

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

1	Networks of Timed Automata	4
1.1	Syntax and Operational Semantics	4
1.2	Product Automaton	5
2	Networks of Timed Automata With Discrete State	10
2.1	Syntax and Operational Semantics	10
2.2	Product Automaton	11
3	Networks of Timed Automata – UPPAAL Style	21
3.1	Syntax and Operational Semantics	21
3.2	Equivalent State Network Automaton	23
4	Implementation of UPPAAL Style Networks	34
4.1	Pre-compiled Networks With States and Clocks as Natural Numbers	36
5	Imperative Implementation of UPPAAL Style Networks	52
5.1	Executable Successor Computation	52
5.2	Check Preconditions	77
6	Simple Networks of Automata With Broadcast Channels and Committed Locations	95
6.1	Product Construction	98
7	Product Bisimulation	144
8	The Final Semantics	146
9	Instantiating the Model Checking Locale	149
9.1	Extracting an Efficient Implementation	156
10	Deadlock Checking	160
10.1	Notes	160
10.2	Abstract Reachability Checking	160
10.3	Operations	163
10.4	Structural Properties	172
10.5	Correctness of <i>dbm_subset_fed</i>	175
10.6	Refined DBM Operations	177
10.7	Transferring Properties	178

10.8 Functional Refinement	183
10.9 Imperative Refinement	186
10.10 Implementation for a Concrete Automaton	189
10.11 Checking a Property on the Reachable Set	190
10.12 Complete Deadlock Checker	191
11 Renaming Identifiers in Simple Networks	200
12 Assembling the Model Checker and Generating Code	235
13 Build and Test Exported Program With MLton	261

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 261) 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 261). 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 261). 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:
```

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

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

```
step_n_i:
```

```
 $\llbracket$ 
```

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

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

```
 $\rrbracket$ 
```

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

```
step_n_s:
```

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

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

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

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

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

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

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

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

lemma *stepI2*:

$N \vdash_N \langle l, Z \rangle \rightarrow^* \langle l'', Z'' \rangle$ **if** $N \vdash_N \langle l', Z' \rangle \rightarrow^* \langle l'', Z'' \rangle$ $N \vdash_N \langle l, Z \rangle \rightarrow_b \langle l', Z' \rangle$
 $\langle proof \rangle$

lemma *step_n_t_I*:

$\llbracket \forall p \in \{..<length\ N\}. u \vdash inv_of\ (N!p)\ (L!p); \forall p \in \{..<length\ N\}. u \oplus d \vdash inv_of\ (N!p)\ (L!p);$
 $d \geq 0$
 $\rrbracket \implies N \vdash_N \langle L, u \rangle \rightarrow_{Del} \langle L, u \oplus d \rangle$
 $\langle proof \rangle$

1.2 Product Automaton

abbreviation *state_set* :: ('a, 'c, 'time, 's) transition set $\Rightarrow$'s set **where**
 $state_set\ T \equiv fst\ 'T \cup (snd\ o\ snd\ o\ snd\ o\ snd)\ 'T$

lemma *guard_concat*:

assumes $\forall g \in set\ xs. u \vdash g$
shows $u \vdash concat\ xs$
 $\langle proof \rangle$

lemma *guard_append*:

assumes $u \vdash g1\ u \vdash g2$
shows $u \vdash g1\ @\ g2$
 $\langle proof \rangle$

lemma *concat_guard*:

assumes $u \vdash concat\ xs\ g \in set\ xs$
shows $u \vdash g$
 $\langle proof \rangle$

lemma *guard_consE*:

assumes $u \vdash ac\ \# \ cc$
obtains $u \vdash_a\ ac\ u \vdash cc$
 $\langle proof \rangle$

lemma *guard_consI*:

assumes $u \vdash_a\ ac\ u \vdash cc$
shows $u \vdash ac\ \# \ cc$
 $\langle proof \rangle$

lemma *collect_clock_pairs_append_cases*:

assumes $x \in collect_clock_pairs\ (g1\ @\ g2)$
shows $x \in collect_clock_pairs\ g1 \vee x \in collect_clock_pairs\ g2$
 $\langle proof \rangle$

lemma *list_update_nth_split*:

assumes $j < length\ xs$
shows $P\ (xs[i := x] ! j) = ((i = j \longrightarrow P\ x) \wedge (i \neq j \longrightarrow P\ (xs ! j)))$
 $\langle proof \rangle$

locale *Product_TA_Defs* =

fixes $N :: ('a, 'b, 'c, 't, 's)\ nta$

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

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

definition

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

definition

$\text{product_trans} \equiv \text{product_trans_i} \cup \text{product_trans_s}$

lemma $\text{product_state_set_subs}$:

assumes $q < \text{length } N \text{ } l \in \text{state_set } \text{product_trans}$

shows $l ! q \in \text{state_set } (T ! q)$

$\langle \text{proof} \rangle$

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 $\text{collect_clki_product_invariant}$:

$\text{Timed_Automata.collect_clki } \text{product_invariant} = \bigcup (\text{Timed_Automata.collect_clki } ' \text{set } I)$

$\langle \text{proof} \rangle$

lemma states_length :

assumes $L \in \text{states}$

shows $\text{length } L = \text{length } N$

$\langle \text{proof} \rangle$

lemma $\text{collect_clkt_product_trans_subs}$:

$\text{Timed_Automata.collect_clkt } \text{product_trans} \subseteq \bigcup (\text{Timed_Automata.collect_clkt } ' \text{set } T)$

$\langle \text{proof} \rangle$

lemma $\text{collect_clkvt_product_trans_subs}$:

$\text{collect_clkvt } \text{product_trans} \subseteq \bigcup (\text{collect_clkvt } ' \text{set } T)$

$\langle \text{proof} \rangle$

lemma statesI_aux :

fixes L

assumes $L: L \in \text{states}$

assumes

$p < \text{length } T$
 $(l, g, a, r, l') \in T ! p$
shows $(L[p := l]) \in \text{states}$
 $\langle \text{proof} \rangle$

lemma *product_trans_s_alt_def*:

$\text{product_trans_s} =$
 $\{(L, g, (a, \text{Syn } b1 \ b2), r, L') \mid L \ g \ a \ b1 \ b2 \ r \ L'. \exists \ p \ q \ g1 \ g2 \ r1 \ r2 \ l1' \ l2'.$
 $L \in \text{states} \wedge p < \text{length } L \wedge q < \text{length } L \wedge p \neq q \wedge$
 $g = g1 \ @ \ g2 \wedge r = r1 \ @ \ r2 \wedge L' = L[p:=l1', q:=l2']$
 $\wedge (L!p, g1, (\text{In } a, b1), r1, l1') \in T!p \wedge (L!q, g2, (\text{Out } a, b2), r2, l2') \in T!q$
 $\}$
 $\langle \text{proof} \rangle$

context

assumes *states_not_empty*: $\text{states} \neq \{\}$

assumes *trans_complete*:

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

True

begin

lemma *collect_clkt_product_trans_sups*:

$\text{Timed_Automata.collect_clkt product_trans} \supseteq \bigcup (\text{Timed_Automata.collect_clkt 'set } T)$

$\langle \text{proof} \rangle$

lemma *collect_clkvt_product_trans_sups*:

$\text{collect_clkvt product_trans} \supseteq \bigcup (\text{collect_clkvt 'set } T)$

$\langle \text{proof} \rangle$

lemma *collect_clkt_product_trans*:

$\text{Timed_Automata.collect_clkt product_trans} = \bigcup (\text{Timed_Automata.collect_clkt 'set } T)$

$\langle \text{proof} \rangle$

lemma *collect_clkvt_product_trans*:

$\text{collect_clkvt product_trans} = \bigcup (\text{collect_clkvt 'set } T)$

$\langle \text{proof} \rangle$

end

context

assumes *finite_trans*: $\forall \ A \in \text{set } N. \text{finite } (\text{trans_of } A)$

begin

lemma *finite_states*:

finite states

$\langle \text{proof} \rangle$

lemma *finite_product_trans_i*:

finite product_trans_i

$\langle \text{proof} \rangle$

lemma *finite_Collect_bounded_ex_5'* [simp]:
assumes *finite* $\{(a,b,c,d,e) . P\ a\ b\ c\ d\ e\}$
shows
 $finite\ \{x. \exists\ a\ b\ c\ d\ e. P\ a\ b\ c\ d\ e \wedge Q\ x\ a\ b\ c\ d\ e\}$
 $\longleftrightarrow (\forall\ a\ b\ c\ d\ e. P\ a\ b\ c\ d\ e \longrightarrow finite\ \{x. Q\ x\ a\ b\ c\ d\ e\})$
 $\langle proof \rangle$

lemma *finite_product_trans_s*:
 $finite\ product_trans_s$
 $\langle proof \rangle$

lemma *finite_trans_of_product*:
shows *finite* (*trans_of_product_ta*)
 $\langle proof \rangle$

end

lemma *inv_of_product*[simp]:
 $inv_of\ product_ta = product_invariant$
 $\langle proof \rangle$

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

lemma *finite_invariant_of_product*[folded *inv_of_product*]:
assumes $\forall\ A \in set\ N. finite\ (range\ (inv_of\ A))$
shows *finite* (*range product_invariant*)
 $\langle proof \rangle$

end

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

lemma *product_invariantD*:
assumes $u \vdash product_invariant\ L\ p < length\ N\ length\ L = length\ N$
shows $u \vdash inv_of\ (N\ !\ p)\ (L!p)$
 $\langle proof \rangle$

lemma *states_step*:
 $L' \in states\ \text{if } N \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle\ L \in states$
 $\langle proof \rangle$

lemma *states_steps*:
 $L' \in states\ \text{if } N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle\ L \in states$
 $\langle proof \rangle$

end

lemma *steps_altI*:

$A \vdash \langle l, Z \rangle \rightarrow^* \langle l'', Z'' \rangle$ **if**
 $A \vdash \langle l, Z \rangle \rightarrow^* \langle l', Z' \rangle \quad A \vdash \langle l', Z' \rangle \rightarrow \langle l'', Z'' \rangle$
 $\langle \text{proof} \rangle$

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

lemma
 $\text{len}[\text{simp}]: \text{length } L = \text{length } N$
 $\langle \text{proof} \rangle$

lemma *product_delay_complete*:
assumes $\text{step}: N \vdash_N \langle L, u \rangle \rightarrow_{\text{Del}} \langle L', u' \rangle$
obtains d **where** $\text{product_ta} \vdash \langle L, u \rangle \rightarrow^d \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_int_complete*:
assumes $\text{step}: N \vdash_N \langle L, u \rangle \rightarrow_{\text{Act } b} \langle L', u' \rangle$
obtains a **where** $\text{product_ta} \vdash \langle L, u \rangle \rightarrow_{(a, \text{Act } b)} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_sync_complete*:
assumes $\text{step}: N \vdash_N \langle L, u \rangle \rightarrow_{\text{Syn } b1 \ b2} \langle L', u' \rangle$
obtains a **where** $\text{product_ta} \vdash \langle L, u \rangle \rightarrow_{(a, \text{Syn } b1 \ b2)} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_complete*:
assumes $\text{step}: N \vdash_N \langle L, u \rangle \rightarrow_b \langle L', u' \rangle$
shows $\text{product_ta} \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_ta_cases*:
assumes $\text{product_ta} \vdash L \longrightarrow^{g,a,r} L'$
shows $(L, g, a, r, L') \in \text{product_trans_i} \vee (L, g, a, r, L') \in \text{product_trans_s}$
 $\langle \text{proof} \rangle$

lemma *product_delay_sound*:
assumes $\text{step}: \text{product_ta} \vdash \langle L, u \rangle \rightarrow^d \langle L', u' \rangle$
shows $N \vdash_N \langle L, u \rangle \rightarrow_{\text{Del}} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_action_sound*:
assumes $\text{step}: \text{product_ta} \vdash \langle L, u \rangle \rightarrow_{(a, b)} \langle L', u' \rangle$
shows $N \vdash_N \langle L, u \rangle \rightarrow_b \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *product_sound*:
assumes $\text{step}: \text{product_ta} \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$
obtains a **where** $N \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$

```

    <proof>

lemma states_product_step[intro]:
   $L' \in \text{states}$  if  $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$ 
  <proof>

lemma product_steps_sound:
   $N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$  if  $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$ 
  <proof>

lemma product_steps_complete:
   $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$  if  $N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$ 
  <proof>

lemma product_correct:
   $\text{product\_ta} \vdash \langle L, u \rangle \rightarrow^* \langle L', u' \rangle \longleftrightarrow N \vdash_N \langle L, u \rangle \rightarrow^* \langle L', u' \rangle$ 
  <proof>

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*
 ($_ \vdash \langle _, _, _ \rangle \rightarrow _ \langle _, _, _ \rangle$ [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 $\forall p \in \{..<length\ N\}. u \oplus d \vdash snd\ (N!p)\ (L!p)$
 $d \geq 0\ length\ N = length\ I \mid$

step_sn_i:
 (N, I) $\vdash \langle L, s, u \rangle \rightarrow_{Act\ a} \langle L', s', u' \rangle$
if ($l, g, (Sil\ a, c, m), f, l'$) $\in fst\ (N!p)$
 $u \vdash g\ s\ \forall p \in \{..<length\ N\}. u' \vdash snd\ (N!p)\ (L!p)$
 $r = f\ s$
 $L!p = l\ p < length\ L\ L' = L[p := l']\ u' = [r \rightarrow 0]u$
 $c\ s\ \forall p < length\ I. (I!p)\ (L'!p)\ s'\ Some\ s' = m\ s$
 $length\ N = length\ I \mid$

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

where

refl: $N \vdash \langle L, s, u \rangle \rightarrow^* \langle L, s, u \rangle \mid$
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**
 $N \vdash \langle l', s', u' \rangle \rightarrow^* \langle l'', s'', u'' \rangle\ N \vdash \langle l, s, u \rangle \rightarrow_a \langle l', s', u' \rangle$
 <proof>

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

$$N_s\ s = map\ (\lambda\ p.\ (T_s\ p\ s, snd\ (fst\ A\ !\ p)))\ [0..<length\ (fst\ A)]$$

abbreviation $P \equiv snd\ A$

definition $p \equiv length\ (fst\ A)$

abbreviation $product\ s \equiv Product_TA_Defs.product_ta\ (N_s\ s)$

abbreviation $T'\ s \equiv trans_of\ (product\ s)$

abbreviation $I'\ s \equiv inv_of\ (product\ s)$

definition

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

definition

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

definition

$$prod_trans \equiv prod_trans_i \cup prod_trans_s$$

definition

$$prod_invariant \equiv \lambda\ (L, s).\ I'\ s\ L$$

definition $prod_ta :: ('a, 'c, 't, 's\ list \times 'st)\ ta\ \mathbf{where}$

$$prod_ta \equiv (prod_trans, prod_invariant)$$

lemma $prod_ta_cases:$

assumes $prod_ta \vdash L \longrightarrow^{g,a,r} L'$

shows $(L, g, a, r, L') \in prod_trans_i \vee (L, g, a, r, L') \in prod_trans_s$
 $\langle proof \rangle$

lemma $inv_of_simp:$

$$inv_of\ prod_ta\ (L, s) = I'\ s\ L$$

$\langle proof \rangle$

lemma $I'_simp:$

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

$\langle proof \rangle$

lemma *collect_clki_prod_invariant*:

Timed_Automata.collect_clki_prod_invariant = Timed_Automata.collect_clki (I' s)
 $\langle \text{proof} \rangle$

lemma *collect_clki_prod_invariant'*:

Timed_Automata.collect_clki_prod_invariant
 $\subseteq \bigcup \{ \text{Timed_Automata.collect_clki (snd (fst A ! p))} \mid p. p < \text{length (fst A)} \}$
 $\langle \text{proof} \rangle$

lemma *collect_clkt_prod_trans_subs*:

Timed_Automata.collect_clkt_prod_trans $\subseteq$ *Timed_Automata.collect_clkt* ($\bigcup (T' \text{ ' } UNIV)$)
 $\langle \text{proof} \rangle$

lemma *collect_clkvt_prod_trans_subs*:

collect_clkvt_prod_trans $\subseteq$ *collect_clkvt* ($\bigcup (T' \text{ ' } UNIV)$)
 $\langle \text{proof} \rangle$

lemma *T_simp*:

T ! q = trans_of (N ! q) **if** *q < length N*
 $\langle \text{proof} \rangle$

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)**
 $\langle \text{proof} \rangle$

lemma *finite_trans'*:

$\forall A \in \text{set } (N_s s). \text{finite (trans_of A)}$
 $\langle \text{proof} \rangle$

lemma *finite_states*:

finite (Product_TA_Defs.states (N_s s))
 $\langle \text{proof} \rangle$

lemma

finite (T' s)
 $\langle \text{proof} \rangle$

lemma *finite_product_1*:

finite (T' s)

$\langle \text{proof} \rangle$

lemma *prod_trans_i_alt_def*:

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

lemma *Some_finite*:

finite $\{x. \text{Some } x = y\}$
 $\langle \text{proof} \rangle$

lemma *finite_prod_trans*:

finite prod_trans **if** $p > 0$
 $\langle \text{proof} \rangle$

end

abbreviation $\text{states}' \text{ s} \equiv \text{Product_TA_Defs.states } (N \text{ s } s)$

lemma *N_s_length*:

length $(N \text{ s } s) = p$
 $\langle \text{proof} \rangle$

end

locale *Prod_TA_Defs'* =

Prod_TA_Defs A **for** $A :: ('a, 'c, 't :: \text{time}, 's, 'st) \text{ snta}$

begin

lemma *A_unfold*:

$A \equiv (N, P)$
 $\langle \text{proof} \rangle$

lemma *network_step_delay*:

assumes *step*: $(N, P) \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$ **and** *len*: $\text{length } L = p$
shows $N \text{ s } s \vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.label.Del}} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *network_step_silent*:

assumes *step*: $(N, P) \vdash \langle L, s, u \rangle \rightarrow_{\text{Act } a} \langle L', s', u' \rangle$ **and** *len*: $\text{length } L = p$
obtains *a* **where** $N \text{ s } s \vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.label.Act } a} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *network_step_sync*:

assumes *step*: $(N, P) \vdash \langle L, s, u \rangle \rightarrow_{\text{Syn } a} \langle L', s', u' \rangle$ **and** *len*: $\text{length } L = p$
obtains *a b* **where** $N \text{ s } s \vdash_N \langle L, u \rangle \rightarrow_{\text{Networks.label.Syn } a \text{ b}} \langle L', u' \rangle$
 $\langle \text{proof} \rangle$

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** $N \text{ s } s \vdash_N \langle L, u \rangle \rightarrow_a \langle L', u' \rangle$

$\langle \text{proof} \rangle$

lemma *trans_of_N_s_1*:

$(fst \text{ ' trans_of } (N_s \ s \ ! \ q)) = fst \text{ ' fst } (N \ ! \ q) \text{ if } q < p$

$\langle \text{proof} \rangle$

lemma *trans_of_N_s_2*:

$((snd \ o \ snd \ o \ snd \ o \ snd) \text{ ' trans_of } (N_s \ s \ ! \ q)) = (snd \ o \ snd \ o \ snd \ o \ snd) \text{ ' fst } (N \ ! \ q) \text{ if } q < p$

$\langle \text{proof} \rangle$

lemma

$fst \text{ ' trans_of } (N_s \ s \ ! \ q) = fst \text{ ' trans_of } (N_s \ s' \ ! \ q) \text{ if } q < p$

$\langle \text{proof} \rangle$

lemma *states'_simp*:

$states' \ s = states' \ s'$

$\langle \text{proof} \rangle$

lemma *states_step*:

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

$\langle \text{proof} \rangle$

lemma *states_steps*:

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

$\langle \text{proof} \rangle$

lemma *inv_step*:

$\forall p < \text{length } P. (P \ ! \ p) (L' \ ! \ p) \ s' \text{ if } A \vdash \langle L, \ s, \ u \rangle \rightarrow_a \langle L', \ s', \ u' \rangle \ \forall p < \text{length } P. (P \ ! \ p) (L \ ! \ p) \ s$

$\langle \text{proof} \rangle$

lemma *inv_steps*:

$\forall p < \text{length } P. (P \ ! \ p) (L' \ ! \ p) \ s' \text{ if } A \vdash \langle L, \ s, \ u \rangle \rightarrow^* \langle L', \ s', \ u' \rangle \ \forall p < \text{length } P. (P \ ! \ p) (L \ ! \ p) \ s$

$\langle \text{proof} \rangle$

end

locale *Prod_TA* =

Prod_TA_Defs' A for A :: ('a, 'c, 't :: time, 's, 'st) snta +

fixes *L :: 's list and s :: 'st*

assumes *states[intro]: L ∈ states' s*

assumes *Len: length N = length P*

and *inv: ∀ p < length P. (P ! p) (L ! p) s*

begin

sublocale *Product_TA N_s s L* $\langle \text{proof} \rangle$

lemma *inv_prod_simp*:

$\text{inv_of_prod_ta } (l, \ s') = \text{Product_TA_Defs.product_invariant } (N_s \ s') \ l \text{ if length } l = p$

$\langle \text{proof} \rangle$

lemma *inv_of_N_simp*:

map inv_of (N_s s') ! q = I ! q if q < p
 $\langle \text{proof} \rangle$

lemma *product_inv_prod_simp*:

inv_of prod_ta (l, s') = I' s l if length l = p
 $\langle \text{proof} \rangle$

lemma *product_inv_prod[intro]*:

u $\vdash$ *inv_of prod_ta (l, s')* **if** *u* $\vdash$ *inv_of product_ta l* *length l = p*
 $\langle \text{proof} \rangle$

lemma *A_simp[simp]*:

N' = N P' = P if A = (N', P')
 $\langle \text{proof} \rangle$

lemma *length_L[intro]*:

length L = p
 $\langle \text{proof} \rangle$

lemma *prod_complete_delay*:

assumes *step*: *A* $\vdash$ $\langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$
obtains *d* **where** *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$
 $\langle \text{proof} \rangle$

lemma *prod_complete_silent*:

assumes *step*: *A* $\vdash$ $\langle L, s, u \rangle \rightarrow_{Act\ a} \langle L', s', u' \rangle$
obtains *a* **where** *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$
 $\langle \text{proof} \rangle$

lemma *prod_complete_sync*:

assumes *step*: *A* $\vdash$ $\langle L, s, u \rangle \rightarrow_{Syn\ a} \langle L', s', u' \rangle$
obtains *a* **where** *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$
 $\langle \text{proof} \rangle$

lemma *prod_complete*:

assumes *step*: *A* $\vdash$ $\langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
shows *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$
 $\langle \text{proof} \rangle$

lemma *A_unfold*:

A = (N, P)
 $\langle \text{proof} \rangle$

lemma *prod_sound'_delay*:

assumes *step*: *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$
shows *A* $\vdash$ $\langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle \wedge$ *product_ta* $\vdash$ $\langle L, u \rangle \rightarrow \langle L', u' \rangle$
 $\wedge (\forall p < \text{length } P. (P ! p) (L' ! p) s')$
 $\langle \text{proof} \rangle$

lemma *prod_sound'_action*:

assumes *step*: *prod_ta* $\vdash$ $\langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$
obtains *a* **where** (*A* $\vdash$ $\langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq Del$) $\wedge$ *product_ta* $\vdash$ $\langle L, u \rangle \rightarrow \langle L', u' \rangle$
 $\wedge (\forall p < \text{length } P. (P ! p) (L' ! p) s')$

$\langle proof \rangle$

lemma *prod_sound'*:

assumes *step*: $prod_ta \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$

obtains *a* **where** $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge product_ta \vdash \langle L, u \rangle \rightarrow \langle L', u' \rangle$
 $\wedge (\forall p < length\ P. (P ! p) (L' ! p) s')$

$\langle proof \rangle$

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*]:

$L' \in states$ **if** $prod_ta \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$

$\langle proof \rangle$

lemma *inv_prod_step*[*intro*]:

$\forall p < length\ P. (P ! p) (L' ! p) s'$ **if** $prod_ta \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$

$\langle proof \rangle$

lemma *prod_steps_sound*:

assumes *step*: $prod_ta \vdash \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle$

shows $A \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$

$\langle proof \rangle$

lemma *prod_steps_complete*:

$prod_ta \vdash \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle$ **if** $A \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$

$\langle proof \rangle$

lemma *prod_correct*:

$prod_ta \vdash \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \longleftrightarrow A \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$

$\langle proof \rangle$

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'

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' 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*
 <proof>

lemma *steps_halt*:
assumes *steps prog n* (*pc*, *st*, *m*, *f*, *rs*) *s prog pc* = *Some HALT*
shows *s* = (*pc*, *st*, *m*, *f*, *rs*)
 <proof>

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'*)
 <proof>

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*)
 <proof>

lemma *exec_length_mono*:
assumes *exec prog n s pcs* = *Some* (*s'*, *pcs'*)
shows *length pcs'* > *length pcs*
 <proof>

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 $\text{exec prog } n \ s \ pcs' = \text{Some } ((pc, st, m, f, rs), pc \# pcs \ @ \ pcs')$
 $\langle \text{proof} \rangle$

lemma visited_exec' :

assumes $\text{visited prog } n \ s \ (pc, st, m, f, rs) \ pcs \ \text{prog } pc = \text{Some } \text{HALT}$

shows $\text{exec prog } n \ s \ [] = \text{Some } ((pc, st, m, f, rs), pc \# pcs)$

$\langle \text{proof} \rangle$

end

theory UPPAAL_Asm_Clocks

imports $\text{Timed_Automata.Timed_Automata Timed_Automata.Normalized_Zone_Semantics}$

$\text{UPPAAL_Asm Complex_Main}$

begin

datatype $'t \ \text{instrc} =$

$\text{INSTR instr} \mid$

$\text{CEXP } (nat, 't) \ \text{acconstraint}$

fun $\text{stepc} :: 't \ \text{instrc} \Rightarrow (nat, 't :: \text{time}) \ \text{cval} \Rightarrow \text{state} \Rightarrow \text{state} \ \text{option} \ \text{where}$

$\text{stepc } (\text{INSTR instr}) \ u \ s =$

$(\text{case step instr } s \ \text{of}$

$\text{Some } s' \Rightarrow \text{Some } s'$

$\mid \text{None} \Rightarrow \text{None}) \mid$

$\text{stepc } (\text{CEXP cc}) \ u \ (pc, st, m, f, rs) = \text{Some } (pc + 1, st, m, u \vdash_a cc, rs)$

type_synonym $'t \ \text{programc} = \text{addr} \Rightarrow 't \ \text{instrc} \ \text{option}$

inductive $\text{stepsc} :: 't \ \text{programc} \Rightarrow \text{fuel} \Rightarrow (nat, 't :: \text{time}) \ \text{cval} \Rightarrow \text{state} \Rightarrow \text{state} \Rightarrow \text{bool} \ \text{where}$

$\text{stepsc prog } (\text{Suc } n) \ u \ \text{start} \ \text{start} \mid$

$\text{stepsc prog } (\text{Suc } n) \ u \ (pc, st, m, f, rs) \ s' \ \text{if}$

$\text{stepc cmd } u \ (pc, st, m, f, rs) = \text{Some } s$

$\text{prog } pc = \text{Some } \text{cmd}$

$\text{stepsc prog } n \ u \ s \ s'$

inductive $\text{visitedc} :: 't \ \text{programc} \Rightarrow \text{fuel} \Rightarrow (nat, 't :: \text{time}) \ \text{cval} \Rightarrow \text{state} \Rightarrow \text{state} \Rightarrow \text{addr list} \Rightarrow \text{bool} \ \text{where}$

$\text{visitedc prog } (\text{Suc } n) \ u \ \text{start} \ \text{start} \ [] \mid$

$\text{visitedc prog } (\text{Suc } n) \ u \ (pc, st, m, f, rs) \ s' \ (pcs \ @ \ [pc]) \ \text{if}$

$\text{stepc cmd } u \ (pc, st, m, f, rs) = \text{Some } s$

$\text{prog } pc = \text{Some } \text{cmd}$

$\text{visitedc prog } n \ u \ s \ s' \ pcs$

definition $\text{stepst prog } n \ u \ \text{start} \equiv \lambda \ (pc, st, m, f, rs).$

$\text{stepc prog } n \ u \ \text{start} \ (pc, st, m, f, rs) \wedge \text{prog } pc = \text{Some } (\text{INSTR } \text{HALT})$

fun $\text{strip} :: 't \ \text{instrc} \Rightarrow \text{instr} \ \text{where}$

```

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 ∀ i < r. finite (S i)
  shows finite {s. length s = r ∧ (∀ i < r. s ! i ∈ S i)} (is finite ?R)
⟨proof⟩

```

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) invassn = 's ⇒ ('c, 'time) cconstraint

```

```

type_synonym
  ('a, 's) transition = 's * addr * 'a * addr * 's

```

```

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

```

```

type_synonym
  ('a, 'time, 's) unta =
  'time programc × ('a act, nat, 'time, 's) uta list × ('s ⇒ addr) list × (int * int) 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 $\text{step_u} ::$

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

where

$\text{step_u_t}:$

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

$\text{step_u_i}:$

$\llbracket$
 $\text{stepst } P \ n \ u \ (pc_g, [], s, \text{True}, []) (_, _, _, \text{True}, _);$
 $\text{stepst } P \ n \ u \ (pc_u, [], s, \text{True}, []) (_, _, s', _, r);$
 $\forall p < \text{length } N. \exists pc \ st \ s \ rs.$
 $\text{stepst } P \ n \ u' ((I ! p) (L' ! p), [], s', \text{True}, []) (pc, st, s, \text{True}, rs);$
 $(l, pc_g, \text{Sil } a, pc_u, l') \in \text{fst} (N ! p);$
 $\forall p < \text{length } N. u' \vdash \text{snd} (N ! p) (L' ! p);$
 $L!p = l; p < \text{length } L; L' = L[p := l']; u' = [r \rightarrow 0]u;$
 $\text{bounded } B \ s'$
 $\rrbracket$
 $\Longrightarrow (P, N, I, B) \vdash_n \langle L, s, u \rangle \rightarrow_{\text{Act } a} \langle L', s', u' \rangle \mid$

