply** (*rule Bisimulation\_Invariant\_refl*)  
**apply** *auto*  
**done**

**lemmas** *renum\_bisim = renum\_bisim.Bisimulation\_Invariant\_axioms*

**end**

**locale** *Simple\_Network\_Rename\_Start'* =  
*Simple\_Network\_Rename'* **where** *automata = automata*  
**for** *automata* ::  
 $('s \text{ list} \times 's \text{ list}$   
 $\times (('a :: \text{countable}) \text{ act}, 's, 'c, 't, 'x :: \text{countable}, \text{int}) \text{ transition list}$   
 $\times (('s :: \text{countable}) \times ('c :: \text{countable}, 't) \text{ cconstraint}) \text{ list}) \text{ list} +$   
**fixes** *s<sub>0</sub> :: ('x × int) list*  
**and** *L<sub>0</sub> :: 's list*  
**assumes** *L<sub>0</sub>\_states: L<sub>0</sub> ∈ states*  
**assumes** *s<sub>0</sub>\_dom: fst ' set s<sub>0</sub> = var\_set and s<sub>0</sub>\_distinct: distinct (map fst s<sub>0</sub>)*  
**begin**

**end**

**locale** *Simple\_Network\_Rename\_Start\_int* =  
*Simple\_Network\_Rename\_int* **where** *automata = automata +*  
*Simple\_Network\_Rename\_Start'* **where** *automata = automata*  
**for** *automata* ::  
 $('s \text{ list} \times 's \text{ list}$   
 $\times (('a :: \text{countable}) \text{ act}, 's, 'c, \text{int}, 'x :: \text{countable}, \text{int}) \text{ transition list}$   
 $\times (('s :: \text{countable}) \times ('c :: \text{countable}, \text{int}) \text{ cconstraint}) \text{ list}) \text{ list}$   
**begin**

**definition** *a<sub>0</sub>* **where**  
 $a_0 = (L_0, \text{map\_of } s_0, \lambda\_ \text{. } 0)$

**definition** *a<sub>0</sub>'* **where**  
 $a_0' = (\text{map\_index renum\_states } L_0, \text{map\_of } (\text{map } (\lambda(x, v). (\text{renum\_vars } x, v)) \text{ } s_0), \lambda\_ \text{. } 0)$

```

lemma state_eq_aux:
  assumes  $x \notin \text{renum\_vars}$  '  $\text{var\_set}$ 
  shows  $\text{vars\_inv } x \notin \text{var\_set}$ 
  unfolding vars_inv_def
proof safe
  assume the_inv renum_vars  $x \in \text{var\_set}$ 
  then have renum_vars (the_inv renum_vars  $x$ ) =  $x$ 
    by (intro f_the_inv_f real.inj_renum_vars) (simp add: real.surj_renum_vars)
  with assms  $\langle \_ \in \text{var\_set} \rangle$  show False
    by force
qed

lemma state_eq:
  assumes fst '  $\text{set } s_0 = \text{var\_set}$  distinct (map fst  $s_0$ )
  shows map_of (map ( $\lambda(x, y). (\text{renum\_vars } x, y)$ )  $s_0$ ) = (map_of  $s_0 \circ \circ \circ \text{the\_inv\_into}$ ) UNIV
  renum_vars
  (is ?l = ?r)
proof (rule ext)
  fix  $x$ 
  show ?l  $x$  = ?r  $x$ 
proof (cases  $x \in \text{renum\_vars}$  ' fst '  $\text{set } s_0$ )
  case True
  then show ?thesis
    apply clarsimp
    apply (subst map_of_mapk_SomeI')
    subgoal
      using real.inj_renum_vars by (auto intro: inj_on_subset)
      apply (rule map_of_is_SomeI, rule assms, assumption)
      apply (simp add: the_inv_f_f[OF real.inj_renum_vars]  $s_0$ _distinct)
    done
  next
  case False
  then have vars_inv  $x \notin \text{fst ' set } s_0$ 
    using state_eq_aux assms(1) unfolding vars_inv_def by auto
  with False show ?thesis
    apply -
    apply (frule map_of_NoneI)
    apply (simp add: vars_inv_def)
    apply (auto simp: map_of_eq_None_iff)
  done
qed
qed

lemma start_equiv:
  renum_bisim.A_B.equiv'  $a_0$   $a_0'$ 
  unfolding renum_bisim.A_B.equiv'_def  $a_0$ _def  $a_0'$ _def
  apply (clarsimp simp: vars_inv_def, intro conjI)
  subgoal
    by (intro state_eq  $s_0$ _dom  $s_0$ _distinct)
  subgoal
    unfolding map_u_def by auto
  subgoal

```

```

    unfolding sem_states_eq by (rule L0_states)
  subgoal
    using s0_dom dom_map_of_conv_image fst[of s0] by fastforce
  done

lemma check_sexp_equiv:
  assumes renum_bisim.A_B.equiv' (L, s, u) (L', s', u') locs_of_sexp e  $\subseteq$  {0.. $n_{ps}$ }
  shows
    check_sexp e L (the  $\circ$  s)  $\longleftrightarrow$ 
    check_sexp (map_sexp renum_states renum_vars id e) L' (the  $\circ$  s')
  using assms unfolding renum_bisim.A_B.equiv'_def
  by (induction e)
    (simp add:
      inj_eq sem.states_lengthD renum_states_inj vars_inv_def the_inv_f_f[OF real.inj_renum_vars])+

lemma check_sexp_compatible:
  assumes locs_of_sexp e  $\subseteq$  {0.. $n_{ps}$ }
  shows renum_bisim.compatible
    ( $\lambda(L, s, u). \text{check\_sexp } e \text{ } L \text{ } (the \circ s)$ )
    ( $\lambda(L', s', u'). \text{check\_sexp } (map\_sexp \text{ renum\_states } renum\_vars \text{ id } e) \text{ } L' \text{ } (the \circ s')$ )
  using check_sexp_equiv[OF _ assms] by auto

lemma has_deadlock_iff:
  has_deadlock sem a0  $\longleftrightarrow$  has_deadlock renum.sem a0'
  unfolding has_deadlock_def using start_equiv
  by (intro renum_bisim.deadlock_iff, unfold renum_bisim.A_B.equiv'_def) auto

lemma state_formula_compatible:
  ( $\bigcup x \in \text{set\_state\_formula } \varphi. \text{locs\_of\_sexp } x \subseteq \{0.. $n_{ps}$ \} \implies$ 
  rel_state_formula renum_bisim.compatible
    (map_state_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (the \circ s)$ )  $\varphi$ )
    (map_state_formula ( $\lambda P (L, s, \_). \text{check\_sexp } (map\_sexp (\lambda p. \text{renum\_states } p) \text{ renum\_vars id } P) \text{ } L \text{ } (the \circ s)$ )
     $\varphi$ ) and path_formula_compatible:
  ( $\bigcup x \in \text{set\_path\_formula } \psi. \text{locs\_of\_sexp } x \subseteq \{0.. $n_{ps}$ \} \implies$ 
  rel_path_formula renum_bisim.compatible
    (map_path_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (the \circ s)$ )  $\psi$ )
    (map_path_formula ( $\lambda P (L, s, \_). \text{check\_sexp } (map\_sexp (\lambda p. \text{renum\_states } p) \text{ renum\_vars id } P) \text{ } L \text{ } (the \circ s)$ )
     $\psi$ )
  by (induction  $\varphi$  and  $\psi$ ) (auto simp: check_sexp_equiv prop_of_def rel_fun_def)

lemmas models_state_compatible = renum_bisim.models_state_compatible[OF state_formula_compatible]

end

locale Simple_Network_Rename_Formula_int =
  Simple_Network_Rename_Start_int where automata = automata
  for automata ::
    ('s list  $\times$  's list
     $\times$  (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list
     $\times$  (('s :: countable)  $\times$  ('c :: countable, int) cconstraint) list) list +
  fixes  $\Phi$  :: (nat, 's, 'x, int) formula
begin

```

**definition**  $\Phi'$  where

$\Phi' = \text{map\_formula } \text{renum\_states } \text{renum\_vars } \text{id } \Phi$

**lemma** *models\_iff*:

$\text{sem}, a_0 \models \Phi = \text{renum.sem}, a_0' \models \Phi'$  **if**  $\text{locs\_of\_formula } \Phi \subseteq \{0..<n\_ps\}$

**proof** –

**have**  $\text{rel\_ctl\_formula } \text{renum\_bisim.compatible } (\text{ctl\_of } \Phi) (\text{ctl\_of } \Phi')$

**using** *that unfolding  $\Phi'$ \_def*

**by** (cases  $\Phi$ ; *auto simp: check\_sexp\_equiv prop\_of\_def rel\_fun\_def*)

**with** *start\_equiv show ?thesis*

**by** (*simp add: models\_ctl\_iff renum\_bisim.CTL\_compatible[THEN rel\_funD, symmetric]*)

**qed**

**end**

**lemma** *vars\_of\_bexp\_finite[finite\_intros]*:

*finite (vars\_of\_bexp (b::('a, 'b) bexp))*

**and** *vars\_of\_exp\_finite[finite\_intros]*:

*finite (vars\_of\_exp (e::('a, 'b) exp))*

**by** (*induction b and e auto*)

**lemma** *set\_bexp\_vars\_of\_bexp*:

$\text{set\_bexp } (b::('a, 'b) \text{ bexp}) = \text{vars\_of\_bexp } b$

**and** *set\_exp\_vars\_of\_exp*:

$\text{set\_exp } (e::('a, 'b) \text{ exp}) = \text{vars\_of\_exp } e$

**by** (*induction b and e auto*)

**definition** (*in Prod\_TA\_Defs*)

$\text{act\_set} \equiv (\bigcup p \in \{0..<n\_ps\}. \bigcup (l, e, g, a, \_) \in \text{trans } (N \ p). \text{act.set\_act } a) \cup \text{broadcast}$

**lemma** (*in Simple\_Network\_Impl*) *act\_set\_compute*:

$\text{act\_set} =$

$\bigcup ((\lambda(\_, \_, t, \_). \bigcup ((\lambda(l, e, g, a, \_). \text{act.set\_act } a) \text{ ‘ set } t)) \text{ ‘ set automata}) \cup \text{set broadcast}$

**unfolding** *act\_set\_def*

**apply** (*fo\_rule arg\_cong2*)

**apply** (*clarsimp simp: N\_p\_trans\_eq n\_ps\_def act\_set\_def broadcast\_def*)

**apply** *safe*

**apply** (*clarsimp simp: N\_p\_trans\_eq n\_ps\_def act\_set\_def broadcast\_def*)

**apply** (*drule nth\_mem, erule bexI[rotated], force*)

**apply** (*drule mem\_nth, force*)

**unfolding** *broadcast\_def*

**apply** *simp+*

**done**

**lemma** *set1\_acconstraint\_elim*:

**assumes**  $c \in \text{set1\_acconstraint } ac$

**obtains**  $x$  **where**  $(c, x) = \text{constraint\_pair } ac$

**using** *assms by (cases ac) auto*

**lemma** (*in Simple\_Network\_Impl*)

**assumes**  $(x1, x2, T, I) \in \text{set automata } (l, b, g, a, f, r, l') \in \text{set } T$

```

shows  $clk\_set'I1[intro]: c \in set\ r \implies c \in clk\_set'$ 
and  $clk\_set'I2[intro]: ac \in set\ g \implies c \in set1\_aconstraint\ ac \implies c \in clk\_set'$ 
and  $loc\_setI1[intro]: l \in loc\_set$  and  $loc\_setI2[intro]: l' \in loc\_set$ 
and  $act\_setI[intro]: a' \in set\_act\ a \implies a' \in act\_set$ 
and  $var\_setI1[intro]: v \in set\_bexp\ b \implies v \in var\_set$ 
and  $var\_setI2[intro]: (x, e) \in set\ f \implies x \in var\_set$ 
and  $var\_setI3[intro]: (x, e) \in set\ f \implies v \in set\_exp\ e \implies v \in var\_set$ 
using assms unfolding
 $loc\_set\_compute\ act\_set\_compute\ var\_set\_compute\ set\_bexp\_vars\_of\_bexp\ set\_exp\_vars\_of\_exp$ 
 $clk\_set'\_def$ 
apply -
apply force
apply (fastforce elim:  $set1\_aconstraint\_elim$  simp:  $clkp\_set'\_def\ collect\_clock\_pairs\_def$ )
apply (simp; blast)+
done

```

```

lemma (in Simple_Network_Impl)  $clk\_set'I3[intro]:$ 
assumes  $(x1, x2, T, I) \in set\ automata$ 
shows  $(l, g') \in set\ I \implies ac \in set\ g' \implies c \in set1\_aconstraint\ ac \implies c \in clk\_set'$ 
using assms unfolding  $clk\_set'\_def\ clkp\_set'\_def\ collect\_clock\_pairs\_def$ 
by (force elim!:  $set1\_aconstraint\_elim$ )

```

#### definition

```

 $find\_remove\ P = map\_option\ (\lambda(xs, x, ys). (x, xs @ ys))\ o\ List.extract\ P$ 

```

```

fun  $merge\_pairs :: ('a \times 'b\ list)\ list \Rightarrow ('a \times 'b\ list)\ list \Rightarrow ('a \times 'b\ list)\ list$  where
 $merge\_pairs\ []\ ys = ys$  |
 $merge\_pairs\ ((k, v) \# xs)\ ys = (case\ find\_remove\ (\lambda(k', \_). k' = k)\ ys\ of$ 
 $None \Rightarrow (k, v) \# merge\_pairs\ xs\ ys$ 
 $| Some\ ((\_, v'), ys) \Rightarrow (k, v @ v') \# merge\_pairs\ xs\ ys$ 
 $)$ 

```

#### definition

```

 $conv\_urge\ urge \equiv \lambda(committed, urgent, trans, inv).$ 
 $(committed,$ 
 $[],$ 
 $map\ (\lambda(l, b, g, a, f, r, l'). (l, b, g, a, f, urge \# r, l'))\ trans,$ 
 $merge\_pairs\ (map\ (\lambda l. (l, [aconstraint.LE\ urge\ 0]))\ urgent)\ inv)$ 

```

```

lemma  $map\_of\_distinct\_upd2:$ 

```

```

assumes  $x \notin set\ (map\ fst\ xs)$ 
shows  $map\_of\ (xs @ (x, y) \# ys) = (map\_of\ (xs @ ys))(x \mapsto y)$ 
using assms by (induction xs) auto

```

```

lemma  $map\_of\_distinct\_lookup:$ 

```

```

assumes  $x \notin set\ (map\ fst\ xs)$ 
shows  $map\_of\ (xs @ (x, y) \# ys)\ x = Some\ y$ 
using  $map\_of\_distinct\_upd2[OF\ assms]$  by auto

```

```

lemma find_remove_SomeD:
  assumes find_remove P xs = Some (x, ys)
  shows  $\exists$  as bs. xs = as @ x # bs  $\wedge$  ys = as @ bs  $\wedge$  ( $\forall x \in \text{set as. } \neg P x$ )  $\wedge$  P x
  using assms unfolding find_remove_def by (auto dest: extract_SomeE)

lemma find_map:
  find Q (map f xs) = map_option f (find P xs) if  $\forall x \in \text{set xs. } Q (f x) \longleftrightarrow P x$ 
  using that by (induction xs) auto

lemma find_remove_map:
  find_remove Q (map f xs) = map_option ( $\lambda(x, xs). (f x, \text{map } f xs)$ ) (find_remove P xs)
  if  $\forall x \in \text{set xs. } Q (f x) \longleftrightarrow P x$ 
  using that
  by (induction xs)
  (auto simp: find_remove_def extract_Nil_code extract_Cons_code split: option.split)

lemma map_merge_pairs2:
  map ( $\lambda(k, v). (g k, \text{map } f v)$ ) (merge_pairs xs ys)
  = merge_pairs (map ( $\lambda(k, v). (g k, \text{map } f v)$ ) xs) (map ( $\lambda(k, v). (g k, \text{map } f v)$ ) ys)
  if inj: inj_on g (fst ' (set xs  $\cup$  set ys))
proof -
  have *:
    find_remove ( $\lambda(k', \_). k' = g k$ ) (map ( $\lambda(k, v). (g k, \text{map } f v)$ ) ys)
    = map_option
      ( $\lambda((k, v), ys). ((g k, \text{map } f v), \text{map } (\lambda(k, v). (g k, \text{map } f v)) ys))$ )
      (find_remove ( $\lambda(k', \_). k' = k$ ) ys)
    if inj_on g (fst ' (set xs  $\cup$  set ys)) k  $\in$  fst ' set xs for k xs ys
    using that
    by (subst find_remove_map[where P = ( $\lambda(k', \_). k' = k$ )])
      (auto elim: inj_onD[rotated, where A = fst ' (set xs  $\cup$  set ys)]
        intro!: arg_cong2[where f = map_option])
  show ?thesis
  using inj
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case
  by simp
next
  case (Cons x xs)
  then show ?case
  apply (clarsimp split: prod.split option.split simp: *[OF Cons(2)])
  apply (subst Cons.IH)
  apply (drule find_remove_SomeD)
  apply auto
  done
qed
qed

lemma map_merge_pairs:
  map ( $\lambda(k, v). (k, \text{map } f v)$ ) (merge_pairs xs ys)
  = merge_pairs (map ( $\lambda(k, v). (k, \text{map } f v)$ ) xs) (map ( $\lambda(k, v). (k, \text{map } f v)$ ) ys)
  using map_merge_pairs2[where g = id] by auto

lemma map_map_commute:

```

$\text{map } f (\text{map } g \text{ } xs) = \text{map } g (\text{map } f \text{ } xs)$  **if**  $\forall x \in \text{set } xs. f (g \text{ } x) = g (f \text{ } x)$   
**using** *that* **by** *simp*