$\text{step_u_s}:$

$\llbracket$
 $\text{stepst } P \ n \ u \ (pc_g1, [], s, \text{True}, []) (_, _, _, \text{True}, _);$
 $\text{stepst } P \ n \ u \ (pc_g2, [], s, \text{True}, []) (_, _, _, \text{True}, _);$
 $\text{stepst } P \ n \ u \ (pc_u2, [], s, \text{True}, []) (_, _, s1, _, r2);$
 $\text{--- UPPAAL semantics quirk}$
 $((\exists pc \ st \ s' \ f. \text{stepst } P \ n \ u \ (pc_u1, [], s, \text{True}, []) (pc, st, s', f, r1))$
 $\vee (\neg (\exists pc \ st \ s' \ f \ r'. \text{stepst } P \ n \ u \ (pc_u1, [], s, \text{True}, []) (pc, st, s', f, r')) \wedge r1 = []));$
 $\text{stepst } P \ n \ u \ (pc_u1, [], s1, \text{True}, []) (_, _, s', _, _);$
 ~~$\text{stepst } P \ n \ u \ (pc_u1, [], s1, \text{True}, []) (_, _, s', _, _);$~~
 $\forall p < \text{length } N. \exists pc \ st \ s \ rs.$
 $\text{stepst } P \ n \ u' ((I ! p) (L' ! p), [], s', \text{True}, []) (pc, st, s, \text{True}, rs);$
 $(l1, pc_g1, \text{In } a, pc_u1, l1') \in \text{fst} (N ! p);$
 $(l2, pc_g2, \text{Out } a, pc_u2, l2') \in \text{fst} (N ! q);$
 $\forall p < \text{length } N. u' \vdash \text{snd} (N ! p) (L' ! p);$
 $L!p = l1; L!q = l2; p < \text{length } L; q < \text{length } L; p \neq q;$
 $L' = L[p := l1', q := l2']; u' = [(r1 @ r2) \rightarrow 0]u;$
 $\text{bounded } B \ s'$
 $\rrbracket \Longrightarrow (P, N, I, B) \vdash_n \langle L, s, u \rangle \rightarrow_{\text{Syn } a} \langle L', s', u' \rangle$

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

inductive $\text{steps_un} ::$

$(\text{'a}, \text{'t} :: \text{time}, \text{'s}) \text{unta} \Rightarrow \text{nat} \Rightarrow \text{'s list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, \text{'t}) \text{cval}$

$\Rightarrow 's \text{ list} \Rightarrow \text{int list} \Rightarrow (\text{nat}, 't) \text{ cval} \Rightarrow \text{bool}$
 $(_ \vdash_ \langle _, _, _ \rangle \rightarrow^* \langle _, _, _ \rangle) [61,61,61,61,61,61] \ 61)$
where
 $\text{refl}: A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L, s, u \rangle \mid$
 $\text{step}: A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \Longrightarrow A \vdash_n \langle L', s', u' \rangle \rightarrow_a \langle L'', s'', u'' \rangle$
 $\Longrightarrow A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$

declare *steps_un.intros*[*intro*]

lemma *stepI2*:

$A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$ **if**
 $A \vdash_n \langle L', s', u' \rangle \rightarrow^* \langle L'', s'', u'' \rangle \ A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
 $\langle \text{proof} \rangle$

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 *make_f pc_u* $\equiv \lambda \ s.$

$\text{case } (\text{exec } P' \ n \ (\text{pc_u}, [], s, \text{True}, []) \ []) \text{ of}$

$\text{None} \Rightarrow []$

$\mid \text{Some } ((_, _, _, _, r), _) \Rightarrow r$

definition *make_mt pc_u* $\equiv \lambda \ s.$

$\text{case } (\text{exec } PT \ n \ (\text{pc_u}, [], s, \text{True}, []) \ []) \text{ of}$

$\text{None} \Rightarrow \text{None}$

$\mid \text{Some } ((_, _, s', _, r), _) \Rightarrow \text{Some } s'$

definition *make_mf pc_u* $\equiv \lambda \ s.$

$\text{case } (\text{exec } PF \ n \ (\text{pc_u}, [], s, \text{True}, []) \ []) \text{ of}$

$\text{None} \Rightarrow \text{None}$

$\mid \text{Some } ((_, _, s', _, r), _) \Rightarrow \text{Some } s'$

definition *make_c pc_g* $\equiv \lambda \ s.$

$\text{case } (\text{exec } PT \ n \ (\text{pc_g}, [], s, \text{True}, []) \ []) \text{ of}$

$\text{None} \Rightarrow \text{False}$

$\mid \text{Some } ((_, _, _, f, _), _) \Rightarrow f$

definition *make_g pc_g* $\equiv \lambda s.$
case (*exec PT n* (*pc_g*, [], *s*, *True*, []) []) *of*
None $\Rightarrow$ []
| *Some* ((_, _, _, _, _), *pcs*) $\Rightarrow$
List.map_filter ($\lambda pc.$
case P pc of
Some (*CEXP ac*) $\Rightarrow$ *Some ac*
| _ $\Rightarrow$ *None*
)
pcs

definition
state_trans_t i $\equiv$
{(*l*, *make_g pc_g*, (*a*, *make_c pc_g*, *make_mf pc_u*), *make_f pc_u*, *l'*) | *l a l' pc_g pc_u.*
(*l*, *pc_g*, *a*, *pc_u*, *l'*) $\in$ *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* $\langle$ *proof* $\rangle$

lemma *bounded_finite*:
finite {*s. bounded B s*} (**is** *finite ?S*)
 $\langle$ *proof* $\rangle$

lemma *finite_state*:
 $\forall\ q < p. \forall\ l. finite\ \{s. (defs.P\ !\ q)\ l\ s\}$
 $\langle$ *proof* $\rangle$

end

fun *is_instr* :: '*t instrc* $\Rightarrow$ *bool* **where**
is_instr (*INSTR* _) = *True* |

$is_instr _ = False$

lemma *step_stripf*:

assumes

$is_instr \text{ cmd}$

shows

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

$\langle proof \rangle$

lemma *step_stript*:

assumes

$is_instr \text{ cmd}$

shows

$stepc \text{ cmd } u \text{ (pc, st, s, f, rs) } = step \text{ (stript cmd) (pc, st, s, f, rs)}$

$\langle proof \rangle$

lemmas $[intro] = stepsc.intros$

lemma *stepsc_f_complete*:

assumes

$stepsc \text{ P } n' \text{ u start end}$

$\bigwedge pc' \text{ st } s' f' \text{ rs cmd.}$

$stepsc \text{ P } n' \text{ u start (pc', st, s', f', rs) } \implies P \text{ pc'} = Some \text{ cmd}$

$\implies is_instr \text{ cmd}$

shows

$steps \text{ (stripfp P) } n' \text{ start end}$

$\langle proof \rangle$

lemma *stepsc_f_sound*:

assumes

$steps \text{ (stripfp P) } n' \text{ start end}$

$\bigwedge pc' \text{ st } s' f' \text{ rs cmd.}$

$stepsc \text{ P } n' \text{ u start (pc', st, s', f', rs) } \implies P \text{ pc'} = Some \text{ cmd}$

$\implies is_instr \text{ cmd}$

shows

$stepsc \text{ P } n' \text{ u start end}$

$\langle proof \rangle$

definition

$time_indep \text{ P } n \text{ start} \equiv$

$\forall pc' \text{ st } s' f' \text{ rs cmd u.}$

$stepsc \text{ P } n \text{ u start (pc', st, s', f', rs) } \wedge P \text{ pc'} = Some \text{ cmd}$

$\longrightarrow is_instr \text{ cmd}$

lemma *stepsc_t_complete*:

assumes

$stepsc \text{ P } n' \text{ u start end}$

$\bigwedge pc' \text{ st } s' f' \text{ rs ac.}$

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

shows

$steps \text{ (striptp P) } n' \text{ start end}$

$\langle proof \rangle$

lemma *stepsc_t_complete2:*

assumes

stepsc P n' u start (pc', st', s', f', rs')

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

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

shows

steps (striptp P) n' start (pc', st', s', f', rs') $\wedge$ ($\forall ac. P pc' = Some (CEXP ac) \longrightarrow u \vdash_a ac$)

<proof>

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

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

<proof>

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

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

<proof>

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

<proof>

lemma *stepsc_visitedc:*

$\exists cc. visitedc P n u start end cc$ **if** *stepsc P n u start end*

<proof>

lemma *visitedc_stepsc:*

stepsc P n u start end **if** *visitedc P n u start end cc*

<proof>

lemma *steps_visited:*

$\exists cc. visited P n start end cc$ **if** *steps P n start end*

<proof>

lemma *visited_steps:*

steps P n start end **if** *visited P n start end cc*

<proof>

context

fixes *P n u start*

```

assumes constraints_conj:  $\forall pc' st s' f' rs ac.$ 
   $stepsc P n u start (pc', st, s', f', rs) \wedge P pc' = Some (CEXP ac) \longrightarrow u \vdash_a ac$ 
begin

lemma stepsc_t_sound':
assumes
  steps (striptp P) n start end
shows
  stepsc P n u start end
   $\langle proof \rangle$ 

lemma stepsc_t_complete':
assumes
  stepsc P n u start end
shows
  steps (striptp P) n start end
   $\langle proof \rangle$ 

lemma stepsc_t_complete'':
assumes
  stepsc P n u start end
shows
   $\exists pcs. visitedc P n u start end pcs \wedge (\forall pc \in set pcs. \forall ac. P pc = Some (CEXP ac) \longrightarrow u \vdash_a ac)$ 
   $\langle proof \rangle$ 

lemma stepsc_t_visited:
assumes
  stepsc P n u start end
shows
   $\exists pcs. visited (striptp P) n start end pcs \wedge (\forall pc \in set pcs. \forall ac. P pc = Some (CEXP ac) \longrightarrow u \vdash_a ac)$ 
   $\langle proof \rangle$ 

lemma stepst_t_complete:
assumes
  stepst P n u start end
shows
   $\exists pcs. exec (striptp P) n start [] = Some (end, pcs) \wedge (\forall pc \in set pcs. \forall ac. P pc = Some (CEXP ac) \longrightarrow u \vdash_a ac)$ 
   $\langle proof \rangle$ 

lemma stepst_t_equiv:
   $(\exists pcs'. exec (striptp P) n start pcs = Some ((pc, st, m, f, rs), pcs'))$ 
   $\longleftrightarrow stepst P n u start (pc, st, m, f, rs)$ 
   $\langle proof \rangle$ 

end

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

```

lemma *time_indep'*:
 $\bigwedge pc' st s' f' rs \text{ cmd.}$
 $\text{stepsc } P \ n \ u \ \text{start } (pc', st, s', f', rs) \implies P \ pc' = \text{Some cmd}$
 $\implies \text{is_instr cmd}$
 $\langle \text{proof} \rangle$

lemma *stepsc_f_complete'*:
assumes
 $\text{stepsc } P \ n \ u \ \text{start end}$
shows
 $\text{steps } (\text{stripfp } P) \ n \ \text{start end}$
 $\langle \text{proof} \rangle$

lemma *stepsc_f_sound'*:
assumes
 $\text{steps } (\text{stripfp } P) \ n \ \text{start end}$
shows
 $\text{stepsc } P \ n \ u \ \text{start end}$
 $\langle \text{proof} \rangle$

lemma *stepsc_f_equiv*:
 $\text{steps } (\text{stripfp } P) \ n \ \text{start end} \longleftrightarrow \text{stepsc } P \ n \ u \ \text{start end}$
 $\langle \text{proof} \rangle$

lemma *stepst_f_equiv*:
 $(\exists pcs'. \text{exec } (\text{stripfp } P) \ n \ \text{start pcs} = \text{Some } ((pc, st, m, f, rs), pcs'))$
 $\longleftrightarrow \text{stepst } P \ n \ u \ \text{start } (pc, st, m, f, rs)$
 $\langle \text{proof} \rangle$

end

lemma *exec_acc*:
assumes $\text{exec } P \ n \ s \ pcs = \text{Some } (s', pcs')$
shows $\exists pcs''. pcs' = pcs'' @ pcs$
 $\langle \text{proof} \rangle$

lemma *exec_acc'*:
assumes $\text{Some } (s', pcs') = \text{exec } P \ n \ s \ pcs$
shows $\exists pcs''. pcs' = pcs'' @ pcs$
 $\langle \text{proof} \rangle$

lemma *exec_min_steps*:
assumes $\text{exec } P \ n \ s \ pcs = \text{Some } (s', pcs' @ pcs)$
shows $\text{exec } P \ (\text{length } pcs') \ s \ pcs = \text{Some } (s', pcs' @ pcs)$
 $\langle \text{proof} \rangle$

lemma *exec_steps_visited*:
assumes
 $\text{exec } P \ (\text{length } pcs') \ s \ pcs = \text{Some } (s', pcs' @ pcs)$
 $\text{steps } P \ (\text{length } pcs') \ s \ (pc, st, m, f, rs)$
shows $pc \in \text{set } pcs'$
 $\langle \text{proof} \rangle$

lemma *stepsc_mono*:

assumes $\text{stepsc } P \ n \ u \ \text{start} \ \text{end} \ n' \geq n$
shows $\text{stepsc } P \ n' \ u \ \text{start} \ \text{end}$
 $\langle \text{proof} \rangle$

lemma stepst_mono :
assumes $\text{stepst } P \ n \ u \ \text{start} \ \text{end} \ n' \geq n$
shows $\text{stepst } P \ n' \ u \ \text{start} \ \text{end}$
 $\langle \text{proof} \rangle$

definition
 $\text{state_indep } P \ n \equiv$
 $\forall \ pc \ f \ pc' \ st \ f' \ rs \ u \ s1 \ s1' \ s2 \ s2' \ rs1 \ rs2.$
 $\text{stepsc } P \ n \ u \ (pc, st, s1, f, rs) \ (pc', st, s1', f', rs1) \wedge$
 $\text{stepsc } P \ n \ u \ (pc, st, s2, f, rs) \ (pc', st, s2', f', rs2)$
 $\longrightarrow rs1 = rs2$

lemma exec_len :
 $n \geq (\text{length } pcs' - \text{length } pcs) \text{ if } \text{exec } P \ n \ s \ pcs = \text{Some } (s', pcs')$
 $\langle \text{proof} \rangle$

lemma $\text{steps_striptp_stepsc}$:
assumes
 $\bigwedge \ pc \ st \ m \ f \ rs \ ac.$
 $\text{steps } (\text{striptp } P) \ n \ s' \ (pc, st, m, f, rs) \Longrightarrow P \ pc = \text{Some } (CEXP \ ac)$
 $\Longrightarrow u' \vdash_a ac$
and $\text{steps } (\text{striptp } P) \ n \ s' \ s''$
shows $\text{stepsc } P \ n \ u' \ s' \ s''$
 $\langle \text{proof} \rangle$

locale $\text{Equiv_TA} =$
 $\text{Equiv_TA_Defs } A \ n \text{ for } A :: ('a, 't :: \text{time}, 's) \text{ unta and } n :: \text{nat} +$
fixes $L :: 's \text{ list and } s :: \text{int list}$
assumes $\text{states}[\text{intro}]: L \in \text{defs.states}' \ s$
and pred_time_indep :
 $\forall \ s. \forall \ L \in \text{defs.states}' \ s. \forall \ q < p. \text{time_indep } P \ n \ ((I ! q) \ (L ! q), [], s, \text{True}, [])$
and upd_time_indep :
 $\forall \ l \ pc_g \ a \ l' \ pc_u \ s. \forall \ q < p. (l, pc_g, a, pc_u, l') \in \text{fst } (N ! q)$
 $\longrightarrow \text{time_indep } P \ n \ (pc_u, [], s, \text{True}, [])$
and clock_conj :
 $\forall \ l \ pc_g \ a \ l' \ pc_u \ s \ u. \forall \ q < p. (l, pc_g, a, pc_u, l') \in \text{fst } (N ! q) \wedge$
 $(\exists \ pc' \ st \ s' \ rs. \text{stepst } P \ n \ u \ (pc_g, [], s, \text{True}, []) \ (pc', st, s', \text{True}, rs)) \longrightarrow$
 $(\forall \ pc' \ st \ s' \ f' \ rs \ ac.$
 $\text{stepsc } P \ n \ u \ (pc_g, [], s, \text{True}, []) \ (pc', st, s', f', rs) \wedge P \ pc' = \text{Some } (CEXP \ ac) \longrightarrow$
 $u \vdash_a ac)$

assumes $\text{Len}: \text{length } N = \text{length } I$
and $\text{inv}: \forall \ q < p. \exists \ pc \ st \ s' \ rs \ pcs.$
 $\text{exec } P' \ n \ ((I ! q) \ (L ! q), [], s, \text{True}, []) \ [] = \text{Some } ((pc, st, s', \text{True}, rs), pcs)$
and $\text{bounded}: \text{bounded } B \ s$
begin

lemma *length_defs_N[simp]*:
 $\text{length } \text{defs}.N = p$
 $\langle \text{proof} \rangle$

lemma *length_defs_P[simp]*:
 $\text{length } \text{defs}.P = p$
 $\langle \text{proof} \rangle$

lemma *inv'*:
 $\forall p < \text{length } \text{defs}.P. (\text{defs}.P ! p) (L ! p) s$
 $\langle \text{proof} \rangle$

lemma *inv''*:
 $\exists pc\ st\ s'\ rs. \text{stepst } P\ n\ u' ((I ! q) (L ! q), [], s, \text{True}, []) (pc, st, s', \text{True}, rs)$
if $q < p$ **for** u'
 $\langle \text{proof} \rangle$

lemma *A_simp[simp]*:
 $PP = P\ N' = N\ I' = I\ B' = B$ **if** $A = (PP, N', I', B')$
 $\langle \text{proof} \rangle$

lemma *A_unfold*:
 $A = (P, N, I, B)$
 $\langle \text{proof} \rangle$

sublocale *prod*: *Prod_TA state_ta* $\langle \text{proof} \rangle$

lemma *inv_simp*:
 $\text{snd } (\text{defs}.N ! q) (L' ! q) = \text{snd } (N ! q) (L' ! q)$ **if** $q < p$ **for** L'
 $\langle \text{proof} \rangle$

lemma *defs_p_eq[simp]*:
 $\text{defs}.p = p$
 $\langle \text{proof} \rangle$

lemma *ball_lessThan[simp]*:
 $(\forall x \in \{..<m\}. Q\ x) \longleftrightarrow (\forall x < m. Q\ x)$
 $\langle \text{proof} \rangle$

lemma *trans_state_taD*:
assumes $(l, g, (a, c, m), f, l') \in \text{fst } (\text{defs}.N ! q)\ q < p$
shows
 $(l, g, (a, c, m), f, l') \in \text{state_trans_t } q$
 $\langle \text{proof} \rangle$

lemma *N_transD*:
assumes $(l, pc_g, a, pc_u, l') \in \text{fst } (N ! q)\ q < p$
shows $(l, \text{make_g } pc_g, (a, \text{make_c } pc_g, \text{make_mf } pc_u), \text{make_f } pc_u, l') \in \text{fst } (\text{defs}.N ! q)$
 $\langle \text{proof} \rangle$

lemma *pred_time_indep'*:
 $\forall L'\ s'\ u'. \forall p' < p. A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$
 $\rightarrow \text{time_indep } P\ n\ ((I ! p') (L' ! p'), [], s', \text{True}, [])$

$\langle \text{proof} \rangle$

lemma P_steps_upd :

assumes

$\text{Some } s'' = \text{make_mf } pc_u \ s' \ (l, pc_g, a, pc_u, l') \in fst \ (N \ ! \ q) \ q < p$

shows

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

$\langle \text{proof} \rangle$

lemma P_steps_reset :

assumes

$q < p \ (l, pc_g, a, pc_u, l') \in fst \ (N \ ! \ q)$

shows

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

$\langle \text{proof} \rangle$

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 $\text{make_f } pc_u \ s' = r$

$\langle \text{proof} \rangle$

lemma $steps_P_upd$:

assumes

$\text{stepst } P \ n \ u' \ (pc_u, [], s', True, []) \ (pc, st, s'', f, r)$
 $q < p \ (l, pc_g, a, pc_u, l') \in fst \ (N \ ! \ q)$

shows

$\text{Some } s'' = \text{make_mf } pc_u \ s' \ (\text{is } ?A) \ r = \text{make_f } pc_u \ s' \ (\text{is } ?B)$

$\langle \text{proof} \rangle$

lemma $steps_P_guard$:

assumes

$\text{stepst } P \ n \ u' \ (pc_g, [], s', True, []) \ (pc, st, s'', True, rs)$
 $q < p \ (l, pc_g, a, pc_u, l') \in fst \ (N \ ! \ q)$

shows

$\text{make_c } pc_g \ s' \ (\text{is } ?A) \ u' \vdash \text{make_g } pc_g \ s' \ (\text{is } ?B)$

$\langle \text{proof} \rangle$

lemma P_steps_guard :

assumes

$\text{make_c } pc_g \ s' \ u' \vdash \text{make_g } pc_g \ s'$
 $q < p \ (l, pc_g, a, pc_u, l') \in fst \ (N \ ! \ q)$

shows

$\exists \ pc \ s'' \ st \ rs. \ stepst \ P \ n \ u' \ (pc_g, [], s', True, []) \ (pc, st, s'', True, rs)$

$\langle \text{proof} \rangle$

lemma $P_bounded$:

assumes

$(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s' \ q < p$

shows *bounded B s'*
 $\langle \text{proof} \rangle$

lemma *P_steps*:

assumes

$(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s'$
 $q < p \ L' \in \text{defs}.states' \ s'$

shows

$\exists \ pc \ st \ s'' \ rs. \ \text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs)$
 $\langle \text{proof} \rangle$

lemma *steps_P*:

assumes

$\text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs)$
 $q < p \ L' \in \text{defs}.states' \ s'$
 $\text{bounded } B \ s'$

shows

$(\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s'$
 $\langle \text{proof} \rangle$

lemma *P_iff*:

$(\exists \ pc \ st \ rs \ s''. \ \text{stepst } P \ n \ u' \ ((I \ ! \ q) \ (L' \ ! \ q), \ [], \ s', \ \text{True}, \ []) \ (pc, \ st, \ s'', \ \text{True}, \ rs)$
 $\wedge \ \text{bounded } B \ s')$
 $\longleftrightarrow (\text{defs}.P \ ! \ q) \ (L' \ ! \ q) \ s' \ \text{if } q < p \ L' \in \text{defs}.states' \ s'$
 $\langle \text{proof} \rangle$

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 $L'[q := l'] \in \text{defs}.states' \ s'$
 $\langle \text{proof} \rangle$

lemma *states'_updI*:

assumes $(L \ ! \ q, \ g, \ (a, \ c, \ m), \ f, \ l') \in \text{fst} \ (\text{defs}.N \ ! \ q)$
shows $L[q := l'] \in \text{defs}.states' \ s'$
 $\langle \text{proof} \rangle$

lemma *states'_updI''*:

assumes

$(L' \ ! \ q, \ g, \ (a, \ c, \ m), \ f, \ l') \in \text{fst} \ (\text{defs}.N \ ! \ q)$
 $(L' \ ! \ q', \ g', \ (a', \ c', \ m'), \ f', \ l'') \in \text{fst} \ (\text{defs}.N \ ! \ q')$
 $L' \in \text{defs}.states' \ s'' \ q \neq q'$

shows $L'[q := l', \ q' := l''] \in \text{defs}.states' \ s'$
 $\langle \text{proof} \rangle$

lemma *equiv_sound*:

assumes $\text{step: state_ta} \vdash \langle L, \ s, \ u \rangle \rightarrow_a \langle L', \ s', \ u' \rangle$
shows $A \vdash_n \langle L, \ s, \ u \rangle \rightarrow_a \langle L', \ s', \ u' \rangle$
 $\langle \text{proof} \rangle$

lemma *state_ta_unfold*:

$\text{state_ta} = (\text{defs}.N, \ \text{defs}.P)$
 $\langle \text{proof} \rangle$

lemma *equiv_complete*:

assumes $step: A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
shows $state_ta \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *equiv_sound'*:

assumes $step: state_ta \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
shows $A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge L' \in defs.states' s' \wedge (\forall q < p. \exists pc\ st\ s''\ rs\ pcs.$
 $exec\ PF\ n\ ((I\ !\ q)\ (L'\ !\ q), [], s', True, []) [] =$
 $Some\ ((pc, st, s'', True, rs), pcs))$
 $\langle proof \rangle$

lemma *equiv_complete'*:

assumes $step: A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
shows $state_ta \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge L' \in defs.states' s'$
 $\wedge (\forall q < p. (defs.P\ !\ q)\ (L'\ !\ q)\ s')$
 $\langle proof \rangle$

lemma *equiv_complete''*:

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

lemma *equiv_steps_sound'*:

assumes $step: state_ta \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$
shows $A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \wedge L' \in defs.states' s' \wedge$
 $(\forall q < p. \exists pc\ st\ s''\ rs\ pcs.$
 $exec\ PF\ n\ ((I\ !\ q)\ (L'\ !\ q), [], s', True, []) [] =$
 $Some\ ((pc, st, s'', True, rs), pcs)) \wedge bounded\ B\ s'$
 $\langle proof \rangle$

lemma *equiv_steps_complete'*:

$state_ta \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \wedge L' \in defs.states' s' \wedge$
 $(\forall q < p. \exists pc\ st\ s''\ rs\ pcs.$
 $exec\ PF\ n\ ((I\ !\ q)\ (L'\ !\ q), [], s', True, []) [] =$
 $Some\ ((pc, st, s'', True, rs), pcs)) \wedge bounded\ B\ s'$
if $A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle\ p > 0$
 $\langle proof \rangle$

lemmas $equiv_steps_sound = equiv_steps_sound'[THEN\ conjunct1]$

lemmas $equiv_steps_complete = equiv_steps_complete'[THEN\ conjunct1]$

lemma *equiv_correct*:

$state_ta \vdash \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle \longleftrightarrow A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle\ \text{if } p > 0$
 $\langle proof \rangle$

lemma *prod_correct*:

$defs.prod_ta \vdash \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \longleftrightarrow A \vdash_n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle\ \text{if } p > 0$
 $\langle proof \rangle$

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:*

```

 $\forall c \in \text{set } r'. \exists x \text{ pc. Some (INSTR (STOREC c x)) = P pc}$ 
if stepc cmd u (pc, st, s, f, r) = Some (pc', st', s', f', r')
 $\forall c \in \text{set } r. \exists x \text{ pc. Some (INSTR (STOREC c x)) = P pc P pc = Some cmd}$ 
<proof>

```

lemma *step_resets':*

```

 $\forall c \in \text{set } r'. \exists x \text{ pc. Some (INSTR (STOREC c x)) = P pc}$ 
if step instr (pc, st, s, f, r) = Some (pc', st', s', f', r')
 $\forall c \in \text{set } r. \exists x \text{ pc. Some (INSTR (STOREC c x)) = P pc P pc = Some (INSTR instr)}$ 
<proof>

```

lemma *step_resets'':*

```

 $\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC c x) = P pc}$ 
if step instr (pc, st, s, f, r) = Some (pc', st', s', f', r')
 $\forall c \in \text{set } r. \exists x \text{ pc. Some (STOREC c x) = P pc P pc = Some instr}$ 
<proof>

```

lemma *steps_reset:*

```

 $\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC c x) = P pc}$ 
if steps P n (pc, st, s, f, r) (pc', st', s', f', r')  $\forall c \in \text{set } r. \exists x \text{ pc. Some (STOREC c x) = P pc}$ 
<proof>

```

lemma *exec_reset:*

```

 $\forall c \in \text{set } r'. \exists x \text{ pc. Some (STOREC c x) = P pc}$ 
if Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, []) pcs
<proof>

```

lemma *exec_pointers:*

```

 $\forall pc \in \text{set pcs'}. \exists pc \text{ instr. Some instr = P pc}$ 
if Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, r) pcs
 $\forall pc \in \text{set pcs}. \exists pc \text{ instr. Some instr = P pc}$ 
<proof>

```

lemma *exec_pointers':*

```

 $\forall pc \in \text{set pcs'}. \exists pc \text{ instr. Some instr = P pc}$ 
if Some ((pc', st', s', f', r'), pcs') = exec P n (pc, st, s, f, r) []
<proof>