**lemma** *conv\_automaton\_conv\_urge\_commute*:  
 $\text{conv\_automaton } (\text{conv\_urge } c \text{ } A) = \text{conv\_urge } c (\text{conv\_automaton } A)$   
**unfolding** *comp\_def conv\_automaton\_def conv\_urge\_def*  
**by** (*simp split: prod.split add: map\_merge\_pairs comp\_def*)

**lemma** *conv\_automaton\_conv\_urge\_commute'*:  
 $\text{conv\_automaton } o \text{ conv\_urge } c = \text{conv\_urge } c o \text{ conv\_automaton}$   
**unfolding** *comp\_def conv\_automaton\_conv\_urge\_commute ..*

**locale** *Simple\_Network\_Rename\_Defs\_int\_urge* =  
*Simple\_Network\_Rename\_Defs\_int* **where** *automata* = *automata* **for** *automata* ::  
 ('s list  $\times$  's list  
 $\times ((\text{'a} :: \text{countable}) \text{ act, 's, 'c, int, 'x} :: \text{countable, int}) \text{ transition list}$   
 $\times ((\text{'s} :: \text{countable}) \times (\text{'c} :: \text{countable, int}) \text{ cconstraint}) \text{ list}) \text{ list}$   
 + **fixes** *urges* :: 'c

**lemma** (**in** *Simple\_Network\_Rename\_Defs*) *conv\_automaton\_of*:  
 $\text{Simple\_Network\_Language.conv\_A } (\text{automaton\_of } A) = \text{automaton\_of } (\text{conv\_automaton } A)$   
**unfolding** *automaton\_of\_def conv\_automaton\_def*  
*Simple\_Network\_Language.conv\_A\_def*  
**by** (*force*  
*simp: default\_map\_of\_alt\_def map\_of\_map Simple\_Network\_Language.conv\_t\_def split:*  
*prod.splits*  
 )

**locale** *Simple\_Network\_Rename* =  
*Simple\_Network\_Rename\_Defs\_int\_urge* **where** *automata* = *automata* **for** *automata* ::  
 ('s list  $\times$  's list  
 $\times ((\text{'a} :: \text{countable}) \text{ act, 's, 'c, int, 'x} :: \text{countable, int}) \text{ transition list}$   
 $\times ((\text{'s} :: \text{countable}) \times (\text{'c} :: \text{countable, int}) \text{ cconstraint}) \text{ list}) \text{ list}$  +  
**assumes** *renum\_states\_inj*:  
 $\forall i < n\_ps. \forall x \in \text{loc\_set}. \forall y \in \text{loc\_set}. \text{renum\_states } i \text{ } x = \text{renum\_states } i \text{ } y \longrightarrow x = y$   
**and** *renum\_clocks\_inj*: *inj\_on* *renum\_clocks* (*insert urges clk\_set'*)  
**and** *renum\_vars\_inj*: *inj\_on* *renum\_vars* *var\_set*  
**and** *renum\_acts\_inj*: *inj\_on* *renum\_acts* *act\_set*  
**and** *infinite\_types*:  
*infinite* (*UNIV* :: 'c set) *infinite* (*UNIV* :: 'x set) *infinite* (*UNIV* :: 's set)  
*infinite* (*UNIV* :: 'a set)  
**and** *bounds'\_var\_set*: *fst* 'set *bounds'* = *var\_set*  
**and** *loc\_set\_invs*:  $\bigcup ((\lambda g. \text{fst 'set } g) \text{ 'set } (\text{map } (\text{snd } o \text{ snd } o \text{ snd}) \text{ automata})) \subseteq \text{loc\_set}$   
**and** *loc\_set\_committed*:  $\bigcup ((\text{set } o \text{ fst}) \text{ 'set } \text{ automata}) \subseteq \text{loc\_set}$   
**and** *loc\_set\_urgent*:  $\bigcup ((\text{set } o (\text{fst } o \text{ snd})) \text{ 'set } \text{ automata}) \subseteq \text{loc\_set}$   
**and** *urges\_not\_in\_clk\_set*: *urges*  $\notin \text{clk\_set}'$   
**begin**

**lemma** *clk\_set'\_finite*:  
*finite* *clk\_set'*  
**unfolding** *clk\_set'\_def* **unfolding** *clkp\_set'\_def* **by** (*intro finite\_intros*) *auto*

**lemmas** [*finite\_intros*] = *trans\_N\_finite*

```

lemma set_act_finite[finite_intros]:
  finite (set_act a)
  by (cases a) auto

lemma loc_set_finite:
  finite loc_set
  unfolding loc_set_def by (auto intro!: finite_intros)

lemma var_set_finite[finite_intros]:
  finite var_set
  unfolding var_set_def by (auto intro!: finite_intros)

lemma act_set_finite:
  finite act_set
  unfolding act_set_def broadcast_def by (auto intro!: finite_intros)

lemma bij_extend_bij_renum_clocks:
  bij (extend_bij renum_clocks (insert urge clk_set'))
  by (intro extend_bij_bij renum_clocks_inj clk_set'_finite infinite_types finite.insertI) simp

lemma renum_vars_bij_extends[simp]:
  extend_bij renum_vars var_set x = renum_vars x if x ∈ var_set
  by (intro extend_bij_extends[rule_format] renum_vars_inj var_set_finite infinite_types that)

lemma bij_extend_bij_renum_states:
  bij (extend_bij renum_vars var_set)
  by (intro extend_bij_bij renum_vars_inj var_set_finite infinite_types) simp

lemma inj_extend_bij_renum_states: inj (extend_bij (renum_states p) loc_set) if p < n_ps
  using renum_states_inj infinite_types loc_set_finite ⟨p < n_ps⟩
  by (intro extend_bij_inj) (auto intro!: inj_onI)

lemma renum_states_extend:
  extend_bij (renum_states p) loc_set l = renum_states p l if l ∈ loc_set p < n_ps
  using renum_states_inj infinite_types loc_set_finite ⟨p < n_ps⟩ ⟨l ∈ loc_set⟩
  by (intro extend_bij_extends[rule_format]) (auto intro!: inj_onI)

lemma renum_acts_bij_extends[simp]:
  extend_bij renum_acts act_set x = renum_acts x if x ∈ act_set
  by (intro extend_bij_extends[rule_format] renum_acts_inj act_set_finite infinite_types that)

lemma inj_extend_bij_renum_acts: inj (extend_bij renum_acts act_set)
  using renum_acts_inj infinite_types act_set_finite by (intro extend_bij_inj) auto

lemma constraint_clk_conv_ac:
  constraint_clk (conv_ac ac) = constraint_clk ac
  by (cases ac) auto

interpretation urge: Prod_TA_sem_urge
  (set broadcast, map (Simple_Network_Language.conv_A o automaton_of) automata, map_of
  bounds')
  urge
  apply (standard; clarsimp)
  subgoal

```

```

using urge_not_in_clk_set
unfolding conv_automaton_of
unfolding automaton_of_def conv_automaton_def
apply (clarsimp split: prod.split_asm)
apply (drule default_map_of_in_listD)
unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
apply clarsimp
subgoal premises prems
  using prems(1,7,8,10)
  unfolding constraint_clk_conv_ac unfolding constraint_clk_constraint_pair
  by force
done
subgoal
  using urge_not_in_clk_set
  unfolding conv_automaton_of
  unfolding automaton_of_def conv_automaton_def
  apply (clarsimp split: prod.split_asm)
  unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
  unfolding constraint_clk_conv_ac unfolding constraint_clk_constraint_pair
  apply force
done

subgoal
  using urge_not_in_clk_set
  unfolding conv_automaton_of
  unfolding automaton_of_def conv_automaton_def
  apply (clarsimp split: prod.split_asm)
  unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
  apply force
done
done

```

```

sublocale rename: Simple_Network_Rename_Defs_int
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set
  extend_bij renum_clocks (insert urge clk_set')
  λp. extend_bij (renum_states p) loc_set
  map (conv_urge urge) automata
.

```

```

sublocale rename: Simple_Network_Rename_int
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set
  extend_bij renum_clocks (insert urge clk_set')
  λp. extend_bij (renum_states p) loc_set
  map (conv_urge urge) automata
  apply (standard;
    (intro allI impI bij_extend_bij_renum_clocks inj_extend_bij_renum_states
      inj_extend_bij_renum_acts bij_extend_bij_renum_states bounds'_var_set)?)
  apply (simp add: Prod_TA_Defs.n_ps_def; fail)
subgoal

```

```

unfolding bounds' var_set rename.var_set_compute var_set_compute unfolding conv_urge_def
  by (fo_rule arg_cong2; fastforce)
subgoal
  unfolding conv_urge_def by auto
done

```

**definition**

```

renum_upd' = map (λ(x, upd). (renum_vars x, map_exp renum_vars upd))

```

**definition**

```

renum_reset' = map renum_clocks

```

**definition** renum\_automaton' **where**

```

renum_automaton' i ≡ λ(committed, trans, inv). let
  committed' = map (renum_states i) committed;
  trans' = map (λ(l, g, a, upd, r, l').
    (renum_states i l,
     map_constraint renum_clocks id g, renum_act a, renum_upd' upd, map renum_clocks r,
     renum_states i l'))
  ) trans;
  inv' = map (λ(l, g). (renum_states i l, map_constraint renum_clocks id g)) inv
in (committed', trans', inv')

```

**lemmas** renum\_automaton'\_alt\_def = renum\_automaton'\_def[unfolded  
 renum\_reset'\_def renum\_upd'\_def map\_constraint\_def renum\_act\_def  
 ]

**lemma** renum\_automaton\_eq:

```

rename.renum_automaton p (automata ! p) = renum_automaton p (automata ! p)
if p < n_ps

```

**proof** –

```

have renum_clocks: extend_bij renum_clocks (insert urge clk_set') c = renum_clocks c
  if c ∈ clk_set' for c
  apply (rule extend_bij_extends[rule_format])
  apply (rule renum_clocks_inj clk_set'_finite finite.insertI)+
  using that infinite_types by auto
have renum_constraint: rename.renum_constraint g = renum_constraint g
  if ∪ (setI_acconstraint ' (set g)) ⊆ clk_set' for g
  unfolding rename.renum_constraint_def renum_constraint_def map_constraint_def
  apply (rule map_cong, rule HOL.refl)
  apply (rule acconstraint.map_cong_pred, rule HOL.refl)
  using that
  apply (auto intro: renum_clocks[simplified] simp: pred_acconstraint_def)
done

```

**show** ?thesis

```

using that[folded length_automata_eq_n_ps]
unfolding rename.renum_automaton_def renum_automaton_def
apply (clarsimp split: prod.split)
apply safe
subgoal committed
  using loc_set_committed
  by (subst renum_states_extend) (auto 4 3 simp: n_ps_def dest!: nth_mem)
subgoal urgent

```

```

    using loc_set_urgent
    by (subst renum_states_extend) (auto 4 3 simp: n_ps_def dest!: nth_mem)
  subgoal start_locs
    by (subst renum_states_extend) (auto simp: n_ps_def dest!: nth_mem)
  subgoal state
    unfolding rename.renum_bexp_def renum_bexp_def
    by (auto dest!: nth_mem intro!: renum_vars_bij_extends bexp.map_cong_pred simp:
pred_bexp_def)
  subgoal guards
    by (auto dest!: nth_mem intro!: renum_cconstraint)
  subgoal actions
    unfolding rename.renum_act_def renum_act_def
    by (auto simp: pred_act_def dest!: nth_mem intro!: renum_acts_bij_extends act.map_cong_pred)
  subgoal upds
    unfolding renum_upd_def rename.renum_upd_def rename.renum_exp_def renum_exp_def
    by (auto dest!: nth_mem intro!: renum_vars_bij_extends exp.map_cong_pred simp: pred_exp_def)
  subgoal clock_resets
    unfolding rename.renum_reset_def renum_reset_def by (auto dest!: nth_mem intro!:
renum_clocks)
  subgoal dest_locs
    by (subst renum_states_extend) (auto simp: n_ps_def dest!: nth_mem)
  subgoal inv_locs
    using loc_set_invs
    by (subst renum_states_extend) (auto 4 4 simp: n_ps_def dest!: nth_mem)
  subgoal renum_clocks
    by (auto dest!: nth_mem intro!: renum_cconstraint)
  done
qed

```

```

lemma renum_urge_automaton_eq1:
  renum_automaton p (conv_urge urge (automata ! p))
= conv_urge (renum_clocks urge) (renum_automaton p (automata ! p)) if p < n_ps
  unfolding renum_automaton_def conv_urge_def
  apply (simp split: prod.split)
  apply safe
  subgoal
    unfolding renum_reset_def by simp
  unfolding renum_cconstraint_def map_cconstraint_def
  apply (subst map_merge_pairs2)
  subgoal
    using loc_set_urgent loc_set_invs ⟨p < n_ps⟩ unfolding inj_on_def n_ps_def
    by (auto; force
      elim!: renum_states_inj[unfolded n_ps_def, simplified, rule_format, rotated -1]
      intro: nth_mem)
  apply (fo_rule arg_cong2; simp)
  done

```

```

lemma renum_urge_automaton_eq2:
  rename.real.renum_automaton p (conv_urge urge (automata ! p))
= conv_urge
  (extend_bij renum_clocks (insert_urge clk_set') urge)
  (rename.renum_automaton p (automata ! p))
  if p < n_ps
  unfolding rename.renum_automaton_def conv_urge_def

```

```

apply (simp split: prod.split)
apply safe
subgoal
  unfolding rename.renum_reset_def by simp
unfolding rename.renum_cconstraint_def map_cconstraint_def
apply (subst map_merge_pairs2)
subgoal
  by (rule inj_extend_bij_renum_states that subset_UNIV inj_on_subset)+
apply (fo_rule arg_cong2)
apply simp+
done

lemma renum_urge_automaton_eq:
  rename.real.renum_automaton p (conv_urge urge (automata ! p)) =
  renum_automaton p (conv_urge urge (automata ! p)) if p < n_ps
using that
unfolding renum_urge_automaton_eq1 [OF that] renum_urge_automaton_eq2 [OF that]
apply (fo_rule arg_cong2)
subgoal
  by (intro
    extend_bij_extends[rule_format] renum_clocks_inj clk_set'_finite finite.insertI infinite_types
    ) auto
by (rule renum_automaton_eq)

lemma rename_N_eq_sem:
  rename.renum.sem =
  Simple_Network_Language_Model_Checking.N
  (map renum_acts broadcast)
  (map_index renum_automaton (map (conv_urge urge) automata))
  (map (λ(a,p). (renum_vars a,p)) bounds')

unfolding rename.renum.sem_def
unfolding Simple_Network_Language.conv_def
apply simp
apply (intro conjI)
subgoal
  by (rule image_cong; intro HOL.refl renum_acts_bij_extends; simp add: act_set_def broadcast_def)
subgoal
  apply (rule map_index_cong)
unfolding conv_automaton_of
apply safe
apply (fo_rule arg_cong)+
apply (erule renum_urge_automaton_eq[folded length_automata_eq_n_ps])
done
subgoal
  using bounds'_var_set by - (fo_rule arg_cong, auto intro: renum_vars_bij_extends)
done

lemma map_of_merge_pairs:
  map_of (merge_pairs xs ys) = (λx.
  (if x ∈ fst ' set xs ∧ x ∈ fst ' set ys then Some (the (map_of xs x) @ the (map_of ys x)))
  else if x ∈ fst ' set xs then map_of xs x
  else map_of ys x))

```

**proof** –

**have** 1: *False* **if** *find\_remove* ( $\lambda(k', \_). k' = k$ ) *ys* = *None* ( $k, x$ )  $\in$  *set ys* **for** *k x ys*  
**using** *that* **unfolding** *find\_remove\_def* **by** (*auto simp: extract\_None\_iff*)  
**have** 2: *map\_of xs k* = *Some x* **if**  
*find\_remove* ( $\lambda(k', \_). k' = k$ ) *xs* = *Some ((k', x), ys)* **for** *k k' x xs ys*  
**using** *that*  
**by** (*auto 4 4 split: option.split dest: map\_of\_SomeD find\_remove\_SomeD simp: map\_add\_def*)  
**show** *?thesis*  
**apply** (*rule ext*)  
**apply** (*induction xs arbitrary: ys*)  
**apply** (*simp; fail*)  
**apply** (*clarsimp split: if\_split\_asm option.split*)  
**apply** (*auto 4 3 split: option.split simp: map\_add\_def dest: find\_remove\_SomeD 2 1*)  
**done**  
**qed**

**lemma** *default\_map\_of\_merge\_pairs*:

*default\_map\_of [] (merge\_pairs xs ys) = ( $\lambda x.$*   
*(if*  $x \in \text{fst } \text{'set } xs$  *then the* (*map\_of xs x*) *@* *default\_map\_of [] ys x*  
*else* *default\_map\_of [] ys x*)  
**unfolding** *default\_map\_of\_alt\_def map\_of\_merge\_pairs*  
**by** (*rule ext*) (*auto 4 3 dest: map\_of\_SomeD weak\_map\_of\_SomeI split: if\_split\_asm*)

**lemma** *automaton\_of\_conv\_urge\_commute*:

*automaton\_of (conv\_urge urge A) = urge.add\_reset (urge.add\_inv (automaton\_of A))*  
**unfolding** *conv\_urge\_def urge.add\_reset\_def urge.add\_inv\_def automaton\_of\_def*  
**apply** (*clarsimp split: prod.splits*)  
**apply** (*rule ext*)  
**unfolding** *default\_map\_of\_merge\_pairs*  
**apply** (*clarsimp, safe*)  
**apply** (*clarsimp*)  
**subgoal** *premises* *prems* **for** *\_\_ urge \_\_ l*  
**proof** –  
**have** \*: *map\_of (map ( $\lambda x. (x, [\text{acconstraint.LE } urge \ 0])$ ) *urgent*)*  
*= ( $\lambda l. \text{if } l \in \text{set } urgent \text{ then } \text{Some } [\text{acconstraint.LE } urge \ 0] \text{ else } \text{None}$ )*  
**using** *map\_of\_map\_keys* **[where**  
*m =  $\lambda l. \text{if } l \in \text{set } urgent \text{ then } \text{Some } [\text{acconstraint.LE } urge \ 0] \text{ else } \text{None}$*   
**by** (*force cong: map\_cong simp: dom\_def*)  
**from**  $\langle l \in \text{set } urgent \rangle$  **show** *?thesis*  
**by** (*subst \**) *auto*  
**qed**  
**by** *auto*

**lemma** *urge\_commute*:

*rename.sem = urge.A\_urge*  
**unfolding** *rename.sem\_def urge.A\_urge\_def*  
**apply** *simp*  
**unfolding** *conv\_automaton\_conv\_urge\_commute conv\_automaton\_of automaton\_of\_conv\_urge\_commute*  
**by** *fast*

**lemma** *conv\_automaton\_of'*:

*automaton\_of  $\circ$  conv\_automaton = Simple\_Network\_Language.conv\_A  $\circ$  automaton\_of*  
**unfolding** *comp\_def conv\_automaton\_of ..*

```

sublocale urge_bisim: Bisimulation_Invariant
   $\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ sem } L \text{ s } u \text{ } L' \text{ s' } u'$ 
   $\lambda(L, s, u) (L', s', u'). \text{step\_}u' \text{ rename.sem } L \text{ s } u \text{ } L' \text{ s' } u'$ 
   $\lambda(L, s, u) (L', s', u'). L' = L \wedge s' = s \wedge u' = u (\text{urge} := 0)$ 
   $\lambda(L, s, u). L \in \text{states } \lambda(L, s, u). \text{True}$ 
  unfolding urge_commute sem_def conv_automaton_of' by (rule urge.urge_bisim[unfolded conv_states])

lemma check_sexp_equiv:
  assumes urge_bisim.A_B.equiv' (L, s, u) (L', s', u')
  shows check_sexp e L (the o s) = check_sexp e L' (the o s')
  using assms unfolding urge_bisim.A_B.equiv'_def by simp

context
  fixes a0 :: 's list  $\times$  ('x  $\Rightarrow$  int option)  $\times$  ('c  $\Rightarrow$  real)
  assumes start: fst a0  $\in$  states snd (snd a0) urge = 0
begin

lemma start_equiv: urge_bisim.A_B.equiv' a0 a0
  using start unfolding urge_bisim.A_B.equiv'_def by auto

lemma urge_models_iff:
  sem, a0  $\models \Phi \longleftrightarrow \text{rename.sem}, a_0 \models \Phi$ 
proof –
  have rel_ctl_formula urge_bisim.compatible (ctl_of  $\Phi$ ) (ctl_of  $\Phi$ )
  by (cases  $\Phi$ ) (auto simp: prop_of_def rel_fun_def check_sexp_equiv)
  with start_equiv show ?thesis
  by (simp only: models_ctl_iff urge_bisim.CTL_compatible[THEN rel_funD, symmetric])
qed

lemma urge_has_deadlock_iff:
  has_deadlock sem a0  $\longleftrightarrow$  has_deadlock rename.sem a0
  unfolding has_deadlock_def using start_equiv
  by (intro urge_bisim.deadlock_iff, unfold urge_bisim.A_B.equiv'_def) auto

lemma state_formula_compatible:
  ( $\bigcup x \in \text{set\_state\_formula } \varphi. \text{locs\_of\_sexp } x \subseteq \{0..<n\_ps\}$ )  $\implies$ 
  rel_state_formula urge_bisim.compatible
  (map_state_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (\text{the } o \text{ } s)) \varphi$ )
  (map_state_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (\text{the } o \text{ } s)) \varphi$ ) and path_formula_compatible:
  ( $\bigcup x \in \text{set\_path\_formula } \psi. \text{locs\_of\_sexp } x \subseteq \{0..<n\_ps\}$ )  $\implies$ 
  rel_path_formula urge_bisim.compatible
  (map_path_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (\text{the } o \text{ } s)) \psi$ )
  (map_path_formula ( $\lambda P (L, s, \_). \text{check\_sexp } P \text{ } L \text{ } (\text{the } o \text{ } s)) \psi$ )
  by (induction  $\varphi$  and  $\psi$ ) (auto simp: check_sexp_equiv prop_of_def rel_fun_def)

lemmas urge_models_state_compatible =
  urge_bisim.models_state_compatible[OF state_formula_compatible]

end

end

```

```

locale Simple_Network_Rename_Start =
  Simple_Network_Rename where automata = automata
for automata ::
  ('s list  $\times$  's list
    $\times$  (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list
    $\times$  (('s :: countable)  $\times$  ('c :: countable, int) cconstraint) list) list +
fixes s0 :: ('x  $\times$  int) list
and L0 :: 's list
assumes L0_states: L0  $\in$  states
and s0_dom: fst ' set s0 = var_set and s0_distinct: distinct (map fst s0)
begin

lemma rename_n_ps_eq:
  rename.n_ps = n_ps
unfolding rename.length_automata_eq_n_ps[symmetric] n_ps_def by simp

lemma rename_states_eq:
  rename.states = states
unfolding rename.states_def states_def rename_n_ps_eq
by (simp add: rename.N_eq[unfolded rename_n_ps_eq] N_eq n_ps_def del: map_map)
  (auto simp: automaton_of_def conv_urge_def trans_def split: prod.splits)

lemma state_eq:
  map_of (map ( $\lambda(x, y).$  (extend_bij renum_vars var_set x, y)) s0) =
  map_of (map ( $\lambda(x, y).$  (renum_vars x, y)) s0)
using s0_dom by - (rule arg_cong, auto intro: renum_vars_bij_extends)

lemma sexp_eq:
assumes
   $\langle$ vars_of_sexp e  $\subseteq$  var_set $\rangle$ 
   $\langle$ set2_sexp e  $\subseteq$  loc_set $\rangle$ 
   $\langle$ locs_of_sexp e  $\subseteq$  {0.. $n_{ps}$ } $\rangle$ 
shows  $\langle$ map_sexp ( $\lambda p.$  extend_bij (renum_states p) loc_set) (extend_bij renum_vars var_set)
  id e =
  map_sexp renum_states renum_vars id e $\rangle$ 
using assms by (induction e; clarsimp simp: renum_states_extend)

lemma L0_dom:
  length L0 = n_ps set L0  $\subseteq$  loc_set
using L0_states by (auto intro!: states_loc_setD)

definition
  a0 = (L0, map_of s0,  $\lambda\_.$  0)

definition
  a0' = (
    map_index ( $\lambda p.$  renum_states p) L0,
    map_of (map ( $\lambda(x, y).$  (renum_vars x, y)) s0),
     $\lambda\_.$  0)

sublocale rename: Simple_Network_Rename_Start_int
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set

```

```

    extend_bij renum_clocks (insert urge clk_set')
  λp. extend_bij (renum_states p) loc_set
s0 L0
map (conv_urge urge) automata
apply (standard; (rule L0_states[folded rename_states_eq] s0_distinct)?)
unfolding s0_dom rename.bounds'_var_set[symmetric] bounds'_var_set ..

lemma rename_a0_eq:
  rename.a0 = a0
unfolding rename.a0_def a0_def ..

lemma rename_a0'_eq:
  rename.a0' = a0'
unfolding a0'_def rename.a0'_def
apply (clarsimp, rule conjI)
subgoal
  using L0_dom by (auto intro!: map_index_cong renum_states_extend)
subgoal
  unfolding vars_inv_def using s0_dom s0_distinct
  by (auto simp: state_eq Simple_Network_Rename_Defs.vars_inv_def)
done

lemma has_deadlock_iff:
  has_deadlock rename.renum.sem a0'  $\longleftrightarrow$  has_deadlock sem a0
proof -
  have has_deadlock sem a0  $\longleftrightarrow$  has_deadlock rename.sem a0
    by (rule urge_has_deadlock_iff) (auto intro: L0_states simp: a0_def)
  also have ...  $\longleftrightarrow$  has_deadlock rename.renum.sem a0'
    by (rule rename.has_deadlock_iff[unfolded rename_a0_eq rename_a0'_eq])
  finally show ?thesis ..
qed

lemma N_eq_sem:
  Simple_Network_Language_Model_Checking.N broadcast automata bounds' = sem
unfolding conv_alt_def sem_def
by safe (rule nth_equalityI; simp add: conv_N_eq N_eq sem_N_eq conv_automaton_of_n_ps_def)

lemma rename_N_eq_sem':
  Simple_Network_Language_Model_Checking.N
    (map renum_acts broadcast)
    (map_index renum_automaton automata)
    (map (λ(a,p). (renum_vars a,p)) bounds')
  = renum.sem
unfolding renum.sem_def
unfolding renum.conv_alt_def
by safe (rule nth_equalityI; simp add: conv_N_eq N_eq sem_N_eq conv_automaton_of_n_ps_def)

lemmas has_deadlock_iff' =
  has_deadlock_iff[unfolded rename_N_eq_sem, folded N_eq_sem, unfolded a0_def a0'_def]

lemmas start_equiv = start_equiv[of a0, unfolded a0_def, simplified, OF L0_states]

lemmas urge_models_state_compatible =
  urge_models_state_compatible[THEN rel_funD, OF _ _ _ start_equiv,

```

of  $a_0$ , unfolded  $a_0\_def$ , simplified, OF  $L_0\_states$ ]

**lemmas** *rename\_models\_state\_compatible* =  
*rename.models\_state\_compatible*[*THEN* *rel\_funD*, OF *rename.start\_equiv*,  
 unfolded *rename\_a0\_eq a0\_def rename\_n\_ps\_eq rename\_a0'\_eq a0'\_def*]

**lemmas** *models\_state\_compatible* =  
*transp\_on\_equality*[*THEN* *transpD*, OF *urge\_models\_state\_compatible rename\_models\_state\_compatible*]

**lemmas** *models\_state\_compatible'* = *models\_state\_compatible*[unfolded *rename\_N\_eq\_sem*, folded  
*N\_eq\_sem*]

**end**

**locale** *Simple\_Network\_Rename\_Formula* =  
*Simple\_Network\_Rename\_Start* **where** *automata* = *automata*  
**for** *automata* ::  
 ('s list  $\times$  's list  
 $\times$  (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list  
 $\times$  (('s :: countable)  $\times$  ('c :: countable, int) cconstraint) list) list +  
**fixes**  $\Phi$  :: (nat, 's, 'x, int) formula  
**assumes** *formula\_dom*:  
*set2\_formula*  $\Phi \subseteq loc\_set$   
*locs\_of\_formula*  $\Phi \subseteq \{0..<n\_ps\}$   
*vars\_of\_formula*  $\Phi \subseteq var\_set$   
**begin**  
  
**sublocale** *rename*: *Simple\_Network\_Rename\_Formula\_int*  
*broadcast bounds'*  
*extend\_bij renum\_acts act\_set*  
*extend\_bij renum\_vars var\_set*  
*extend\_bij renum\_clocks (insert urge clk\_set')*  
 $\lambda p.$  *extend\_bij (renum\_states p) loc\_set*  
 $s_0$   $L_0$   
*map (conv\_urge urge) automata*  
**apply** (*standard*; (*rule*  $L_0\_states$ [folded *rename\_states\_eq*]  $s_0\_distinct$ )?)  
**unfolding**  $s_0\_dom$  *rename.bounds'\_var\_set*[*symmetric*] *bounds'\_var\_set* .

**definition**

$\Phi' = map\_formula (\lambda p. renum\_states p) renum\_vars id \Phi$

**lemma** *rename\_Φ'\_eq*:

*rename.Φ' = Φ'*

**using** *formula\_dom* **unfolding** *rename.Φ'\_def Φ'\_def* **by** (*induction*  $\Phi$ ; *clarsimp simp: serp\_eq*)

**lemma** *models\_iff1*:

*rename.renum.sem, a0'  $\models \Phi' \longleftrightarrow rename.sem, a_0 \models \Phi$*

**by** (*intro* *rename.models\_iff*[unfolded *rename\_a0\_eq rename\_a0'\_eq rename\_Φ'\_eq rename\_n\_ps\_eq*]  
*formula\_dom sym*)

**lemma** *models\_iff2*:

*rename.sem, a0  $\models \Phi \longleftrightarrow sem, a_0 \models \Phi$*

**by** (*rule* *sym*, *intro* *urge\_models\_iff formula\_dom*) (*auto* *intro: formula\_dom L0\_states simp:*  
 $a_0\_def$ )

```

lemma models_iff:
  rename.renum.sem, a0'  $\models \Phi' \longleftrightarrow \textit{sem}, a_0 \models \Phi$ 
  unfolding models_iff1 models_iff2 ..

lemmas models_iff' =
  models_iff[unfolded rename_N_eq_sem, folded N_eq_sem, unfolded a0_def a0'_def  $\Phi'$ _def]

end

end
theory Simple_Network_Language_Deadlock_Checking
  imports Simple_Network_Language_Model_Checking Munta_Model_Checker.Deadlock_Checking
begin

context Simple_Network_Impl_nat_ceiling_start_state — slow: 120s
begin

context
  fixes  $\varphi$ 
  assumes formula_is_false: formula = formula.EX (not sexp.true)
begin

lemma F_is_FalseI:
  shows PR_CONST F = ( $\lambda\_.$  False)
  by (subst formula_is_false) auto

lemma final_fun_is_False:
  Fi a = False
  by (subst formula_is_false) auto

lemmas deadlock_checker_hoare = impl.deadlock_checker_hoare[
  OF final_fun_is_False F_is_FalseI, folded deadlock_start_iff has_deadlock_def]

end

schematic_goal deadlock_checker_alt_def:
  impl.deadlock_checker  $\equiv$  ?impl
  unfolding impl.deadlock_checker_def
  unfolding impl.succs_impl_def
  unfolding impl.check_deadlock_neg_impl_def impl.check_deadlock_impl_def
  apply (abstract_let impl.is_start_in_states_impl is_start)
  unfolding impl.is_start_in_states_impl_def
  apply (abstract_let impl.E_op''_impl E_op''_impl)
  unfolding impl.E_op''_impl_def fw_impl'_int
  apply (abstract_let trans_impl trans_impl)
  unfolding trans_impl_def
  apply (abstract_let int_trans_impl int_trans_impl)
  apply (abstract_let bin_trans_from_impl bin_trans_impl)
  apply (abstract_let broad_trans_from_impl broad_trans_impl)
  unfolding int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def
  apply (abstract_let trans_in_broad_grouped trans_in_broad_grouped)
  apply (abstract_let trans_out_broad_grouped trans_out_broad_grouped)
  apply (abstract_let trans_in_map trans_in_map)

```

```

apply (abstract_let trans_out_map trans_out_map)
apply (abstract_let int_trans_from_all_impl int_trans_from_all_impl)
unfolding int_trans_from_all_impl_def
apply (abstract_let int_trans_from_vec_impl int_trans_from_vec_impl)
unfolding int_trans_from_vec_impl_def
apply (abstract_let int_trans_from_loc_impl int_trans_from_loc_impl)
unfolding int_trans_from_loc_impl_def
apply (abstract_let trans_i_map trans_i_map)
unfolding trans_out_broad_grouped_def trans_out_broad_map_def
unfolding trans_in_broad_grouped_def trans_in_broad_map_def
unfolding trans_in_map_def trans_out_map_def
unfolding trans_i_map_def
apply (abstract_let trans_map trans_map)
apply (abstract_let inv_fun :: nat list  $\times$  int list  $\Rightarrow$  _ inv_fun)
unfolding inv_fun_alt_def
apply (abstract_let invs2 invs)
unfolding invs2_def
unfolding k_impl_alt_def
apply (abstract_let k_i k_i)
apply (abstract_let n_ps n_ps)

unfolding k_i_def impl.state_copy_impl_def impl.a0_impl_def impl.abstr_repair_impl_def
  impl.subsumes_impl_def impl.emptiness_check_impl_def impl.abstra_repair_impl_def
  impl.init_dbm_impl_def
by (rule Pure.reflexive)

```

**end**

**concrete\_definition** deadlock\_checker **uses**

*Simple\_Network\_Impl\_nat\_ceiling\_start\_state.deadlock\_checker\_alt\_def*

**definition** precond\_dc **where**

```

precond_dc
  show_clock show_state broadcast bounds' automata m num_states num_actions k L0 s0 formula
 $\equiv$ 
  if Simple_Network_Impl_nat_ceiling_start_state
    broadcast bounds' automata m num_states num_actions k L0 s0 formula
  then
    deadlock_checker
      broadcast bounds' automata m num_states num_actions k L0 s0 show_clock show_state
       $\gg=$  ( $\lambda x$ . return (Some x))
  else return None

```

**theorem** deadlock\_check:

```

<emp> precond_dc
  show_clock show_state broadcast bounds automata m num_states num_actions k L0 s0
  (formula.EX (not sexp.true))
< $\lambda$  Some r  $\Rightarrow$   $\uparrow$ (
  Simple_Network_Impl_nat_ceiling_start_state
    broadcast bounds automata m num_states num_actions k L0 s0 (formula.EX (not
sexp.true))  $\wedge$ 
    r = has_deadlock (N broadcast automata bounds) (L0, map_of s0,  $\lambda$ _. 0)
  )
| None  $\Rightarrow$   $\uparrow$ ( $\neg$  Simple_Network_Impl_nat_ceiling_start_state

```

```

    broadcast bounds automata m num_states num_actions k L0 s0 (formula.EX (not sexp.true)))
  >t
proof —
  define A where A ≡ N broadcast automata bounds
  define check where check ≡ A, (L0, map_of s0, λ_. 0) ⊨ (formula.EX (not sexp.true))
  note [sep_heap_rules] = Simple_Network_Impl_nat_ceiling_start_state.deadlock_checker_hoare[
    OF_ HOL.refl,
    of broadcast bounds automata m num_states num_actions k L0 s0 _,
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    folded A_def
  ]
  show ?thesis
  unfolding A_def[symmetric] precondition_dc_def
  by (sep_auto simp: deadlock_checker.refine[symmetric] pure_def has_deadlock_def)
qed