```

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))
  ⟨proof⟩

lemma range_prod_invariant:
  range prod_invariant = range (I' s)
  ⟨proof⟩

lemma finite_rangeI:
  assumes  $\forall A \in \{0..<p\}. \text{finite } (\text{range } (\text{snd } (N ! A)))$ 
  shows finite (range prod_invariant)
  ⟨proof⟩

end

context Equiv_TA_Defs
begin

lemma states'_len_simp[simp]:
  length L = p if L ∈ defs.states' s
  ⟨proof⟩

lemma defs_N_p[simp]:
  length defs.N = p
  ⟨proof⟩

lemma defs_p[simp]:
  defs.p = p
  ⟨proof⟩

lemma P_Storec_iff:
  (Some (INSTR (STOREC x xa)) = P pc)  $\longleftrightarrow$  (Some (STOREC x xa) = PF pc)
  ⟨proof⟩

lemma product_trans_i_resets:
  collect_clkvt (Product_TA_Defs.product_trans_i (defs.N_s s))
   $\subseteq \{c. \exists x pc. \text{Some } (\text{INSTR } (\text{STOREC } c x)) = P pc\}$ 
  ⟨proof⟩

lemma product_trans_s_resets:
  collect_clkvt (Product_TA_Defs.product_trans_s (defs.N_s s))
   $\subseteq \{c. \exists x pc. \text{Some } (\text{INSTR } (\text{STOREC } c x)) = P pc\}$ 
  ⟨proof⟩

lemma product_trans_resets:
  collect_clkvt ( $\bigcup s. \text{defs.T}' s$ )  $\subseteq \{c. \exists x pc. \text{Some } (\text{INSTR } (\text{STOREC } c x)) = P pc\}$ 
  ⟨proof⟩

lemma product_trans_guards:
  Timed_Automata.collect_clkvt ( $\bigcup s. \text{defs.T}' s$ )
   $\subseteq \{\text{constraint\_pair } ac \mid ac. \exists pc. \text{Some } (\text{CEXP } ac) = P pc\}$ 
  ⟨proof⟩

```

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* $\Rightarrow$ *nat list* $\Rightarrow$ *int list* $\Rightarrow$ *bool* **where**

check_bexp (*not a*) *L s* $\longleftrightarrow \neg$ *check_bexp a L s* |
check_bexp (*and a b*) *L s* $\longleftrightarrow$ *check_bexp a L s* $\wedge$ *check_bexp b L s* |
check_bexp (*or a b*) *L s* $\longleftrightarrow$ *check_bexp a L s* $\vee$ *check_bexp b L s* |
check_bexp (*imply a b*) *L s* $\longleftrightarrow$ (*check_bexp a L s* $\longrightarrow$ *check_bexp b L s*) |
check_bexp (*loc p l*) *L* $\longleftrightarrow$ *L ! p = l* |
check_bexp (*eq i x*) *s* $\longleftrightarrow$ *s ! i = x* |
check_bexp (*le i x*) *s* $\longleftrightarrow$ *s ! i $\leq$ x* |
check_bexp (*lt i x*) *s* $\longleftrightarrow$ *s ! i < x* |
check_bexp (*ge i x*) *s* $\longleftrightarrow$ *s ! i $\geq$ x* |
check_bexp (*gt i x*) *s* $\longleftrightarrow$ *s ! i > x*

datatype *formula* =

EX bexp | *EG bexp* | *AX bexp* | *AG bexp* | *Leadsto bexp bexp*

abbreviation *repeat x n* $\equiv$ *map* ($\lambda _.$ *x*) [*0*..*n*]

abbreviation *conv_prog P pc* $\equiv$ *map_option* (*map_instrc real_of_int*) (*P pc*)

abbreviation *conv_A'* $\equiv$ $\lambda (T, I).$ (*T*, *conv_cc o I*)

fun *hd_of_formula* :: *formula* $\Rightarrow$ *nat list* $\Rightarrow$ *int list* $\Rightarrow$ *bool* **where**

hd_of_formula (*formula.EX* φ) = *check_bexp* φ |
hd_of_formula (*EG* φ) = *check_bexp* φ |
hd_of_formula (*AX* φ) = *Not oo check_bexp* φ |
hd_of_formula (*AG* φ) = *Not oo check_bexp* φ |
hd_of_formula (*Leadsto* $\varphi _$) = *check_bexp* φ

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 ⟨proof⟩

  abbreviation EA ≡ equiv.state_ta

  abbreviation A ≡ equiv.defs.prod_ta

  lemma equiv_p_eq[simp]:
    equiv.p = p
    ⟨proof⟩

  lemma length_N_s[simp]:
    length (equiv.defs.N_s s) = p
    ⟨proof⟩

  lemma length_N[simp]:
    length equiv.defs.N = p
    ⟨proof⟩

  lemma
    equiv.defs.I' s L = concat (map (λ q. if q < p then I q (L ! q) else []) [0..<length L])
    ⟨proof⟩

end

  lemma snd_comp[simp]:
    snd o (λ i. (f i, g i)) = g
    ⟨proof⟩

  locale UPPAAL_Reachability_Problem_precompiled =
    UPPAAL_Reachability_Problem_precompiled_defs +
    assumes process_length: length inv = p length trans = p length pred = p
    and lengths:
      ∀ i < p. length (pred ! i) = length (trans ! i) ∧ length (inv ! i) = length (trans ! i)
    and state_set: ∀ T ∈ set trans. ∀ xs ∈ set T. ∀ (_, _, _, l) ∈ set xs. l < length T
    assumes consts_nats: snd ' clkp_set' ⊆ 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*: $\forall x \ c. \text{Some} (\text{INSTR} (\text{STOREC } c \ x)) \in \text{set prog} \longrightarrow x = 0$

begin

lemma *consts_nats'*:

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

$\forall ac. \text{Some} (\text{CEXP } ac) \in \text{set prog} \longrightarrow (\text{snd} (\text{constraint_pair } ac) \in \mathbb{N})$
 $\langle \text{proof} \rangle$

lemma *clk_pairs_N_inv*:

$\bigcup (\text{collect_clock_pairs} \text{ 'range } (\text{snd } x)) \subseteq \bigcup (\text{collect_clock_pairs} \text{ 'set } (\text{concat inv}))$
if $x \in \text{set equiv.defs.N}$ **for** x
 $\langle \text{proof} \rangle$

lemma *clkp_set_simp_1*:

$\bigcup (\text{collect_clock_pairs} \text{ 'set } (\text{concat inv})) \supseteq \text{Timed_Automata.collect_clki} (\text{snd } A)$
 $\langle \text{proof} \rangle$

lemma *clk_set_simp_2*:

$\{c. \exists x. \text{Some} (\text{INSTR} (\text{STOREC } c \ x)) \in \text{set prog}\} \supseteq \text{collect_clkvt} (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma *clkp_set_simp_3*:

$\{\text{constraint_pair } ac \mid ac. \text{Some} (\text{CEXP } ac) \in \text{set prog}\} \supseteq \text{Timed_Automata.collect_clkt}$
 $(\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma *clkp_set'_subs*:

$\text{Timed_Automata.clkp_set } A \subseteq \text{clkp_set}'$
 $\langle \text{proof} \rangle$

lemma *clk_set'_subs*:

$\text{clk_set } A \subseteq \text{clk_set}'$
 $\langle \text{proof} \rangle$

lemma *clk_set*:

$\text{clk_set } A \subseteq \{1..m\}$
 $\langle \text{proof} \rangle$

lemma

$\forall (_, d) \in \text{Timed_Automata.clkp_set } A. d \in \mathbb{Z}$
 $\langle \text{proof} \rangle$

lemma *clkp_set'_consts_nat*:

$\forall (_, d) \in \text{clkp_set}'. d \in \mathbb{N}$
 $\langle \text{proof} \rangle$

```

lemma clkp_set_consts_nat:
   $\forall (\_, d) \in \text{Timed\_Automata.clkp\_set } A. d \in \mathbb{N}$ 
   $\langle \text{proof} \rangle$ 

lemma finite_clkp_set':
  finite clkp_set'
   $\langle \text{proof} \rangle$ 

lemma finite_clkp_set_A[intro, simp]:
  finite (Timed_Automata.clkp_set A)
   $\langle \text{proof} \rangle$ 

lemma clkp_set'_bounds:
   $a \in \{\text{Suc } 0..m\} \text{ if } (a, b) \in \text{clkp\_set}'$ 
   $\langle \text{proof} \rangle$ 

lemma finite_range_inv_of_A[intro, simp]:
  finite (range (inv_of A))
   $\langle \text{proof} \rangle$ 

lemma Collect_fold_pair:
   $\{f \ a \ b \mid a \ b. P \ a \ b\} = (\lambda \ (a, b). f \ a \ b) \ ` \ \{(a, b). P \ a \ b\} \text{ for } P$ 
   $\langle \text{proof} \rangle$ 

lemma finite_T[intro, simp]:
  finite (trans_of A)
   $\langle \text{proof} \rangle$ 

sublocale TA_Start_No_Ceiling A (init, s0) m
   $\langle \text{proof} \rangle$ 

lemma P_p[simp]:
  length P = p
   $\langle \text{proof} \rangle$ 

lemma length_I[simp]:
  length equiv.I = p
   $\langle \text{proof} \rangle$ 

lemma length_N[simp]:
  length equiv.N = p
   $\langle \text{proof} \rangle$ 

end

end

theory Program_Analysis
  imports
    UPPAAL_State_Networks_Impl
    Munta_Base.More_Methods
  begin

fun steps_approx :: nat  $\Rightarrow$  't instrc option list  $\Rightarrow$  addr  $\Rightarrow$  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*]:
 $\forall q \in \{..

::nat\}. P\ q \equiv \forall\ q < p. P\ q$
 <proof>

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*]:

$P' \equiv \text{map_option strip } o \text{ (conv_prog } (\lambda i. \text{ if } i < \text{length prog then prog ! } i \text{ else None}))$

lemma *steps_out_of_range'*:

assumes *steps* $P' n$ (pc, st, s, f, rs) (pc', st', s', f', rs') $pc \geq \text{length prog}$
shows $pc' = pc$
 <proof>

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') $pc' < \text{length prog}$
shows $pc' \in \text{steps_approx } n \text{ prog } pc$
 <proof>

lemmas *steps_steps_approx* = *steps_steps_approx'*[*unfolded P'_def*]

end

context

fixes prog :: int instrc option list

begin

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

lemma stepsc_out_of_range:

assumes stepsc (conv_prog P) n u (pc, st, s, f, rs) (pc', st', s', f', rs') pc $\geq$ length prog
shows pc' = pc

$\langle \text{proof} \rangle$

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' $\in$ steps_approx n prog pc

$\langle \text{proof} \rangle$

definition

time_indep_check pc n $\equiv$

$\forall\ pc' \in \text{steps_approx } n\ \text{prog } pc. pc' < \text{length prog}$

$\longrightarrow (\text{case prog ! } pc' \text{ of Some cmd} \Rightarrow \text{is_instr cmd} \mid _ \Rightarrow \text{True})$

lemma time_indep_overapprox:

assumes

time_indep_check pc n

shows time_indep (conv_prog P) n (pc, st, s, f, rs)

$\langle \text{proof} \rangle$

end

context UPPAAL_Reachability_Problem_precompiled_defs

begin

definition

collect_cexp' pc = {ac. Some (CEXP ac) $\in$ ((!) prog) ' steps_approx max_steps prog pc}

definition clkp_set'' i l $\equiv$

collect_clock_pairs (inv ! i ! l) $\cup$

$\bigcup ((\lambda (g, _). \text{constraint_pair ' collect_cexp' } g) ' \text{set (trans ! i ! l)})$

definition

collect_cexp = {ac. Some (CEXP ac) $\in$ set prog}

definition

collect_store' pc =

{(c, x). Some (INSTR (STOREC c x)) $\in$ ((!) prog) ' steps_approx max_steps prog pc}

end

lemma visited_resets_mono:

set r $\subseteq$ set r' if visited P n (pc, st, s, f, r) (pc', st', s', f', r') pcs

$\langle \text{proof} \rangle$

lemma *visitedc_resets_mono*:

set $r \subseteq \text{set } r'$ **if** *visitedc* $P \ n \ u \ (pc, st, s, f, r) \ (pc', st', s', f', r') \ pcs$

$\langle \text{proof} \rangle$

lemma *visited_reset*:

$\exists \ x. \exists \ pc \in \text{set } pcs. \text{Some } (STOREC \ c \ x) = P \ pc$

if *visited* $P \ n \ (pc, st, s, f, r) \ (pc', st', s', f', r') \ pcs \ c \in \text{set } r' - \text{set } r$

$\langle \text{proof} \rangle$

lemma *visitedc_reset*:

$\exists \ x. \exists \ pc \in \text{set } pcs. \text{Some } (INSTR \ (STOREC \ c \ x)) = P \ pc$

if *visitedc* $P \ n \ u \ (pc, st, s, f, r) \ (pc', st', s', f', r') \ pcs \ c \in \text{set } r' - \text{set } r$

$\langle \text{proof} \rangle$

lemma *visited_fuel_mono*:

visited $P \ n' \ s \ s' \ pcs$ **if** *visited* $P \ n \ s \ s' \ pcs \ n' \geq n$

$\langle \text{proof} \rangle$

lemma *visitedc_fuel_mono*:

visitedc $P \ n' \ u \ s \ s' \ pcs$ **if** *visitedc* $P \ n \ u \ s \ s' \ pcs \ n' \geq n$

$\langle \text{proof} \rangle$

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 $P \ n \ (pc', st', s', f', r') \ s'' \ (pcs' @ [pc'])$

$\langle \text{proof} \rangle$

lemma *visited_steps'*:

assumes *visited* $P \ n \ (pc, st, s, f, r) \ s'' \ pcs \ pc' \in \text{set } pcs$

obtains $st' \ s' \ f' \ r'$ **where** *steps* $P \ n \ (pc, st, s, f, r) \ (pc', st', s', f', r')$

$\langle \text{proof} \rangle$

lemma *vistedc_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 $P \ n \ u \ (pc', st', s', f', r') \ s'' \ (pcs' @ [pc'])$

$\langle \text{proof} \rangle$

lemma *visitedc_stepsc'*:

assumes *visitedc* $P \ n \ u \ (pc, st, s, f, r) \ s'' \ pcs \ pc' \in \text{set } pcs$

obtains $st' \ s' \ f' \ r'$ **where** *stepsc* $P \ n \ u \ (pc, st, s, f, r) \ (pc', st', s', f', r')$

$\langle \text{proof} \rangle$

lemma *steps_fuel_mono*:

steps $P \ n' \ s \ s'$ **if** *steps* $P \ n \ s \ s' \ n' \geq n$

$\langle \text{proof} \rangle$

lemma *exec_steps'*:

assumes *exec prog* $n \ s \ pcs = \text{Some } (s', pcs') \ pc' \in \text{set } pcs' - \text{set } pcs$

obtains $st\ m\ f\ rs$ **where** $steps\ prog\ n\ s\ (pc',\ st,\ m,\ f,\ rs)$
 $\langle proof \rangle$

lemma *exec_visited*:

assumes $exec\ prog\ n\ s\ pcs = Some\ ((pc,\ st,\ m,\ f,\ rs),\ pcs')$

obtains pcs'' **where**

$visited\ prog\ n\ s\ (pc,\ st,\ m,\ f,\ rs)\ pcs'' \wedge pcs' = pc \# pcs'' @ pcs \wedge prog\ pc = Some\ HALT$

$\langle proof \rangle$

lemma *exec_reset'*:

$\exists\ x.\ \exists\ pc \in set\ pcs'.\ Some\ (STOREC\ c\ x) = P\ pc$

if $exec\ P\ n\ (pc,\ st,\ s,\ f,\ r)\ pcs = Some\ ((pc',\ st',\ s',\ f',\ r'),\ pcs')$ $c \in set\ r' - set\ r$

$\langle proof \rangle$

lemma *steps_approx_out_of_range*:

$steps_approx\ n\ prog\ pc = \{\}$ **if** $pc \geq length\ prog$

$\langle proof \rangle$

lemma *steps_resets_mono*:

$set\ r \subseteq set\ r' \text{ if } steps\ P\ n\ (pc,\ st,\ s,\ f,\ r)\ (pc',\ st',\ s',\ f',\ r')$

$\langle proof \rangle$

lemma *resets_start*:

assumes

$\forall\ pc \in \{pc..pc'\}.\ \exists\ c\ x.\ prog\ !\ pc = Some\ (INSTR\ (STOREC\ c\ x))$

$steps$

$(map_option\ stripf\ o\ (\lambda pc.\ if\ pc < size\ prog\ then\ prog\ !\ pc\ else\ None))$

$n\ (pc,\ st,\ s,\ f,\ r)\ (pc_t,\ st',\ s',\ f',\ r')$

$prog\ !\ pc_t = Some\ (INSTR\ HALT)$

shows $\{c.\ \exists\ x.\ \exists\ pc \in \{pc..pc'\}.\ prog\ !\ pc = Some\ (INSTR\ (STOREC\ c\ x))\} \subseteq set\ r'$

$\langle proof \rangle$

unbundle *lattice_syntax*

function *find_resets_start* **where**

$find_resets_start\ prog\ pc =$

(

$if\ pc < length\ prog$

$then$

$case\ prog\ !\ pc\ of$

$Some\ (INSTR\ (STOREC\ c\ x)) \Rightarrow (Some\ pc \sqcup find_resets_start\ prog\ (pc + 1)) \mid$

$_ \Rightarrow None$

$else\ None$

)

$\langle proof \rangle$

termination

$\langle proof \rangle$

lemma *find_resets_start*:

$\forall\ pc \in \{pc..pc'\}.\ \exists\ c\ x.\ prog\ !\ pc = Some\ (INSTR\ (STOREC\ c\ x))$ **if**

$find_resets_start\ prog\ pc = Some\ pc'$

$\langle proof \rangle$

lemmas $resets_start' = resets_start[OF\ find_resets_start]$

context *UPPAAL_Reachability_Problem_precompiled_defs*
begin

definition

$collect_store''\ pc \equiv$
 $case\ find_resets_start\ prog\ pc\ of$
 $None \Rightarrow \{\}$ |
 $Some\ pc' \Rightarrow$
 $\{(c, x). Some\ (INSTR\ (STOREC\ c\ x)) \in (!\ prog)\ ' \{pc .. pc'\}\}$

end

lemma *bexp_atLeastAtMost_iff*:

$(\forall\ pc \in \{pc_s..pc_t\}. P\ pc) \longleftrightarrow (\forall\ pc. pc_s \leq pc \wedge pc \leq pc_t \longrightarrow P\ pc)$
 $\langle proof \rangle$

lemma *bexp_atLeastLessThan_iff*:

$(\forall\ pc \in \{pc_s..<pc_t\}. P\ pc) \longleftrightarrow (\forall\ pc. pc_s \leq pc \wedge pc < pc_t \longrightarrow P\ pc)$
 $\langle proof \rangle$

lemma *guaranteed_execution*:

assumes

$\forall\ pc \in \{pc..<pc_t\}.$
 $prog\ !\ pc \neq None$
 $\wedge\ prog\ !\ pc \notin Some\ ' INSTR\ '$
 $\{STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,$
 $instr.LE, instr.EQ\}$
 $\wedge\ (\forall\ c\ d. prog\ !\ pc = Some\ (INSTR\ (STOREC\ c\ d)) \longrightarrow d = 0)$

$\forall\ pc \in \{pc..<pc_t\}. \forall\ pc'. prog\ !\ pc = Some\ (INSTR\ (JMPZ\ pc')) \longrightarrow pc' > pc \wedge pc' \leq pc_t$
 $prog\ !\ pc_t = Some\ (INSTR\ HALT)\ pc_t \geq pc\ n > pc_t - pc\ pc_t < length\ prog$

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

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

$\langle proof \rangle$

function *find_next_halt* **where**

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

$\langle proof \rangle$

termination

$\langle proof \rangle$

lemma *find_next_halt_finds_halt*:
 $prog ! pc' = \text{Some } (\text{INSTR } \text{HALT}) \wedge pc \leq pc' \wedge pc' < \text{length } prog$
if *find_next_halt* *prog* *pc* = *Some pc'*
 <proof>

definition

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

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 $\Rightarrow$
 (
 $\forall pc \in \{pc_s..<pc_t\}.$
 $prog ! pc \neq \text{None}$
 $\wedge prog ! pc \notin \text{Some } ' \text{INSTR } '$
 $\{ \text{STORE}, \text{HALT}, \text{POP}, \text{CALL}, \text{RETURN}, \text{instr.AND}, \text{instr.NOT}, \text{instr.ADD}, \text{instr.LT},$
 $\text{instr.LE}, \text{instr.EQ} \}$
 $\wedge (\text{case } prog ! pc \text{ of } \text{Some } (\text{INSTR } (\text{STOREC } c d)) \Rightarrow d = 0 \mid _ \Rightarrow \text{True})$
 $) \wedge$
 $(\forall pc \in \{pc_s..<pc_t\}.$
 $\text{case } prog ! pc \text{ of } \text{Some } (\text{INSTR } (\text{JMPZ } pc')) \Rightarrow pc' > pc \wedge pc' \leq pc_t \mid _ \Rightarrow \text{True})$
 $\wedge n > pc_t - pc_s$

<proof>

lemma *guaranteed_execution'*:
 $\exists pc_t st' s' f' r' pcs'. \text{exec}$
 $(\text{map_option stripf } o (\lambda pc. \text{if } pc < \text{size } prog \text{ then } prog ! pc \text{ else } \text{None}))$
 $n (pc, st, s, f, r) pcs = \text{Some } ((pc_t, st', s', f', r'), pcs')$
if *guaranteed_execution_cond prog pc n*
 <proof>

context *UPPAAL_Reachability_Problem_precompiled_defs*
begin

lemma *collect_cexp_alt_def*:

collect_cexp =
 set (*List.map_filter*
 ($\lambda x. \text{case } x \text{ of } \text{Some } (CExp \text{ } ac) \Rightarrow \text{Some } ac \mid _ \Rightarrow \text{None}$)
prog)
 ⟨*proof*⟩

lemma *clkp_set'_alt_def*:

clkp_set' =
 $\bigcup (\text{collect_clock_pairs} \text{ ' set (concat inv) } \cup (\text{constraint_pair} \text{ ' collect_cexp})$
 ⟨*proof*⟩

definition

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

lemma *collect_store_alt_def*:

collect_store =
 set (*List.map_filter*
 ($\lambda x. \text{case } x \text{ of } \text{Some } (INSTR \text{ } (STOREC \text{ } c \text{ } x)) \Rightarrow \text{Some } (c, x) \mid _ \Rightarrow \text{None}$)
prog)
 ⟨*proof*⟩

lemma *clk_set'_alt_def*: *clk_set'* = (*fst* ' *clkp_set'* $\cup$ *fst* ' *collect_store*)

⟨*proof*⟩

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*) |
conj_instr (*INSTR instr.AND*) _ = True |
conj_instr _ _ = False

inductive *is_conj_block'* :: 't instrc option list $\Rightarrow$ addr $\Rightarrow$ addr $\Rightarrow$ bool **where**

is_conj_block' *prog pc pc* **if**
pc < length *prog*
prog ! *pc* = Some (*INSTR HALT*) |

is_conj_block' *prog pc pc'* **if**
pc' < length *prog*
prog ! *pc* = Some (*INSTR COPY*) *prog* ! (*pc* + 1) = Some (*CExp ac*)
prog ! (*pc* + 2) = Some (*INSTR instr.AND*)
is_conj_block' *prog* (*pc* + 3) *pc'* |
is_conj_block' *prog pc pc'* **if**
pc' < length *prog*
prog ! *pc* = Some (*INSTR COPY*)
prog ! (*pc* + 1) = Some (*INSTR (JMPZ pc')*)
prog ! (*pc* + 2) = Some (*CExp ac*)
prog ! (*pc* + 3) = Some (*INSTR instr.AND*)

$is_conj_block' \text{ prog } (pc + 4) \text{ pc}'$

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

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

$\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

lemma $is_conj_block' _ len_prog$:
 $pc' < \text{length prog} \text{ if } is_conj_block' \text{ prog } pc \text{ pc'}$
 $\langle \text{proof} \rangle$

lemma $check_conj_block'$:
 $check_conj_block' \text{ prog } pc = \text{Some pc'} \Rightarrow is_conj_block' \text{ prog } pc \text{ pc'}$
 $\langle \text{proof} \rangle$

lemma $stepsc_reverseE$:
assumes $stepsc \text{ prog } (Suc \text{ n}) \text{ u } s \text{ s'' s''} \neq s$
obtains $pc' \text{ st' m' f' rs' cmd where}$
 $\quad stepc \text{ cmd } u (pc', st', m', f', rs') = \text{Some s''}$
 $\quad \text{prog } pc' = \text{Some cmd}$
 $\quad stepsc \text{ prog } n \text{ u } s (pc', st', m', f', rs')$
 $\langle \text{proof} \rangle$

lemma $stepsc_reverseE$:
assumes $stepsc \text{ prog } n \text{ u } s \text{ s'' s''} \neq s$
obtains $n' \text{ pc' st' m' f' rs' cmd where}$
 $\quad n = Suc \text{ n'}$
 $\quad stepc \text{ cmd } u (pc', st', m', f', rs') = \text{Some s''}$
 $\quad \text{prog } pc' = \text{Some cmd}$
 $\quad stepsc \text{ prog } n' \text{ u } s (pc', st', m', f', rs')$
 $\langle \text{proof} \rangle$

lemma

$pc = pc' - 1$ **if**
 $stepc (CEXP cc) u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$
 $\langle proof \rangle$

lemma

$pc = pc' - 1$ **if**
 $stepc (INSTR instr) u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$
 $\neg (\exists x. instr = JMPZ pc') \text{ instr} \neq RETURN \text{ instr} \neq CALL$
 $\langle proof \rangle$

lemma *stepc_pc_no_jump*:

$pc = pc' - 1$ **if**
 $stepc cmd u (pc, st, m, f, rs) = Some (pc', st', m', f', rs')$
 $cmd \neq INSTR (JMPZ pc') \text{ cmd} \neq INSTR RETURN \text{ cmd} \neq INSTR CALL$
 $\langle proof \rangle$

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

for *prog* **where**

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

declare *stepsn.intros*[intro]

lemma *stepsc_stepsn*:

assumes $stepsc P n u s s'$
obtains n' **where** $stepsn P n' u s s' n' < n$
 $\langle proof \rangle$

lemma *stepsn_stepsc*:

assumes $stepsn P n' u s s' n' < n$
shows $stepsc P n u s s'$
 $\langle proof \rangle$

lemma *stepsn_extend*:

assumes $stepsn P n1 u s s1 \text{ stepsn } P n2 u s s2 \text{ } n1 \leq n2$
shows $stepsn P (n2 - n1) u s1 s2$
 $\langle proof \rangle$

lemma *stepsc_halt*:

$s' = (pc, s)$ **if** $stepsc P n u (pc, s) s' P pc = Some (INSTR HALT)$
 $\langle proof \rangle$

lemma *stepsn_halt*:

$s' = (pc, s)$ **if** $stepsn P n u (pc, s) s' P pc = Some (INSTR HALT)$
 $\langle proof \rangle$

lemma *is_conj_block'_pc_mono*:

$pc \leq pc'$ **if** $is_conj_block' \text{ prog } pc pc'$
 $\langle proof \rangle$

lemma *is_conj_block'_halt*:

prog ! pc' = Some (INSTR HALT) if is_conj_block' prog pc pc'
 ⟨proof⟩

lemma *numeral_4_eq_4*:

4 = Suc (Suc (Suc (Suc 0)))
 ⟨proof⟩

lemma *is_conj_block'_is_conj*:

assumes *is_conj_block' P pc pc'*

and *stepsn* ($\lambda i. \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$) *n u* (*pc, st, s, f, rs*) (*pc_t, st_t, s_t, True, rs_t*)

and *P ! pc_t = Some (INSTR HALT)*

shows *f ∧ pc_t = pc'*

⟨proof⟩

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 ∧ pc_t = pc'*

⟨proof⟩

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 ⊢_a ac*) ∧ *pc_t = pc'*

⟨proof⟩

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 ⊢_a ac*) ∧ *pc_t = pc'*

⟨proof⟩

definition

is_conj_block P pc pc' ≡

($\exists ac. P ! pc = \text{Some } (CEXP ac)$) ∧ *is_conj_block' P (pc + 1) pc'*

∨ ($\exists ac.$

P ! pc = Some (CEXP ac)) ∧ *P ! (pc + 1) = Some (INSTR instr.AND)*

∧ *is_conj_block' P (pc + 2) pc'*

lemma *is_conj_block_alt_def[code]*:

is_conj_block P pc pc' ≡

(*case P ! pc of Some (CEXP ac) ⇒ True | _ ⇒ False*) ∧ *is_conj_block' P (pc + 1) pc'*

∨ (*case P ! pc of Some (CEXP ac) ⇒ True | _ ⇒ False*) ∧ *P ! (pc + 1) = Some (INSTR instr.AND)*

∧ *is_conj_block' P (pc + 2) pc'*

$\langle proof \rangle$

lemma *is_conj_block is_conj*:

assumes *is_conj_block* P pc 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 $(u \vdash_a ac) \wedge pc_t = pc'$

$\langle proof \rangle$

lemma *is_conj_block'_decomp*:

is_conj_block P pc' pc'' **if**

is_conj_block' P pc pc'' $P ! pc' = Some (CEXP ac)$ $pc \leq pc' \leq pc''$

$\langle proof \rangle$

lemma *is_conj_block_decomp*:

is_conj_block P pc' pc'' **if**

is_conj_block P pc pc'' $P ! pc' = Some (CEXP ac)$ $pc \leq pc' \leq pc''$

$\langle proof \rangle$

lemma *steps_approx_finite*[*intro,simp*]:

finite $(\text{steps_approx } n \ P \ pc_s)$

$\langle proof \rangle$

abbreviation *conv_P* $\equiv \text{map } (\text{map_option } (\text{map_instrc } \text{real_of_int}))$

lemma *stepst_stepc_extend*:

stepst $P \ n \ u \ (pc', s') \ (pc'', s'')$

if *stepst* $P \ n \ u \ (pc, s) \ (pc'', s'')$ *stepsc* $P \ n \ u \ (pc, s) \ (pc', s')$

$\langle proof \rangle$

lemma *conv_P_conj_block'*[*intro*]:

is_conj_block' $(\text{conv_P } P) \ pc \ pc'$ **if** *is_conj_block'* $P \ pc \ pc'$

$\langle proof \rangle$

lemma *conv_P_conj_block*[*intro*]:

is_conj_block $(\text{conv_P } P) \ pc \ pc'$ **if** *is_conj_block* $P \ pc \ pc'$

$\langle proof \rangle$

context

fixes $P :: \text{int instrc option list}$

and $pc_s :: \text{addr}$

and $n :: \text{nat}$

begin

private abbreviation *prog* $i \equiv \text{if } i < \text{length } P \text{ then } P ! i \text{ else None}$

lemma *stepst_conv_P*:

stepst $(\lambda i. \text{if } i < \text{length } (\text{conv_P } P) \text{ then } \text{conv_P } P ! i \text{ else None}) \ n \ u \ s \ s'$ **if**

stepst $(\text{conv_prog prog}) \ n \ u \ s \ s'$ $\langle proof \rangle$

lemma *is_conj*:

fixes $u :: \text{nat} \Rightarrow \text{real}$
defines $S \equiv \text{steps_approx } n \ P \ pc_s$
defines $pc_c \equiv \text{Min } \{pc. \exists \ ac. pc \in S \wedge P ! pc = \text{Some } (CEXP \ ac)\}$
assumes $\text{is_conj_block } P \ pc_c \ (\text{Max } S)$
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 $(u \vdash_a \text{conv_ac } ac) \wedge pc_t = \text{Max } S$
 $\langle \text{proof} \rangle$

lemma $\text{is_conj}'$:
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
 $\langle \text{proof} \rangle$

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 check_conj_block :
 $\text{check_conj_block } pc \ pc' \implies \text{is_conj_block } P \ pc \ pc'$
 $\langle \text{proof} \rangle$

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 $\text{conjunction_check_alt_def}[\text{code}]$:
 $\text{conjunction_check} =$
 $($
 let
 $\quad S = \text{steps_approx } n \ P \ pc_s;$
 $\quad S' = \{pc. pc \in S \wedge (\text{case } P ! pc \text{ of } \text{Some } (CEXP \ ac) \Rightarrow \text{True} \mid _ \Rightarrow \text{False})\}$
 in
 $\quad S' = \{\} \vee \text{check_conj_block } (\text{Min } S') \ (\text{Max } S)$
 $)$
 $\langle \text{proof} \rangle$

lemma conjunction_check :
fixes $u :: \text{nat} \Rightarrow \text{real}$
assumes 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  $P \vdash pc' = \text{Some } (CEXP \ ac) \ pc' < \text{length } P$ 
  shows  $u \vdash_a \text{conv\_ac } ac$ 
  <proof>

end

end

theory UPPAAL_State_Networks_Impl_R refine
  imports
    Program_Analysis
    Munta_Base.Normalized_Zone_Semantics_Impl_R 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  $\text{exec prog } n \ (pc, st, s, f, rs) \ pcs = \text{Some } ((pc', st', s', f', rs'), pcs')$ 
  shows  $\text{length } s = \text{length } s'$ 
  <proof>

```

```

locale UPPAAL_Reachability_Problem_precompiled_defs' =
  UPPAAL_Reachability_Problem_precompiled_defs +
  fixes  $na :: \text{nat}$ 
begin

```

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

definition

```

trans_i_map =
  map (map (List.map_filter
    ( $\lambda \ (g, a, m, l'). \text{case } a \text{ of } \text{Sil } a \Rightarrow \text{Some } (g, a, m, l') \mid \_ \Rightarrow \text{None}$ )
  )) trans

```

definition

```

trans_in_map  $\equiv$ 
  map (map (map
    ( $\lambda \ (g, a, m, l'). \text{case } a \text{ of } \text{In } a \Rightarrow (g, a, m, l') \mid \_ \Rightarrow \text{filter } (\lambda \ (g, a, m, l').$ 
       $\text{case } a \text{ of } \text{In } a \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})$ 
    ) trans

```

definition

```

trans_out_map  $\equiv$ 
  map (map (map
    ( $\lambda \ (g, a, m, l'). \text{case } a \text{ of } \text{Out } a \Rightarrow (g, a, m, l') \mid \_ \Rightarrow \text{filter } (\lambda \ (g, a, m, l').$ 
       $\text{case } a \text{ of } \text{Out } a \Rightarrow \text{True} \mid \_ \Rightarrow \text{False})$ 
    ) trans

```

abbreviation

$nested_list_to_iarray\ xs \equiv IArray\ (map\ IArray\ xs)$

definition

$actions_by_state\ i \equiv fold\ (\lambda\ t\ acc.\ acc[fst\ (snd\ t) := (i, t) \# (acc ! fst\ (snd\ t))])$

definition

$all_actions_by_state\ t\ L \equiv$
 $fold\ (\lambda\ i.\ actions_by_state\ i\ (t !! i !! (L ! i)))\ [0..<p]\ (repeat\ []\ na)$

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 $PROG'_PROG\ [simp]:$

$PROG' = PROG$

$\langle proof \rangle$

definition $bounded'\ s \equiv$

$(\forall\ i < length\ s.\ fst\ (IArray\ bounds !! i) < s ! i \wedge s ! i < snd\ (IArray\ bounds !! i))$

definition

$check_pred\ L\ s \equiv$
 $list_all$
 $(\lambda\ q.$
 $\quad case\ runf\ (pred ! q ! (L ! q))\ s\ of$
 $\quad\quad Some\ ((_, _, _, f, _), _) \Rightarrow f \wedge bounded'\ s$
 $\quad\quad | None \Rightarrow False$
 $\quad)$
 $[0..<p]$

definition

$make_cconstr\ pcs = List.map_filter$
 $(\lambda\ pc.$
 $\quad case\ PROG\ pc\ of$
 $\quad\quad Some\ (CEXP\ ac) \Rightarrow Some\ ac$
 $\quad\quad | _ \Rightarrow None$
 $\quad)$
 pcs

definition

$check_g\ pc\ s \equiv$
 $case\ runt\ pc\ s\ of$
 $\quad Some\ ((_, _, _, True, _), pcs) \Rightarrow Some\ (make_cconstr\ pcs)$
 $\quad | _ \Rightarrow None$

definition

$trans_i_from \equiv \lambda\ (L, s)\ i.$
 $List.map_filter\ (\lambda\ (g, a, m, l').$
 $\quad case\ check_g\ g\ s\ of$

```

Some cc ⇒
(case runf m s of
  Some ((_, _, s', _, r), _) ⇒
    if check_pred (L[i := l']) s'
    then Some (cc, a, r, (L[i := l'], s'))
    else None
| _ ⇒ None)
| _ ⇒ None)
((IArray (map IArray trans_i_map)) !! i !! (L ! i))

```

definition

$trans_i_fun\ L \equiv concat\ (map\ (trans_i_from\ L)\ [0..<p])$

definition

```

make_reset m1 s ≡
case runf m1 s of
  Some ((_, _, _, _, r1), _) ⇒ r1
| None ⇒ []

```

definition

```

pairs_by_action ≡ λ (L, s) OUT. concat o
map (λ (i, g1, a, m1, l1). List.map_filter
(λ (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) ⇒ (
      case runf m2 s of
        Some ((_, _, s1, _, r2), _) ⇒ (
          case runf m1 s1 of
            Some ((_, _, s', _, _), _) ⇒
              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
          | _ ⇒ None)
        | _ ⇒ None)
      | _ ⇒ None)
    )
  OUT)

```

definition

```

trans_s_fun ≡ λ (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 (λ 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') ∈ set (trans_i_fun L)
shows (g, a, r, L') ∈ set (trans_fun L)
⟨proof⟩

lemma trans_s_fun_trans_fun:
assumes (g, a, r, L') ∈ set (trans_s_fun L)
shows (g, a, r, L') ∈ set (trans_fun L)
⟨proof⟩

end

context Prod_TA_Defs
begin

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') ∈ Product_TA_Defs.product_trans_i (N_s s)
    }
  ∧
    (∀ q < p. (P ! q) (L ! q) s) ∧ (∀ q < p. (P ! q) (L' ! q) s')
    ∧ c s ∧ Some s' = m s
  ⟨proof⟩

lemma prod_trans_s_alt_def:
  prod_trans_s =
    {((L, s), g, a, r, (L', s')) | L s g ci co a r mi mo L' s1 s'.
      ci s ∧ co s
      ∧ (∀ q < p. (P ! q) (L ! q) s) ∧ (∀ q < p. (P ! q) (L' ! q) s')
      ∧ (L, g, (a, Networks.label.Syn (ci, mi) (co, mo)), r, L')
      ∈ Product_TA_Defs.product_trans_s (N_s s)
      ∧ Some s' = mi s1 ∧ Some s1 = mo s
    }
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

end

lemma (in Equiv_TA_Defs) p_p:
  defs.p = p
  ⟨proof⟩

```

context

Prod_TA_Defs

begin

lemma *states'_alt_def*:

states' s =

$\{L. \text{length } L = p \wedge$

$(\forall q < p. (L ! q) \in \text{fst } '(\text{fst } (\text{fst } A ! q)) \cup (\text{snd } o \text{ snd } o \text{ snd } o \text{ snd}) '(\text{fst } (\text{fst } A ! q)))\}$

$\langle \text{proof} \rangle$

end

context *UPPAAL_Reachability_Problem_precompiled*

begin

lemma *PF_unfold*:

equiv.PF = stripfp (conv_prog PROG)

$\langle \text{proof} \rangle$

lemma *PT_unfold*:

equiv.PT = striptp (conv_prog PROG)

$\langle \text{proof} \rangle$

lemma *states'I*:

$l \in \text{equiv.defs.states}' s \text{ if } A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$

$\langle \text{proof} \rangle$

lemma *A_lengthD*:

$\text{length } l = p \text{ if } A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$

$\langle \text{proof} \rangle$

lemma *N_s_state_trans*:

assumes *equiv.defs.N_s s ! q* $\vdash l ! q \longrightarrow^{g,(a, c, m'),r} l' q < p$

obtains *f' g' where*

$(l ! q, g', (a, c, m'), f', l') \in \text{equiv.state_trans } q \text{ } g = g' s \text{ } r = f' s$

$\langle \text{proof} \rangle$

lemma *make_f_collect_store*:

assumes $(l, pc_g, a, pc_u, l') \in \text{fst } (\text{equiv.N ! } q) \text{ } c \in \text{set } (\text{equiv.make_f } pc_u \text{ } s) \text{ } q < p$

shows $c \in \text{fst } ' \text{collect_store}' pc_u$

$\langle \text{proof} \rangle$

lemma *resets_approx*:

set r $\subseteq$

$\bigcup \{ \text{fst } ' \text{collect_store}' r \mid i \text{ } g \text{ } a \text{ } r. (g, a, r, (l' ! i)) \in \text{set } (\text{trans ! } i ! (l ! i)) \}$

if $A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$

$\langle \text{proof} \rangle$

lemma *make_g_clkp_set''*:

fixes *x*

assumes

$(l, pc_g, a, pc_u, l') \in \text{fst } (\text{equiv.N ! } q) \text{ } x \in \text{collect_clock_pairs } (\text{equiv.make_g } pc_g \text{ } s)$

$q < p$
shows $x \in \text{clkp_set}'' q l$
 $\langle \text{proof} \rangle$

lemma *guard_approx*:
 $\text{collect_clock_pairs } g \subseteq$
 $\bigcup \{ \text{clkp_set}'' i (l ! i) \mid i g a r. (g, a, r, (l' ! i)) \in \text{set } (\text{trans} ! i ! (l ! i)) \wedge l \in \text{equiv.defs.states}' s \wedge i < p$
 $\}$
if $A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$
 $\langle \text{proof} \rangle$

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'*:
 $\text{Equiv_TA_Defs conv } N \text{ max_steps } \langle \text{proof} \rangle$

lemma *equiv_states'_alt_def*:
 $\text{equiv.defs.states}' s =$
 $\{ L. \text{length } L = p \wedge$
 $(\forall q < p. L ! q \in \text{fst } ' \text{fst } (\text{equiv.N} ! q)$
 $\vee L ! q \in (\text{snd } o \text{snd } o \text{snd } o \text{snd}) ' \text{fst } (\text{equiv.N} ! q)) \}$
 $\langle \text{proof} \rangle$

lemma *init_states*:
 $\text{init} \in \text{equiv.defs.states}' s_0$
 $\langle \text{proof} \rangle$

lemma *p_p[simp]*:
 $\text{defs'.p} = p$
 $\langle \text{proof} \rangle$

lemma *T_s_unfold_1'*:
 $\text{fst } ' \text{defs'.defs.T_s } q s = \text{fst } ' \text{fst } (\text{defs'.N} ! q) \text{ if } q < p$
 $\langle \text{proof} \rangle$

lemma *T_s_unfold_2'*:
 $(\text{snd } o \text{snd } o \text{snd } o \text{snd}) ' \text{defs'.defs.T_s } q s = (\text{snd } o \text{snd } o \text{snd } o \text{snd}) ' \text{fst } (\text{defs'.N} ! q)$
if $q < p$
 $\langle \text{proof} \rangle$

lemma *product_states'_alt_def*:
 $\text{defs'.defs.states}' s =$
 $\{ L. \text{length } L = p \wedge$
 $(\forall q < p. L ! q \in \text{fst } ' \text{fst } (\text{defs'.N} ! q)$
 $\vee L ! q \in (\text{snd } o \text{snd } o \text{snd } o \text{snd}) ' \text{fst } (\text{defs'.N} ! q)) \}$
 $\langle \text{proof} \rangle$

lemma *states'_conv[simp]*:
defs'.defs.states' s = equiv.defs.states' s
 ⟨*proof*⟩

lemma *init_states_defs'[intro]*:
init ∈ *defs'.defs.states' s₀*
 ⟨*proof*⟩

lemma
defs'.I = equiv.I
 ⟨*proof*⟩

lemma *PF_PF[simp]*:
defs'.PF = equiv.PF
 ⟨*proof*⟩

lemma *PF_PROG[simp]*:
equiv.PF = stripfp PROG
 ⟨*proof*⟩

lemma *I_simp[simp]*:
 (*equiv.I ! q*) *l* = *pred ! q ! l* **if** *q < p*
 ⟨*proof*⟩

lemma
defs'.P = conv_prog PROG
 ⟨*proof*⟩

lemma *states_len[intro]*:
assumes
 q < p *L* ∈ *equiv.defs.states' s*
shows
 L ! q < length (trans ! q)
 ⟨*proof*⟩

end

locale *UPPAAL_Reachability_Problem_precompiled_ceiling* =
UPPAAL_Reachability_Problem_precompiled +
fixes *k :: nat list list list*
assumes *k_ceiling*:
 ∀ *i* < *p*. ∀ *l* < *length (trans ! i)*. ∀ (*x, m*) ∈ *clkp_set'' i l*. *m* ≤ *k ! i ! l ! x*
 ∀ *i* < *p*. ∀ *l* < *length (trans ! i)*. ∀ (*x, m*) ∈ *collect_clock_pairs (inv ! i ! l)*.
 m ≤ *k ! i ! l ! x*
and *k_resets*:
 ∀ *i* < *p*. ∀ *l* < *length (trans ! i)*. ∀ (*g, a, r, l'*) ∈ *set (trans ! i ! l)*.
 ∀ *c* ∈ {0..*m*+1} − *fst 'collect_store'' r. k ! i ! l' ! c* ≤ *k ! i ! l ! c*
and *k_length*:
 length k = *p* ∀ *i* < *p*. *length (k ! i)* = *length (trans ! i)*
 ∀ *xs* ∈ *set k*. ∀ *xxs* ∈ *set xs*. *length xxs* = *m + 1*
and *k_0*:
 ∀ *i* < *p*. ∀ *l* < *length (trans ! i)*. *k ! i ! l ! 0* = 0
and *guaranteed_resets*:

$\forall i < p. \forall l < \text{length } (\text{trans } ! i). \forall (g, a, r, l') \in \text{set } (\text{trans } ! i ! l).$
 $\text{guaranteed_execution_cond prog } r \text{ max_steps}$

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' :

$\text{equiv}.p = p$

$\langle \text{proof} \rangle$

lemma $\text{clkp_set_clk_set_subs}$:

$\text{fst } ' \text{ clkp_set } A \ (l, s) \subseteq \text{clk_set } A$

$\langle \text{proof} \rangle$

lemma $k_ceiling_1$:

$\forall l. \forall (x, m) \in \text{clkp_set } A \ l. m \leq k_fun \ l \ x$

$\langle \text{proof} \rangle$

lemma $k_ceiling_2$:

$\forall l \ g \ a \ r \ l' \ c. A \vdash l \longrightarrow^{g, a, r} l' \wedge c \notin \text{set } r \longrightarrow k_fun \ l' \ c \leq k_fun \ l \ c$

$\langle \text{proof} \rangle$

lemma

shows $k_ceiling'$:

$\forall l. \forall (x, m) \in \text{clkp_set } A \ l. m \leq k_fun \ l \ x$

$\forall l \ g \ a \ r \ l' \ c. A \vdash l \longrightarrow^{g, a, r} l' \wedge c \notin \text{set } r \longrightarrow k_fun \ l' \ c \leq k_fun \ l \ c$

and k_bound' :

$\forall l. \forall i > m. k_fun \ l \ i = 0$

and k_0' :

$\forall l. k_fun \ l \ 0 = 0$

$\langle \text{proof} \rangle$

sublocale $\text{Reachability_Problem } A \ (\text{init}, s_0) \ m \ k_fun \ \text{PR_CONST } (\lambda (l, s). F \ l \ s)$

$\langle \text{proof} \rangle$

end

locale $\text{UPPAAL_Reachability_Problem_precompiled_start_state} =$

$\text{UPPAAL_Reachability_Problem_precompiled } _ _ _ _ \text{pred}$

for $\text{pred} :: \text{nat list list} +$

fixes $s_0 :: \text{int list}$

assumes start_pred :

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

$\text{exec } (\text{stripfp } \text{PROG}) \ \text{max_steps } ((\text{pred } ! q ! (\text{init } ! q)), [], s_0, \text{True}, []) []$

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

and bounded : $\text{bounded bounds } s_0$

and pred_time_indep : $\forall x \in \text{set pred}. \forall pc \in \text{set } x. \text{time_indep_check prog } pc \ \text{max_steps}$

and upd_time_indep :

$\forall T \in \text{set trans}. \forall xs \in \text{set } T. \forall (_, _, pc_u, _) \in \text{set } xs.$

$\text{time_indep_check prog } pc_u \ \text{max_steps}$

```

and clock_conj:
   $\forall T \in \text{set trans. } \forall xs \in \text{set } T. \forall (pc\_g, \_, \_, \_) \in \text{set } xs.$ 
  conjunction_check prog pc_g max_steps
begin

lemma B0[intro]:
  bounded defs'.B s0
  <proof>

lemma equiv_P_simp:
  equiv.P = PROG
  <proof>

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>

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>

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)
  stepsc defs'.P max_steps u (pc_g, [], s, True, []) (pc', st, s', f', rs)
  defs'.P pc' = Some (CEXP ac)
  <proof>

sublocale product':
  Equiv_TA conv N max_steps init s0
  <proof>

lemma fst_S[simp]:
  fst ' (λ(l, g, a, r, l'). (l, map conv_ac g, a, r, l')) ' S = fst ' S
  <proof>

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
  <proof>

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:
   $\forall T \in \text{set trans. } \forall xs \in \text{set } T. \forall (\_, a, \_) \in \text{set } xs. \text{pred\_act } (\lambda a. a < na) a$ 
begin

```

sublocale *Reachability_Problem_Impl_Defs* $___ A$ (*init*, *s*₀) *m* *k_fun* *PR_CONST* (λ (*l*, *s*). *F l s*) $\langle$ *proof* $\rangle$

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 $(g1, a, r1) \in set (IArray (map \ IArray \ trans_in_map) \ !! \ q \ !! \ l)$

$\langle$ *proof* $\rangle$

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 $(g1, a, r1) \in set (IArray (map \ IArray \ trans_out_map) \ !! \ q \ !! \ l)$

$\langle$ *proof* $\rangle$

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**

$i < length (trans \ ! \ q \ ! \ l) \wedge trans \ ! \ q \ ! \ l \ ! \ i = (g1, In \ a, r1)$

$\langle$ *proof* $\rangle$

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**

$i < length (trans \ ! \ q \ ! \ l) \wedge trans \ ! \ q \ ! \ l \ ! \ i = (g1, Out \ a, r1)$

$\langle$ *proof* $\rangle$

lemma *in_actions_by_stateI*:

assumes

$(g1, a, r1) \in set \ xs \ a < length \ acc$

shows

$(q, g1, a, r1) \in set (actions_by_state \ q \ xs \ acc \ ! \ a)$

$\wedge a < length (actions_by_state \ q \ xs \ acc)$

$\langle$ *proof* $\rangle$

lemma *in_actions_by_state_preserv*:

assumes

$(q, g1, a, r1) \in set (acc \ ! \ a) \ a < length \ acc$

shows

$(q, g1, a, r1) \in set (actions_by_state \ y \ xs \ acc \ ! \ a)$

$\wedge a < length (actions_by_state \ y \ xs \ acc)$

$\langle$ *proof* $\rangle$

lemma *length_actions_by_state_preserv[simp]*:
shows $\text{length } (\text{actions_by_state } y \text{ } xs \text{ } acc) = \text{length } acc$
 $\langle \text{proof} \rangle$

lemma *in_all_actions_by_stateI*:
assumes
 $a < na \quad q < p \quad (g1, a, r1) \in \text{set } (M !! q !! (L ! q))$
shows
 $(q, g1, a, r1) \in \text{set } (\text{all_actions_by_state } M \text{ } L ! a)$
 $\langle \text{proof} \rangle$

lemma *actions_by_state_inj*:
assumes $j < \text{length } acc$
shows $\forall (q, a) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc ! j). (q, a) \notin \text{set } (acc ! j) \longrightarrow i = q$
 $\langle \text{proof} \rangle$

lemma *actions_by_state_inj'*:
assumes $j < \text{length } acc \quad (q, a) \notin \text{set } (acc ! j) \quad (q, a) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc ! j)$
shows $i = q$
 $\langle \text{proof} \rangle$

lemma *in_actions_by_stateD*:
assumes
 $(q, g, a, t) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc ! j) \quad (q, g, a, t) \notin \text{set } (acc ! j)$
 $j < \text{length } acc$
shows
 $(g, a, t) \in \text{set } xs \wedge j = a$
 $\langle \text{proof} \rangle$

lemma *in_all_actions_by_stateD*:
assumes
 $(q, g1, a, r1) \in \text{set } (\text{all_actions_by_state } M \text{ } L ! a') \quad a' < na$
shows
 $(g1, a, r1) \in \text{set } (M !! q !! (L ! q)) \wedge q < p \wedge a' = a$
 $\langle \text{proof} \rangle$

lemma *length_all_actions_by_state_preserv*:
 $\text{length } (\text{all_actions_by_state } M \text{ } L) = na$
 $\langle \text{proof} \rangle$

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 $\text{pred_act } (\lambda a. a < na) \text{ } a$
 $\langle \text{proof} \rangle$

lemma *in_actions_trans_in_mapI*:
assumes
 $pa < p$
 $(g1, \text{In } a, r1) = \text{trans } ! pa ! (L ! pa) ! j$
 $L ! pa < \text{length } (\text{trans } ! pa)$

$j < \text{length } (\text{trans} ! \text{pa} ! (L ! \text{pa}))$
shows $(\text{pa}, g1, a, r1) \in \text{set } (\text{all_actions_by_state } (IArray (\text{map } IArray \text{trans_in_map}))) L !$
 a) $\langle \text{proof} \rangle$

lemma *in_actions_trans_out_mapI*:
assumes
 $\text{pa} < p$
 $(g1, \text{Out } a, r1) = \text{trans} ! \text{pa} ! (L ! \text{pa}) ! j$
 $L ! \text{pa} < \text{length } (\text{trans} ! \text{pa})$
 $j < \text{length } (\text{trans} ! \text{pa} ! (L ! \text{pa}))$
shows $(\text{pa}, g1, a, r1) \in \text{set } (\text{all_actions_by_state } (IArray (\text{map } IArray \text{trans_out_map}))) L$
 ! a) $\langle \text{proof} \rangle$

lemma *in_pairs_by_actionD2*:
assumes
 $(g, a, r, L', s') \in \text{set } (\text{pairs_by_action } (L, s) \text{ xs ys})$
 $\forall (q, g, a'', m, l) \in \text{set xs. } a'' = a'$
 $\forall (q, g, a'', m, l) \in \text{set ys. } a'' = a'$
shows $\text{check_pred } L' s'$
 $\langle \text{proof} \rangle$

lemma *in_pairs_by_actionD1*:
assumes
 $(g, a, r, L', s') \in \text{set } (\text{pairs_by_action } (L, s) \text{ xs ys})$
 $\forall (q, g, a'', m, l) \in \text{set xs. } a'' = a'$
 $\forall (q, g, a'', m, l) \in \text{set ys. } a'' = a'$
obtains
 $\text{pa } q \text{ pc_g1 pc_g2 } g1 \text{ g2 } r1 \text{ r2 pc_u1 pc_u2 } l1' \text{ l2'} \text{ s1}$
 $x1 \text{ x2 } x3 \text{ x4 } x5 \text{ y1 } y2 \text{ y3 } y4$
where
 $\text{pa} \neq q$
 $(\text{pa}, \text{pc_g1}, a, \text{pc_u1}, l1') \in \text{set ys}$
 $(q, \text{pc_g2}, a, \text{pc_u2}, l2') \in \text{set xs}$
 $L' = L[\text{pa} := l1', q := l2']$
 $\text{runf } \text{pc_u1 } s1 = \text{Some } ((x1, x2, s', x3, x4), x5)$
 $\text{runf } \text{pc_u2 } s = \text{Some } ((y1, y2, s1, y3, r2), y4)$
 $\text{check_g } \text{pc_g1 } s = \text{Some } g1 \text{ check_g } \text{pc_g2 } s = \text{Some } g2$
 $r1 = \text{make_reset } \text{pc_u1 } s$
 $g = g1 @ g2 \text{ r} = r1 @ r2$
 $\langle \text{proof} \rangle$

lemma *in_pairs_by_actionD*:
assumes
 $(g, a, r, L', s') \in \text{set } (\text{pairs_by_action } (L, s) \text{ xs ys})$
 $\forall (q, g, a'', m, l) \in \text{set xs. } a'' = a'$
 $\forall (q, g, a'', m, l) \in \text{set ys. } a'' = a'$
obtains
 $\text{pa } q \text{ pc_g1 pc_g2 } p1 \text{ p2 } g1 \text{ g2 } r1 \text{ r2 pc_u1 pc_u2 } l1' \text{ l2'} \text{ s1}$
 $x1 \text{ x2 } x3 \text{ x4 } x5 \text{ y1 } y2 \text{ y3 } y4$
where
 $\text{pa} \neq q$
 $(\text{pa}, \text{pc_g1}, a, \text{pc_u1}, l1') \in \text{set ys}$

$(q, pc_g2, a, pc_u2, l2') \in set\ xs$
 $L' = L[pa := l1', q := l2']$

$runf\ pc_u1\ s1 = Some\ ((x1, x2, s', x3, x4), x5)$
 $runf\ pc_u2\ s = Some\ ((y1, y2, s1, y3, r2), y4)$
 $check_g\ pc_g1\ s = Some\ g1\ check_g\ pc_g2\ s = Some\ g2$
 $r1 = make_reset\ pc_u1\ s$
 $g = g1\ @\ g2\ r = r1\ @\ r2$

$check_pred\ L'\ s'$

$\langle proof \rangle$

lemma *in_trans_funD*:

assumes $y \in set\ (trans_fun\ L)$

shows $y \in set\ (trans_s_fun\ L) \vee y \in set\ (trans_i_fun\ L)$

$\langle proof \rangle$

lemma *states'_states'[intro]*:

$L \in equiv.defs.states'\ s$ **if** $(L, s) \in states'$

$\langle proof \rangle$

lemma *bounded'_bounded*:

$bounded'\ s \longleftrightarrow bounded\ bounds\ s$ **if** $length\ s = length\ bounds$

$\langle proof \rangle$

lemma *bounded_bounded'*:

$bounded\ bounds\ s \implies bounded'\ s$

$\langle proof \rangle$

lemma *P_unfold*:

$(\forall q < p. (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \longleftrightarrow (check_pred\ L\ s)$ **if** $length\ s = length\ bounds$

$\langle proof \rangle$

lemma *P_unfold_1*:

$(\forall q < p. (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \implies (check_pred\ L\ s)$

$\langle proof \rangle$

lemma *equiv_PT[simp]*:

$equiv.PT = PT$

$\langle proof \rangle$

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 $(L\ !\ q, pc_g, a, pc_u, l') \in fst\ (equiv.N\ !\ q)$

$\langle proof \rangle$

lemma *trans_ND*:

assumes

```

    (L ! q, pc_g, a, pc_u, l') ∈ fst (equiv.N ! q)
    q < p
shows
    equiv.defs.N_s s ! q ⊢ L ! q
    → equiv.make_g pc_g s, (a, equiv.make_c pc_g, equiv.make_mf pc_u), equiv.make_f pc_u s l'
    ⟨proof⟩

lemma make_f_unfold:
    equiv.make_f pc s = make_reset pc s
    ⟨proof⟩

lemma make_g_simp:
    assumes check_g pc_g s = Some g1
    shows g1 = equiv.make_g pc_g s
    ⟨proof⟩

lemma make_c_simp:
    assumes check_g pc_g s = Some g1
    shows equiv.make_c pc_g s
    ⟨proof⟩

lemma make_reset_simp:
    assumes runf pc_u s = Some ((y1, y2, s1, y3, r2), y4)
    shows make_reset pc_u s = r2
    ⟨proof⟩

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
    ⟨proof⟩

lemma trans_fun_trans_of':
    (trans_fun, trans_of A) ∈ transition_rel states'
    ⟨proof⟩

lemma transition_rel_mono:
    (a, b) ∈ transition_rel B if (a, b) ∈ transition_rel C B ⊆ C
    ⟨proof⟩