```

```

end
theory Shortest_SCC_Paths
  imports Gabow_SCC.Gabow_SCC_Code
begin

```

```

definition compute_SCC_tr ::
  ('a :: hashable ⇒ bool, 'a ⇒ 'a list, 'a list, 'b) gen_g_impl_scheme ⇒ _ where
  compute_SCC_tr =
    Gabow_SCC_Code.compute_SCC_tr (⇒) bounded_hashcode_nat (def_hashmap_size TYPE('a))

```

— Example usage:

```

term compute_SCC_tr (| gi_V = (λ x. True), gi_E = (λ x. [3]), gi_V0 = [1], ... = 3 |)

```

Efficiently calculate shortest paths in a graph with non-negative edge weights, and where the only cycles are 0-cycles.

```

definition
  calc_shortest_scc_paths G n ≡
  let
    sccs = compute_SCC_tr G;
    d = replicate n None @ [Some 0];
    d = (
      fold
        (λ vs d.
          fold
            (λ u d.
              fold
                (λ v d.
                  case d ! u of
                    None ⇒ d
                  | Some du ⇒ (
                      case d ! v of
                        None ⇒ d[v := Some (du + more G u v)]
                      | Some dv ⇒
                        if du + more G u v < dv
                        then d[v := Some (du + more G u v)]
                        else d
                    )
                )
            )
          )
    )

```

```

      (gi_E G u)
    d
  )
  vs
  d
)
sccs
d
);
d = (
  fold
  (λ vs d.
    let
      dsc =
        fold
        (λ v dsc.
          case dsc of
            None ⇒ d ! v
          | Some d' ⇒ (
              case d ! v of
                None ⇒ dsc
              | Some dv ⇒ Some (min dv d')
            )
        )
      vs
      None
    in
      fold (λ v d. d[v:=dsc]) vs d
  )
  sccs
  d
)
in d

```

**end**

## 12 Assembling the Model Checker and Generating Code

We assemble the model checker:

- A parser is added to parse JSON files
- The resulting JSON data is validated and converted to the simple networks language
- The verified model checker is run on this input, and the output is printed
- The verified part includes a “renaming” step to normalize identifiers in the input  
Then the code is exported:
- We add some logging utilities (via code printings)
- Code generation is setup
- We export this to SML via reflection

- And test it on a few small examples

```

theory Simple_Network_Language_Export_Code
imports
  Munta_Model_Checker.JSON_Parsing
  Munta_Model_Checker.Simple_Network_Language_Renaming
  Munta_Model_Checker.Simple_Network_Language_Deadlock_Checking
  Shortest_SCC_Paths
  Munta_Base.Error_List_Monad
begin

datatype result =
  Renaming_Failed | Preconds_Unsat | Sat | Unsat

abbreviation renum_automaton  $\equiv$  Simple_Network_Rename_Defs.renum_automaton

locale Simple_Network_Rename_Formula_String_Defs =
  Simple_Network_Rename_Defs_int where automata = automata for automata ::
    (nat list  $\times$  nat list  $\times$ 
      (String.literal act, nat, String.literal, int, String.literal, int) transition list
       $\times$  (nat  $\times$  (String.literal, int) cconstraint) list) list
begin

definition check_renaming where check_renaming urge  $\Phi$   $L_0$   $s_0 \equiv$  combine [
  assert ( $\forall i < n\_ps. \forall x \in loc\_set. \forall y \in loc\_set. renum\_states\ i\ x = renum\_states\ i\ y \longrightarrow x = y$ )
    STR "Location renamings are injective",
  assert (inj_on renum_clocks (insert urge clk_set'))
    STR "Clock renaming is injective",
  assert (inj_on renum_vars var_set)
    STR "Variable renaming is injective",
  assert (inj_on renum_acts act_set)
    STR "Action renaming is injective",
  assert (fst 'set bounds' = var_set)
    STR "Bound set is exactly the variable set",
  assert ( $\bigcup ((\lambda g. fst\ 'set\ g)\ 'set\ (map\ (snd\ o\ snd\ o\ snd)\ automata)) \subseteq loc\_set$ )
    STR "Invariant locations are contained in the location set",
  assert ( $\bigcup ((set\ o\ fst)\ 'set\ automata) \subseteq loc\_set$ )
    STR "Broadcast locations are contained in the location set",
  assert ( $((\bigcup x \in set\ automata. set\ (fst\ (snd\ x))) \subseteq loc\_set$ )
    STR "Urgent locations are contained in the location set",
  assert (urge  $\notin$  clk_set')
    STR "Urge not in clock set",
  assert ( $L_0 \in states$ )
    STR "Initial location is in the state set",
  assert (fst 'set  $s_0$  = var_set)
    STR "Initial state has the correct domain",
  assert (distinct (map fst  $s_0$ ))
    STR "Initial state is unambiguous",
  assert (set2_formula  $\Phi \subseteq loc\_set$ )
    STR "Formula locations are contained in the location set",
  assert (locs_of_formula  $\Phi \subseteq \{0..<n\_ps\}$ )
    STR "Formula automata are contained in the automata set",
  assert (vars_of_formula  $\Phi \subseteq var\_set$ )
    STR "Variables of the formula are contained in the variable set"

```

```

]

end

locale Simple_Network_Rename_Formula_String =
  Simple_Network_Rename_Formula_String_Defs +
  fixes urge :: String.literal
  fixes  $\Phi$  :: (nat, nat, String.literal, int) formula
    and  $s_0$  :: (String.literal  $\times$  int) list
    and  $L_0$  :: nat list
  assumes renum_states_inj:
     $\forall i < n\_ps. \forall x \in loc\_set. \forall y \in loc\_set. renum\_states\ i\ x = renum\_states\ i\ y \longrightarrow x = y$ 
  and renum_clocks_inj: inj_on renum_clocks (insert urge clk_set')
  and renum_vars_inj: inj_on renum_vars var_set
  and renum_acts_inj: inj_on renum_acts act_set
  and bounds'_var_set: fst 'set bounds' = var_set
  and loc_set_invs:  $\bigcup ((\lambda g. fst\ 'set\ g)\ 'set\ (map\ (snd\ o\ snd\ o\ snd)\ automata)) \subseteq loc\_set$ 
  and loc_set_broadcast:  $\bigcup ((set\ o\ fst)\ 'set\ automata) \subseteq loc\_set$ 
  and loc_set_urgent:  $\bigcup ((set\ o\ (fst\ o\ snd))\ 'set\ automata) \subseteq loc\_set$ 
  and urge_not_in_clk_set:  $urge \notin clk\_set'$ 
  assumes  $L_0\_states: L_0 \in states$ 
    and  $s_0\_dom: fst\ 'set\ s_0 = var\_set$  and  $s_0\_distinct: distinct\ (map\ fst\ s_0)$ 
  assumes formula_dom:
    set2_formula  $\Phi \subseteq loc\_set$ 
    locs_of_formula  $\Phi \subseteq \{0..<n\_ps\}$ 
    vars_of_formula  $\Phi \subseteq var\_set$ 
begin

interpretation Simple_Network_Rename_Formula
  by (standard;
    rule renum_states_inj renum_clocks_inj renum_vars_inj bounds'_var_set renum_acts_inj
      loc_set_invs loc_set_broadcast loc_set_urgent urge_not_in_clk_set
      infinite_literal infinite_UNIV_nat  $L_0\_states$   $s_0\_dom$   $s_0\_distinct$  formula_dom
  )+ — slow: 40s

lemmas Simple_Network_Rename_intro = Simple_Network_Rename_Formula_axioms

end

lemma is_result_assert_iff:
  is_result (assert b m)  $\longleftrightarrow$  b
  unfolding assert_def by auto

lemma is_result_combine_Cons_iff:
  is_result (combine (x # xs))  $\longleftrightarrow$  is_result x  $\wedge$  is_result (combine xs)
  by (cases x; cases combine xs) auto

lemma is_result_combine_iff:
  is_result (a <|> b)  $\longleftrightarrow$  is_result a  $\wedge$  is_result b
  by (cases a; cases b) (auto simp: combine2_def)

context Simple_Network_Rename_Formula_String_Defs
begin

```

```

lemma check_renaming:
  Simple_Network_Rename_Formula_String
  broadcast bounds'
  renum_acts renum_vars renum_clocks renum_states
  automata urge  $\Phi$   $s_0$   $L_0 \longleftrightarrow$ 
  is_result (check_renaming urge  $\Phi$   $L_0$   $s_0$ )

  unfolding check_renaming_def Simple_Network_Rename_Formula_String_def
  by (simp add: is_result_combine_Cons_iff is_result_assert_iff del: combine.simps(2))

end

context Simple_Network_Impl_nat_defs
begin

definition check_precond1 where
check_precond1 =
  combine [
    assert (m > 0)
    (STR "At least one clock"),
    assert (0 < length automata)
    (STR "At least one automaton"),
    assert ( $\forall i < n\_ps$ . let ( $\_, \_, trans, \_$ ) = (automata ! i) in  $\forall (l, \_, \_, \_, \_, l') \in set\ trans.$ 
      l < num_states i  $\wedge$   $l' < num\_states\ i$ )
    (STR "Number of states is correct (transitions)"),
    assert ( $\forall i < n\_ps$ . let ( $\_, \_, \_, inv$ ) = (automata ! i) in  $\forall (x, \_) \in set\ inv. x < num\_states$ 
    i)
    (STR "Number of states is correct (invariants)"),
    assert ( $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, \_, \_, f, \_, \_) \in set\ trans.$ 
       $\forall (x, upd) \in set\ f. x < n\_vs \wedge (\forall i \in vars\_of\_exp\ upd. i < n\_vs)$ )
    (STR "Variable set bounded (updates)"),
    assert ( $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, b, \_, \_, \_, \_) \in set\ trans.$ 
       $\forall i \in vars\_of\_bexp\ b. i < n\_vs$ )
    (STR "Variable set bounded (guards)"),
    assert ( $\forall i < n\_vs. fst (bounds' ! i) = i$ )
    (STR "Bounds first index"),
    assert ( $\forall a \in set\ broadcast. a < num\_actions$ )
    (STR "Broadcast actions bounded"),
    assert ( $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, \_, a, \_, \_, \_) \in set\ trans.$ 
      pred_act ( $\lambda a. a < num\_actions$ ) a)
    (STR "Actions bounded (transitions)"),
    assert ( $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, g, \_, \_, r, \_) \in set\ trans.$ 
      ( $\forall c \in set\ r. 0 < c \wedge c \leq m$ )  $\wedge$ 
      ( $\forall (c, x) \in collect\_clock\_pairs\ g. 0 < c \wedge c \leq m \wedge x \in \mathbb{N}$ )
    (STR "Clock set bounded (transitions)"),
    assert ( $\forall (\_, \_, \_, inv) \in set\ automata. \forall (l, g) \in set\ inv.$ 
      ( $\forall (c, x) \in collect\_clock\_pairs\ g. 0 < c \wedge c \leq m \wedge x \in \mathbb{N}$ )
    (STR "Clock set bounded (invariants)"),
    assert ( $\forall (\_, \_, trans, \_) \in set\ automata. \forall (\_, \_, g, a, \_, \_, \_) \in set\ trans.$ 
      case a of In a  $\Rightarrow a \in set\ broadcast \longrightarrow g = [] \mid \_ \Rightarrow True$ )
    (STR "Broadcast receivers are unguarded"),
    assert ( $\forall (\_, U, \_, \_) \in set\ automata. U = []$ )
    STR "Urgency was removed"
  ]

```

]

**lemma** *check\_precond1*:

*is\_result check\_precond1*

$\longleftrightarrow$  *Simple\_Network\_Impl\_nat\_urge broadcast bounds' automata m num\_states num\_actions*

**unfolding** *check\_precond1\_def Simple\_Network\_Impl\_nat\_def Simple\_Network\_Impl\_nat\_urge\_def*  
*Simple\_Network\_Impl\_nat\_urge\_axioms\_def*

**by** (*simp add: is\_result\_combine Cons\_iff is\_result\_assert\_iff del: combine.simps(2)*)

**context**

**fixes** *k* :: *nat list list list*

**and** *L<sub>0</sub>* :: *nat list*

**and** *s<sub>0</sub>* :: (*nat* × *int*) *list*

**and** *formula* :: (*nat, nat, nat, int*) *formula*

**begin**

**definition** *check\_precond2* **where**

*check\_precond2*  $\equiv$

*combine* [

*assert* ( $\forall i < n\_ps. \forall (l, g) \in \text{set } ((\text{snd} \circ \text{snd} \circ \text{snd}) (\text{automata} ! i)).$

$\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x))$

(*STR "Ceiling invariants"*),

*assert* ( $\forall i < n\_ps. \forall (l, \_, g, \_) \in \text{set } ((\text{fst} \circ \text{snd} \circ \text{snd}) (\text{automata} ! i)).$

$\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x))$

(*STR "Ceiling transitions"*),

*assert* ( $\forall i < n\_ps. \forall (l, b, g, a, \text{upd}, r, l') \in \text{set } ((\text{fst} \circ \text{snd} \circ \text{snd}) (\text{automata} ! i)).$

$\forall c \in \{0..<m+1\} - \text{set } r. k ! i ! l' ! c \leq k ! i ! l ! c$ )

(*STR "Ceiling resets"*),

*assert* ( $\text{length } k = n\_ps$ )

(*STR "Ceiling length"*),

*assert* ( $\forall i < n\_ps. \text{length } (k ! i) = \text{num\_states } i$ )

(*STR "Ceiling length automata"*),

*assert* ( $\forall xs \in \text{set } k. \forall xxs \in \text{set } xs. \text{length } xxs = m + 1$ )

(*STR "Ceiling length clocks"*),

*assert* ( $\forall i < n\_ps. \forall l < \text{num\_states } i. k ! i ! l ! 0 = 0$ )

(*STR "Ceiling zero clock"*),

*assert* ( $\forall (\_, \_, \_, \text{inv}) \in \text{set } \text{automata}. \text{distinct } (\text{map } \text{fst } \text{inv})$ )

(*STR "Unambiguous invariants"*),

*assert* ( $\text{bounded } \text{bounds } (\text{map\_of } s_0)$ )

(*STR "Initial state bounded"*),

*assert* ( $\text{length } L_0 = n\_ps$ )

(*STR "Length of initial state"*),

*assert* ( $\forall i < n\_ps. L_0 ! i \in \text{fst } ' \text{set } ((\text{fst} \circ \text{snd} \circ \text{snd}) (\text{automata} ! i))$ )

(*STR "Initial state has outgoing transitions"*),

*assert* ( $\text{vars\_of\_formula } \text{formula} \subseteq \{0..<n\_vs\}$ )

(*STR "Variable set of formula"*)

]

**lemma** *check\_precond2*:

*is\_result check\_precond2*  $\longleftrightarrow$

*Simple\_Network\_Impl\_nat\_ceiling\_start\_state\_axioms broadcast bounds' automata m num\_states*  
*k L<sub>0</sub> s<sub>0</sub> formula*

**unfolding** *check\_precond2\_def Simple\_Network\_Impl\_nat\_ceiling\_start\_state\_axioms\_def*

```

    by (simp add: is_result_combine_Cons_iff is_result_assert_iff del: combine.simps(2))

end

definition
  check_precond k L0 s0 formula ≡ check_precond1 <|> check_precond2 k L0 s0 formula

lemma check_precond:
  Simple_Network_Impl_nat_ceiling_start_state broadcast bounds' automata m num_states
    num_actions k L0 s0 formula ⟷ is_result (check_precond k L0 s0 formula)
unfolding check_precond_def is_result_combine_iff check_precond1 check_precond2
  Simple_Network_Impl_nat_ceiling_start_state_def
  ..

end

derive show acconstraint act sexp formula

fun shows_exp and shows_bexp where
  shows_exp (const c) = show c |
  shows_exp (var v) = show v |
  shows_exp (if_then_else b e1 e2) =
    shows_bexp b @ " ? " @ shows_exp e1 @ " : " @ shows_exp e2 |
  shows_exp (binop _ e1 e2) = "binop " @ shows_exp e1 @ " " @ shows_exp e2 |
  shows_exp (unop _ e) = "unop " @ shows_exp e |
  shows_bexp (bexp.lt a b) = shows_exp a @ " < " @ shows_exp b |
  shows_bexp (bexp.le a b) = shows_exp a @ " <= " @ shows_exp b |
  shows_bexp (bexp.eq a b) = shows_exp a @ " = " @ shows_exp b |
  shows_bexp (bexp.ge a b) = shows_exp a @ " >= " @ shows_exp b |
  shows_bexp (bexp.gt a b) = shows_exp a @ " > " @ shows_exp b |
  shows_bexp bexp.true = "true" |
  shows_bexp (bexp.not b) = "! " @ shows_bexp b |
  shows_bexp (bexp.and a b) = shows_bexp a @ " && " @ shows_bexp b |
  shows_bexp (bexp.or a b) = shows_bexp a @ " || " @ shows_bexp b |
  shows_bexp (bexp.imply a b) = shows_bexp a @ " -> " @ shows_bexp b

instantiation bexp :: (show, show) show
begin

definition shows_prec p (e :: (_, _) bexp) rest = shows_bexp e @ rest for p

definition shows_list (es) s =
  map shows_bexp es |> intersperse ", " |> (λxs. "[" @ concat xs @ "]" @ s)
instance
  by standard (simp_all add: shows_prec_bexp_def shows_list_bexp_def showLaw_simps)

end

instantiation exp :: (show, show) show
begin

definition shows_prec p (e :: (_, _) exp) rest = shows_exp e @ rest for p

definition shows_list (es) s =

```

```

    map shows_exp es |> intersperse ", " |> (λxs. "[" @ concat xs @ "]" @ s)
instance
  by standard (simp_all add: shows_prec_exp_def shows_list_exp_def show_law_simps)

end

```

**definition**

```
pad m s = replicate m CHR " " @ s
```

**definition** *shows\_rat*  $r \equiv \text{case } r \text{ of } \text{fract.Rat } s \text{ } f \text{ } b \Rightarrow$

```
(if s then "" else "-") @ show f @ (if b ≠ 0 then "." @ show b else "")
```

**fun** *shows\_json* :: *nat* ⇒ *JSON* ⇒ *string* **where**

```

  shows_json n (Nat m) = show m |> pad n
| shows_json n (Rat r) = shows_rat r |> pad n
| shows_json n (JSON.Int r) = show r |> pad n
| shows_json n (Boolean b) = (if b then "true" else "false") |> pad n
| shows_json n Null = "null" |> pad n
| shows_json n (String s) = pad n ("" @ s @ "")
| shows_json n (JSON.Array xs) = (
  if xs = Nil then
    pad n "[]"
  else
    pad n "["
    @ concat (map (shows_json (n + 2)) xs |> intersperse ",")
    @ "]"
    @ pad n "]"
)
| shows_json n (JSON.Object xs) = (
  if xs = Nil then
    pad n "{}"
  else
    pad n "{"
    @ concat (
      map (λ(k, v). pad (n + 2) ("" @ k @ "")) @ ":" @ shows_json (n + 4) v xs
    |> intersperse ",")
    @ "}"
    @ pad n "}"
)
)

```

**instantiation** *JSON* :: *show*

**begin**

**definition** *shows\_prec*  $p$  ( $x :: \text{JSON}$ ) *rest* = *shows\_json* 0  $x$  @ *rest* **for**  $p$

**definition** *shows\_list* *jsons*  $s =$

```
map (shows_json 0) jsons |> intersperse ", " |> (λxs. "[" @ concat xs @ "]" @ s)
```

**instance**

```
by standard (simp_all add: shows_prec_JSON_def shows_list_JSON_def show_law_simps)
```

**end**

**definition** *rename\_network* **where**

```

rename_network broadcast bounds' automata renum_acts renum_vars renum_clocks renum_states
≡
let
  automata = map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
automata;
  broadcast = map renum_acts broadcast;
  bounds' = map (λ(a,b,c). (renum_vars a, b, c)) bounds'
in
  (broadcast, automata, bounds')
```

**definition**

```

show_clock inv_renum_clocks = show o inv_renum_clocks
```

**definition**

```

show_locs inv_renum_states = show o map_index inv_renum_states
```

**definition**

```

show_vars inv_renum_vars = show o map_index (λi v. show (inv_renum_vars i) @ "=" @
show v)
```

**definition**

```

show_state inv_renum_states inv_renum_vars ≡ λ(L, vs).
let L = show_locs inv_renum_states L; vs = show_vars inv_renum_vars vs in
"<" @ L @ ">", "<" @ vs @ ">"
```

**definition** *do\_rename\_mc* **where**

```

do_rename_mc f dc broadcast bounds' automata k urge L0 s0 formula
  m num_states num_actions renum_acts renum_vars renum_clocks renum_states
  inv_renum_states inv_renum_vars inv_renum_clocks
≡
let
  _ = println (STR "Checking renaming");
  formula = (if dc then formula.EX (not sexp.true) else formula);
  renaming_valid = Simple_Network_Rename_Formula_String_Defs.check_renaming
    broadcast bounds' renum_acts renum_vars renum_clocks renum_states automata urge for-
mula L0 s0;
  _ = println (STR "Renaming network");
  (broadcast, automata, bounds') = rename_network
    broadcast bounds' (map (conv_urge urge) automata)
    renum_acts renum_vars renum_clocks renum_states;
  _ = trace_level 4 (λ_. return (STR "Automata after renaming"));
  _ = map (λa. show a |> String.implode |> (λs. trace_level 4 (λ_. return s))) automata;
  _ = println (STR "Renaming formula");
  formula =
    (if dc then formula.EX (not sexp.true) else map_formula renum_states renum_vars id for-
mula);
  _ = println (STR "Renaming state");
  L0 = map_index renum_states L0;
  s0 = map (λ(x, v). (renum_vars x, v)) s0;
  show_clock = show o inv_renum_clocks;
  show_state = show_state inv_renum_states inv_renum_vars
```

```

in
if is_result renaming_valid then do {
  let _ = println (STR "Checking preconditions");
  let r = Simple_Network_Impl_nat_defs.check_precond
    broadcast bounds' automata m num_states num_actions k L0 s0 formula;
  let _ = (case r of Result _ ⇒ ()
    | Error es ⇒
      let
        _ = println STR ""';
        _ = println STR "The following pre-conditions were not satisfied:"
      in
        let _ = map println es in println STR ""');
  let _ = println (STR "Running precondition mc");
  let r = f show_clock show_state
    broadcast bounds' automata m num_states num_actions k L0 s0 formula;
  Some r
}
else do {
  let _ = println (STR "The following conditions on the renaming were not satisfied:");
  let _ = the_errors renaming_valid |> map println;
  None}

```

**definition** *rename\_mc where*

```

rename_mc dc broadcast bounds' automata k urge L0 s0 formula
  m num_states num_actions renum_acts renum_vars renum_clocks renum_states
  inv_renum_states inv_renum_vars inv_renum_clocks
≡
do {
  let r = do_rename_mc (if dc then precondition_dc else precondition_mc)
    dc broadcast bounds' automata k urge L0 s0 formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks;
  case r of Some r ⇒ do {
    r ← r;
    case r of
      None ⇒ return Preconds_Unsat
    | Some False ⇒ return Unsat
    | Some True ⇒ return Sat
  }
  | None ⇒ return Renaming_Failed
}

```

**theorem** *model\_check\_rename:*

```

<emp> rename_mc False broadcast bounds automata k urge L0 s0 formula
  m num_states num_actions renum_acts renum_vars renum_clocks renum_states
  inv_renum_states inv_renum_vars inv_renum_clocks
<λ Sat ⇒ ↑(
  (¬ has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0)) →
  N broadcast automata bounds, (L0, map_of s0, λ_. 0) ⊨ formula
))

```

```

| Unsat  $\Rightarrow \uparrow($ 
   $(\neg \text{has\_deadlock } (N \text{ broadcast automata bounds}) (L_0, \text{map\_of } s_0, \lambda\_ . 0) \longrightarrow$ 
     $\neg N \text{ broadcast automata bounds}, (L_0, \text{map\_of } s_0, \lambda\_ . 0) \models \text{formula}$ 
   $)$ 
| Renaming_Failed  $\Rightarrow \uparrow(\neg \text{Simple\_Network\_Rename\_Formula}$ 
  broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
formula)
| Preconds_Unsat  $\Rightarrow \uparrow(\neg \text{Simple\_Network\_Impl\_nat\_ceiling\_start\_state}$ 
   $(\text{map renum\_acts broadcast})$ 
   $(\text{map } (\lambda(a,p). (\text{renum\_vars } a, p)) \text{ bounds})$ 
   $(\text{map\_index } (\text{renum\_automaton renum\_acts renum\_vars renum\_clocks renum\_states})$ 
     $(\text{map } (\text{conv\_urge urge}) \text{ automata}))$ 
  m num_states num_actions k
   $(\text{map\_index renum\_states } L_0) (\text{map } (\lambda(x, v). (\text{renum\_vars } x, v)) s_0)$ 
   $(\text{map\_formula renum\_states renum\_vars id formula}))$ 
 $>_t$ 
proof –
have *:
  Simple_Network_Rename_Formula_String
    broadcast bounds renum_acts renum_vars renum_clocks renum_states automata urge
formula s0 L0
  = Simple_Network_Rename_Formula
    broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
formula

unfolding
  Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
  Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
  Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
using infinite_literal by auto
define A where A  $\equiv N \text{ broadcast automata bounds}$ 
define check where check  $\equiv A, (L_0, \text{map\_of } s_0, \lambda\_ . 0) \models \text{formula}$ 
define A' where A'  $\equiv N$ 
   $(\text{map renum\_acts broadcast})$ 
   $(\text{map\_index } (\text{renum\_automaton renum\_acts renum\_vars renum\_clocks renum\_states})$ 
     $(\text{map } (\text{conv\_urge urge}) \text{ automata}))$ 
   $(\text{map } (\lambda(a,p). (\text{renum\_vars } a, p)) \text{ bounds})$ 
define check' where check'  $\equiv$ 
  A', (map_index renum_states L0, map_of (map (λ(x, v). (renum_vars x, v)) s0), λ_ . 0)  $\models$ 
map_formula renum_states renum_vars id formula
define preconds_sat where preconds_sat  $\equiv$ 
  Simple_Network_Impl_nat_ceiling_start_state
   $(\text{map renum\_acts broadcast})$ 
   $(\text{map } (\lambda(a,p). (\text{renum\_vars } a, p)) \text{ bounds})$ 
   $(\text{map\_index } (\text{renum\_automaton renum\_acts renum\_vars renum\_clocks renum\_states})$ 
     $(\text{map } (\text{conv\_urge urge}) \text{ automata}))$ 
  m num_states num_actions k
   $(\text{map\_index renum\_states } L_0) (\text{map } (\lambda(x, v). (\text{renum\_vars } x, v)) s_0)$ 
   $(\text{map\_formula renum\_states renum\_vars id formula})$ 
define renaming_valid where renaming_valid  $\equiv$ 
  Simple_Network_Rename_Formula
    broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
formula

```

```

have [simp]: check  $\longleftrightarrow$  check'
  if renaming_valid
  using that unfolding check_def check'_def A_def A'_def renaming_valid_def
  by (rule Simple_Network_Rename_Formula.models_iff[symmetric])
have test[symmetric, simp]:
  Simple_Network_Language_Model_Checking.has_deadlock A (L0, map_of s0, λ_. 0)
 $\longleftrightarrow$  Simple_Network_Language_Model_Checking.has_deadlock A'
  (map_index renum_states L0, map_of (map (λ(x, y). (renum_vars x, y)) s0), λ_. 0)
  if renaming_valid
  using that unfolding check_def check'_def A_def A'_def renaming_valid_def
  unfolding Simple_Network_Rename_Formula_def
  by (elim conjE) (rule Simple_Network_Rename_Start.has_deadlock_iff[symmetric])
note [sep_heap_rules] =
  model_check[
    of _ _
    map renum_acts broadcast map (λ(a, p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata)
    m num_states num_actions k map_index renum_states L0 map (λ(x, v). (renum_vars x, v))
  ]
so
  map_formula renum_states renum_vars id formula,
  folded A'_def preconds_sat_def renaming_valid_def, folded check'_def, simplified
]
show ?thesis
  unfolding rename_mc_def do_rename_mc_def rename_network_def
  unfolding if_False
  unfolding Simple_Network_Rename_Formula_String_Defs.check_renaming[symmetric] *
Let_def
  unfolding
    A_def[symmetric] check_def[symmetric]
    preconds_sat_def[symmetric] renaming_valid_def[symmetric]
  by (sep_auto simp: model_checker.refine[symmetric] split: bool.splits)
qed

theorem deadlock_check_rename:
  <emp> rename_mc True broadcast bounds automata k urge L0 s0 formula
  m num_states num_actions renum_acts renum_vars renum_clocks renum_states
  inv_renum_states inv_renum_vars inv_renum_clocks
  <λ Sat  $\Rightarrow$   $\uparrow$ ( has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0))
  | Unsat  $\Rightarrow$   $\uparrow$ ( $\neg$  has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0))
  | Renaming_Failed  $\Rightarrow$   $\uparrow$ ( $\neg$  Simple_Network_Rename_Formula
    broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
    (formula.EX (not sexp.true)))
  | Preconds_Unsat  $\Rightarrow$   $\uparrow$ ( $\neg$  Simple_Network_Impl_nat_ceiling_start_state
    (map renum_acts broadcast)
    (map (λ(a, p). (renum_vars a, p)) bounds)
    (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata))
    m num_states num_actions k
    (map_index renum_states L0) (map (λ(x, v). (renum_vars x, v)) s0)
    (formula.EX (not sexp.true)))
  >t
proof -
  have *:

```

```

Simple_Network_Rename_Formula_String
  broadcast bounds renum_acts renum_vars renum_clocks renum_states automata urge
  (formula.EX (not sexp.true)) s0 L0
= Simple_Network_Rename_Formula
  broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
  (formula.EX (not sexp.true))

unfolding
  Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
  Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
  Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
using infinite_literal by auto
define A where A ≡ N broadcast automata bounds
define A' where A' ≡ N
  (map renum_acts broadcast)
  (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
    (map (conv_urge urge) automata))
  (map (λ(a,p). (renum_vars a, p)) bounds)
define preconds_sat where preconds_sat ≡
  Simple_Network_Impl_nat_ceiling_start_state
  (map renum_acts broadcast)
  (map (λ(a,p). (renum_vars a, p)) bounds)
  (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
    (map (conv_urge urge) automata))
  m num_states num_actions k
  (map_index renum_states L0) (map (λ(x, v). (renum_vars x, v)) s0)
  (formula.EX (not sexp.true))
define renaming_valid where renaming_valid ≡
  Simple_Network_Rename_Formula
  broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
  (formula.EX (not sexp.true))
have test[symmetric, simp]:
  Simple_Network_Language_Model_Checking.has_deadlock A (L0, map_of s0, λ_. 0)
  ↔ Simple_Network_Language_Model_Checking.has_deadlock A'
  (map_index renum_states L0, map_of (map (λ(x, y). (renum_vars x, y)) s0), λ_. 0)
if renaming_valid
using that unfolding check_def A_def A'_def renaming_valid_def Simple_Network_Rename_Formula_def
by (elim conjE) (rule Simple_Network_Rename_Start.has_deadlock_iff[symmetric])
note [sep_heap_rules] =
  deadlock_check[
    of _ _
    map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata)
    m num_states num_actions k map_index renum_states L0 map (λ(x, v). (renum_vars x, v))
  ] s0,
  folded preconds_sat_def A'_def renaming_valid_def,
  simplified
]
show ?thesis
unfolding rename_mc_def do_rename_mc_def rename_network_def
unfolding if_True
unfolding Simple_Network_Rename_Formula_String_Defs.check_renaming[symmetric] *
Let_def

```

```

    unfolding A_def[symmetric] preconds_sat_def[symmetric] renaming_valid_def[symmetric]
    by (sep_auto simp: deadlock_checker.refine[symmetric] split: bool.splits)
qed

```

**Code Setup for the Model Checker**    `lemmas [code] =`  
*reachability\_checker\_def*  
*Alw\_ev\_checker\_def*  
*leadsto\_checker\_def*  
*model\_checker\_def*[*unfolded PR\_CONST\_def*]

`lemmas [code] =`  
*Prod\_TA\_Defs.n\_ps\_def*  
*Simple\_Network\_Impl\_Defs.n\_vs\_def*  
*automaton\_of\_def*  
*Simple\_Network\_Impl\_nat\_defs.pairs\_by\_action\_impl\_def*  
*Simple\_Network\_Impl\_nat\_defs.all\_actions\_from\_vec\_def*  
*Simple\_Network\_Impl\_nat\_defs.all\_actions\_by\_state\_def*  
*Simple\_Network\_Impl\_nat\_defs.compute\_upds\_impl\_def*  
*Simple\_Network\_Impl\_nat\_defs.actions\_by\_state\_def*  
*Simple\_Network\_Impl\_nat\_defs.check\_boundedi\_def*  
*Simple\_Network\_Impl\_nat\_defs.get\_committed\_def*  
*Simple\_Network\_Impl\_nat\_defs.make\_combs\_def*  
*Simple\_Network\_Impl\_nat\_defs.trans\_map\_def*  
*Simple\_Network\_Impl\_nat\_defs.actions\_by\_state'\_def*  
*Simple\_Network\_Impl\_nat\_defs.bounds\_map\_def*  
*Simple\_Network\_Impl\_nat\_defs.bin\_actions\_def*  
*mk\_updsi\_def*

**lemma** (in *Simple\_Network\_Impl\_nat\_defs*) *bounded\_s0\_iff*:  
*bounded bounds (map\_of s0)  $\longleftrightarrow$  bounded (map\_of bounds') (map\_of s0)*  
**unfolding** *bounds\_def snd\_conv ..*

**lemma** *int\_Nat\_range\_iff*:  
*(n :: int)  $\in \mathbb{N} \longleftrightarrow n \geq 0$  for n*  
**using** *zero\_le\_imp\_eq\_int* **unfolding** *Nats\_def* **by** *auto*

`lemmas [code] =`  
*Simple\_Network\_Impl\_nat\_ceiling\_start\_state\_def*  
*Simple\_Network\_Impl\_nat\_ceiling\_start\_state\_axioms\_def*[  
   *unfolded Simple\_Network\_Impl\_nat\_defs.bounded\_s0\_iff*]  
*Simple\_Network\_Impl\_nat\_def*[*unfolded int\_Nat\_range\_iff*]  
*Simple\_Network\_Impl\_nat\_urge\_def*  
*Simple\_Network\_Impl\_nat\_urge\_axioms\_def*

`lemmas [code_unfold] =` *bounded\_def dom\_map\_of\_conv\_image\_fst*

**export\_code** *Simple\_Network\_Impl\_nat\_ceiling\_start\_state\_axioms*

**export\_code** *precond\_mc* **in** *SML* **module\_name** *Test*

**export\_code** *precond\_dc* **checking** *SML*

**Code Setup for Renaming**    `lemmas [code] =`

*Simple\_Network\_Rename\_Formula\_String\_def*  
*Simple\_Network\_Impl.clk\_set'\_def*  
*Simple\_Network\_Impl.clkp\_set'\_def*