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)))}
        ⟨proof⟩

end

context UPPAAL_Reachability_Problem_precompiled_defs

```

begin

lemma *N_s_state_indep*:

assumes $(L \vdash q, g, a, r, l') \in \text{map trans_of } (\text{equiv.defs.N_s } s) \vdash q < p$
obtains $g \ r$ **where** $(L \vdash q, g, a, r, l') \in \text{map trans_of } (\text{equiv.defs.N_s } s') \vdash q$
 $\langle \text{proof} \rangle$

lemma *fst_product_state_indep*:

$\text{fst } ' \text{fst } (\text{equiv.defs.product } s) = \text{fst } ' \text{fst } (\text{equiv.defs.product } s')$
 $\langle \text{proof} \rangle$

lemma *last_product_state_indep*:

$(\text{snd } o \text{ snd } o \text{ snd } o \text{ snd}) \ ' \text{fst } (\text{equiv.defs.product } s)$
 $= (\text{snd } o \text{ snd } o \text{ snd } o \text{ snd}) \ ' \text{fst } (\text{equiv.defs.product } s')$
 $\langle \text{proof} \rangle$

lemma *state_set_T'*:

$\text{state_set } (\text{equiv.defs.T' } s'') \supseteq \text{fst } ' \text{state_set } (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma *state_set_T'2*:

$\text{length } L = \text{equiv.defs.p}$
 $\forall q < \text{equiv.defs.p}. L \vdash q \in \text{State_Networks.state_set } (\text{fst } (\text{equiv.defs.N } \vdash q))$
if $(L, s) \in \text{state_set } (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma *state_set_states'*:

$L \in \text{equiv.defs.states' } s$ **if** $(L, s) \in \text{state_set } (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma *state_set_pred*:

$\forall q < p. (\text{equiv.defs.P } \vdash q) (L \vdash q) \ s$ **if** $(L, s) \in \text{state_set } (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

end

context *UPPAAL_Reachability_Problem_precompiled'*

begin

lemma *bounded_bounded''*:

$\text{bounded bounds } s \implies \text{length } s = \text{length bounds}$
 $\langle \text{proof} \rangle$

lemma *P_bounded*:

$(\forall q < p. (\text{equiv.defs.P } \vdash q) (L \vdash q) \ s) \implies \text{bounded bounds } s$
 $\langle \text{proof} \rangle$

lemma *P_state_length*:

$(\forall q < p. (\text{equiv.defs.P } \vdash q) (L \vdash q) \ s) \implies \text{length } s = \text{length bounds}$
 $\langle \text{proof} \rangle$

lemma *state_set_state_length*:

$\text{length } s = \text{length bounds}$ **if** $(L, s) \in \text{state_set } (\text{trans_of } A)$

$\langle proof \rangle$

lemma *state_set_states*:
 $state_set\ (trans_of\ A) \subseteq states'$
 $\langle proof \rangle$

lemma *p_p_2[simp]*:
 $defs'.defs.p = p$
 $\langle proof \rangle$

lemma *len_product'_N[simp]*:
 $length\ defs'.defs.N = p$
 $\langle proof \rangle$

lemma *len_equiv_N*:
 $length\ equiv.defs.N = p$
 $\langle proof \rangle$

lemma
 $defs'.p = p$
 $\langle proof \rangle$

lemma *equiv_p_p*: $equiv.p = p$
 $\langle proof \rangle$

lemma *init_states*:
 $init \in equiv.defs.states'\ s_0$
 $\langle proof \rangle$

lemma *start_pred'*:
 $check_pred\ init\ s_0$
 $\langle proof \rangle$

lemma *start_states'*:
 $(init, s_0) \in states'$
 $\langle proof \rangle$

lemma *trans_fun_trans_of[intro, simp]*:
 $(trans_fun, trans_of\ A) \in transition_rel\ states$
 $\langle proof \rangle$

definition
 $inv_fun \equiv \lambda\ (L, _). concat\ (map\ (\lambda\ i. IArray\ (map\ IArray\ inv)\ !!\ i\ !!\ (L\ !\ i))\ [0..<p])$

lemma *states_states'*:
 $states \subseteq states'$
 $\langle proof \rangle$

lemma *states'_length[simp]*:
 $length\ L = p\ \text{if}\ (L, s) \in states'$
 $\langle proof \rangle$

lemma *inv_simp*:

$I\ q\ (L\ !\ q) = inv\ !\ q\ !\ (L\ !\ q)\ \text{if}\ q < p\ (L, s) \in states'$
 $\langle proof \rangle$

lemma *inv_fun_inv_of'*:
 $(inv_fun, inv_of\ A) \in inv_rel\ Id\ states'$
 $\langle proof \rangle$

lemma *inv_rel_mono*:
 $(a, b) \in inv_rel\ Id\ B\ \text{if}\ (a, b) \in inv_rel\ Id\ C\ B \subseteq C$
 $\langle proof \rangle$

lemma *inv_fun_inv_of[intro, simp]*:
 $(inv_fun, inv_of\ A) \in inv_rel\ Id\ states$
 $\langle proof \rangle$

definition *final_fun* $\equiv \lambda\ (L, s). hd_of_formula\ formula\ L\ s$

lemma *final_fun_final'*:
 $(final_fun, (\lambda\ (l, s). F\ l\ s)) \in inv_rel\ Id\ states'$
 $\langle proof \rangle$

lemma *final_fun_final[intro, simp]*:
 $(final_fun, (\lambda\ (l, s). F\ l\ s)) \in inv_rel\ Id\ states$
 $\langle proof \rangle$

lemma *fst_clkp_setD*:
assumes $(c, d) \in clkp_set\ A\ l$
shows $c > 0\ c \leq m\ d \in range\ int$
 $\langle proof \rangle$

lemma *init_has_trans*:
 $(init, s_0) \in fst\ ' (trans_of\ A) \longleftrightarrow trans_fun\ (init, s_0) \neq []$
 $\langle proof \rangle$

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

abbreviation $k_i \equiv IArray\ (map\ (IArray\ o\ (map\ (IArray\ o\ map\ int)))\ k)$

definition
 $k_impl \equiv \lambda\ (l, _). IArray\ (map\ (\lambda\ c. Max\ \{k_i\ !!\ i\ !!\ (l\ !\ i)\ !!\ c\ |\ i. i < p\})\ [0..<m+1])$

lemma *k_impl_alt_def*:
 $k_impl =$
 $(\lambda\ (l, _). IArray\ (map\ (\lambda\ c. Max\ ((\lambda\ i. k_i\ !!\ i\ !!\ (l\ !\ i)\ !!\ c)\ ' \{0..<p\}))\ [0..<m+1]))$
 $\langle proof \rangle$

lemma *k_length_alt*:
 $\forall\ i < p. \forall\ j < length\ (k\ !\ i). length\ (k\ !\ i\ !\ j) = m + 1$
 $\langle proof \rangle$

lemma *Max_int_commute*:

$\text{int } (\text{Max } S) = \text{Max } (\text{int } ' S) \text{ if finite } S \text{ } S \neq \{\}$
 $\langle \text{proof} \rangle$

lemma $k_impl_k'[intro]$:
 $k_impl (l, s) = IArray (k' (l, s)) \text{ if } (l, s) \in \text{states}'$
 $\langle \text{proof} \rangle$

lemma $k_impl_k'2[intro]$:
 $k_impl (l, s) = IArray (k' (l, s)) \text{ if}$
 $(l, s) \in \text{Normalized_Zone_Semantics_Impl_Refine.state_set } (\text{trans_of } A)$
 $\langle \text{proof} \rangle$

lemma $k_impl_k'_0[intro]$:
 $k_impl (\text{init}, s_0) = IArray (k' (\text{init}, s_0))$
 $\langle \text{proof} \rangle$

sublocale $impl$:
 $\text{Reachability_Problem_Impl}$
where $\text{trans_fun} = \text{trans_fun}$
and $\text{trans_impl} = \text{trans_fun}$
and $\text{inv_fun} = \text{inv_fun}$
and $F_fun = \text{final_fun}$
and $\text{ceiling} = k_impl$
and $A = A$
and $l_0 = (\text{init}, s_0)$
and $l_0 i = (\text{init}, s_0)$
and $F = \text{PR_CONST } ((\lambda (l, s). F \ l \ s))$
and $n = m$
and $k = k_fun$
and $\text{loc_rel} = \text{Id}$
and $\text{show_clock} = \text{show}$
and $\text{show_state} = \text{show}$
and $\text{states}' = \text{states}$
 $\langle \text{proof} \rangle$

lemma $F_reachable_correct'$:
 $impl.op.F_reachable$
 $\longleftrightarrow (\exists L' s' u u'.$
 $\quad \text{conv_A } A \vdash' \langle (\text{init}, s_0), u \rangle \rightarrow^* \langle (L', s'), u \rangle$
 $\quad \wedge (\forall c \in \{1..m\}. u \ c = 0) \wedge \text{check_bexp } \varphi \ L' s'$
 $\quad) \text{ if formula} = \text{formula.EX } \varphi$
 $\langle \text{proof} \rangle$

lemma PT_PT :
 $\text{defs'.PT} = \text{equiv.PT}$
 $\langle \text{proof} \rangle$

lemma $P_P[simp]$:
 $\text{defs'.defs.P} = \text{equiv.defs.P}$
 $\langle \text{proof} \rangle$

lemma map_map_filter :
 $\text{map } f \ (\text{List.map_filter } g \ xs) = \text{List.map_filter } (\text{map_option } f \ o \ g) \ xs$
 $\langle \text{proof} \rangle$

lemma *make_g_conv*:

defs'.make_g = conv_cc oo equiv.make_g
⟨proof⟩

lemma *make_c_conv*:

defs'.make_c = equiv.make_c
⟨proof⟩

lemma *make_f_conv*:

defs'.make_f = equiv.make_f
⟨proof⟩

lemma *make_mf_conv*:

defs'.make_mf = equiv.make_mf
⟨proof⟩

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⟩
⟨proof⟩

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⟩*
⟨proof⟩

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)
⟨proof⟩

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)
⟨proof⟩

lemma *prod_trans_s_conv*:

defs'.defs.prod_trans_s = conv_t ' equiv.defs.prod_trans_s
⟨proof⟩

lemma *prod_trans_i_conv*:

defs'.defs.prod_trans_i = conv_t ' equiv.defs.prod_trans_i
⟨proof⟩

lemma *prod_trans_conv*:

defs'.defs.prod_trans = conv_t ' equiv.defs.prod_trans
⟨proof⟩

lemma *prod_invariant_conv*:

defs'.defs.prod_invariant = (map conv_ac oo Prod_TA_Defs.prod_invariant) EA
⟨proof⟩

lemma *prod_conv*: *defs'.defs.prod_ta = conv_A A*
<proof>

lemma *F_reachable_correct*:
impl.op.F_reachable
 $\longleftrightarrow (\exists L' s' u u'. \\
\text{conv } N \vdash_m \text{ax_steps } \langle \text{init}, s_0, u \rangle \rightarrow^* \langle L', s', u' \rangle \\
\wedge (\forall c \in \{1..m\}. u \text{ c} = 0) \wedge \text{check_bexp } \varphi L' s' \\
) \text{ if } \text{formula} = \text{formula.EX } \varphi \text{ start_inv_check}$
<proof>

definition

reachability_checker_old $\equiv$
worklist_algo2_impl
impl.subsumes_impl impl.a0_impl impl.F_impl impl.succs_impl impl.emptiness_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.emptiness_check_impl

theorem *reachability_check'*:

(uncurry0 reachability_checker',
uncurry0 (
*Refine_Basic.RETURN (\exists L' s' u u'. *
conv_A A \vdash' \langle (init, s_0), u \rangle \rightarrow^ \langle (L', s'), u' \rangle*
\wedge (\forall c \in \{1..m\}. u \text{ c} = 0) \wedge \text{check_bexp } \varphi L' s'
)
)
)
 $\in \text{unit_assn}^k \rightarrow_a \text{bool_assn}$ **if** *formula = formula.EX \varphi*
<proof>

corollary *reachability_checker'_hoare*:

<emp> reachability_checker'
 $\langle \lambda r. \uparrow(r = (\exists L' s' u u'. \\
\text{conv}_A A \vdash' \langle (init, s_0), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \\
\wedge (\forall c \in \{1..m\}. u \text{ c} = 0) \wedge \text{check_bexp } \varphi L' s' \\
)) \\
>_t \text{ if } \text{formula} = \text{formula.EX } \varphi$
<proof>

definition *reachability_checker* **where**

reachability_checker $\equiv$ *do*
{
init_sat $\leftarrow$ *impl.start_inv_check_impl*;
if init_sat then do
{ x $\leftarrow$ *reachability_checker'*;
return (if x then REACHABLE else UNREACHABLE)
}
else
return INIT_INV_ERR

}

theorem *reachability_check*:

```
(uncurry0 reachability_checker,
  uncurry0 (
    Refine_Basic.RETURN (
      if start_inv_check
      then
        if
          (
            ∃ L' s' u u'.
              conv N ⊢m ax_steps ⟨init, s0, u⟩ →* ⟨L', s', u'⟩
              ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
          )
        then REACHABLE
        else UNREACHABLE
      else INIT_INV_ERR
    )
  ))
∈ unit_assnk →a id_assn if formula = formula.EX φ
⟨proof⟩
```

corollary *reachability_checker_hoare*:

```
<emp> reachability_checker
<λ r. ↑(r =
  (
    if start_inv_check
    then
      if
        (
          ∃ L' s' u u'.
            conv N ⊢m ax_steps ⟨init, s0, u⟩ →* ⟨L', s', u'⟩
            ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
        )
      then REACHABLE
      else UNREACHABLE
    else INIT_INV_ERR
  )
)>t if formula = formula.EX φ
⟨proof⟩
```

lemma *list_all_concat*:

```
list_all Q (concat xs) ⟷ (∀ xs ∈ set xs. list_all Q xs)
⟨proof⟩
```

lemma *inv_of_init_unfold*:

```
u ⊢ inv_of (conv_A A) (init, s0) ⟷ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0))
⟨proof⟩
```

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 ⊢m ax_steps ⟨init, s0, u⟩ →* ⟨L', s', u'⟩
            ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
        )
      then REACHABLE
      else UNREACHABLE
    else INIT_INV_ERR
  )
)
>t if formula = formula.EX φ
⟨proof⟩

```

Post-processing **schematic_goal** *succs_impl_alt_def*:

```

impl.succs_impl ≡ ?impl
⟨proof⟩

```

lemma *reachability_checker'_alt_def*:

```

reachability_checker' ≡
  let
    key = return ∘ fst;
    sub = impl.subsumes_impl;
    copy = impl.state_copy_impl;
    start = impl.a0_impl;
    final = impl.F_impl;
    succs = impl.succs_impl;
    empty = impl.emptiness_check_impl;
    trace = impl.tracei
  in pw_impl key copy trace sub start final succs empty
⟨proof⟩

```

schematic_goal *reachability_checker_alt_def*:

```

reachability_checker ≡ ?impl
⟨proof⟩

```

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 ⇒ int ⇒ int **where**
plus_int a b ≡ a + b

fun *dbm_add_int* :: int DBMEntry ⇒ int DBMEntry ⇒ int DBMEntry
where
dbm_add_int ∞ _ = ∞ |
dbm_add_int _ ∞ = ∞ |

$dbm_add_int (Le\ a) (Le\ b) = (Le\ (plus_int\ a\ b)) \mid$
 $dbm_add_int (Le\ a) (Lt\ b) = (Lt\ (plus_int\ a\ b)) \mid$
 $dbm_add_int (Lt\ a) (Le\ b) = (Lt\ (plus_int\ a\ b)) \mid$
 $dbm_add_int (Lt\ a) (Lt\ b) = (Lt\ (plus_int\ a\ b))$

lemma *dbm_add_int*:
 $dbm_add = dbm_add_int$
 $\langle proof \rangle$

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
 $fw_upd_impl_int\ n \equiv \lambda ai\ bib\ bia\ bi. do \{$
 $xa \leftarrow mtx_get\ (Suc\ n)\ ai\ (bia, bib);$
 $xb \leftarrow mtx_get\ (Suc\ n)\ ai\ (bib, bi);$
 $x \leftarrow mtx_get\ (Suc\ n)\ ai\ (bia, bi);$
 $let\ e = (dbm_add_int\ xa\ xb);$
 $if\ e < x\ then\ mtx_set\ (Suc\ n)\ ai\ (bia, bi)\ e\ else\ Heap_Monad.return\ ai$
 $\}$

lemma *fw_upd_impl_int_eq*:
 $fw_upd_impl_int = fw_upd_impl$
 $\langle proof \rangle$

definition
 $fw_impl_int\ n \equiv$
 $imp_for'\ 0\ (n + 1)$
 $(\lambda xb. imp_for'\ 0\ (n + 1)$
 $(\lambda xd. imp_for'\ 0\ (n + 1)\ (\lambda xf\ \sigma'''''. fw_upd_impl_int\ n\ \sigma'''''\ xb\ xd\ xf)))$

lemma *fw_impl'_int*:
 $fw_impl = fw_impl_int$
 $\langle proof \rangle$

context *UPPAAL_Reachability_Problem_precompiled_defs'*
begin

definition *run_impl* $program\ pc\ s \equiv exec\ program\ max_steps\ (pc, [], s, True, []) []$

lemma *runf_impl*:
 $runf = run_impl\ PF$
 $\langle proof \rangle$

lemma *runt_impl*:
 $runt = run_impl\ PT$
 $\langle proof \rangle$

definition
 $make_cconstr_impl\ program\ pcs =$

List.map_filter
 ($\lambda pc. \text{case program } pc \text{ of } None \Rightarrow None \mid \text{Some } (INSTR\ x) \Rightarrow \text{Map.empty } x$
 $\mid \text{Some } (CEXP\ ac) \Rightarrow \text{Some } ac$)
pcs

lemma *make_constr_impl*:
make_constr = *make_constr_impl* *PROG*
 $\langle \text{proof} \rangle$

definition
check_g_impl *programf* *program* *pc* *s* $\equiv$
case run_impl *programf* *pc* *s* of *None* $\Rightarrow$ *None*
 $\mid \text{Some } ((x, xa, xb, \text{True}, xc), pcs) \Rightarrow \text{Some } (\text{make_constr_impl } \text{program } pcs)$
 $\mid \text{Some } ((x, xa, xb, \text{False}, xc), pcs) \Rightarrow \text{None}$

lemma *check_g_impl*:
check_g = *check_g_impl* *PT* *PROG'*
 $\langle \text{proof} \rangle$

definition
make_reset_impl *program* *m1* *s* $\equiv$
case run_impl *program* *m1* *s* of
Some $((_, _, _, _, r1), _)$ $\Rightarrow$ *r1*
 $\mid \text{None} \Rightarrow []$

lemma *make_reset_impl*:
make_reset = *make_reset_impl* *PF*
 $\langle \text{proof} \rangle$

definition
check_pred_impl *program* *bnds* *L* *s* $\equiv$
list_all
 $(\lambda q. \text{case run_impl } \text{program } (\text{pred } ! q ! (L ! q)) \text{ of } \text{None} \Rightarrow \text{False}$
 $\mid \text{Some } ((x, xa, xb, f, xc), xd) \Rightarrow$
 $f \wedge (\forall i < \text{length } s. \text{fst } (bnds !! i) < s ! i \wedge s ! i < \text{snd } (bnds !! i)))$
 $[0..<p]$

lemma *check_pred_impl*:
check_pred = *check_pred_impl* *PF* (*IArray* *bounds*)
 $\langle \text{proof} \rangle$

definition
pairs_by_action_impl *pf* *pt* *porig* *bnds* $\equiv$ $\lambda (L, s) \text{ OUT. concat o}$
 $\text{map } (\lambda (i, g1, a, m1, l1). \text{List.map_filter}$
 $(\lambda (j, g2, a, m2, l2).$
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 ((_, _, s', _, _), _) =>
      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
  | _ => None)
| _ => None)
| _ => None
)
OUT)

```

lemma *pairs_by_action_impl*:

pairs_by_action = *pairs_by_action_impl* PF PT PROG' (IArray bounds)
 <proof>

definition

all_actions_by_state_impl *upt_p* *empty_ran* *i* *L* ≡
 fold (λia. *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)
 <proof>

definition

trans_i_from_impl *programf* *programt* *program* *bnds* *trans_i_array* ≡
 λ(L, s) i.
 List.map_filter
 (λ(g, a, m, l').
 case check_g_impl *programt* *program* g s of None => None
 | Some cc =>
 (case run_impl *programf* m s of None => None
 | Some ((xx, xa, s', xb, r), xc) =>
 if check_pred_impl *programf* (IArray bounds) (L[i := l']) s'
 then Some (cc, a, r, L[i := l'], s') else None))
 (trans_i_array !! i !! (L ! i))

lemma *trans_i_from_impl*:

trans_i_from = *trans_i_from_impl* PF PT PROG' (IArray bounds) (IArray (map IArray
trans_i_map))
 <proof>

end

context UPPAAL_Reachability_Problem_precompiled'

begin

lemma *PF_alt_def*:

PF = (λ pc. if pc < length prog then (IArray (map (map_option stripf) prog)) !! pc else None)
 <proof>

lemma *PT_alt_def*:

PT = (λ pc. if pc < length prog then (IArray (map (map_option stript) prog)) !! pc else None)
 <proof>

schematic_goal *reachability_checker_alt_def_refined*:

$reachability_checker \equiv ?impl$
 $\langle proof \rangle$

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*:

$check_nat_subs \longleftrightarrow snd \text{ ' } clkp_set' \subseteq \mathbb{N}$
 $\langle proof \rangle$

definition

$check_resets \equiv \forall x \ c. \text{ Some } (INSTR (STOREC \ c \ x)) \in set \ prog \longrightarrow x = 0$

lemma *check_resets_alt_def*:

$check_resets =$
 $(\forall (c, x) \in collect_store. x = 0)$
 $\langle proof \rangle$

definition

$check_pre \equiv$
 $length \ inv = p \wedge length \ trans = p \wedge length \ pred = p$
 $\wedge (\forall i < p. length \ (pred \ ! \ i) = length \ (trans \ ! \ i) \wedge length \ (inv \ ! \ i) = length \ (trans \ ! \ i))$
 $\wedge (\forall T \in set \ trans. \forall xs \in set \ T. \forall (_, _, _, l) \in set \ xs. l < length \ T)$
 $\wedge p > 0 \wedge m > 0$
 $\wedge (\forall i < p. trans \ ! \ i \neq []) \wedge (\forall q < p. trans \ ! \ q \neq [])$
 $\wedge check_nat_subs \wedge clk_set' = \{1..m\}$
 $\wedge check_resets$

lemma *finite_clkp_set'*[*intro, simp*]:

$finite \ clkp_set'$
 $\langle proof \rangle$

lemma *check_pre*:

$UPPAAL_Reachability_Problem_precompiled \ p \ m \ inv \ pred \ trans \ prog \longleftrightarrow check_pre$
 $\langle proof \rangle$

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
  <proof>

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 <proof>

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))))
⟨proof⟩

lemmas [code] =
UPPAAL_Reachability_Problem_precompiled'_axioms_def
UPPAAL_Reachability_Problem_precompiled'_def
pred_act_def

definition

init_pred_check $\equiv \lambda p \text{ prog max_steps pred } s_0.$
 $(\forall q < p.$
case (exec
(stripp (UPPAAL_Reachability_Problem_precompiled_defs.PROG prog))
max_steps
((pred ! q ! (UPPAAL_Reachability_Problem_precompiled_defs.init p ! q)), [], s₀, True,
 $\square)$
 $\square)$
of Some ((pc, st, s', True, rs), pcs) ⇒ True | _ ⇒ False)

definition

time_indep_check1 $\equiv \lambda \text{pred prog max_steps.}$
 $(\forall x \in \text{set pred. } \forall pc \in \text{set } x. \text{time_indep_check prog pc max_steps})$

definition

time_indep_check2 $\equiv \lambda \text{trans prog max_steps.}$
 $(\forall T \in \text{set trans. } \forall xs \in \text{set } T. \forall (_, _, pc_u, _) \in \text{set } xs. \text{time_indep_check prog pc_u max_steps})$

definition

conjunction_check2 $\equiv \lambda \text{trans prog max_steps.}$
 $(\forall T \in \text{set trans. } \forall xs \in \text{set } T. \forall (pc_g, _, _, _) \in \text{set } xs. \text{conjunction_check prog pc_g max_steps})$

lemma *start_pred*[code]:

UPPAAL_Reachability_Problem_precompiled_start_state_axioms = $(\lambda p \text{ max_steps trans prog}$
bounds pred } s_0.
init_pred_check p prog max_steps pred } s_0
 $\wedge$ *bounded bounds } s_0*
 $\wedge$ *time_indep_check1 pred prog max_steps*
 $\wedge$ *time_indep_check2 trans prog max_steps*

```

 $\wedge$  conjunction_check2 trans prog max_steps
)
⟨proof⟩

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'  $\Rightarrow$ 
       $\bigcup$  (
        ( $\lambda$  cmd. case cmd of Some (INSTR (STOREC c x))  $\Rightarrow$  {(c, x)} | _  $\Rightarrow$  {}) ‘
        (!) prog) ‘ {pc .. pc'}
      )
  ⟨proof⟩

lemma collect_cexp'_alt_def:
  collect_cexp' pc  $\equiv$ 
   $\bigcup$  (( $\lambda$  cmd. case cmd of Some (CEXP ac)  $\Rightarrow$  {ac} | _  $\Rightarrow$  {})) ‘
  (!) prog) ‘ steps_approx max_steps prog pc
  ⟨proof⟩

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_cconstr_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*
 $\gg (\lambda x. \text{return } (\text{Some } x))$
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 ($\forall u. (\forall c \in \{1..m\}. u \ c = 0) \longrightarrow (\forall i < p. u \vdash \text{conv_cc } (I \ ! \ i \ ! \ 0))$)
then
if
(
 $\exists L' \ s' \ u \ u'.$
 $\text{conv } (N \ p \ I \ P \ T \ \text{prog} \ \text{bounds}) \vdash_m \text{max_steps}$
 $\langle \text{repeat } 0 \ p, \ s_0, \ u \rangle \rightarrow^* \langle L', \ s', \ u' \rangle$
 $\wedge (\forall c \in \{1..m\}. u \ c = 0) \wedge \text{check_bexp } \text{formula } L' \ s'$
)
then REACHABLE
else UNREACHABLE
else INIT_INV_ERR
)
else None
)
)
)
 $\in \text{unit_assn}^k \rightarrow_a \text{id_assn}$
<proof>)

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* $\vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$
 $\langle proof \rangle$

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* $\vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq Del$)
 $\langle proof \rangle$

lemma *prod_correct'_delay*:

($\exists$ *d. defs.prod_ta* $\vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$) =
 $state_ta \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *equiv_correct*:

$state_ta \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle = A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *prod_correct_action*:

($\exists$ *a. defs.prod_ta* $\vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$) =
($\exists$ *a. A* $\vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \wedge a \neq State_Networks.label.Del$)
 $\langle proof \rangle$

lemma *prod_correct_delay*:

($\exists$ *d. defs.prod_ta* $\vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$) =
A $\vdash_n \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *prod_correct*:

$defs.prod_ta \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle =$
 $(\exists a. A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle)$
 $\langle proof \rangle$

context

assumes $0 < p$

begin

lemmas $equiv_complete'' = equiv_complete''[OF_ \langle 0 < p \rangle]$

definition

$all_prop L' s' \equiv$
 $(\forall q < p. \exists pc\ st\ s''\ rs\ pcs.$
 $exec\ PF\ n\ ((I\ !\ q)\ (L'\ !\ q), [], s', True, [])\ [] =$
 $Some\ ((pc,\ st,\ s'',\ True,\ rs), pcs)$
 $) \wedge bounded\ B\ s' \wedge L' \in defs.states'\ s' \wedge q < p \wedge pc \in defs.P\ q \wedge L' \in L\ s'$

lemma $step_u_inv$:

$all_prop L' s' \text{ if } A \vdash_n \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma $step_inv$:

$all_prop L' s' \text{ if } state_ta \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma $Equiv_TA_I$:

$Equiv_TA\ A\ n\ L'\ s' \text{ if } *[unfolded\ all_prop_def]: all_prop L' s'$
 $\langle proof \rangle$

lemma $step_u'_inv$:

$all_prop L'' s'' \wedge defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L'', s''), u'' \rangle$
 $\text{if } A \vdash^n \langle L, s, u \rangle \rightarrow \langle L'', s'', u'' \rangle$
 $\langle proof \rangle$

lemma $step'_inv$:

$all_prop L'' s'' \wedge A \vdash^n \langle L, s, u \rangle \rightarrow \langle L'', s'', u'' \rangle$
 $\text{if } defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L'', s''), u'' \rangle$
 $\langle proof \rangle$

lemma $prod_correct_step'$:

$defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle =$
 $A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma all_prop_start :

$all_prop L\ s$
 $\langle proof \rangle$

lemma $steps_u'_inv$:

$all_prop L'' s'' \wedge defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u'' \rangle$
 $\text{if } A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u'' \rangle$
 $\langle proof \rangle$

lemma *steps'_inv*:
 $all_prop\ L''\ s'' \wedge A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u' \rangle$
if $defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u' \rangle$
 $\langle proof \rangle$

lemma *steps_un'_complete*:
 $defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u' \rangle$
if $A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u' \rangle$
 $\langle proof \rangle$

lemma *steps'_sound*:
 $A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L'', s'', u' \rangle$
if $defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L'', s''), u' \rangle$
 $\langle proof \rangle$

lemma *prod_reachable_correct*:
 $defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \longleftrightarrow A \vdash^n \langle L, s, u \rangle \rightarrow^* \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *Bisimulation_Invariant_I*:
Bisimulation_Invariant
 $(\lambda (L, s, u) (L', s', u').\ defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$
 $(\lambda (L, s, u) (L', s', u').\ A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(=)$
 $(\lambda (L, s, u).\ all_prop\ L\ s)$
 $(\lambda (L, s, u).\ all_prop\ L\ s)$
 $\langle proof \rangle$

interpretation *Bisimulation_Invariant*
 $\lambda (L, s, u) (L', s', u').\ defs.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$
 $\lambda (L, s, u) (L', s', u').\ A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$
 $(=)$
 $\lambda (L, s, u).\ all_prop\ L\ s$
 $\lambda (L, s, u).\ all_prop\ L\ s$
 $\langle proof \rangle$

end

end

definition *models* $(_, _ \models _ _ [61, 61]\ 61)$ **where**
 $A, a_0 \models_n \Phi \equiv (case\ \Phi\ of$
 $formula.EX\ \varphi \Rightarrow$
 $Graph_Defs.Ex_ev$
 $(\lambda (L, s, u) (L', s', u').\ A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(\lambda (L, s, _).\ check_bexp\ \varphi\ L\ s)$
 $| formula.EG\ \varphi \Rightarrow$
 $Graph_Defs.Ex_alw$
 $(\lambda (L, s, u) (L', s', u').\ A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(\lambda (L, s, _).\ check_bexp\ \varphi\ L\ s)$
 $| formula.AX\ \varphi \Rightarrow$
 $Graph_Defs.Alw_ev$
 $(\lambda (L, s, u) (L', s', u').\ A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(\lambda (L, s, _).\ check_bexp\ \varphi\ L\ s)$

```

| formula.AG  $\varphi \Rightarrow$ 
  Graph_Defs.Alw_alw
  ( $\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ )
  ( $\lambda (L, s, \_). \text{check\_bexp } \varphi L s$ )
| formula.Leadsto  $\varphi \psi \Rightarrow$ 
  Graph_Defs.leadsto
  ( $\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ )
  ( $\lambda (L, s, \_). \text{check\_bexp } \varphi L s$ )
  ( $\lambda (L, s, \_). \text{check\_bexp } \psi L s$ )
) a0

```

definition

```

has_deadlock A n a0  $\equiv$ 
  Graph_Defs.deadlock ( $\lambda (L, s, u) (L', s', u'). A \vdash^n \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle$ ) a0

```

lemmas *models_iff* = *models_def*[*unfolded* Graph_Defs.Ex_alw_iff Graph_Defs.Alw_alw_iff]

context *Reachability_Problem*

begin

lemma *reaches_steps'*:

```

reaches (l, u) (l', u')  $\longleftrightarrow \text{conv\_A } A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$ 
<proof>

```

lemma *clocks_I*:

```

( $\forall c. c \in \text{clk\_set } (\text{conv\_A } A) \longrightarrow u c = u' c$ ) if  $\forall c \in \{1..n\}. u c = u' c$ 
<proof>

```

lemma *init_dbm_reaches_iff*:

```

( $\exists u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle$ )
 $\longleftrightarrow ([\text{curry } (\text{init\_dbm} :: \text{real DBM}')]_{v,n} \neq \{\}) \wedge$ 
  ( $\forall u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle$ )

```

<proof>

theorem *reachable_decides_emptiness_new*:

```

( $\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry } (\text{conv\_M } D')]_{v,n} \neq \{\}$ )
 $\longleftrightarrow [\text{curry } (\text{init\_dbm} :: \text{real DBM}')]_{v,n} \neq \{\} \wedge$ 
  ( $\forall u \in [\text{curry init\_dbm}]_{v,n}. \exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle$ )
<proof>

```

lemma *reachable_decides_emptiness'_new*:

```

( $\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry } (\text{conv\_M } D')]_{v,n} \neq \{\}$ )
 $\longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle))$ 
<proof>

```

lemma *reachability_check_new_aux*:

```

( $\exists D'. E^{**} a_0 (l', D') \wedge [\text{curry } (\text{conv\_M } D')]_{v,n} \neq \{\} \wedge F l'$ )
 $\longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u c = 0) \longrightarrow (\exists u'. \text{conv\_A } A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle \wedge F l'))$ 
<proof>

```

theorem *reachability_check_new*:

```

( $\exists D'. E^{**} a_0 (l', D') \wedge F\_rel (l', D')$ )

```

$\longleftrightarrow (\forall u. (\forall c \in \{1..n\}. u\ c = 0) \longrightarrow (\exists u'. \text{conv_}A\ A \vdash' \langle l_0, u \rangle \rightarrow^* \langle l', u' \rangle \wedge F\ l'))$
 $\langle \text{proof} \rangle$

lemma *init_state_in_state_set*:
 $l_0 \in \text{state_set} (\text{trans_of}\ A) \text{ if } \neg \text{deadlock}\ (l_0, u_0)$
 $\langle \text{proof} \rangle$

lemma *init_state_in_state_set'*:
 $l_0 \in \text{state_set} (\text{trans_of}\ A) \text{ if } (\forall u_0. (\forall c \in \{1..n\}. u_0\ c = 0) \longrightarrow \neg \text{deadlock}\ (l_0, u_0))$
 $\langle \text{proof} \rangle$

end

context *Reachability_Problem_Impl*
begin

context
fixes $Q :: 's \Rightarrow \text{bool}$ **and** Q_fun
assumes Q_fun : $(Q_fun, Q) \in \text{inv_rel}\ \text{loc_rel}\ \text{states}'$
begin

lemma *leadsto_spec_refine*:
 $\text{leadsto_spec_alt}\ Q$
 $\leq \text{SPEC}\ (\lambda r. \neg r \longleftrightarrow$
 $(\nexists x. (\lambda a\ b. E_op''. E_from_op\ a\ b \wedge \neg \text{check_diag}\ n\ (\text{snd}\ b))^{**}\ (l_0, \text{init_dbm})\ x \wedge$
 $F\ (\text{fst}\ x) \wedge Q\ (\text{fst}\ x) \wedge$
 $(\exists a. (\lambda a\ b. E_op''. E_from_op\ a\ b \wedge \neg \text{check_diag}\ n\ (\text{snd}\ b) \wedge Q\ (\text{fst}\ b))^{**}\ x\ a \wedge$
 $(\lambda a\ b. E_op''. E_from_op\ a\ b \wedge \neg \text{check_diag}\ n\ (\text{snd}\ b) \wedge Q\ (\text{fst}\ b))^{++}\ a\ a)$
 $))$
 $\langle \text{proof} \rangle$

lemma *leadsto_impl_hnr'*:
 $(\text{uncurry0}$
 $(\text{leadsto_impl}\ \text{state_copy_impl}$
 $(\text{succs_P_impl}'\ Q_fun)\ a_0_impl\ \text{subsumes_impl}\ (\text{return} \circ \text{fst})$
 $\text{succs_impl}'\ \text{emptiness_check_impl}\ F_impl\ (Q_impl\ Q_fun)\ \text{tracei}),$
 uncurry0
 $(\text{SPEC}$
 $(\lambda r. (\forall u_0. (\forall c \in \{1..n\}. u_0\ c = 0) \longrightarrow \neg \text{deadlock}\ (l_0, u_0)) \longrightarrow$
 $(\neg r) =$
 $(\forall u_0. (\forall c \in \{1..n\}. u_0\ c = 0) \longrightarrow$
 $\text{leadsto}\ (\lambda(l, u). F\ l)\ (\lambda(l, u). \neg Q\ l)\ (l_0, u_0))))$
 $\in \text{unit_assn}^k \rightarrow_a \text{bool_assn}$
 $\langle \text{proof} \rangle$

end

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *F_reachable_correct'_new*:
 $\text{impl.op.F_reachable}$

$\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv_A } A \vdash' \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \text{check_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.EX } \varphi \\
\langle \text{proof} \rangle$

lemma *F_reachable_correct'_new'*:

impl.op.F_reachable
 $\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv_A } A \vdash' \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \neg \text{check_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.AG } \varphi \\
\langle \text{proof} \rangle$

lemma *F_reachable_correct_new*:

impl.op.F_reachable
 $\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv } N \vdash^m \text{ax_steps } \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \text{check_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.EX } \varphi \\
\langle \text{proof} \rangle$

lemma *F_reachable_correct_new'*:

impl.op.F_reachable
 $\longleftrightarrow (\exists L' s'. \forall u. (\forall c \in \{1..m\}. u\ c = 0) \longrightarrow (\exists u'. \\
\text{conv } N \vdash^m \text{ax_steps } \langle \text{init}, s_0 \rangle, u \rangle \rightarrow^* \langle L', s' \rangle, u' \rangle \\
\wedge \neg \text{check_bexp } \varphi\ L' s') \\
) \text{ if formula} = \text{formula.AG } \varphi \\
\langle \text{proof} \rangle$

definition

$\text{Alw_ev_checker} = \text{dfs_map_impl'}$
 $(\text{impl.succs_P_impl' final_fun}) \text{impl.a0_impl impl.subsumes_impl (return } \circ \text{fst)}$
 $\text{impl.state_copy_impl}$

definition

$\text{leadsto_checker } \psi = \text{do } \{$
 $r \leftarrow \text{leadsto_impl}$
 $\text{impl.state_copy_impl } (\text{impl.succs_P_impl' } (\lambda (L, s). \neg \text{check_bexp } \psi\ L\ s))$

 $\text{impl.a0_impl impl.subsumes_impl (return } \circ \text{fst)}$
 $\text{impl.succs_impl' impl.emptiness_check_impl impl.F_impl}$
 $(\text{impl.Q_impl } (\lambda (L, s). \neg \text{check_bexp } \psi\ L\ s))$
 $\text{impl.tracei};$
 $\text{return } (\neg r)$
 $\}$

definition

$\text{model_checker} = ($
 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} :

$0 < \text{defs}'.p$

$\langle \text{proof} \rangle$

interpretation *Bisim_A: Bisimulation_Invariant*

$(\lambda(L, s, u) (L', s', u').$
 $\text{defs}'.\text{defs}.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$
 $(\lambda(L, s, u) (L', s', u').$
 $conv\ N \vdash^m ax_steps \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(=) (\lambda(L, s, u). product'.all_prop\ L\ s)$
 $(\lambda(L, s, u). product'.all_prop\ L\ s)$
 $\langle \text{proof} \rangle$

interpretation *Bisim_B: Bisimulation_Invariant*

$(\lambda(l, u) (l', u'). conv_A\ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda(L, s, u) (L', s', u').$
 $\text{defs}'.\text{defs}.prod_ta \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$
 $(\lambda(l, u') (L, s, u). (l, u') = ((L, s), u))\ \lambda\ _.\ True$
 $(\lambda(L, s, u). product'.all_prop\ L\ s)$
 $\langle \text{proof} \rangle$

interpretation *Bisimulation_Invariant*

$(\lambda(l, u) (l', u'). conv_A\ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda(L, s, u) (L', s', u').$
 $conv\ N \vdash^m ax_steps \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(\lambda(l, u') (L, s, u). (l, u') = ((L, s), u))\ \lambda\ _.\ True$
 $(\lambda(L, s, u). product'.all_prop\ L\ s)$
 $\langle \text{proof} \rangle$

lemma *models_correct*:

$conv\ N, (init, s_0, u_0) \models_m ax_steps\ \Phi = (\text{case } \Phi \text{ of}$
 $formula.EX\ \varphi \Rightarrow$
 $(\lambda((L, s), u). \exists\ L'\ s'\ u'. conv_A\ A \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge check_bexp\ \varphi\ L'\ s'))$
 $| formula.EG\ \varphi \Rightarrow$
 $Not\ o\ Graph_Defs.Alw_ev$
 $(\lambda(l, u) (l', u'). conv_A\ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda((L, s), _). \neg check_bexp\ \varphi\ L\ s)$
 $| formula.AX\ \varphi \Rightarrow$
 $Graph_Defs.Alw_ev$

$(\lambda (l, u) (l', u'). \text{conv_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda ((L, s), _). \text{check_bexp } \varphi \ L \ s)$
 $| \text{formula.AG } \varphi \Rightarrow$
 $\text{Not } o (\lambda ((L, s), u).$
 $\exists L' s' u'. \text{conv_A } A \vdash' \langle (L, s), u \rangle \rightarrow^* \langle (L', s'), u' \rangle \wedge \neg \text{check_bexp } \varphi \ L' s'$
 $)$
 $| \text{formula.Leadsto } \varphi \ \psi \Rightarrow$
 $\text{Graph_Defs.leadsto}$
 $(\lambda (l, u) (l', u'). \text{conv_A } A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda ((L, s), _). \text{check_bexp } \varphi \ L \ s)$
 $(\lambda ((L, s), _). \text{check_bexp } \psi \ L \ s)$
 $) ((\text{init}, s_0), u_0) \text{ if } \neg \text{deadlock } ((\text{init}, s_0), u_0)$
 $\langle \text{proof} \rangle$

lemma *final_fun_final'*:
 $((\lambda (L, s). P \ L \ s), (\lambda (L, s). P \ L \ s)) \in \text{inv_rel } Id \ \text{states}' \text{ for } P$
 $\langle \text{proof} \rangle$

lemma *final_fun_final[intro, simp]*:
 $((\lambda (L, s). P \ L \ s), (\lambda (L, s). P \ L \ s)) \in \text{inv_rel } Id \ \text{states}' \text{ for } P$
 $\langle \text{proof} \rangle$

abbreviation $u_0 \equiv (\lambda _. 0 :: \text{real})$

lemma *deadlock_start_iff*:
 $\text{Bisim_A.B.deadlock } (\text{init}, s_0, \lambda _. 0) \longleftrightarrow \text{deadlock } ((\text{init}, s_0), u_0)$
 $\langle \text{proof} \rangle$

theorem *model_check'*:
 $(\text{uncurry0 } \text{model_checker},$
 $\text{uncurry0 } ($
 $\text{SPEC } (\lambda r.$
 $\neg \text{Graph_Defs.deadlock}$
 $(\lambda (L, s, u) (L', s', u'). \text{conv } N \vdash^m \text{ax_steps } \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle) (\text{init}, s_0, u_0) \longrightarrow$
 $r = (\text{conv } N, (\text{init}, s_0, u_0) \models_m \text{ax_steps } \text{formula})$
 $)$
 $)$
 $)$
 $\in \text{unit_assn}^k \rightarrow_a \text{bool_assn}$
 $\langle \text{proof} \rangle$

theorem *model_check'_hoare*:
 $\langle \text{emp} \rangle$
 model_checker
 $\langle \lambda r. \uparrow ((\neg \text{Bisim_A.B.deadlock } (\text{init}, s_0, \lambda _. 0)) \longrightarrow r = ($
 $\text{conv } N, (\text{init}, s_0, u_0) \models_m \text{ax_steps } \text{formula}$
 $)) \rangle_t$
 $\langle \text{proof} \rangle$

lemma *Alw_ev_checker_alt_def'*:
 $\text{Alw_ev_checker} \equiv$
 $\text{do } \{$
 $x \leftarrow \text{let}$
 $\text{key} = \text{return} \circ \text{fst};$

```

    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;
_ ← return ();
return x
}
⟨proof⟩

```

lemma *leadsto_checker_alt_def'*:

```

leadsto_checker ψ ≡
do {
  r ← let
    key = return ∘ fst;
    sub = impl.subsumes_impl;
    copy = impl.state_copy_impl;
    start = impl.a0_impl;
    final = impl.F_impl;
    final' = (impl.Q_impl (λ(L, s). ¬ check_bexp ψ L s));
    succs = impl.succs_P_impl' (λ(L, s). ¬ check_bexp 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)
}
⟨proof⟩

```

schematic_goal *succs_P_impl_alt_def*:

```

impl.succs_P_impl (λ(L, s). P L s) ≡ ?impl for P
⟨proof⟩

```

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' ≡
do {
  x ← do {
    let key = return ∘ 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
  }
}

```

```

    };
    _ ← return ();
    return x
  }
  ⟨proof⟩

schematic_goal reachability_checker'_alt_def:
  reachability_checker' ≡ ?impl
  ⟨proof⟩

schematic_goal Alw_ev_checker_alt_def:
  Alw_ev_checker ≡ ?impl
  ⟨proof⟩

schematic_goal leadsto_checker_alt_def:
  leadsto_checker ≡ ?impl
  ⟨proof⟩

schematic_goal reachability_checker'_alt_def_refined:
  reachability_checker' ≡ ?impl
  ⟨proof⟩

schematic_goal Alw_ev_checker_alt_def_refined:
  Alw_ev_checker ≡ ?impl
  ⟨proof⟩

schematic_goal leadsto_checker_alt_def_refined:
  leadsto_checker ≡ ?impl
  ⟨proof⟩

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]:

```
precond_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> precond_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>
```

theorem model_check_alt:

```
<emp> precond_mc 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) ⊨max_steps formula
      ))
  else r = None
  )>t
<proof>
```

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

```
precond_mc 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 ⇒ 'b ⇒ 'b ('a, 'b) exp ('a, 'b) exp | unop 'b ⇒ 'b ('a, 'b) exp

```

inductive *check_bexp* :: ('a → 'b) ⇒ ('a, 'b :: linorder) bexp ⇒ bool ⇒ bool **and** *is_val*
where

```

check_bexp s true True |
check_bexp s (not e) (¬ b) if check_bexp s e b |
check_bexp s (and e1 e2) (a ∧ b) if check_bexp s e1 a check_bexp s e2 b |
check_bexp s (or e1 e2) (a ∨ b) if check_bexp s e1 a check_bexp s e2 b |
check_bexp s (imply e1 e2) (a → 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 ≤ 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 ≥ 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

$('a, 'b)\ upd = ('a * ('a, 'b)\ exp)\ list$

definition is_upd **where**

$is_upd\ s\ upd\ s' \equiv \exists x\ e\ v.\ upd = (x, e) \wedge is_val\ s\ e\ v \wedge s' = s(x := Some\ v)$ **for** upd

inductive is_upds **where**

$is_upds\ s\ []\ s$ |
 $is_upds\ s\ (x \# xs)\ s''$ **if** $is_upd\ s\ x\ s'\ is_upds\ s'\ xs\ s''$

inductive_cases is_upds_NilE :

$is_upds\ s\ []\ s'$

and is_upds_ConsE :

$is_upds\ s\ (e \# es)\ s'$

inductive_simps $is_upds_Nil_iff$: $is_upds\ s\ []\ s'$

and $is_upds_Cons_iff$: $is_upds\ s\ (f \# upds)\ s'$

lemma $is_upds_appendI$:

assumes $is_upds\ s\ upds1\ s'\ is_upds\ s'\ upds2\ s''$

shows $is_upds\ s\ (upds1\ @\ upds2)\ s''$

$\langle proof \rangle$

lemma $is_upds_appendE$:

assumes $is_upds\ s\ (upds1\ @\ upds2)\ s''$

obtains s' **where** $is_upds\ s\ upds1\ s'\ is_upds\ s'\ upds2\ s''$

$\langle proof \rangle$

lemma is_upds_NilD :

$s' = s$ **if** $is_upds\ s\ []\ s'$

$\langle proof \rangle$

lemma $is_upds_all_NilD$:

assumes $is_upds\ s\ (concat\ upds)\ s' (\forall xs \in set\ upds.\ xs = [])$

shows $s' = s$

$\langle proof \rangle$

```

lemma is_upd_const_simp:
  is_upd s (x, const c) s'  $\longleftrightarrow$  s' = s(x := Some c)
  <proof>

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,conv_cc g,a,f,r,l')

qualified definition conv_A where conv_A  $\equiv$   $\lambda$  (C, U, T, I). (C, U, conv_t 'T, conv_cc o I)

definition conv where
  conv  $\equiv$   $\lambda$ (broadcast, automata, bounds). (broadcast, map conv_A automata, bounds)

end

datatype 'b label = Del | Internal 'b | Bin 'b | Broad 'b

```

definition *bounded where*

$bounded\ bounds\ s \equiv dom\ s = dom\ bounds \wedge$
 $(\forall x \in dom\ s. fst\ (the\ (bounds\ x)) \leq the\ (s\ x) \wedge the\ (s\ x) \leq snd\ (the\ (bounds\ x)))$

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:

$\llbracket$
"target invariant" | $\forall p < length\ N. u \oplus d \vdash inv\ (N\ !\ p)\ (L\ !\ p);$
"nonnegative" | $d \geq 0;$
"urgent" | $(\exists p < length\ N. L\ !\ p \in urgent\ (N\ !\ p)) \longrightarrow d = 0;$
"bounded" | $bounded\ B\ s$
 $\rrbracket$

$\Longrightarrow (broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L, s, u \oplus d \rangle \mid$

step_int:

$\llbracket$
TRANS "silent" | $(l, b, g, Sil\ a, f, r, l') \in trans\ (N\ !\ p);$
"committed" | $l \in committed\ (N\ !\ p) \vee (\forall p < length\ N. L\ !\ p \notin committed\ (N\ !\ p));$
"bexp" | $check_bexp\ s\ b\ True;$
"guard" | $u \vdash g;$
"target invariant" | $\forall p < length\ N. u' \vdash inv\ (N\ !\ p)\ (L'\ !\ p);$
"loc" | $L!p = l;$
"range" | $p < length\ L;$
"new loc" | $L' = L[p := l'];$
"new valuation" | $u' = [r \rightarrow 0]u;$
"is_upd" | $is_upds\ s\ f\ s';$
"bounded" | $bounded\ B\ s'$
 $\rrbracket$

$\Longrightarrow (broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Internal\ a} \langle L', s', u' \rangle \mid$

step_bin:

$\llbracket$
"not broadcast" | $(a \notin broadcast);$
TRANS "in" | $(l1, b1, g1, In\ a, f1, r1, l1') \in trans\ (N\ !\ p);$
 $\rrbracket$

$TRANS \text{ "out" } \mid (l2, b2, g2, Out\ a, f2, r2, l2') \in trans\ (N !\ q);$
 $\text{"committed" } \mid$
 $l1 \in committed\ (N !\ p) \vee l2 \in committed\ (N !\ q) \vee (\forall p < length\ N. L !\ p \notin committed\ (N !\ p));$
 $\text{"bexp" } \mid check_bexp\ s\ b1\ True; \text{"bexp" } \mid check_bexp\ s\ b2\ True;$
 $\text{"guard" } \mid u \vdash g1; \text{"guard" } \mid u \vdash g2;$
 $\text{"target invariant" } \mid \forall p < length\ N. u' \vdash inv\ (N !\ p)\ (L' !\ p);$
 $RECV \text{ "loc" } \mid L!p = l1; SEND \text{ "loc" } \mid L!q = l2;$
 $RECV \text{ "range" } \mid p < length\ L; SEND \text{ "range" } \mid q < length\ L;$
 $\text{"different" } \mid p \neq q;$
 $\text{"new loc" } \mid L' = L[p := l1', q := l2'];$
 $\text{"new valuation" } \mid u' = [r1 @ r2 \rightarrow 0]u;$
 $\text{"upd" } \mid is_upds\ s\ f1\ s'; \text{"upd" } \mid is_upds\ s'\ f2\ s'';$
 $\text{"bounded" } \mid bounded\ B\ s''$
 $\Downarrow$
 $\implies (broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Bin\ a} \langle L', s'', u' \rangle \mid$
 $step_broad:$
 $\Downarrow$
 $\text{"broadcast" } \mid a \in broadcast;$
 $TRANS \text{ "out" } \mid (l, b, g, Out\ a, f, r, l') \in trans\ (N !\ p);$
 $TRANS \text{ "in" } \mid (\forall p \in set\ ps. (L !\ p, bs\ p, gs\ p, In\ a, fs\ p, rs\ p, ls'\ p) \in trans\ (N !\ p));$
 $\text{"committed" } \mid (l \in committed\ (N !\ p) \vee (\exists p \in set\ ps. L !\ p \in committed\ (N !\ p)))$
 $\vee (\forall p < length\ N. L !\ p \notin committed\ (N !\ p));$
 $\text{"bexp" } \mid check_bexp\ s\ b\ True;$
 $\text{"bexp" } \mid \forall p \in set\ ps. check_bexp\ s\ (bs\ p)\ True;$
 $\text{"guard" } \mid u \vdash g;$
 $\text{"guard" } \mid \forall p \in set\ ps. u \vdash gs\ p;$
 $\text{"maximal" } \mid \forall q < length\ N. q \notin set\ ps \wedge p \neq q$
 $\implies (\forall b\ g\ f\ r\ l'. (L!q, b, g, In\ a, f, r, l') \in trans\ (N !\ q))$
 $\implies \neg check_bexp\ s\ b\ True \vee \neg u \vdash g;$
 $\text{"target invariant" } \mid \forall p < length\ N. u' \vdash inv\ (N !\ p)\ (L' !\ p);$
 $SEND \text{ "loc" } \mid L!p = l;$
 $SEND \text{ "range" } \mid p < length\ L;$
 $SEL \text{ "range" } \mid set\ ps \subseteq \{0..<length\ N\};$
 $SEL \text{ "not sender" } \mid p \notin set\ ps;$
 $SEL \text{ "distinct" } \mid distinct\ ps;$
 $SEL \text{ "sorted" } \mid sorted\ ps;$
 $\text{"new loc" } \mid L' = (fold\ (\lambda p\ L. L[p := ls'\ p])\ ps\ L)[p := l'];$
 $\text{"new valuation" } \mid u' = [r @ concat\ (map\ rs\ ps) \rightarrow 0]u;$
 $\text{"upd" } \mid is_upds\ s\ f\ s';$
 $\text{"upds" } \mid is_upds\ s'\ (concat_map\ fs\ ps)\ s'';$
 $\text{"bounded" } \mid bounded\ B\ s''$
 $\Downarrow$
 $\implies (broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Broad\ a} \langle L', s'', u' \rangle$
lemmas $[intro?] = step_u.intros[unfolded\ TAG_def]$

Comments:

- Should there be an error transition + state if states run out 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*:

$\langle u \vdash \text{concat } gs \longleftrightarrow (\forall g \in \text{set } gs. u \vdash g) \rangle$
 $\langle \text{proof} \rangle$

lemma *clock_val_append_iff*:

$\langle u \vdash g1 @ g2 \longleftrightarrow u \vdash g1 \wedge u \vdash g2 \rangle$
 $\langle \text{proof} \rangle$

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

$\text{broadcast} = \text{fst } A$

definition

$N \ i = \text{fst } (\text{snd } A) \ ! \ i$

definition

$\text{bounds} = \text{snd } (\text{snd } A)$

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

$\text{trans_int} =$
 $\{((L, s), g, \text{Internal } a, r, (L', s')) \mid L \ s \ l \ b \ g \ f \ p \ a \ r \ l' \ L' \ s'.$
 $(l, b, g, \text{Sil } a, f, r, l') \in \text{trans } (N \ p) \wedge$
 $(l \in \text{committed } (N \ p) \vee (\forall p < n_ps. L \ ! \ p \notin \text{committed } (N \ p))) \wedge$
 $L[p := l] \wedge p < \text{length } L \wedge L' = L[p := l'] \wedge \text{is_upds } s \ f \ s' \wedge \text{check_bexp } s \ b \ \text{True} \wedge$
 $L \in \text{states} \wedge \text{bounded } \text{bounds } s \wedge \text{bounded } \text{bounds } s'$
 $\}$

definition

$\text{trans_bin} =$
 $\{((L, s), g1 @ g2, \text{Bin } a, r1 @ r2, (L', s'')) \mid$
 $L \ s \ L' \ s' \ s'' \ a \ p \ q \ l1 \ b1 \ g1 \ f1 \ r1 \ l1' \ l2 \ b2 \ g2 \ f2 \ r2 \ l2'.$

$a \notin \text{broadcast} \wedge$
 $(l1, b1, g1, \text{In } a, f1, r1, l1') \in \text{trans } (N p) \wedge$
 $(l2, b2, g2, \text{Out } a, f2, r2, l2') \in \text{trans } (N q) \wedge$
 $(l1 \in \text{committed } (N p) \vee l2 \in \text{committed } (N q) \vee (\forall p < n_ps. L ! p \notin \text{committed } (N p))) \wedge$
 $L!p = l1 \wedge L!q = l2 \wedge p < \text{length } L \wedge q < \text{length } L \wedge p \neq q \wedge$
 $\text{check_bexp } s b1 \text{ True} \wedge \text{check_bexp } s b2 \text{ True} \wedge$
 $L' = L[p := l1', q := l2'] \wedge \text{is_upds } s f1 s' \wedge \text{is_upds } s' f2 s'' \wedge$
 $L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s''$
 $\}$

definition

$\text{trans_broad} =$
 $\{((L, s), g @ \text{concat } (\text{map } gs \ ps), \text{Broad } a, r @ \text{concat } (\text{map } rs \ ps), (L', s'')) \mid$
 $L \ s \ L' \ s' \ s'' \ a \ p \ l \ b \ g \ f \ r \ l' \ bs \ gs \ fs \ rs \ ls' \ ps.$
 $a \in \text{broadcast} \wedge$
 $(l, b, g, \text{Out } a, f, r, l') \in \text{trans } (N p) \wedge$
 $(\forall p \in \text{set } ps. (L ! p, bs \ p, gs \ p, \text{In } a, fs \ p, rs \ p, ls' \ p) \in \text{trans } (N p)) \wedge$
 $(l \in \text{committed } (N p) \vee (\exists p \in \text{set } ps. L ! p \in \text{committed } (N p)))$
 $\vee (\forall p < n_ps. L ! p \notin \text{committed } (N p))) \wedge$
 $(\forall q < n_ps. q \notin \text{set } ps \wedge p \neq q \longrightarrow$
 $\neg (\exists b \ g \ f \ r \ l'. (L!q, b, g, \text{In } a, f, r, l') \in \text{trans } (N q) \wedge \text{check_bexp } s \ b \ \text{True})) \wedge$
 $L!p = l \wedge$
 $p < \text{length } L \wedge \text{set } ps \subseteq \{0..<n_ps\} \wedge p \notin \text{set } ps \wedge \text{distinct } ps \wedge \text{sorted } ps \wedge$
 $\text{check_bexp } s \ b \ \text{True} \wedge (\forall p \in \text{set } ps. \text{check_bexp } s \ (bs \ p) \ \text{True}) \wedge$
 $L' = (\text{fold } (\lambda p \ L. L[p := ls' \ p]) \ ps \ L)[p := l'] \wedge$
 $\text{is_upds } s \ f \ s' \wedge \text{is_upds } s' \ (\text{concat_map } fs \ ps) \ s'' \wedge$
 $L \in \text{states} \wedge \text{bounded bounds } s \wedge \text{bounded bounds } s''$
 $\}$

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 $\text{inv_of_prod[simp]}:$

$\text{inv_of } \text{prod_ta} = \text{prod_inv}$
 $\langle \text{proof} \rangle$

lemma $\text{trans_of_prod[simp]}:$

$\text{trans_of } \text{prod_ta} = \text{trans_prod}$
 $\langle \text{proof} \rangle$

lemma $A_split:$

$\langle A = (\text{broadcast}, \text{map } N \ [0..<n_ps], \text{bounds}) \rangle$
 $\langle \text{proof} \rangle$

lemma $\text{map_N_n_ps_simp}:$

$\text{map } N \ [0..<n_ps] ! p = N \ p$
 $\langle \text{proof} \rangle$

lemma $N_split_simp[simp]:$

assumes $A = (\text{broadcast}', N', B)$
shows $N' ! i = N \ i$

$\langle proof \rangle$

lemma *state_preservation_updI*:
assumes $l' \in (\bigcup (l, b, g, a, r, u, l') \in trans\ (N\ p). \{l, l'\})\ L \in states$
shows $L[p := l'] \in states$
 $\langle proof \rangle$

lemma *state_preservation_fold_updI*:
assumes $\forall p \in set\ ps. ls'\ p \in (\bigcup (l, b, g, a, r, u, l') \in trans\ (N\ p). \{l, l'\})\ L \in states$
shows $fold\ (\lambda p\ L. L[p := ls'\ p])\ ps\ L \in states$
 $\langle proof \rangle$

lemma *state_set_states*:
 $Simulation_Graphs_TA.state_set\ prod_ta \subseteq \{(l, s). l \in states\}$
 $\langle proof \rangle$

lemma *trans_prod_bounded_inv*:
 $\langle bounded\ bounds\ s' \rangle\ \mathbf{if}\ \langle ((L, s), g, a, r, (L', s')) \in trans_prod \rangle$
 $\langle proof \rangle$

lemma *trans_prod_states_inv*:
 $\langle L' \in states \rangle\ \mathbf{if}\ \langle ((L, s), g, a, r, (L', s')) \in trans_prod \rangle\ \langle L \in states \rangle$
 $\langle proof \rangle$

end

locale *Prod_TA_sem* =
 $Prod_TA_Defs\ A\ \mathbf{for}\ A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder)\ nta$
begin

lemma *prod_invI[intro]*:
 $\langle u \vdash prod_inv\ (L, s) \rangle\ \mathbf{if}\ \langle \forall p < n_ps. u \vdash Simple_Network_Language.inv\ (N\ p)\ (L\ !\ p) \rangle$
 $\langle proof \rangle$

lemma *prod_invD[dest]*:
 $\langle \forall p < n_ps. u \vdash Simple_Network_Language.inv\ (N\ p)\ (L\ !\ p) \rangle\ \mathbf{if}\ \langle u \vdash prod_inv\ (L, s) \rangle\ \langle L \in states \rangle$
 $\langle proof \rangle$

lemma *delay_sound*:
assumes $prod_ta \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle\ L \in states\ bounded\ bounds\ s$
 $\forall N \in set\ (fst\ (snd\ A)).\ urgent\ N = \{\}$
shows $A \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$
 $\langle proof \rangle$

lemma *action_sound*:
 $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle\ \mathbf{if}\ prod_ta \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$
 $\langle proof \rangle$

lemma *bounded_inv*:
 $\langle bounded\ bounds\ s' \rangle\ \mathbf{if}\ \langle A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle \rangle$
 $\langle proof \rangle$