**lemmas** [code] =

*Simple\_Network\_Rename\_Defs.renum\_automaton\_def*  
*Simple\_Network\_Rename\_Defs.renum\_constraint\_def*  
*Simple\_Network\_Rename\_Defs.map\_constraint\_def*  
*Simple\_Network\_Rename\_Defs.renum\_reset\_def*  
*Simple\_Network\_Rename\_Defs.renum\_upd\_def*  
*Simple\_Network\_Rename\_Defs.renum\_act\_def*  
*Simple\_Network\_Rename\_Defs.renum\_exp\_def*  
*Simple\_Network\_Rename\_Defs.renum\_bexp\_def*  
*Simple\_Network\_Rename\_Formula\_String\_Defs.check\_renaming\_def*  
*Simple\_Network\_Impl\_nat\_defs.check\_precond\_def*  
*Simple\_Network\_Impl\_nat\_defs.check\_precond1\_def*[unfolded int\_Nat\_range\_iff]  
*Simple\_Network\_Impl\_nat\_defs.check\_precond2\_def*[  
 unfolded Simple\_Network\_Impl\_nat\_defs.bounded\_s0\_iff]

**lemma** (in Prod\_TA\_Defs) states\_mem\_iff:

$L \in \text{states} \iff \text{length } L = n\_ps \wedge (\forall i. i < n\_ps \longrightarrow$   
 $(\exists (l, b, g, a, r, u, l') \in \text{fst} (\text{snd} (\text{snd} (\text{fst} (\text{snd } A) ! i))). L ! i = l \vee L ! i = l'))$

**unfolding** states\_def trans\_def N\_def **by** (auto split: prod.split)

**lemmas** [code\_unfold] =

*Prod\_TA\_Defs.states\_mem\_iff*  
*Simple\_Network\_Impl.act\_set\_compute*  
*Simple\_Network\_Impl\_Defs.var\_set\_compute*  
*Simple\_Network\_Impl\_Defs.loc\_set\_compute*  
*setcompr\_eq\_image*  
*Simple\_Network\_Impl.length\_automata\_eq\_n\_ps*[symmetric]

**export\_code** rename\_mc **in** SML **module\_name** Test

**Calculating the Clock Ceiling** context *Simple\_Network\_Impl\_nat\_defs*  
**begin**

**definition** *clkp\_inv* *i l* ≡

$\bigcup g \in \text{set} (\text{filter } (\lambda(a, b). a = l) (\text{snd} (\text{snd} (\text{snd} (\text{automata} ! i))))). \text{collect\_clock\_pairs } (\text{snd } g)$

**definition** *clkp\_set''* *i l* ≡

$\text{clkp\_inv } i l \cup (\bigcup (l', b, g, \_) \in \text{set} (\text{fst} (\text{snd} (\text{snd} (\text{automata} ! i)))).$   
 $\text{if } l' = l \text{ then collect\_clock\_pairs } g \text{ else } \{\})$

**definition**

$\text{collect\_resets } i l = (\bigcup (l', b, g, a, f, r, \_) \in \text{set} (\text{fst} (\text{snd} (\text{snd} (\text{automata} ! i)))).$   
 $\text{if } l' = l \text{ then set } r \text{ else } \{\})$

**context**

**fixes** *q c* :: nat

**begin**

**definition** *n* ≡ num\_states *q*

**definition**  $V \equiv \lambda v. v \leq n$

**definition**

$bound\_g\ l \equiv$   
 $Max\ (\{0\} \cup \bigcup ((\lambda (x, d). \text{ if } x = c \text{ then } \{d\} \text{ else } \{ \}) \text{ ' clkp\_set'' } q\ l))$

**definition**

$bound\_inv\ l \equiv$   
 $Max\ (\{0\} \cup \bigcup ((\lambda (x, d). \text{ if } x = c \text{ then } \{d\} \text{ else } \{ \}) \text{ ' clkp\_inv } q\ l))$

**definition**

$bound\ l \equiv max\ (bound\_g\ l)\ (bound\_inv\ l)$

**definition**

$resets\ l \equiv$   
 $fold$   
 $(\lambda (l1, b, g, a, f, r, l')\ xs. \text{ if } l1 \neq l \vee l' \in set\ xs \vee c \in set\ r \text{ then } xs \text{ else } (l' \# xs))$   
 $(fst\ (snd\ (snd\ (automata\ !\ q))))$   
 $\square$

Edges in the direction nodes to single sink.

**definition**

$E'\ l \equiv resets\ l$

Turning around the edges to obtain a single source shortest paths problem.

**definition**

$E\ l \equiv \text{ if } l = n \text{ then } [0..<n] \text{ else filter } (\lambda l'. l \in set\ (E'\ l'))\ [0..<n]$

Weights already turned around.

**definition**

$W\ l\ l' \equiv \text{ if } l = n \text{ then } -\ bound\ l' \text{ else } 0$

**definition**  $G$  where

$G \equiv (\mid gi\_V = V, gi\_E = E, gi\_V0 = [n], \dots = W \mid)$

**definition**

$local\_ceiling\_single \equiv$   
 $let$   
 $w = calc\_shortest\_scc\_paths\ G\ n$   
 $in$   
 $map\ (\lambda x. \text{ case } x \text{ of } None \Rightarrow 0 \mid Some\ x \Rightarrow nat(-x))\ w$

**end**

**definition**

```

local_ceiling ≡
  rev $
  fold
    (λ q xs.
      (λ x. rev x # xs) $
      fold
        (λ l xs.
          (λ x. (0 # rev x) # xs) $
          fold
            (λ c xs. local_ceiling_single q c ! l # xs)
            [1..<Suc m]
            []
          )
        [0..<n q]
        []
      )
    [0..<n_ps]
    []

```

**end****lemmas** [code] =

```

Simple_Network_Impl_nat_defs.local_ceiling_def
Simple_Network_Impl_nat_defs.local_ceiling_single_def
Simple_Network_Impl_nat_defs.n_def
Simple_Network_Impl_nat_defs.G_def
Simple_Network_Impl_nat_defs.W_def
Simple_Network_Impl_nat_defs.V_def
Simple_Network_Impl_nat_defs.E'_def
Simple_Network_Impl_nat_defs.E_def
Simple_Network_Impl_nat_defs.resets_def
Simple_Network_Impl_nat_defs.bound_def
Simple_Network_Impl_nat_defs.bound_inv_def
Simple_Network_Impl_nat_defs.bound_g_def
Simple_Network_Impl_nat_defs.collect_resets_def
Simple_Network_Impl_nat_defs.clkp_set''_def
Simple_Network_Impl_nat_defs.clkp_inv_def

```

**export\_code** Simple\_Network\_Impl\_nat\_defs.local\_ceiling **checking** SML\_imp**Calculating the Renaming** **definition** mem\_assoc x = list\_ex (λ(y, \_). x = y)**definition** mk\_renaming str xs ≡

```

do {
  mapping ← fold_error
    (λx m.
      if mem_assoc x m then Error [STR "Duplicate name: " + str x] else (x,length m) # m |>
    ) xs [];
  Result

```

```

Result (let
  m = map_of mapping;
  f = (λx.
    case m x of
      None ⇒ let _ = println (STR "Key error: " + str x) in undefined
    | Some v ⇒ v);
  m = map_of (map prod.swap mapping);
  f_inv = (λx.
    case m x of
      None ⇒ let _ = println (STR "Key error: " + String.implode (show x)) in undefined
    | Some v ⇒ v)
  in (f, f_inv)
)
}

```

**definition**

```

extend_domain m d n ≡
  let
    (i, xs) = fold
      (λx (i, xs). if x ∈ set d then (i + 1, (x, i + 1) # xs) else (i, xs)) d (n, []);
    m' = map_of xs
  in
    (λx. if x ∈ set d then the (m' x) else m x)

```

**lemma** *is\_result\_make\_err*:

```

is_result (make_err m e) ⟷ False
by (cases e) auto

```

**lemma** *is\_result\_assert\_msg\_iff*:

```

is_result (assert_msg b m r) ⟷ is_result r ∧ b
unfolding assert_msg_def by (simp add: is_result_make_err)

```

**context** *Simple\_Network\_Impl*

**begin**

**definition** *action\_set* **where**

```

action_set ≡
  (⋃ (_, _, trans, _) ∈ set automata. ⋃ (_, _, _, a, _, _, _) ∈ set trans. set_act a)
  ∪ set broadcast

```

**definition** *loc\_set'* **where**

```

loc_set' p = (⋃ (l, _, _, _, _, l') ∈ set (fst (snd (snd (automata ! p)))). {l, l'}) for p

```

**end**

**definition**

```

concat_str = String.implode o concat o map String.explode

```

**Unsafe Glue Code** **definition** *list\_of\_set'* :: 'a set ⇒ 'a list **where**

```

list_of_set' xs = undefined

```

**definition** *list\_of\_set* :: 'a set  $\Rightarrow$  'a list **where**  
*list\_of\_set* *xs* = *list\_of\_set'* *xs* |> *remdups*

**code\_printing**

**constant** *list\_of\_set'*  $\rightarrow$  (*SML*) (*fn Set xs => xs*) \_  
**and** (*OCaml*) (*fun x -> match x with Set xs -> xs*) \_

**definition**

*mk\_renaming'* *xs*  $\equiv$  *mk\_renaming* (*String.implode o show*) *xs*

**definition** *make\_renaming*  $\equiv$   $\lambda$  *broadcast automata bounds*.

*let*  
*action\_set* = *Simple\_Network\_Impl.action\_set automata broadcast* |> *list\_of\_set*;  
*clk\_set* = *Simple\_Network\_Impl.clk\_set' automata* |> *list\_of\_set*;  
*clk\_set* = *clk\_set* @ [*STR "urge"*];  
*loc\_set'* = ( $\lambda i$ . *Simple\_Network\_Impl.loc\_set' automata i* |> *list\_of\_set*);  
*loc\_set* = *Prod\_TA\_Defs.loc\_set*  
 (*set broadcast, map automaton\_of automata, map\_of bounds*);  
*loc\_set\_diff* = ( $\lambda i$ . *loc\_set* - *Simple\_Network\_Impl.loc\_set' automata i* |> *list\_of\_set*);  
*loc\_set* = *list\_of\_set loc\_set*;  
*var\_set* = *Prod\_TA\_Defs.var\_set*  
 (*set broadcast, map automaton\_of automata, map\_of bounds*) |> *list\_of\_set*;  
*n\_ps* = *length automata*;  
*num\_actions* = *length action\_set*;  
*m* = *length (remdups clk\_set)*;  
*num\_states\_list* = *map* ( $\lambda i$ . *loc\_set' i* |> *remdups* |> *length*) [*0..<n\_ps*];  
*num\_states* = ( $\lambda i$ . *num\_states\_list* ! *i*);  
*mk\_renaming* = *mk\_renaming* ( $\lambda x$ . *x*)  
*in do* {  
 (*renum\_acts*, \_), (*renum\_clocks*, *inv\_renum\_clocks*), (*renum\_vars*, *inv\_renum\_vars*)  $\leftarrow$   
*mk\_renaming action\_set* <|> *mk\_renaming clk\_set* <|> *mk\_renaming var\_set*;  
*let renum\_clocks* = *Suc o renum\_clocks*;  
*let inv\_renum\_clocks* = ( $\lambda c$ . *if* *c* = 0 *then STR "0"* *else inv\_renum\_clocks (c - 1)*);  
*renum\_states\_list'*  $\leftarrow$  *combine\_map* ( $\lambda i$ . *mk\_renaming' (loc\_set' i)*) [*0..<n\_ps*];  
*let renum\_states\_list* = *map fst renum\_states\_list'*;  
*let renum\_states\_list* = *map\_index*  
 ( $\lambda i m$ . *extend\_domain m (loc\_set\_diff i) (length (loc\_set' i)) renum\_states\_list*);  
*let renum\_states* = ( $\lambda i$ . *renum\_states\_list* ! *i*);  
*let inv\_renum\_states* = ( $\lambda i$ . *map snd renum\_states\_list' i*);  
*assert* (*fst* ' *set bounds*  $\subseteq$  *set var\_set*)  
*STR "State variables are declared but do not appear in model"*;  
*Result* (*m, num\_states, num\_actions, renum\_acts, renum\_vars, renum\_clocks, renum\_states,*  
*inv\_renum\_states, inv\_renum\_vars, inv\_renum\_clocks*)  
}

**definition** *preproc\_mc*  $\equiv$   $\lambda dc$  *ids\_to\_names (broadcast, automata, bounds)* *L<sub>0</sub> s<sub>0</sub> formula*.

*let* \_ = *println (STR "Make renaming")* *in*  
*case make\_renaming broadcast automata bounds of*  
*Error e*  $\Rightarrow$  *return (Error e)*  
| *Result* (*m, num\_states, num\_actions, renum\_acts, renum\_vars, renum\_clocks, renum\_states,*  
*inv\_renum\_states, inv\_renum\_vars, inv\_renum\_clocks*)  $\Rightarrow$  *do* {

```

let _ = println (STR "Renaming");
let (broadcast', automata', bounds') = rename_network
  broadcast bounds automata renum_acts renum_vars renum_clocks renum_states;
let _ = println (STR "Calculating ceiling");
let k = Simple_Network_Impl_nat_defs.local_ceiling broadcast' bounds' automata' m num_states;
let _ = println (STR "Running model checker");
let inv_renum_states = ( $\lambda i$ . ids_to_names i o inv_renum_states i);
r  $\leftarrow$  rename_mc dc broadcast bounds automata k STR "_urge" L0 s0 formula
  m num_states num_actions renum_acts renum_vars renum_clocks renum_states
  inv_renum_states inv_renum_vars inv_renum_clocks;
return (Result r)
}

```

#### definition

```
err s = Error [s]
```

#### definition

```

do_preproc_mc  $\equiv$   $\lambda dc$  ids_to_names (broadcast, automata, bounds) L0 s0 formula. do {
  r  $\leftarrow$  preproc_mc dc ids_to_names (broadcast, automata, bounds) L0 s0 formula;
  return (case r of
    Error es  $\Rightarrow$ 
      intersperse (STR "[ $\Leftarrow$ ]") es
      |> concat_str
      |> ( $\lambda e$ . STR "Error during preprocessing:[ $\Leftarrow$ ]" + e)
      |> err
    | Result Renaming_Failed  $\Rightarrow$  STR "Renaming failed" |> err
    | Result Preconds_Unsat  $\Rightarrow$  STR "Input invalid" |> err
    | Result Unsat  $\Rightarrow$ 
      (if dc then STR "Model has no deadlock!" else STR "Property is not satisfied!") |> Result
    | Result Sat  $\Rightarrow$ 
      (if dc then STR "Model has a deadlock!" else STR "Property is satisfied!") |> Result
  )
}

```

#### lemmas [code] =

```

Simple_Network_Impl.action_set_def
Simple_Network_Impl.loc_set'_def

```

```
export_code do_preproc_mc in SML module_name Main
```

#### definition parse where

```

parse parser s  $\equiv$  case parse_all lx_ws parser s of
  Inl e  $\Rightarrow$  Error [e () "Parser: " |> String.implode]
  | Inr r  $\Rightarrow$  Result r

```

#### definition get\_nat :: string $\Rightarrow$ JSON $\Rightarrow$ nat Error\_List\_Monad.result where

```

get_nat s json  $\equiv$  case json of
  Object as  $\Rightarrow$  Error []
  | _  $\Rightarrow$  Error [STR "JSON Get: expected object"]
for json

```

#### definition of\_object :: JSON $\Rightarrow$ (string $\rightarrow$ JSON) Error\_List\_Monad.result where

```
of_object json  $\equiv$  case json of
```

```

    Object as ⇒ map_of as |> Result
  | _ ⇒ Error [STR "json_to_map: expected object"]
for json

```

**definition** *get* **where**

```

get m x ≡ case m x of
  None ⇒ Error [STR "(Get) key not found: " + String.implode (show x)]
  | Some a ⇒ Result a

```

**definition**

```

get_default def m x ≡ case m x of None ⇒ def | Some a ⇒ a

```

**definition** *default* **where**

```

default def x ≡ case x of Result s ⇒ s | Error e ⇒ def

```

**definition** *of\_array* **where**

```

of_array json ≡ case json of
  JSON.Array s ⇒ Result s
  | _ ⇒ Error [STR "of_array: expected sequence"]
for json

```

**definition** *of\_string* **where**

```

of_string json ≡ case json of
  JSON.String s ⇒ Result (String.implode s)
  | _ ⇒ Error [STR "of_array: expected sequence"]
for json

```

**definition** *of\_nat* **where**

```

of_nat json ≡ case json of
  JSON.Nat n ⇒ Result n
  | _ ⇒ Error [STR "of_nat: expected natural number"]
for json

```

**definition** *of\_int* **where**

```

of_int json ≡ case json of
  JSON.Int n ⇒ Result n
  | _ ⇒ Error [STR "of_int: expected integral number"]
for json

```

**definition** [*consuming*]:

```

lx_underscore = exactly "_" with (λ_. CHR "_")

```

**definition** [*consuming*]:

```

lx_hyphen = exactly "-" with (λ_. CHR "-")

```

**definition** [*consuming*]:

```

ta_var_ident ≡
  (lx_alphanum || lx_underscore)
  -- Parser_Combinator.repeat (lx_alphanum || lx_underscore || lx_hyphen)
  with uncurry (#)

```

**definition** [*consuming*]:

```

parse_bound ≡ ta_var_ident --

```

*exactly* "[*''* \*--- *lx\_int* --- *exactly* "*''* \*--- *lx\_int* ---\* *exactly* '*''*]"

**definition** *parse\_bounds*  $\equiv$  *parse\_list'* (*lx\_ws* \*--- *parse\_bound* with ( $\lambda(s,p). (String.implode\ s, p)$ )))

**lemma**

*parse* *parse\_bounds* (*STR* "*id*[-1:2], *id*[-1:0]'")  
 $=$  *Result* [(*STR* "*id*", -1, 2), (*STR* "*id*", -1, 0)]  
**by** *eval*

**lemma** *parse* *parse\_bounds* (*STR* "") = *Result* []  
**by** *eval*

**definition** [*consuming*]:

*scan\_var* = *ta\_var\_ident*

**abbreviation** *seq\_ignore\_left\_ws* (**infixr** \*\*--- 60)  
**where**  $p$  \*\*---  $q \equiv$  *token*  $p$  \*---  $q$  **for**  $p\ q$

**abbreviation** *seq\_ignore\_right\_ws* (**infixr** ---\*\* 60)  
**where**  $p$  ---\*\*  $q \equiv$  *token*  $p$  ---\*  $q$  **for**  $p\ q$

**abbreviation** *seq\_ws* (**infixr** --- 60)  
**where** *seq\_ws*  $p\ q \equiv$  *token*  $p$  ---  $q$  **for**  $p\ q$

**definition** *scan\_acconstraint* **where** [*unfolded Let\_def, consuming*]:

*scan\_acconstraint*  $\equiv$   
*let* *scan* =  
( $\lambda s\ c. scan\_var$  --- *exactly*  $s$  \*\*--- *token* *lx\_int* with ( $\lambda(x, y). c\ (String.implode\ x)\ y)$ ) *in*  
(  
*scan* "<" *lt* ||  
*scan* "<=" *le* ||  
*scan* "==" *eq* ||  
*scan* "=" *eq* ||  
*scan* ">=" *ge* ||  
*scan* ">" *gt*  
)

**definition** [*consuming*]:

*scan\_parens* *lparen* *rparen* *inner*  $\equiv$  *exactly* *lparen* \*\*--- *inner* ---\*\* *token* (*exactly* *rparen*)

**definition** [*consuming*]: *scan\_loc*  $\equiv$

(*scan\_var* --- (*exactly* "*''*" \*--- *scan\_var*))  
with ( $\lambda\ (p, s). loc\ (String.implode\ p)\ (String.implode\ s)$ )

**definition** [*consuming*]: *scan\_bexp\_elem*  $\equiv scan\_acconstraint \parallel scan\_loc$

**abbreviation** *scan\_parens'*  $\equiv scan\_parens\ "("\ ")"$

**definition** [*consuming*]: *scan\_infix\_pair*  $a\ b\ s \equiv a$  --- *exactly*  $s$  \*\*--- *token*  $b$

**lemma** [*fundef\_cong*]:

**assumes**  $\bigwedge l2. ll\_fuel\ l2 \leq ll\_fuel\ l' \implies A\ l2 = A'\ l2$

```

assumes  $\bigwedge l2. ll\_fuel\ l2 + (if\ length\ s > 0\ then\ 1\ else\ 0) \leq ll\_fuel\ l' \implies B\ l2 = B'\ l2$ 
assumes  $s = s'\ l=l'$ 
shows  $scan\_infix\_pair\ A\ B\ s\ l = scan\_infix\_pair\ A'\ B'\ s'\ l'$ 
using assms unfolding scan_infix_pair_def gen_token_def
by (cases s; intro Parser_Combinator.bind_cong repeat_cong assms) auto

```

```

lemma [fundef_cong]:
assumes  $\bigwedge l2. ll\_fuel\ l2 < ll\_fuel\ l \implies A\ l2 = A'\ l2\ l = l'$ 
shows  $scan\_parens'\ A\ l = scan\_parens'\ A'\ l'$ 
using assms unfolding scan_parens_def gen_token_def
by (intro Parser_Combinator.bind_cong repeat_cong assms) auto

```

```

lemma token_cong[fundef_cong]:
assumes  $\bigwedge l2. ll\_fuel\ l2 \leq ll\_fuel\ l \implies A\ l2 = A'\ l2\ l = l'$ 
shows  $token\ A\ l = token\ A'\ l'$ 
using assms unfolding scan_parens_def gen_token_def
by (intro Parser_Combinator.bind_cong repeat_cong assms) auto

```

```

lemma is_cparser_scan_parens'[parser_rules]:
is_cparser (scan_parens' a)
unfolding scan_parens_def by simp

```

**fun** *aexp* **and** *mexp* **and** *scan\_exp* **and** *scan\_7* **and** *scan\_6* **and** *scan\_0* **where** — slow: 90s

```

aexp ::=
  token lx_int with exp.const || token scan_var with exp.var o String.implode ||
  scan_parens' (scan_exp --- exactly "?" **--- scan_7 --- exactly ":" **--- scan_exp)
  with ( $\lambda (e1, b, e2). exp.if\_then\_else\ b\ e1\ e2$ ) ||
  tk_lparen **--- scan_exp ---** tk_rparen
| mexp ::= chainL1 aexp (multiplicative_op with ( $\lambda f\ a\ b. exp.binop\ f\ a\ b$ ))
| scan_exp ::= chainL1 mexp (additive_op with ( $\lambda f\ a\ b. exp.binop\ f\ a\ b$ ))
| scan_7 ::=
  scan_infix_pair scan_6 scan_7 "->" with uncurry bexp.imply ||
  scan_infix_pair scan_6 scan_7 "||" with uncurry bexp.or ||
  scan_6 |
scan_6 ::=
  scan_infix_pair scan_0 scan_6 "&&" with uncurry bexp.and ||
  scan_0 |
scan_0 ::=
  (exactly "~" || exactly "!") **--- scan_parens' scan_7 with bexp.not ||
  token (exactly "true") with ( $\lambda \_. bexp.true$ ) ||
  scan_infix_pair aexp aexp "<=" with uncurry bexp.le ||
  scan_infix_pair aexp aexp "<" with uncurry bexp.lt ||
  scan_infix_pair aexp aexp "==" with uncurry bexp.eq ||
  scan_infix_pair aexp aexp ">" with uncurry bexp.gt ||
  scan_infix_pair aexp aexp ">=" with uncurry bexp.ge ||
  scan_parens' scan_7

```

```

context
fixes elem :: (char, 'bexp) parser
and ImPLY Or And :: 'bexp  $\Rightarrow$  'bexp  $\Rightarrow$  'bexp
and Not :: 'bexp  $\Rightarrow$  'bexp
begin

```

**fun** *scan\_7'* **and** *scan\_6'* **and** *scan\_0'* **where**

```

scan_7' ::=
  scan_infix_pair scan_6' scan_7' ">" with uncurry Imply ||
  scan_infix_pair scan_6' scan_7' "||" with uncurry Or ||
  scan_6' |
scan_6' ::=
  scan_infix_pair scan_0' scan_6' "&&" with uncurry And ||
  scan_0' |
scan_0' ::=
  (exactly "~" || exactly "!=") *-- scan_parens' scan_7' with Not ||
  elem ||
  scan_parens' scan_7'

context
  assumes [parser_rules]: is_cparser elem
begin

lemma [parser_rules]:
  is_cparser scan_0'
  by (simp add: scan_0'.simps[abs_def])

lemma [parser_rules]:
  is_cparser scan_6'
  by (subst scan_6'.simps[abs_def]) simp

end
end

abbreviation scan_bexp ≡ scan_7' scan_bexp_elem sexp.implies sexp.or sexp.and sexp.not

lemma [parser_rules]:
  is_cparser scan_bexp
  by (subst scan_7'.simps[abs_def]) simp

lemma parse scan_bexp (STR "a < 3 && b >= 2 || ~ (c <= 4)")
= Result (sexp.or (and (lt STR "a" 3) (ge STR "b" 2)) (not (sexp.le STR "c" 4)))
  by eval

definition [consuming]: scan_prefix p head = exactly head *-- p for p

definition [consuming]: scan_formula ≡
  scan_prefix scan_bexp "E<>" with formula.EX ||
  scan_prefix scan_bexp "E[]" with EG ||
  scan_prefix scan_bexp "A<>" with AX ||
  scan_prefix scan_bexp "A[]" with AG ||
  scan_infix_pair scan_bexp scan_bexp "-->" with uncurry Leadsto

lemma is_cparser_token[parser_rules]:
  is_cparser (token a) if is_cparser a
  using that unfolding gen_token_def by simp

definition [consuming]: scan_action ≡
  (scan_var --* token (exactly "?")) with In o String.implode ||
  (scan_var --* token (exactly "!")) with Out o String.implode ||

```

*scan\_var* with *Sil* o *String.implode*

**abbreviation** *orelse* (**infix** *orelse* 58) **where**

*a orelse b*  $\equiv$  *default b a*

**definition**

*parse\_action s*  $\equiv$  *parse scan\_action s orelse Sil (STR "'')*

**fun** *chop\_sexp* **where**

*chop\_sexp* *clocks* (*and a b*) (*cs, es*) =  
*chop\_sexp* *clocks a* (*cs, es*) |> *chop\_sexp* *clocks b* |  
*chop\_sexp* *clocks* (*eq a b*) (*cs, es*) =  
(*if a*  $\in$  *set clocks* then (*eq a b*  $\#$  *cs, es*) else (*cs, eq a b*  $\#$  *es*)) |  
*chop\_sexp* *clocks* (*le a b*) (*cs, es*) =  
(*if a*  $\in$  *set clocks* then (*le a b*  $\#$  *cs, es*) else (*cs, le a b*  $\#$  *es*)) |  
*chop\_sexp* *clocks* (*lt a b*) (*cs, es*) =  
(*if a*  $\in$  *set clocks* then (*lt a b*  $\#$  *cs, es*) else (*cs, lt a b*  $\#$  *es*)) |  
*chop\_sexp* *clocks* (*ge a b*) (*cs, es*) =  
(*if a*  $\in$  *set clocks* then (*ge a b*  $\#$  *cs, es*) else (*cs, ge a b*  $\#$  *es*)) |  
*chop\_sexp* *clocks* (*gt a b*) (*cs, es*) =  
(*if a*  $\in$  *set clocks* then (*gt a b*  $\#$  *cs, es*) else (*cs, gt a b*  $\#$  *es*)) |  
*chop\_sexp* *clocks a* (*cs, es*) = (*cs, a*  $\#$  *es*)

**fun** *sexp\_to\_acconstraint* :: (*String.literal, String.literal, String.literal, int*) *sexp*  $\Rightarrow$  *\_* **where**

*sexp\_to\_acconstraint* (*lt a* (*b* :: *int*)) = *acconstraint.LT a b* |  
*sexp\_to\_acconstraint* (*le a b*) = *acconstraint.LE a b* |  
*sexp\_to\_acconstraint* (*eq a b*) = *acconstraint.EQ a b* |  
*sexp\_to\_acconstraint* (*ge a b*) = *acconstraint.GE a b* |  
*sexp\_to\_acconstraint* (*gt a b*) = *acconstraint.GT a b*

**no\_notation** *top\_assn* (*true*)

**fun** *sexp\_to\_bexp* :: (*String.literal, String.literal, String.literal, int*) *sexp*  $\Rightarrow$  *\_* **where**

*sexp\_to\_bexp* (*lt a* (*b* :: *int*)) = *bexp.lt* (*exp.var a*) (*exp.const b*) |> *Result* |  
*sexp\_to\_bexp* (*le a b*) = *bexp.le* (*exp.var a*) (*exp.const b*) |> *Result* |  
*sexp\_to\_bexp* (*eq a b*) = *bexp.eq* (*exp.var a*) (*exp.const b*) |> *Result* |  
*sexp\_to\_bexp* (*ge a b*) = *bexp.ge* (*exp.var a*) (*exp.const b*) |> *Result* |  
*sexp\_to\_bexp* (*gt a b*) = *bexp.gt* (*exp.var a*) (*exp.const b*) |> *Result* |  
*sexp\_to\_bexp* (*and a b*) =  
do {*a*  $\leftarrow$  *sexp\_to\_bexp a*; *b*  $\leftarrow$  *sexp\_to\_bexp b*; *bexp.and a b* |> *Result*} |  
*sexp\_to\_bexp* (*sexp.or a b*) =  
do {*a*  $\leftarrow$  *sexp\_to\_bexp a*; *b*  $\leftarrow$  *sexp\_to\_bexp b*; *bexp.or a b* |> *Result*} |  
*sexp\_to\_bexp* (*imply a b*) =  
do {*a*  $\leftarrow$  *sexp\_to\_bexp a*; *b*  $\leftarrow$  *sexp\_to\_bexp b*; *bexp.imply a b* |> *Result*} |  
*sexp\_to\_bexp x* = *Error [STR "Illegal construct in binary operation"]*

**definition** [*consuming*]:

*scan\_update*  $\equiv$   
*scan\_var* --- (*exactly* *"=*" || *exactly* *"':=*"') \*\*--- *scan\_exp*  
*with* ( $\lambda(s, x).$  (*String.implode s, x*))

**abbreviation** *scan\_updates*  $\equiv$  *parse\_list scan\_update*

**value** *parse scan\_updates* (*STR* "y2 := 0")

**value** *parse scan\_updates* (*STR* "y2 := 0, x2 := 0")

**value** *parse scan\_exp* (*STR* "( 1 ? L == 0 : 0 )")

**definition** *compile\_invariant* **where**

*compile\_invariant* *clocks vars inv*  $\equiv$

let

(*cs*, *es*) = *chop\_sexp* *clocks inv* ([], []);

*g* = *map\_sexp\_to\_acconstraint cs*

in

if *es* = []

then *Result* (*g*, *bexp.true*)

else do {

let *e* = *fold* (*and*) (*tl es*) (*hd es*);

*b*  $\leftarrow$  *sexp\_to\_bexp* *e*;

assert (*set\_bexp b*  $\subseteq$  *set vars*) (*String.implode* ("Unknown variable in bexp: " @ *show b*));

*Result* (*g*, *b*)

} **for** *inv*

**definition** *compile\_invariant'* **where**

*compile\_invariant'* *clocks vars inv*  $\equiv$

if *inv* = *STR* "" then

*Result* ([], *bexp.true*)

else do {

*inv*  $\leftarrow$  *parse\_scan\_bexp inv* |> *err\_msg* (*STR* "Failed to parse guard in " + *inv*);

*compile\_invariant* *clocks vars inv*

}

**for** *inv*

**definition** *convert\_node* **where**

*convert\_node* *clocks vars n*  $\equiv$  do {

*n*  $\leftarrow$  *of\_object n*;

*ID*  $\leftarrow$  *get n "id"*  $\gg$  *of\_nat*;

*name*  $\leftarrow$  *get n "name"*  $\gg$  *of\_string*;

*inv*  $\leftarrow$  *get n "invariant"*  $\gg$  *of\_string*;

(*inv*, *inv\_vars*)  $\leftarrow$

*compile\_invariant'* *clocks vars inv* |> *err\_msg* (*STR* "Failed to parse invariant!");

assert (case *inv\_vars* of *bexp.true*  $\Rightarrow$  *True* | \_  $\Rightarrow$  *False*)

(*STR* "State invariants on nodes are not supported");

*Result* ((*name*, *ID*), *inv*)

}

**definition** *convert\_edge* **where**

*convert\_edge* *clocks vars e*  $\equiv$  do {

*e*  $\leftarrow$  *of\_object e*;

*source*  $\leftarrow$  *get e "source"*  $\gg$  *of\_nat*;

*target*  $\leftarrow$  *get e "target"*  $\gg$  *of\_nat*;

```

guard ← get e "guard" ≫ of_string;
label ← get e "label" ≫ of_string;
update ← get e "update" ≫ of_string;
label ← if label = STR "" then STR "" |> Sil |> Result else
  parse_scan_action label |> err_msg (STR "Failed to parse label in " + label);
(g, check) ← compile_invariant' clocks vars guard |> err_msg (STR "Failed to parse guard!");
upd ← if update = STR "" then Result [] else
  parse_scan_updates update |> err_msg (STR "Failed to parse update in " + update);
let resets = filter (λx. fst x ∈ set clocks) upd;
assert
  (list_all (λ(⟦, d). case d of exp.const x ⇒ x = 0 | ⟦ ⇒ undefined) resets)
  (STR "Clock resets to values different from zero are not supported");
let resets = map fst resets;
let upds = filter (λx. fst x ∉ set clocks) upd;
assert
  (list_all (λ(x, ⟦). x ∈ set vars) upds)
  (STR "Unknown variable in update: " + update);
Result (source, check, g, label, upds, resets, target)
}

```

**definition** *convert\_automaton where*

```

convert_automaton clocks vars a ≡ do {
  nodes ← get a "nodes" ≫ of_array;
  edges ← get a "edges" ≫ of_array;
  nodes ← combine_map (convert_node clocks vars) nodes;
  let invs = map (λ (⟦, n), g). (n, g)) nodes;
  let names_to_ids = map fst nodes;
  assert (map fst names_to_ids |> filter (λs. s ≠ STR "") |> distinct)
    (STR "Node names are ambiguous" + (show (map fst names_to_ids) |> String.implode));
  assert (map snd names_to_ids |> distinct) (STR "Duplicate node id");
  let ids_to_names = map_of (map prod.swap names_to_ids);
  let names_to_ids = map_of names_to_ids;
  let committed = default [] (get a "committed" ≫ of_array);
  committed ← combine_map of_nat committed;
  let urgent = default [] (get a "urgent" ≫ of_array);
  urgent ← combine_map of_nat urgent;
  edges ← combine_map (convert_edge clocks vars) edges;
  Result (names_to_ids, ids_to_names, (committed, urgent, edges, invs))
}

```

**fun** *rename\_locs\_sexp where*

```

rename_locs_sexp f (not a) =
  do {a ← rename_locs_sexp f a; not a |> Result} |
rename_locs_sexp f (imply a b) =
  do {a ← rename_locs_sexp f a; b ← rename_locs_sexp f b; imply a b |> Result} |
rename_locs_sexp f (sexp.or a b) =
  do {a ← rename_locs_sexp f a; b ← rename_locs_sexp f b; sexp.or a b |> Result} |
rename_locs_sexp f (and a b) =
  do {a ← rename_locs_sexp f a; b ← rename_locs_sexp f b; and a b |> Result} |
rename_locs_sexp f (loc n x) = do {x ← f n x; loc n x |> Result} |
rename_locs_sexp f (eq a b) = Result (eq a b) |
rename_locs_sexp f (lt a b) = Result (lt a b) |
rename_locs_sexp f (le a b) = Result (le a b) |
rename_locs_sexp f (ge a b) = Result (ge a b) |