```

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

end

locale Prod_TA =
  Prod_TA_sem for  $A :: ('a, 's, 'c, 't :: \text{time}, 'x, 'v :: \text{linorder}) \text{nta} +$ 
  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:
   $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ 
  if  $A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle$   $a \neq \text{Del } L \in \text{states bounded bounds } s$ 
   $\langle \text{proof} \rangle$ 

lemma delay_complete:
  assumes  $A \vdash \langle L, s, u \rangle \rightarrow_{\text{Del}} \langle L', s', u' \rangle$ 
  obtains  $d$  where  $\text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow^d \langle (L', s'), u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma step_iff:
   $(\exists a. A \vdash \langle L, s, u \rangle \rightarrow_a \langle L', s', u' \rangle) \longleftrightarrow \text{prod\_ta} \vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  if bounded bounds  $s \ L \in \text{states} \ \forall N \in \text{set } (\text{fst } (\text{snd } A)). \text{urgent } N = \{\}$ 
   $\langle \text{proof} \rangle$ 

end

end

theory TA_Impl_Misc2
imports
  Munta_Base.TA_Impl_Misc
  HOL-Library.Sublist
  List-Index.List_Index
  Automatic_Refinement.Misc
begin

lemma mem_nth:
   $x \in \text{set } xs \implies \exists i < \text{length } xs. xs ! i = x$ 
   $\langle \text{proof} \rangle$ 

lemma union_subsetI:
   $A \cup B \subseteq C \cup D$  if  $A \subseteq C \ B \subseteq D$ 
   $\langle \text{proof} \rangle$ 

lemma map_eq_imageD:
   $f \text{ ' set } xs = \text{set } ys$  if  $\text{map } f \ xs = ys$ 
   $\langle \text{proof} \rangle$ 

lemma if_contract:
   $(\text{if } a \text{ then } x \text{ else if } b \text{ then } x \text{ else } y) = (\text{if } a \vee b \text{ then } x \text{ else } y)$  for  $a \ x \ b \ y$ 
   $\langle \text{proof} \rangle$ 

```

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* ∧ *Q b*} **if** *finite* {a. *P a*} *finite* {a. *Q a*}
 ⟨*proof*⟩

lemma *finite_prodI3*:
 finite {(a,b,c). *P a* ∧ *Q b* ∧ *Q1 c*}
 if *finite* {a. *P a*} *finite* {a. *Q a*} *finite* {a. *Q1 a*}
 ⟨*proof*⟩

lemma *finite_prodI4*:
 finite {(a,b,c,d). *P a* ∧ *Q b* ∧ *Q1 c* ∧ *Q2 d*}
 if *finite* {a. *P a*} *finite* {a. *Q a*} *finite* {a. *Q1 a*} *finite* {a. *Q2 a*}
 ⟨*proof*⟩

lemma *finite_prodI5*:
 finite {(a,b,c,d,e). *P a* ∧ *Q b* ∧ *Q1 c* ∧ *Q2 d* ∧ *Q3 e*}
 if *finite* {a. *P a*} *finite* {a. *Q a*} *finite* {a. *Q1 a*} *finite* {a. *Q2 a*} *finite* {a. *Q3 a*}
 ⟨*proof*⟩

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*) ¬ *P y acc*

 ∧ *acc x*. ¬ *P y acc* ⇒ *Q acc* ⇒ *P y* (*f x acc*) ⇒ *x* ∈ *set xs* ⇒ *x = y*

Q acc ∧ *acc x*. *Q acc* ⇒ *Q* (*f x acc*)

 ∧ *acc x*. ¬ *P y acc* ⇒ *Q acc* ⇒ *P y* (*f x acc*) ⇒ *R y*

shows ∃ *ys zs*. *xs* = *ys* @ *y* # *zs* ∧ ¬ *P y* (*fold f ys acc*) ∧ *P y* (*f y* (*fold f ys acc*)) ∧ *R y*
 ⟨*proof*⟩

lemmas *fold_evD2'* = *fold_evD2*[**where** *R* = λ *_*. *True*, *simplified*]

lemma *filter_map_map_filter*:

filter P (*map f xs*) = *List.map_filter* (λ *x*. *let y = f x in if P y then Some y else None*) *xs*
 ⟨*proof*⟩

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*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *filter_greatest_subseqD*:
 $\exists x \in \text{set } zs. \neg P\ x$ **if** *subseq zs xs $\text{length } zs > \text{length } (\text{filter } P\ xs)$*
 $\langle \text{proof} \rangle$

lemma *filter_eq_iff_greatest_subseq*:
 $\text{filter } P\ xs = ys \longleftrightarrow$
subseq ys xs $\wedge (\forall x \in \text{set } ys. P\ x) \wedge$
 $(\forall zs. \text{subseq } zs\ xs \wedge \text{length } zs > \text{length } ys \longrightarrow (\exists x \in \text{set } zs. \neg P\ x))$
 $\langle \text{proof} \rangle$

lemma *subseq_subsetD*:
 $\text{set } xs \subseteq \text{set } ys$ **if** *subseq xs ys*
 $\langle \text{proof} \rangle$

lemma *subseq_distinct*:
distinct xs **if** *distinct ys subseq xs ys*
 $\langle \text{proof} \rangle$

lemma *finite_subseqs*:
 $\text{finite } \{xs. \text{subseq } xs\ ys\}$ **(is finite ?S)**
 $\langle \text{proof} \rangle$

lemma *filter_distinct_eqI*:
assumes
subseq ys 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 *filter P xs = ys*
 $\langle \text{proof} \rangle$

lemma *subseq_sorted_wrt*:
 $\text{sorted_wrt } R\ xs$ **if** $\text{sorted_wrt } R\ ys$ *subseq xs ys*
 $\langle \text{proof} \rangle$

lemma *subseq_sorted*:
 $\text{sorted } xs$ **if** $\text{sorted } ys$ *subseq xs ys*
 $\langle \text{proof} \rangle$

lemma *sorted_distinct_subset_subseqI*:
assumes $\text{sorted } xs$ *distinct xs sorted ys $\text{set } xs \subseteq \text{set } ys$*
shows *subseq xs ys*
 $\langle \text{proof} \rangle$

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)
 $\langle$ proof $\rangle$ 

lemma subseq_mapE:
assumes subseq xs (map f ys)
obtains xs' where subseq xs' ys map f xs' = xs
 $\langle$ proof $\rangle$ 

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
 $\langle$ proof $\rangle$ 

lemma list_all2_fst_aux:
map fst xs = ys if list_all2 ( $\lambda x y. \text{fst } x = y$ ) xs ys
 $\langle$ proof $\rangle$ 

Stronger version of  $\llbracket \text{inj } ?f; \text{map\_of } ?t \text{ ?k} = \text{Some } ?x \rrbracket \implies \text{map\_of } (\text{map } (\lambda(k, y). (?f \text{ k}, y)) \text{ ?t})$ 
 $(?f \text{ ?k}) = \text{Some } ?x$ 

theorem map_of_mapk_SomeI':
assumes inj_on f (fst ' set t)
and map_of t k = Some x
shows map_of (map ( $\lambda(k, y). (f \text{ k}, g \text{ y})$ ) t) (f k) = Some (g x)
 $\langle$ proof $\rangle$ 

theorem map_of_mapk_SomeI:
assumes inj f
and map_of t k = Some x
shows map_of (map ( $\lambda(k, y). (f \text{ k}, g \text{ y})$ ) t) (f k) = Some (g x)
 $\langle$ proof $\rangle$ 

lemma list_all2_map_eq_iff:
list_all2 ( $\lambda x y. f \text{ x} = g \text{ y}$ ) xs ys  $\longleftrightarrow$  map f xs = map g ys
 $\langle$ proof $\rangle$ 

end
theory TA_More2
imports Munta_Base.TA_More
begin

lemma collect_clock_pairs_concat:
collect_clock_pairs (concat xxs) = ( $\bigcup$   $x \in \text{set } xxs. \text{collect\_clock\_pairs } x$ )
 $\langle$ proof $\rangle$ 

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 :: time, 's) ta

```

```

fixes  $S :: 's \text{ set}$ 
assumes  $\text{state\_set\_trans\_of}: \text{state\_set } (\text{trans\_of } A) \subseteq S$ 
assumes  $\text{invs}: \forall l \in S. \text{inv\_of } A \ l = \text{inv\_of } B \ l$ 
assumes  $\text{trans\_eq}: \text{trans\_of } A = \text{trans\_of } B$ 
begin

lemma delay1:
   $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$  if  $l \in S$   $B \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma action1:
   $A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$  if  $B \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma delay2:
   $B \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$  if  $l \in S$   $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma action2:
   $B \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$  if  $A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma step1:
   $A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$  if  $l \in S$   $B \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma step2:
   $B \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$  if  $l \in S$   $A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ 
   $\langle \text{proof} \rangle$ 

lemma  $S\_inv$ :
   $l' \in S$  if  $A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$   $l \in S$ 
   $\langle \text{proof} \rangle$ 

interpretation interp: Bisimulation_Invariant
   $\lambda (l, u) (l', u'). A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ 
   $\lambda (l, u) (l', u'). B \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ 
   $\lambda lu \ lu'. lu' = lu \ \lambda (l, u). l \in S \ \lambda (l, u). l \in S$ 
   $\langle \text{proof} \rangle$ 

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 x = (if x  $\in$  dom (map_of xs) then the (map_of xs x) else a)
  <proof>

lemma range_default_map_of:
  range (default_map_of x xs)  $\subseteq$  snd ` set xs  $\cup$  {x}
  <proof>

lemma finite_range_default_map_of:
  finite (range (default_map_of x m))
  <proof>

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
     $\forall i < \text{length } L + n. \forall x \in S. \forall y \in T. f \ i \ x = g \ i \ y \longrightarrow x = y$ 
  <proof>

lemma map_index_inj:
  L = L'
  if map_index f L = map_index g L' set L  $\subseteq$  S set L'  $\subseteq$  T
     $\forall i < \text{length } L. \forall x \in S. \forall y \in T. f \ i \ x = g \ i \ y \longrightarrow x = y$ 
  <proof>

lemma map_index_inj1:
  L = L'
  if map_index f L = map_index g L'
     $\forall i < \text{length } L. f \ i \ (L \ ! \ i) = g \ i \ (L' \ ! \ i) \longrightarrow L \ ! \ i = L' \ ! \ i$ 
  <proof>

lemma map_index_update:
  map_index f (xs[i := a]) = (map_index f xs)[i := f i a]
  <proof>

lemma map_trans_broad_aux1:
  map_index map_loc (fold ( $\lambda p \ L. L[p := ls' \ p]$ ) ps L) =
  fold ( $\lambda p \ L. L[p := \text{map\_loc } p \ (ls' \ p)]$ ) ps (map_index map_loc L)
  <proof>

lemma InD2:
  fixes map_action
  assumes inj map_action In (map_action a) = map_act map_action a'
  shows a' = In a

```

$\langle \text{proof} \rangle$

lemma *OutD2*:

fixes *map_action*

assumes *inj map_action Out (map_action a) = map_act map_action a'*

shows *a' = Out a*

$\langle \text{proof} \rangle$

lemma (**in** *Prod_TA_Defs*) *trans_broad_alt_def*:

trans_broad =

$\{((L, s), g @ \text{concat} (\text{map } gs \ ps), \text{Broad } a, r @ \text{concat} (\text{map } rs \ ps), (L', s'')) \mid$
 $L \ s \ L' \ s' \ s'' \ a \ p \ l \ b \ g \ f \ r \ l' \ bs \ gs \ fs \ rs \ ls' \ ps.$
 $a \in \text{broadcast} \ \wedge$
 $(l, b, g, \text{Out } a, f, r, l') \in \text{trans } (N \ p) \ \wedge$
 $(\forall p \in \text{set } ps. (L ! p, bs \ p, gs \ p, \text{In } a, fs \ p, rs \ p, ls' \ p) \in \text{trans } (N \ p)) \ \wedge$
 $(l \in \text{committed } (N \ p) \vee (\exists p \in \text{set } ps. L ! p \in \text{committed } (N \ p)))$
 $\vee (\forall p < n_ps. L ! p \notin \text{committed } (N \ p))) \ \wedge$
 $(\forall q < n_ps. q \notin \text{set } ps \wedge p \neq q \longrightarrow$
 $\neg (\exists b \ g \ f \ r \ l'. (L ! q, b, g, \text{In } a, f, r, l') \in \text{trans } (N \ q) \wedge \text{check_bexp } s \ b \ \text{True})) \ \wedge$
 $L ! p = l \ \wedge$
 $p < \text{length } L \wedge \text{set } ps \subseteq \{0..<n_ps\} \wedge p \notin \text{set } ps \wedge \text{distinct } ps \wedge \text{sorted } ps \wedge$
 $\text{check_bexp } s \ b \ \text{True} \wedge (\forall p \in \text{set } ps. \text{check_bexp } s \ (bs \ p) \ \text{True}) \ \wedge$
 $L' = (\text{fold } (\lambda p \ L. L[p := ls' \ p]) \ ps \ L)[p := l'] \ \wedge$
 $\text{is_upds } s \ f \ s' \wedge \text{is_upds } s' \ (\text{concat_map } fs \ ps) \ s'' \ \wedge$
 $L \in \text{states} \wedge \text{bounded_bounds } s \wedge \text{bounded_bounds } s'' \ \wedge$
 $(\forall p. p \notin \text{set } ps \longrightarrow bs \ p = \text{bexp.true}) \wedge (\forall p. p \notin \text{set } ps \longrightarrow gs \ p = []) \ \wedge$
 $(\forall p. p \notin \text{set } ps \longrightarrow fs \ p = []) \wedge (\forall p. p \notin \text{set } ps \longrightarrow rs \ p = [])$
 $\}$

$\langle \text{proof} \rangle$

definition

conv_automaton $\equiv \lambda(\text{committed}, \text{urgent}, \text{trans}, \text{inv}).$

(committed,

urgent,

map $(\lambda(l, b, g, a, f, r, l'). (l, b, \text{conv_cc } g, a, f, r, l')) \ \text{trans},$

map $(\lambda(s, cc). (s, \text{conv_cc } cc)) \ \text{inv})$

definition

automaton_of $\equiv$

$\lambda(\text{committed}, \text{urgent}, \text{trans}, \text{inv}). (\text{set } \text{committed}, \text{set } \text{urgent}, \text{set } \text{trans}, \text{default_map_of } [] \ \text{inv})$

locale *Simple_Network_Impl_Defs* =

fixes *automata* ::

$(\text{'s list} \times \text{'s list} \times (\text{'a act}, \text{'s}, \text{'c}, \text{'t}, \text{'x}, \text{int}) \ \text{transition list}$

$\times (\text{'s} \times (\text{'c}, \text{'t}) \ \text{cconstraint}) \ \text{list}) \ \text{list}$

and *broadcast* :: *'a list*

and *bounds'* :: $(\text{'x} \times (\text{int} \times \text{int})) \ \text{list}$

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'*) *<proof>*

lemma *L_len*[*intro, dest*]:

length L = n_ps **if** *L ∈ states*

<proof>

lemma *N_eq*:

<N i = automaton_of (automata ! i)> **if** *<i < n_ps>*

<proof>

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'* *<proof>*

end

Mapping through the product construction **lemma** *f_the_inv_f*:

f (the_inv f x) = x **if** *inj f x ∈ range f*

<proof>

method *fprem* =

(*match premises in R: _ ⇒ <rule R[elim_format]>, 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'*) *<proof>*

lemma (**in** *—*) *conv_ac_inj*:

ac = ac' **if** *conv_ac ac = conv_ac ac'*

<proof>

lemma (**in** *—*) *conv_cc_inj*:

cc = cc' **if** *conv_cc cc = conv_cc cc'*

<proof>

context

begin

lemma *conv_alt_def*:

conv (set broadcast, map automaton_of automata, map_of bounds') =
 (set broadcast, map (Simple_Network_Language.conv_A o automaton_of) automata, map_of
 bounds')

⟨proof⟩ **lemma** 2:

Simple_Network_Language.conv_A o automaton_of = (λ(committed, urgent, trans, inv).

(set committed,
 set urgent,
 set (map Simple_Network_Language.conv_t trans),
 default_map_of [] (map (λ (l, g). (l, conv_cc g)) inv)))

⟨proof⟩

lemma *conv_n_ps_eq*:

conv.n_ps = *n_ps*

⟨proof⟩

lemma *conv_N_eq*:

conv.N i = Simple_Network_Language.conv_A (N i) **if** *i* < *n_ps*

⟨proof⟩ **lemma** 5:

inv (*conv.N i*) = *conv_cc* o (*inv* (N i)) **if** *i* < *n_ps*

⟨proof⟩

lemma *trans_conv_N_eq*:

trans (*conv.N i*) = Simple_Network_Language.conv_t ' (*trans* (N i)) **if** *i* < *n_ps*

⟨proof⟩ **lemma** 71:

(*l*, *b*, *conv_cc* *g*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (*conv.N i*)

if (*l*, *b*, *g*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (N i) *i* < *n_ps*

⟨proof⟩ **lemma** 72:

(*l*, *b*, *conv_cc* *g*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (*conv.N i*)

⟷ (*l*, *b*, *g*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (N i) **if** *i* < *n_ps*

⟨proof⟩ **lemma** 73:

∃ *g'*. *g* = *conv_cc* *g'* ∧ (*l*, *b*, *g'*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (N i)

if (*l*, *b*, *g*, *a*, *r*, *u*, *l'*) ∈ Simple_Network_Language.trans (*conv.N i*) *i* < *n_ps*

⟨proof⟩

lemma *conv_bounds[simp]*:

conv.bounds = *bounds*

⟨proof⟩

lemma *conv_states[simp]*:

conv.states = *states*

⟨proof⟩ **lemma** 9:

committed (*conv.N p*) = *committed* (N p) **if** ⟨*p* < *n_ps*⟩

⟨proof⟩ **lemma** 10:

conv.broadcast = set broadcast

⟨proof⟩

lemma *conv_trans_int*:

conv.trans_int = (λ(*l*, *g*, *a*, *r*, *l'*). (*l*, map conv_ac *g*, *a*, *r*, *l'*)) ' *trans_int*

⟨proof⟩

lemma *conv_trans_bin*:

conv.trans_bin = (λ(*l*, *g*, *a*, *r*, *l'*). (*l*, map conv_ac *g*, *a*, *r*, *l'*)) ' *trans_bin*

⟨proof⟩

lemma *n_ps_rangeD*:

$p < n_ps$ **if** $p \in \text{set } ps$ $\text{set } ps \subseteq \{0..<n_ps\}$
 $\langle \text{proof} \rangle$

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 $b \ ps \ p \ q \ L \ a' \ s \ \langle \text{proof} \rangle$

lemma *conv_trans_broad*:

$\text{conv.trans_broad} = (\lambda(l, \ g, \ a, \ r, \ l'). (l, \ \text{map } \text{conv_ac } g, \ a, \ r, \ l')) \ ' \ \text{trans_broad}$
 $\langle \text{proof} \rangle$

lemma *conv_prod_ta*:

$\text{conv.prod_ta} = \text{Normalized_Zone_Semantics_Impl.conv_A prod_ta}$
 $\langle \text{proof} \rangle$

end

Fundamentals definition *clkp_set'* $\equiv$

$(\bigcup A \in \text{set automata}. \bigcup g \in \text{set } (\text{snd } (\text{snd } (\text{snd } A)))) \cdot \text{collect_clock_pairs } (\text{snd } g)) \\$
 $\cup (\bigcup A \in \text{set automata}. \bigcup (l, \ b, \ g, \ _) \in \text{set } (\text{fst } (\text{snd } (\text{snd } A)))) \cdot \text{collect_clock_pairs } g)$

definition *clk_set'* **where**

$\langle \text{clk_set}' = \\$
 $\text{fst } \langle \text{clkp_set}' \cup \\$
 $(\bigcup A \in \text{set automata}. \bigcup (_, \ _, \ _, \ _, \ _, \ r, \ _) \in \text{set } (\text{fst } (\text{snd } (\text{snd } A)))) \cdot \text{set } r) \rangle$

lemma *(in -) default_map_of_in_listD*:

$x \in \bigcup (\text{set } \langle \text{snd } \langle \text{set } \text{invs} \rangle \ \text{if } x \in \text{set } (\text{default_map_of } [] \ \text{invs } l) \\$
 $\langle \text{proof} \rangle$

lemma *collect_clock_pairs_invsI*:

$(a, \ b) \in \bigcup ((\text{collect_clock_pairs } o \ \text{snd}) \ ' \ \text{set } \text{invs}) \\$
if $(a, \ b) \in \text{collect_clock_pairs } (\text{default_map_of } [] \ \text{invs } l) \\$
 $\langle \text{proof} \rangle$

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 $i < n_ps \\$
 $\langle \text{proof} \rangle$

lemma *length_automata_eq_n_ps*:

$\text{length automata} = n_ps$
 $\langle \text{proof} \rangle$

lemma *clkp_set'_subs*:

Timed_Automata.clkp_set prod_ta $\subseteq$ *clkp_set'*
 ⟨proof⟩

lemma *collect_clkvt_subs*:
collect_clkvt (trans_of prod_ta) $\subseteq$
 $(\bigcup A \in \text{set automata}. \bigcup (_, _, _, _, _, r, _) \in \text{set (fst (snd (snd A)))}. \text{set } r)$
 ⟨proof⟩

lemma *clk_set'_subs*: *clk_set prod_ta* $\subseteq$ *clk_set'*
 ⟨proof⟩

end

lemma (in *Prod_TA_Defs*) *finite_range_invI*:
finite (range prod_inv) **if** *assms*: $\forall i < n_ps. \text{finite (range (inv (N i)))}$
 ⟨proof⟩

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*:
 $\text{states} \subseteq \{L. \text{set } L \subseteq \text{loc_set} \wedge \text{length } L = n_ps\}$
 ⟨proof⟩

lemma (in *Prod_TA_Defs*) *finite_states*:
assumes *finite_trans*: $\forall p < n_ps. \text{finite (Simple_Network_Language.trans } (N p))$
shows *finite states*
 ⟨proof⟩

context *Simple_Network_Impl*
begin

lemma *trans_N_finite*:
assumes $p < n_ps$
shows *finite (Simple_Network_Language.trans (N p))*
 ⟨proof⟩

lemma *states_finite*:
finite states
 ⟨proof⟩

lemma *bounded_finite*:
finite {s. bounded bounds s}
 ⟨proof⟩

lemma *trans_prod_finite*:
finite trans_prod
 ⟨proof⟩

lemma *prod_inv_finite*:
finite (range prod_inv)

$\langle \text{proof} \rangle$

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)

∀ (_, _, trans, _) ∈ set automata. ∀ (_, b, _, _, _, _) ∈ set trans.

∀ i ∈ vars_of_bexp b. i < n_vs

assumes bounds:

∀ i < n_vs. fst (bounds' ! i) = i

assumes action_set:

∀ a ∈ 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*:
 $\forall p < n_ps. \forall l\ b\ g\ a\ f\ r\ l'.$
 $(l, b, g, In\ a, f, r, l') \in Simple_Network_Language.trans\ (N\ p) \wedge a \in set\ broadcast \longrightarrow g =$
 $[]$
 $\langle proof \rangle$

sublocale *conv*: *Prod_TA*
 $(set\ broadcast, map\ (Simple_Network_Language.conv_A\ o\ automaton_of)\ automata, map_of$
 $bounds')$
 $\langle proof \rangle$

sublocale *TA_Start_No_Ceiling prod_ta init m*
 $\langle proof \rangle$

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* $\langle proof \rangle$

lemma *sem_N_eq*:
 $sem.N\ p = automaton_of\ (conv_automaton\ (automata\ !\ p))\ if\ \langle p < n_ps \rangle$
 $\langle proof \rangle$

end

inductive_cases *step_u_elims*:
 $A \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$
 $A \vdash \langle L, s, u \rangle \rightarrow_{Internal\ a} \langle L', s', u' \rangle$
 $A \vdash \langle L, s, u \rangle \rightarrow_{Bin\ a} \langle L', s'', u' \rangle$
 $A \vdash \langle L, s, u \rangle \rightarrow_{Broad\ a} \langle L', s'', u' \rangle$

inductive_cases *step_u_elims'*:
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$

$(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Internal\ a} \langle L', s', u' \rangle$
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Bin\ a} \langle L', s'', u' \rangle$
 $(broadcast, N, B) \vdash \langle L, s, u \rangle \rightarrow_{Broad\ a} \langle L', s'', u' \rangle$

lemma (in *Prod_TA_Defs*) *states_lengthD*:
 $length\ L = n_ps$ if $L \in states$
 $\langle proof \rangle$

end

theory *Simple_Network_Language_Impl_Refine*

imports *Simple_Network_Language_Impl*

begin

unbundle *no_library_syntax*

notation *fun_rel_syn* (**infixr** $\rightarrow 60$)

hide_const (**open**) *upd*

Expression evaluation **fun** *bval* :: $('a \Rightarrow 'b) \Rightarrow ('a, 'b :: linorder) \text{ bexp} \Rightarrow bool$ **and** *eval*

where

$bval\ _ \text{ bexp.true} \longleftrightarrow True$ |
 $bval\ s\ (not\ e) \longleftrightarrow \neg\ bval\ s\ e$ |
 $bval\ s\ (and\ e1\ e2) \longleftrightarrow bval\ s\ e1 \wedge bval\ s\ e2$ |
 $bval\ s\ (bexp.or\ e1\ e2) \longleftrightarrow bval\ s\ e1 \vee bval\ s\ e2$ |
 $bval\ s\ (imply\ e1\ e2) \longleftrightarrow bval\ s\ e1 \longrightarrow bval\ s\ e2$ |
 $bval\ s\ (eq\ i\ x) \longleftrightarrow eval\ s\ i = eval\ s\ x$ |
 $bval\ s\ (le\ i\ x) \longleftrightarrow eval\ s\ i \leq eval\ s\ x$ |
 $bval\ s\ (lt\ i\ x) \longleftrightarrow eval\ s\ i < eval\ s\ x$ |
 $bval\ s\ (ge\ i\ x) \longleftrightarrow eval\ s\ i \geq eval\ s\ x$ |
 $bval\ s\ (gt\ i\ x) \longleftrightarrow eval\ s\ i > eval\ s\ x$ |
 $eval\ s\ (const\ c) = c$ |
 $eval\ s\ (var\ x) = s\ x$ |
 $eval\ s\ (if_then_else\ b\ e1\ e2) = (if\ bval\ s\ b\ then\ eval\ s\ e1\ else\ eval\ s\ e2)$ |
 $eval\ s\ (binop\ f\ e1\ e2) = f\ (eval\ s\ e1)\ (eval\ s\ e2)$ |
 $eval\ s\ (unop\ f\ e) = f\ (eval\ s\ e)$

lemma *check_bexp_determ*:

$check_bexp\ s\ b\ b1 \Longrightarrow check_bexp\ s\ b\ b2 \Longrightarrow b1 = b2$
and *is_val_determ*: $is_val\ s\ e\ v1 \Longrightarrow is_val\ s\ e\ v2 \Longrightarrow v1 = v2$
 $\langle proof \rangle$

lemma *is_upd_determ*:

$s1 = s2$ if $is_upd\ s\ f\ s1\ is_upd\ s\ f\ s2$
 $\langle proof \rangle$

lemma *is_upds_determ*:

$s1 = s2$ if $is_upds\ s\ fs\ s1\ is_upds\ s\ fs\ s2$
 $\langle proof \rangle$

lemma *check_bexp_bval*:

$\forall x \in vars_of_bexp\ b. x \in dom\ s \Longrightarrow check_bexp\ s\ b\ (bval\ (the\ o\ s)\ b)$

and *is_val_eval*:

$\forall x \in vars_of_exp\ e. x \in dom\ s \Longrightarrow is_val\ s\ e\ (eval\ (the\ o\ s)\ e)$
 $\langle proof \rangle$

lemma *is_upd_dom*:

$dom\ s' = dom\ s$ **if** $is_upd\ s\ (x, e)\ s'\ x \in dom\ s$
 $\langle proof \rangle$

lemma *is_upds_dom*:

$dom\ s' = dom\ s$ **if** $is_upds\ s\ upds\ s' \forall (x, e) \in set\ upds.\ x \in dom\ s$
 $\langle proof \rangle$

definition

$mk_upd \equiv \lambda(x, e)\ s.\ s(x \mapsto eval\ (the\ o\ s)\ e)$

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** $upd = (x, e)\ \forall x \in vars_of_exp\ e.\ x \in dom\ s$
 $\langle proof \rangle$

lemma *dom_mk_upd*:

$dom\ s \subseteq dom\ (mk_upd\ upd\ s)$
 $\langle proof \rangle$

lemma *is_upds_make_updsI*:

$is_upds\ s\ upds\ (mk_upds\ s\ upds)$ **if** $\forall (_, e) \in set\ upds.\ \forall x \in vars_of_exp\ e.\ x \in dom\ s$
 $\langle proof \rangle$

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*:

$union_map_of\ xs\ x = (let$
 $\quad ys = rev\ (map\ snd\ (filter\ (\lambda\ (x', y).\ x' = x)\ xs))$
 $\quad in\ if\ ys = []\ then\ None\ else\ Some\ ys)$
 $\langle proof \rangle$

lemma *in_union_map_ofI*:

$\exists\ ys.\ union_map_of\ xs\ x = Some\ ys \wedge y \in set\ ys$ **if** $(x, y) \in set\ xs$
 $\langle proof \rangle$

lemma *in_union_map_ofD*:

$(x, y) \in set\ xs$ **if** $union_map_of\ xs\ x = Some\ ys\ y \in set\ ys$
 $\langle proof \rangle$

Implementation of state set context *Simple_Network_Impl_nat_defs*
begin

definition

$states_i\ i = (\bigcup (l, e, g, a, r, u, l') \in set\ (fst\ (snd\ (automata\ !\ i))).\ \{l, l'\})$

lemma *states_mem_compute*[code]:

$L \in \text{states} \longleftrightarrow \text{length } L = n_ps \wedge (\forall i < n_ps. L ! i \in \text{states_}i \ i)$
 $\langle \text{proof} \rangle$

lemma *states_mem_compute'*:

$L \in \text{states} \longleftrightarrow \text{length } L = n_ps \wedge (\forall i < n_ps. L ! i \in \text{map states_}i \ [0..<n_ps] ! i)$
 $\langle \text{proof} \rangle$

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 $i < n_ps$
 $\langle \text{proof} \rangle$

lemma *L_i_len*:

$\langle L ! i < \text{num_states } i \rangle$ **if** $i < n_ps$ $L \in \text{states}$
 $\langle \text{proof} \rangle$

lemma *L_i_simp*:

$\langle [0..<\text{num_states } i] ! (L ! i) = L ! i \rangle$
if $i < n_ps$ $L \in \text{states}$
 $\langle \text{proof} \rangle$

lemma *action_setD*:

$\langle \text{pred_act } (\lambda a'. a' < \text{num_actions}) \ a \rangle$
if $\langle (l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N \ p) \rangle$ $\langle p < n_ps \rangle$
 $\langle \text{proof} \rangle$

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** $(L, s) \in \text{states}'$
 $\langle \text{proof} \rangle$

lemma *states'_dom[intro, dest]*:

$\text{dom } s = \{0..<n_vs\}$ **if** $(L, s) \in \text{states}'$
 $\langle \text{proof} \rangle$

lemma *states'_bounded[intro, dest]*:

$\text{bounded bounds } s$ **if** $(L, s) \in \text{states}'$
 $\langle \text{proof} \rangle$

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*)