```

$\text{rename\_locs\_sexp } f \text{ (gt a b)} = \text{Result (gt a b)}$

**fun**  $\text{rename\_locs\_formula where}$

$\text{rename\_locs\_formula } f \text{ (formula.EX } \varphi) = \text{rename\_locs\_sexp } f \varphi \gg \text{Result o formula.EX |}$   
 $\text{rename\_locs\_formula } f \text{ (EG } \varphi) = \text{rename\_locs\_sexp } f \varphi \gg \text{Result o EG |}$   
 $\text{rename\_locs\_formula } f \text{ (AX } \varphi) = \text{rename\_locs\_sexp } f \varphi \gg \text{Result o AX |}$   
 $\text{rename\_locs\_formula } f \text{ (AG } \varphi) = \text{rename\_locs\_sexp } f \varphi \gg \text{Result o AG |}$   
 $\text{rename\_locs\_formula } f \text{ (Leadsto } \varphi \psi) =$   
 $\text{do } \{\varphi \leftarrow \text{rename\_locs\_sexp } f \varphi; \psi \leftarrow \text{rename\_locs\_sexp } f \psi; \text{Leadsto } \varphi \psi \mid > \text{Result}\}$

**definition**  $\text{convert} :: \text{JSON} \Rightarrow$

$(\text{nat} \Rightarrow \text{nat} \Rightarrow \text{String.literal}) \times (\text{String.literal} \Rightarrow \text{nat}) \times \text{String.literal list} \times$   
 $(\text{nat list} \times \text{nat list} \times$   
 $(\text{String.literal act}, \text{nat}, \text{String.literal}, \text{int}, \text{String.literal}, \text{int}) \text{ transition list}$   
 $\times (\text{nat} \times (\text{String.literal}, \text{int}) \text{ cconstraint}) \text{ list}) \text{ list} \times$   
 $(\text{String.literal} \times \text{int} \times \text{int}) \text{ list} \times$   
 $(\text{nat}, \text{nat}, \text{String.literal}, \text{int}) \text{ formula} \times \text{nat list} \times (\text{String.literal} \times \text{int}) \text{ list}$   
 $) \text{ Error\_List\_Monad.result where}$   
 $\text{convert json} \equiv \text{do } \{$   
 $\text{all} \leftarrow \text{of\_object json};$   
 $\text{automata} \leftarrow \text{get all "automata"};$   
 $\text{automata} \leftarrow \text{of\_array automata};$   
 $\text{let broadcast} = \text{default } [] \text{ (do } \{x \leftarrow \text{get all "broadcast"}; \text{of\_array } x\});$   
 $\text{broadcast} \leftarrow \text{combine\_map of\_string broadcast};$   
 $\text{let } \_ = \text{trace\_level } 3$   
 $(\lambda \_. \text{return (STR "Broadcast channels " + String.implode (show broadcast))});$   
 $\text{let bounds} = \text{default (STR "'')} \text{ (do } \{$   
 $\text{ } x \leftarrow \text{get all "vars"}; \text{of\_string } x\}$   
 $\});$   
 $\text{bounds} \leftarrow \text{parse parse\_bounds bounds} \mid > \text{err\_msg (STR "Failed to parse bounds")};$   
 $\text{clocks} \leftarrow \text{get all "clocks"};$   
 $\text{clocks} \leftarrow \text{of\_string clocks};$   
 $\text{clocks} \leftarrow \text{parse (parse\_list (lx\_ws *-- ta\_var\_ident with String.implode)) clocks}$   
 $\mid > \text{err\_msg (STR "Failed to parse clocks")};$   
 $\text{formula} \leftarrow \text{get all "formula"};$   
 $\text{formula} \leftarrow \text{of\_string formula};$   
 $\text{formula} \leftarrow \text{parse scan\_formula formula} \mid > \text{err\_msg (STR "Failed to parse formula")};$   
 $\text{automata} \leftarrow \text{combine\_map of\_object automata};$   
 $\text{process\_names} \leftarrow \text{combine\_map } (\lambda a. \text{get a "name"} \gg \text{of\_string}) \text{ automata};$   
 $\text{assert (distinct process\_names) (STR "Process names are ambiguous")};$   
 $\text{assert (locs\_of\_formula formula} \subseteq \text{set process\_names) (STR "Unknown process name in}$   
 $\text{formula")};$   
 $\text{let process\_names\_to\_index} = \text{List\_Index.index process\_names};$   
 $\text{init\_locs} \leftarrow \text{combine\_map}$   
 $(\lambda a. \text{do } \{x \leftarrow \text{get a "initial"}; x \leftarrow \text{of\_nat } x; x \mid > \text{Result}\})$   
 $\text{automata};$   
 $\text{let formula} = \text{formula.map\_formula process\_names\_to\_index id id id formula};$   
 $\text{let vars} = \text{map fst bounds};$   
 $\text{let init\_vars} = \text{map } (\lambda x. (x, 0::\text{int})) \text{ vars};$   
 $\text{names\_automata} \leftarrow \text{combine\_map (convert\_automaton clocks vars) automata};$   
 $\text{let automata} = \text{map (snd o snd) names\_automata};$   
 $\text{let names} = \text{map fst names\_automata};$   
 $\text{let ids\_to\_names} = \text{map (fst o snd) names\_automata};$   
 $\text{let ids\_to\_names} =$

```

    (λp i. case (ids_to_names ! p) i of Some n ⇒ n | None ⇒ String.implode (show i));
formula ← rename_locs_formula (λi. get (names ! i)) formula;
Result
(ids_to_names, process_names_to_index,
 broadcast, automata, bounds, formula, init_locs, init_vars)
} for json

```

#### Unsafe Glue Code for Printing `code_printing`

```

constant print → (SML) writeln _
and          (OCaml) print'_string _
code_printing
constant println → (SML) writeln _
and          (OCaml) print'_string _

```

#### definition `parse_convert_run_print` where

```

parse_convert_run_print dc s ≡
case parse json s ≫≡ convert of
  Error es ⇒ do {let _ = map println es; return ()}
| Result (ids_to_names, _, broadcast, automata, bounds, formula, L0, s0) ⇒ do {
  r ← do_preproc_mc dc ids_to_names (broadcast, automata, bounds) L0 s0 formula;
  case r of
    Error es ⇒ do {let _ = map println es; return ()}
  | Result s ⇒ do {let _ = println s; return ()}
}

```

#### definition `parse_convert_run` where

```

parse_convert_run dc s ≡
case
  parse json s ≫≡ (λr.
    let
      s' = show r |> String.implode;
      _ = trace_level 2 (λ_. return s')
    in parse json s' ≫≡ (λr'.
      assert (r = r') STR "Parse-print-parse loop failed!" ≫≡ (λ_. convert r)))
of
  Error es ⇒ return (Error es)
| Result (ids_to_names, _, broadcast, automata, bounds, formula, L0, s0) ⇒
  do_preproc_mc dc ids_to_names (broadcast, automata, bounds) L0 s0 formula

```

#### definition `convert_run` where

```

convert_run dc json_data ≡
case (
  let
    s' = show json_data |> String.implode;
    _ = trace_level 2 (λ_. return s')
  in parse json s' ≫≡ (λr'.
    assert (json_data = r') STR "Parse-print-parse loop failed!" ≫≡ (λ_. convert json_data)))
of
  Error es ⇒ return (Error es)
| Result (ids_to_names, _, broadcast, automata, bounds, formula, L0, s0) ⇒
  do_preproc_mc dc ids_to_names (broadcast, automata, bounds) L0 s0 formula

```

Eliminate Gabow statistics

```
code_printing
  code_module Gabow_Skeleton_Statistics → (SML) ⟨⟩
code_printing
  code_module AStatistics → (SML) ⟨⟩
```

Delete “junk”

```
code_printing code_module Bits_Integer → (SML) ⟨⟩
```

```
code_printing
  constant stat_newnode → (SML) (fn x => ()) _
| constant stat_start   → (SML) (fn x => ()) _
| constant stat_stop    → (SML) (fn x => ()) _
```

```
code_printing
  constant IArray.sub' → (SML) (Vector.sub o (fn (a, b) => (a, IntInf.toInt b)))
```

```
code_printing code_module Logging → (SML)
⟨
  structure Logging : sig
    val set_level : int -> unit
    val trace : int -> (unit -> string) -> unit
    val get_trace : unit -> (int * string) list
  end = struct
    val level = Unsynchronized.ref 0;
    val messages : (int * string) list ref = Unsynchronized.ref [];
    fun set_level i = level := i;
    fun get_trace () = !messages;
    fun trace i f =
      if i > !level
      then ()
      else
        let
          val s = f ();
          val _ = messages := (i, s) :: !messages;
        in () end;
  end
⟩
and (Eval) ⟨⟩
code_reserved (Eval) Logging
code_reserved (SML) Logging
```

```
code_printing constant trace_level → (SML)
  Logging.trace (IntInf.toInt (integer'_of'_int _)) ( _ ())
and (Eval)
  (fn ' _ => fn s => writeln (s ())) _ ( _ ())
```

To disable state tracing:

**SML Code Export** Now we export the actual executable code for MUNTA. First for SML:

```

export_code parse_convert_run Result Error
  in SML module_name Model_Checker
  file_prefix Simple_Model_Checker

```

**OCaml Code Export** This is how to do it for OCaml:

```

export_code
  convert_run Result Error String.explode int_of_integer nat_of_integer
  JSON.Object JSON.Array JSON.String JSON.Int JSON.Nat JSON.Rat JSON.Boolean JSON.Null
  fract.Rat
  in OCaml module_name Model_Checker
  file_prefix Simple_Model_Checker

```

**Running Munta** Adding a bit of wrapper code, so that we can directly run MUNTA from within Isabelle:

```

definition parse_convert_run_test where
  parse_convert_run_test dc s ≡ do {
    x ← parse_convert_run dc s;
    case x of
      Error es ⇒ do {let _ = map println es; return (STR "Fail")}
    | Result r ⇒ return r
  }

```

```

ML <
  fun assert comp exp =
    if comp = exp then () else error (Assertion failed! expected: ^ exp ^ but got: ^ comp)
  fun test dc file =
    let
      val dir = Path.append @{master_dir} @{path ../benchmarks/}
      val file = Path.explode file |> Path.append dir
      val s = File.read file
    in
      @{code parse_convert_run_test} dc s end
    >

```

Now we can run MUNTA on a few examples:

```

ML_val <assert (test false HDDI_02.muntax ()) Property is not satisfied!>
ML_val <assert (test true HDDI_02.muntax ()) Model has no deadlock!>

```

```

ML_val <assert (test false simple.muntax ()) Property is satisfied!>
ML_val <assert (test true simple.muntax ()) Model has no deadlock!>

```

```

ML_val <assert (test false light_switch.muntax ()) Property is satisfied!>
ML_val <assert (test true light_switch.muntax ()) Model has no deadlock!>

```

```

ML_val <assert (test false PM_test.muntax ()) Property is not satisfied!>
ML_val <assert (test true PM_test.muntax ()) Model has a deadlock!>

```

```

ML_val <assert (test false bridge.muntax ()) Property is satisfied!>
ML_val <assert (test true bridge.muntax ()) Model has no deadlock!>

```

```

ML_val <assert (test false fischer.muntax ()) Property is satisfied!>
ML_val <assert (test true fischer.muntax ()) Model has no deadlock!>

```

```
ML_val <assert (test false PM_all_4.muntax ()) Property is not satisfied!>
ML_val <assert (test true PM_all_4.muntax ()) Model has no deadlock!>
```

```
end
```

### 13 Build and Test Exported Program With MLton

```
theory Munta_Compile_MLton
  imports Munta_Model_Checker.Simple_Network_Language_Export_Code
begin

— Produces a command for checking a single benchmark, e.g.: ./check_benchmark.sh 568
"Property is not satisfied" ./munta -m benchmarks/PM_all_5.muntax
ML <
fun mk_benchmark_check_gen munta mk_prop name num_states satisfied =
  let
    val prop = mk_prop satisfied
    val benchmark = name ^ ".muntax"
  in
    — this line checks that number of states and property satisfaction are correct
    ./check_benchmark.sh ^ string_of_int num_states ^ ^ prop
    — this line runs Munta on the actual benchmark
    ^ ^ munta ^ ^ benchmark
  end
val mk_benchmark_check =
  mk_benchmark_check_gen ./munta -m
  (fn satisfied =>
    if satisfied then
      verbatim <Property is satisfied!>
    else
      verbatim <Property is not satisfied!>)
val mk_benchmark_check_dc =
  mk_benchmark_check_gen ./munta -dc -m
  (fn satisfied =>
    if satisfied then
      verbatim <Model has a deadlock!>
    else
      verbatim <Model has no deadlock!>)
val it = mk_benchmark_check PM_all_5 568 false
val it1 = mk_benchmark_check_dc hddi_08 466 false
>
```

Here is how to compile Munta with MLton and then run some benchmarks:

```
compile_generated_files code/Simple_Model_Checker.ML (in Simple_Network_Language_Export_Code)
external_files
  <Unsyncronized.sml>
  <Writeln.sml>
  <Util.sml>
  <Muntax.sml>
  <MLton_Main.sml>
  <library.ML>
  <muntax.mlb>
```

```

    <check_benchmark.sh>
    (in ../ML)
and
    <HDDI_02.muntax>
    <HDDI_02_broadcast.muntax>
    <HDDI_08_broadcast.muntax>
    <PM_all_1.muntax>
    <PM_all_2.muntax>
    <PM_all_3.muntax>
    <PM_all_4.muntax>
    <PM_all_5.muntax>
    <PM_all_6.muntax>
    <PM_all_urgent.muntax>
    <bridge.muntax>
    <csma_05.muntax>
    <csma_06.muntax>
    <fischer.muntax>
    <fischer_05.muntax>
    <hddi_08.muntax>
    <light_switch.muntax> (in ../benchmarks)
export_files <munta> (exe)
where <fn dir =>
    let
        val exec = Generated_Files.execute dir
        val _ =
            exec <Preparation>
            mv code/Simple_Model_Checker.ML Simple_Model_Checker.ML
        val _ =
            exec <Compilation>
            (verbatim $ISABELLE_MLTON $ISABELLE_MLTON_OPTIONS //
            verbatim <$ISABELLE_MLTON> ^
            — these additional settings have been copied from the AFP entry PAC_Checker
            — they bring down benchmarks/PM_all_6.muntax from 1000 s to 180 s on an M1 Mac
            — const 'MLton.safe false' —verbose 1 —inline 700 —cc-opt -O3 ^
            — this one from PAC_Checker does not work on ARM64 though
            //codegen/flat//
            — these used to be the only setting for Munta
            —default-type int64 ^
            —output munta ^
            muntax.mlb)
        val _ = exec <Test HDDI_02> (mk_benchmark_check HDDI_02 34 false)
        val _ = exec <Test HDDI_02_broadcast> (mk_benchmark_check HDDI_02_broadcast 34
false)
        val _ = exec <Test HDDI_08_broadcast> (mk_benchmark_check HDDI_08_broadcast 466
false)
        val _ = exec <Test PM_all_1> (mk_benchmark_check PM_all_4 529 false)
        val _ = exec <Test PM_all_1> (mk_benchmark_check PM_all_1 338 false)
        val _ = exec <Test PM_all_2> (mk_benchmark_check PM_all_2 2828 false)
        val _ = exec <Test PM_all_3> (mk_benchmark_check PM_all_3 3994 false)
        val _ = exec <Test PM_all_4> (mk_benchmark_check PM_all_4 529 false)
        val _ = exec <Test PM_all_5> (mk_benchmark_check PM_all_5 568 false)
        — 180 s on an M1 Mac
        (val _ = exec <Test PM_all_6> (mk_benchmark_check PM_all_6 116243 false)
        val _ = exec <Test PM_all_urgent> (mk_benchmark_check PM_all_urgent 8 true)

```

```

val _ = exec ⟨Test bridge⟩ (mk_benchmark_check bridge 73 true)
val _ = exec ⟨Test csma_05⟩ (mk_benchmark_check csma_05 8217 false)
val _ = exec ⟨Test csma_06⟩ (mk_benchmark_check csma_06 68417 false)
val _ = exec ⟨Test fischer⟩ (mk_benchmark_check fischer 4500 true)
val _ = exec ⟨Test fischer_05⟩ (mk_benchmark_check fischer_05 38579 false)
val _ = exec ⟨Test hddi_08⟩ (mk_benchmark_check hddi_08 466 false)
val _ = exec ⟨Test light_switch⟩ (mk_benchmark_check light_switch 2 true)
— a test for the deadlock checker
val _ = exec ⟨Test hddi_08⟩ (mk_benchmark_check_dc hddi_08 466 false)
in () end

```

end

## References

- [1] S. Wimmer. Timed automata. *Archive of Formal Proofs*, March 2016. [https://isa-afp.org/entries/Timed\\_Automata.html](https://isa-afp.org/entries/Timed_Automata.html), Formal proof development.
- [2] S. Wimmer. Munta: A verified model checker for timed automata. In É. André and M. Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 236–243. Springer, 2019.
- [3] S. Wimmer. *Trustworthy Verification of Realtime Systems*. PhD thesis, Technical University of Munich, Germany, 2020.
- [4] S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.
- [5] S. Wimmer and P. Lammich. Difference bound matrices. *Archive of Formal Proofs*, August 2024. [https://isa-afp.org/entries/Difference\\_Bound\\_Matrices.html](https://isa-afp.org/entries/Difference_Bound_Matrices.html), Formal proof development.
- [6] S. Wimmer and P. Lammich. Worklist algorithms. *Archive of Formal Proofs*, August 2024. [https://isa-afp.org/entries/Worklist\\_Algorithms.html](https://isa-afp.org/entries/Worklist_Algorithms.html), Formal proof development.