$invs1 \equiv \text{map } (\lambda i. \text{let } m = \text{default_map_of } [] \text{ (snd (snd (snd (automata ! i))))});$
 $m' = \text{map } (\lambda j. m j) [0..<\text{num_states } i]$
 $\text{in } m') [0..<n_ps]$

definition (in *Simple_Network_Impl_nat_defs*)
 $invs2 \equiv \text{IArray (map } (\lambda i. \text{let } m = \text{default_map_of } [] \text{ (snd (snd (snd (automata ! i))))});$
 $m' = \text{IArray (map } (\lambda j. m j) [0..<\text{num_states } i])$
 $\text{in } m') [0..<n_ps]$

lemma *refine_invs2*:
 $invs2 !! i !! j = invs1 ! i ! j \text{ if } i < n_ps$
 $\langle \text{proof} \rangle$

definition (in *Simple_Network_Impl_nat_defs*)
 $inv_fun \equiv \lambda(L, _).$
 $\text{concat (map } (\lambda i. invs1 ! i ! (L ! i)) [0..<n_ps])$

lemma *refine_invs1*:
 $\langle invs1 ! i = invs i \rangle \text{ if } \langle i < n_ps \rangle$
 $\langle \text{proof} \rangle$

lemma *invs_simp*:
 $invs1 ! i ! (L ! i) = \text{Simple_Network_Language.inv } (N i) (L ! i)$
 $\text{if } i < n_ps \text{ } L \in \text{states}$
 $\langle \text{proof} \rangle$

lemma *inv_fun_inv_of'*:
 $(inv_fun, inv_of \text{ prod_ta}) \in inv_rel \text{ } R \text{ states' if } R \subseteq Id \times_r S$
 $\langle \text{proof} \rangle$

lemma *inv_fun_alt_def*:
 $inv_fun = (\lambda(L, _). \text{concat (map } (\lambda i. invs2 !! i !! (L ! i)) [0..<n_ps]))$
 $\langle \text{proof} \rangle$

end

Implementation of transitions **context** *Simple_Network_Impl_nat_defs*
begin

definition
 $\text{bounds_map} \equiv \text{the o map_of bounds'}$

definition
 $\text{check_bounded } s =$
 $(\forall x \in \text{dom } s. \text{fst (bounds_map } x) \leq \text{the } (s \text{ } x) \wedge \text{the } (s \text{ } x) \leq \text{snd (bounds_map } x))$

Compute pairs of processes with committed initial locations from location vector.

definition
 $\text{get_committed } L =$
 $\text{List.map_filter } (\lambda p.$
 $\text{let } l = L ! p \text{ in}$
 $\text{if } l \in \text{set (fst (automata ! p)) then Some (p, l) else None) [0..<n_ps]$

Given a process and a location, return the corresponding transitions.

definition

```

trans_map i ≡
  let m = union_map_of (fst (snd (snd (automata ! i)))) in (λj.
    case m j of None ⇒ [] | Some xs ⇒ xs)

```

Filter for internal transitions.

definition

```

trans_i_map i j ≡
  List.map_filter
    (λ (b, g, a, m, l'). case a of Sil a ⇒ Some (b, g, a, m, l') | _ ⇒ 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 ≡

```

$\text{fold } (\lambda i. \text{actions_by_state } i \ (t \ i \ (L \ ! \ i))) \ [0..<n_ps] \ (\text{repeat } [] \ \text{num_actions})$

definition

$\text{all_actions_from_vec } t \ \text{vec} \equiv$
 $\text{fold } (\lambda(p, l). \text{actions_by_state } p \ (t \ p \ l)) \ \text{vec} \ (\text{repeat } [] \ \text{num_actions})$

definition

$\text{pairs_by_action } L \ s \ OUT \ IN \equiv$
 $\text{concat } ($
 $\text{map } (\lambda (p, b1, g1, a1, f1, r1, l1).$
 $\text{List.map_filter } (\lambda (q, b2, g2, a2, f2, r2, l2).$
 $\text{if } p = q \text{ then } None \text{ else}$
 $\text{let } s' = \text{mk_upds } (\text{mk_upds } s \ f1) \ f2 \text{ in}$
 $\text{if } \text{bval } (\text{the } o \ s) \ b1 \wedge \text{bval } (\text{the } o \ s) \ b2 \wedge \text{check_bounded } s'$
 $\text{then } Some \ (g1 \ @ \ g2, Bin \ a1, r1 \ @ \ r2, (L[p := l1, q := l2], s'))$
 $\text{else } None$
 $) \ OUT) \ IN)$

definition

$\text{trans_in_map } i \ j \equiv$
 List.map_filter
 $(\lambda (b, g, a, m, l'). \text{case } a \text{ of } In \ a \Rightarrow Some \ (b, g, a, m, l') \mid _ \Rightarrow None)$
 $(\text{trans_map } i \ j)$

definition

$\text{trans_out_map } i \ j \equiv$
 List.map_filter
 $(\lambda (b, g, a, m, l'). \text{case } a \text{ of } Out \ a \Rightarrow Some \ (b, g, a, m, l') \mid _ \Rightarrow None)$
 $(\text{trans_map } i \ j)$

definition

$\text{bin_actions} = \text{filter } (\lambda a. a \notin \text{set broadcast}) \ [0..<\text{num_actions}]$

lemma $\text{mem_bin_actions_iff}$:

$a \in \text{set bin_actions} \longleftrightarrow a \notin \text{local.broadcast} \wedge a < \text{num_actions}$
 $\langle \text{proof} \rangle$

definition

$\text{bin_trans_from} \equiv \lambda(L, s).$
 let
 $\text{pairs} = \text{get_committed } L;$
 $In = \text{all_actions_by_state } \text{trans_in_map } L;$
 $Out = \text{all_actions_by_state } \text{trans_out_map } L$
 in
 $\text{if } \text{pairs} = [] \text{ then}$
 $\text{concat } (\text{map } (\lambda a. \text{pairs_by_action } L \ s \ (Out \ ! \ a) \ (In \ ! \ a)) \ \text{bin_actions})$
 else
 let
 $In2 = \text{all_actions_from_vec } \text{trans_in_map } \text{pairs};$
 $Out2 = \text{all_actions_from_vec } \text{trans_out_map } \text{pairs}$
 in
 $\text{concat } (\text{map } (\lambda a. \text{pairs_by_action } L \ s \ (Out \ ! \ a) \ (In2 \ ! \ a)) \ \text{bin_actions})$

@ 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 ≡ actions_by_state' (trans_out_broad_map i j)

definition

trans_in_broad_grouped i j ≡ actions_by_state' (trans_in_broad_map i j)

definition

pair_by_action OUT IN ≡
 concat (
 map (λ(g1, a, r1, (L, s)).
 List.map (λ(q, g2, a2, f2, r2, l2).
 (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
) OUT) IN)

definition make_combs where

make_combs p a xs ≡
 let
 ys = List.map_filter
 (λi.
 if i = p then None
 else if xs ! i ! a = [] then None
 else Some (map (λt. (i, t)) (xs ! i ! a))
)
 [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 ≡
 let
 ys = List.map_filter

```

(λi.
  if i = p then None
  else if xs ! i ! a = [] then None
  else Some (map (λt. (i, t)) (xs ! i ! a))
)
[0..<n_ps]
in if ys = [] then [] else product_lists ys

```

definition

```

broad_trans_from ≡ 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, _). bval (the o s) b))) In;
  Out = map (map (filter (λ (b, _). bval (the o 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
              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..<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;

```

```

    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
      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..

```

end

lemma *product_lists_empty*: $\text{product_lists } xss = [] \longleftrightarrow (\exists xs \in \text{set } xss. xs = [])$ **for** xss
 ⟨proof⟩

context *Simple_Network_Impl_nat*
begin

lemma *broad_trans_from_alt_def*:
 $\text{broad_trans_from} \equiv \lambda(L, s).$
 let
 pairs = get_committed L;
 In = map (λp. trans_in_broad_grouped p (L ! p)) [0..
 Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..
 In = map (map (filter (λ (b, _). bval (the o s) b))) In;
 Out = map (map (filter (λ (b, _). bval (the o 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
        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..

```

$$\begin{aligned}
& (g1 \text{ @ } g2, a, r1 \text{ @ } r2, (L[q := l2], mk_upds \ s \ f2)) \\
&) \\
& \text{comb} \\
& \text{init} \\
&) \text{ combs}) \\
&) \\
& [0..<n_ps]) \\
&) \\
& [0..<num_actions])
\end{aligned}$$

$\langle proof \rangle$

Correctness for implementations of primitives lemma *dom_bounds_eq*:

$dom \ bounds = \{0..<n_vs\}$

$\langle proof \rangle$

lemma *check_bounded_iff*:

$Simple_Network_Language.bounded \ bounds \ s \longleftrightarrow check_bounded \ s \text{ if } dom \ s = \{0..<n_vs\}$

$\langle proof \rangle$

lemma *get_committed_mem_iff*:

$(p, l) \in set \ (get_committed \ L) \longleftrightarrow (l = L \ ! \ p \wedge l \in committed \ (N \ p) \wedge p < n_ps)$

$\langle proof \rangle$

lemma *get_committed_empty_iff*:

$(\forall p < n_ps. L \ ! \ p \notin committed \ (N \ p)) \longleftrightarrow get_committed \ L = []$

$\langle proof \rangle$

lemma *get_committed_distinct*: $distinct \ (get_committed \ L)$

$\langle proof \rangle$

lemma *is_upds_make_updsI2*:

$is_upds \ s \ upds \ (mk_upds \ s \ upds)$

if $(l, b, g, a, upds, r, l') \in Simple_Network_Language.trans \ (N \ p) \ p < n_ps$

$dom \ s = \{0..<n_vs\}$

$\langle proof \rangle$

lemma *var_setD*:

$\forall (x, upd) \in set \ f. x < n_vs \wedge (\forall i \in vars_of_exp \ upd. i < n_vs)$

if $(l, b, g, a, f, r, l') \in Simple_Network_Language.trans \ (N \ p) \ p < n_ps$

$\langle proof \rangle$

lemma *var_setD2*:

$\forall i \in vars_of_bexp \ b. i < n_vs$

if $(l, b, g, a, f, r, l') \in Simple_Network_Language.trans \ (N \ p) \ p < n_ps$

$\langle proof \rangle$

lemma *is_upds_dom2*:

$dom \ s' = \{0..<n_vs\} \text{ if } is_upds \ s \ f \ s'$

$(l, b, g, a, f, r, l') \in Simple_Network_Language.trans \ (N \ p) \ p < n_ps$

$dom \ s = \{0..<n_vs\}$

$\langle proof \rangle$

lemma *is_updsD*:

$s' = mk_upds\ s\ f$ **if** $is_upds\ s\ f\ s'$
 $(l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)\ p < n_ps$
 $dom\ s = \{0..<n_vs\}$
 $\langle proof \rangle$

lemma *is_upds_make_upds_concatI2*:

$is_upds\ s\ (concat\ upds)\ (mk_upds\ s\ (concat\ upds))$
if $dom\ s = \{0..<n_vs\}$
 $\forall upd \in set\ upds. \exists p < n_ps. \exists l\ b\ g\ a\ r\ l'.$
 $(l, b, g, a, upd, r, l') \in Simple_Network_Language.trans\ (N\ p)$
 $\langle proof \rangle$

lemma *is_upds_concat_dom2*:

assumes $is_upds\ s\ (concat\ upds)\ s'$
and $\forall f \in set\ upds. \exists p < n_ps. \exists l\ b\ g\ a\ r\ l'.$
 $(l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)$
and $dom\ s = \{0..<n_vs\}$
shows $dom\ s' = dom\ s$
 $\langle proof \rangle$

lemma *is_upds_dom3*:

assumes $is_upds\ s\ (concat_map\ fs\ ps)\ s'$
and $\forall p \in set\ ps. (L\ !\ p, bs\ p, gs\ p, a, fs\ p, rs\ p, ls'\ p) \in trans\ (N\ p)$
and $set\ ps \subseteq \{0..<n_ps\}$
and $dom\ s = \{0..<n_vs\}$
shows $dom\ s' = dom\ s$
 $\langle proof \rangle$

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 $\forall p \in set\ ps. (L\ !\ p, bs\ p, gs\ p, a, fs\ p, rs\ p, ls'\ p) \in trans\ (N\ p)$
and $set\ ps \subseteq \{0..<n_ps\}$
for $s :: nat \Rightarrow int\ option$
 $\langle proof \rangle$

lemma *is_upds_concatD*:

assumes
 $dom\ s = \{0..<n_vs\}$ **and**
 $\forall p \in set\ ps.$
 $(ls\ p, bs\ p, gs\ p, a, fs\ p, rs\ p, ls'\ p)$
 $\in Simple_Network_Language.trans\ (N\ p)$ **and**
 $set\ ps \subseteq \{0..<n_ps\}$ **and**
 $is_upds\ s\ (concat_map\ fs\ ps)\ s'$
shows $s' = mk_upds\ s\ (concat_map\ fs\ ps)$
 $\langle proof \rangle$

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

$(g, a, f, r, l') \in \text{set } (\text{trans_map } p (L ! p))$
 $\langle \text{proof} \rangle$

lemma *trans_i_mapI*:

assumes

$(L ! p, b, g, \text{Sil } a', f, r, l') \in \text{Simple_Network_Language.trans } (N p)$
 $p < n_ps$

shows

$(b, g, a', f, r, l') \in \text{set } (\text{trans_i_map } p (L ! p))$
 $\langle \text{proof} \rangle$

lemma *trans_mapI'*:

assumes

$(l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N p)$
 $p < n_ps$

shows

$(b, g, a, f, r, l') \in \text{set } (\text{trans_map } p l)$
 $\langle \text{proof} \rangle$

lemma *trans_mapD*:

assumes

$(b, g, a, f, r, l') \in \text{set } (\text{trans_map } p l)$
 $p < n_ps$

shows

$(l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N p)$
 $\langle \text{proof} \rangle$

lemma *trans_map_iff*:

assumes

$p < n_ps$

shows

$(b, g, a, f, r, l') \in \text{set } (\text{trans_map } p l)$
 $\longleftrightarrow (l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N p)$
 $\langle \text{proof} \rangle$

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

$(L ! p, b, g, \text{Sil } a', f, r, l') \in \text{Simple_Network_Language.trans } (N p)$
 $\langle \text{proof} \rangle$

An additional brute force method for forward-chaining of facts **method** *frules_all* =
repeat_rotate $\langle \text{frules} \rangle$, *dedup_premis*

Internal transitions **lemma** *get_committed_memI*:

$(p, L ! p) \in \text{set } (\text{get_committed } L)$ **if** $L ! p \in \text{committed } (N p)$ $p < n_ps$

$\langle \text{proof} \rangle$

lemma *check_bexp_bvalI*:

bval (*the o s*) *b* **if** *check_bexp s b* *True*

$(l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N\ p)\ p < n_ps$

dom s = $\{0..<n_vs\}$

$\langle \text{proof} \rangle$

lemma *check_bexp_bvalD*:

check_bexp s b *True* **if** *bval* (*the o s*) *b*

$(l, b, g, a, f, r, l') \in \text{Simple_Network_Language.trans } (N\ p)\ p < n_ps$

dom s = $\{0..<n_vs\}$

$\langle \text{proof} \rangle$

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*:

$(\text{int_trans_from}, \text{trans_int}) \in \text{transition_rel states'}$

$\langle \text{proof} \rangle$

Mostly copied lemma *in_actions_by_stateI*:

assumes

$(b1, g1, a, r1) \in \text{set } xs\ a < \text{length } acc$

shows

$(q, b1, g1, a, r1) \in \text{set } (\text{actions_by_state } q\ xs\ acc\ !\ a)$

$\wedge a < \text{length } (\text{actions_by_state } q\ xs\ acc)$

$\langle \text{proof} \rangle$

lemma *in_actions_by_state'I*:

assumes

$(b1, g1, a, r1) \in \text{set } xs\ a < \text{num_actions}$

shows

$(b1, g1, a, r1) \in \text{set } (\text{actions_by_state' } xs\ !\ a)$

$\wedge a < \text{length } (\text{actions_by_state' } xs)$

$\langle \text{proof} \rangle$

lemma *in_actions_by_state_preserv*:

assumes

$(q, b1, g1, a, r1) \in \text{set } (acc\ !\ a)\ a < \text{length } acc$

shows

$(q, b1, g1, a, r1) \in \text{set } (\text{actions_by_state } y\ xs\ acc\ !\ a)$

$\wedge a < \text{length } (\text{actions_by_state } y\ xs\ acc)$

$\langle \text{proof} \rangle$

lemma *length_actions_by_state_preserv*[simp]:
shows $\text{length } (\text{actions_by_state } y \text{ } xs \text{ } acc) = \text{length } acc$
 $\langle \text{proof} \rangle$

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
 $(q, b1, g1, a, r1) \in \text{set } (\text{all_actions_by_state } M \ L \ ! \ a)$
 $\langle \text{proof} \rangle$

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
 $(q, b1, g1, a, r1) \in \text{set } (\text{all_actions_from_vec } M \ \text{pairs} \ ! \ a)$
 $\langle \text{proof} \rangle$

lemma *actions_by_state_inj*:
assumes $j < \text{length } acc$
shows $\forall \ (q, a) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc \ ! \ j). \ (q, a) \notin \text{set } (acc \ ! \ j) \longrightarrow i = q$
 $\langle \text{proof} \rangle$

lemma *actions_by_state_inj'*:
assumes $j < \text{length } acc \ (q, a) \notin \text{set } (acc \ ! \ j) \ (q, a) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc \ ! \ j)$
shows $i = q$
 $\langle \text{proof} \rangle$

lemma *in_actions_by_stateD*:
assumes
 $(q, b, g, a, t) \in \text{set } (\text{actions_by_state } i \text{ } xs \text{ } acc \ ! \ j) \ (q, b, g, a, t) \notin \text{set } (acc \ ! \ j)$
 $j < \text{length } acc$
shows
 $(b, g, a, t) \in \text{set } xs \wedge j = a$
 $\langle \text{proof} \rangle$

lemma *in_actions_by_state'D*:
assumes
 $(b, g, a, r) \in \text{set } (\text{actions_by_state}' \text{ } xs \ ! \ a') \ a' < \text{num_actions}$
shows
 $(b, g, a, r) \in \text{set } xs \wedge a' = a$
 $\langle \text{proof} \rangle$

lemma *in_all_actions_by_stateD*:
assumes
 $(q, b1, g1, a, r1) \in \text{set } (\text{all_actions_by_state } M \ L \ ! \ a') \ a' < \text{num_actions}$
shows
 $(b1, g1, a, r1) \in \text{set } (M \ q \ (L \ ! \ q)) \wedge q < n_ps \wedge a' = a$
 $\langle \text{proof} \rangle$

lemma *length_all_actions_by_state_preserv*:
 $\text{length } (\text{all_actions_by_state } M \ L) = \text{num_actions}$
 $\langle \text{proof} \rangle$

lemma *length_actions_by_state'_preserv*:
 $\text{length } (\text{actions_by_state}' \text{ } xs) = \text{num_actions}$
 $\langle \text{proof} \rangle$

lemma *all_actions_from_vecD*:
assumes
 $(q, b1, g1, a, r1) \in \text{set } (\text{all_actions_from_vec } M \text{ pairs } ! a') \ a' < \text{num_actions}$
 $\text{distinct } (\text{map } \text{fst } \text{pairs})$
shows
 $\exists l. (q, l) \in \text{set pairs} \wedge (b1, g1, a, r1) \in \text{set } (M \ q \ l) \wedge a' = a$
 $\langle \text{proof} \rangle$

lemma *all_actions_from_vecD2*:
assumes
 $(q, b1, g1, a, r1) \in \text{set } (\text{all_actions_from_vec } M \text{ pairs } ! a') \ a' < \text{num_actions}$
 $(q, l) \in \text{set pairs} \ \text{distinct } (\text{map } \text{fst } \text{pairs})$
shows
 $(b1, g1, a, r1) \in \text{set } (M \ q \ l) \wedge a' = a$
 $\langle \text{proof} \rangle$

Binary transitions **lemma** *bin_trans_from_correct*:
 $(\text{bin_trans_from}, \text{trans_bin}) \in \text{transition_rel } \text{states}'$
 $\langle \text{proof} \rangle$

Broadcast transitions **lemma** *make_combs_alt_def*:
 $\text{make_combs } p \ a \ xs \equiv$
 let
 $\quad ys =$
 $\quad \text{map } (\lambda i. \text{map } (\lambda t. (i, t)) (xs ! i ! a))$
 $\quad (\text{filter}$
 $\quad \quad (\lambda i. xs ! i ! a \neq [] \wedge i \neq p)$
 $\quad \quad [0..<n_ps])$
 $\quad \text{in if } ys = [] \text{ then } [] \text{ else product_lists } ys$
 $\langle \text{proof} \rangle$

lemma *list_all2_fst_aux*:
 $\text{map } \text{fst } xs = ys \ \text{if } \text{list_all2 } (\lambda x y. \text{fst } x = y) \ xs \ ys$
 $\langle \text{proof} \rangle$

lemma *broad_trans_from_correct*:
 $(\text{broad_trans_from}, \text{trans_broad}) \in \text{transition_rel } \text{states}'$
 $\langle \text{proof} \rangle$

Refinement of the State Implementation **definition** *state_rel* :: $(\text{nat} \rightarrow \text{int}) \Rightarrow \text{int list} \Rightarrow \text{bool}$
where
 $\text{state_rel } s \ xs \equiv \text{length } xs = n_vs \wedge \text{dom } s = \{0..<n_vs\} \wedge (\forall i < n_vs. xs ! i = \text{the } (s \ i))$

definition *loc_rel* **where**
 $\text{loc_rel} \equiv \{((L', s'), (L, s)) \mid L \ s \ L' \ s'. L' = L \wedge \text{length } L = n_ps \wedge \text{state_rel } s \ s'\}$

lemma *state_impl_abstract*:

$\exists L s. ((Li, si), (L, s)) \in loc_rel$ **if** $length\ Li = n_ps$ $length\ si = n_vs$
 $\langle proof \rangle$

lemma *state_rel_left_unique*:

$l \in states' \implies (li, l) \in loc_rel \implies (li', l) \in loc_rel \implies li' = li$
 $\langle proof \rangle$

lemma *state_rel_right_unique*:

$l \in states' \implies l' \in states' \implies (li, l) \in loc_rel \implies (li, l') \in loc_rel \implies l' = l$
 $\langle proof \rangle$

end

end

fun *bvali* :: $_ \Rightarrow (nat, 'b :: linorder)$ *bexp* $\Rightarrow bool$ **and** *evali* **where**

bvali *s* *bexp.true* = *True* |
bvali *s* (*not* *e*) $\longleftrightarrow \neg$ *bvali* *s* *e* |
bvali *s* (*and* *e1* *e2*) $\longleftrightarrow$ *bvali* *s* *e1* $\wedge$ *bvali* *s* *e2* |
bvali *s* (*bexp.or* *e1* *e2*) $\longleftrightarrow$ *bvali* *s* *e1* $\vee$ *bvali* *s* *e2* |
bvali *s* (*imply* *e1* *e2*) $\longleftrightarrow$ *bvali* *s* *e1* $\longrightarrow$ *bvali* *s* *e2* |
bvali *s* (*eq* *i* *x*) $\longleftrightarrow$ *evali* *s* *i* = *evali* *s* *x* |
bvali *s* (*le* *i* *x*) $\longleftrightarrow$ *evali* *s* *i* $\leq$ *evali* *s* *x* |
bvali *s* (*lt* *i* *x*) $\longleftrightarrow$ *evali* *s* *i* < *evali* *s* *x* |
bvali *s* (*ge* *i* *x*) $\longleftrightarrow$ *evali* *s* *i* $\geq$ *evali* *s* *x* |
bvali *s* (*gt* *i* *x*) $\longleftrightarrow$ *evali* *s* *i* > *evali* *s* *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* ::

$int\ list \Rightarrow (nat \times (nat, int)\ exp)\ list \Rightarrow int\ list$ **where**
mk_updsi *s* *upds* = *fold* ($\lambda(x, upd)\ s. s[x := evali\ s\ upd]$) *upds* *s*

context *Simple_Network_Impl_nat_defs*

begin

definition

check_boundedi *s* =
 $(\forall x < length\ s. fst\ (bounds_map\ x) \leq s\ !\ x \wedge s\ !\ x \leq snd\ (bounds_map\ x))$

definition

states'_memi $\equiv \lambda(L, s). L \in states \wedge length\ s = n_vs \wedge check_boundedi\ s$

definition

int_trans_from_loc_impl *p* *l* *L* *s* $\equiv$
 $let\ trans = trans_i_map\ p\ l$
 in
 $List.map_filter\ (\lambda\ (b, g, a, f, r, l').$
 $let\ s' = mk_updsi\ s\ f\ in$
 $if\ bvali\ s\ b \wedge check_boundedi\ s'\ then\ Some\ (g, Internal\ a, r, (L[p := l'], s'))$
 $else\ None$

) *trans*

definition

int_trans_from_vec_impl pairs L s $\equiv$
concat (*map* ($\lambda(p, l). \text{int_trans_from_loc_impl } p \text{ } l \text{ } L \text{ } s$) *pairs*)

definition

int_trans_from_all_impl L s $\equiv$
concat (*map* ($\lambda p. \text{int_trans_from_loc_impl } p \text{ } (L \text{ ! } p) \text{ } L \text{ } s$) $[0..<n_ps]$)

definition

int_trans_impl $\equiv \lambda (L, s).$
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 $\equiv$
concat (
map ($\lambda (p, b1, g1, a1, f1, r1, l1).$
List.map_filter ($\lambda (q, b2, g2, a2, f2, r2, l2).$
if p = q then None else
let s' = mk_updsi (mk_updsi s f1) f2 in
if bvali s b1 $\wedge$ bvali s b2 $\wedge$ 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 $\equiv \lambda(L, s).$
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* ($\lambda a. \text{pairs_by_action_impl } L \text{ } s \text{ } (Out \text{ ! } a) \text{ } (In \text{ ! } a)$) *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* ($\lambda a. \text{pairs_by_action_impl } L \text{ } s \text{ } (Out \text{ ! } a) \text{ } (In2 \text{ ! } a)$) *bin_actions*)
 @ *concat* (*map* ($\lambda a. \text{pairs_by_action_impl } L \text{ } s \text{ } (Out2 \text{ ! } a) \text{ } (In \text{ ! } a)$) *bin_actions*)

definition

compute_upds_init $\equiv \text{List.map_filter } (\lambda \text{comb.}$
let (*g*, *a*, *r*, *L'*, *s*) =
fold
 ($\lambda(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).$

```

      (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 ⇒ ∀ x ∈ vars_of_bexp b. x ∈ dom s ⇒ bval (the o s) b = bvali si b

and *eval_evali:*

state_rel s si ⇒ ∀ x ∈ vars_of_exp e. x ∈ dom s ⇒ eval (the o s) e = evali si e
 ⟨proof⟩

lemma *mk_upds_mk_updsi:*

state_rel (mk_upds s upds) (mk_updsi si upds)

if *assms:* *state_rel s si* *∀ (__, e) ∈ set upds. ∀ x ∈ vars_of_exp e. x < n_vs*
∀ (x, e) ∈ set upds. x < n_vs

⟨proof⟩

lemma *check_bounded_check_boundedi:*

check_bounded s = check_boundedi si **if** *state_rel s si*

⟨proof⟩

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*]:

$is_at_least_equality\ (=)$

$\langle proof \rangle$

lemma [*is_at_least_equality*]:

$is_at_least_equality\ R$ **if** $is_equality\ R$ **for** R

$\langle proof \rangle$

lemma [*is_at_least_equality*]:

$is_at_least_equality\ (eq_onp\ P)$

$\langle proof \rangle$

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

$\langle proof \rangle$

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** $R1\ R2$

$\langle proof \rangle$

lemma *is_at_least_equality_cong1*:

$(S1\ ===>\ (=))\ f\ f$ **if** $is_at_least_equality\ S1$ $is_at_least_equality\ S2$ **for** $S1\ f$

$\langle proof \rangle$

lemma *is_at_least_equality_cong2*:

$(S1\ ===>\ S2\ ===>\ (=))\ f\ f$ **if** $is_at_least_equality\ S1$ $is_at_least_equality\ S2$ **for** $S1\ S2\ f$

$\langle proof \rangle$

lemma *is_at_least_equality_cong3*:

$(S1\ ===>\ S2\ ===>\ S3\ ===>\ (=))\ f\ f$

if $is_at_least_equality\ S1$ $is_at_least_equality\ S2$ $is_at_least_equality\ S3$ **for** $S1\ S2\ S3\ f$

$\langle proof \rangle$

lemma *is_at_least_equality_Let*:

$(S\ ===>\ ((=)\ ===>\ R)\ ===>\ R)$ *Let Let* **if** $is_at_least_equality\ S$ **for** R

$\langle proof \rangle$

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
 $\langle \text{proof} \rangle$

lemma upt_0_transfer :

$(\text{eq_onp}\ (\lambda x. x = 0) \implies \text{eq_onp}\ (\lambda x. x = n) \implies \text{list_all2}\ (\text{eq_onp}\ (\lambda x. x < n)))\ \text{upt}$
 $\text{upt for } n$
 $\langle \text{proof} \rangle$

lemma $\text{upt_length_transfer}$:

$(\text{eq_onp}\ (\lambda x. x = 0) \implies \text{eq_onp}\ (\lambda x. x = n)$
 $\implies (\lambda x\ y. \text{list_all2}\ (\text{eq_onp}\ (\lambda x. x < n))\ x\ y \wedge \text{length}\ x = n))\ \text{upt upt for } n$
 $\langle \text{proof} \rangle$

lemma $\text{case_prod_transfer_strong}$:

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 $((A \implies B \implies C) \implies A\ \times_R\ B\ \implies C)\ \text{case_prod case_prod}$
 $\langle \text{proof} \rangle$

lemma $\text{concat_transfer_strong}$:

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 $(C \implies \text{list_all2}\ B)\ \text{concat concat}$
 $\langle \text{proof} \rangle$

lemma $\text{map_transfer_strong}$:

fixes $A\ B\ C$
assumes $\bigwedge xs\ ys. C\ xs\ ys \implies \text{list_all2}\ A\ xs\ ys$
shows $((A \implies B) \implies C \implies \text{list_all2}\ B)\ \text{map map}$
 $\langle \text{proof} \rangle$

lemma $\text{list_update_transfer}'$:

fixes $A :: 'a \Rightarrow 'b \Rightarrow \text{bool}$
shows $(\text{list_all2}\ A \implies \text{eq_onp}\ (\lambda i. i < n_ps) \implies A \implies \text{list_all2}\ A)\ \text{list_update}$
 list_update
 $\langle \text{proof} \rangle$

lemma $\text{list_update_transfer}''$:

fixes $A :: 'a \Rightarrow 'b \Rightarrow \text{bool and } n$
shows $((\lambda x\ y. \text{list_all2}\ A\ x\ y \wedge \text{length}\ x = n) \implies \text{eq_onp}\ (\lambda i. i < n) \implies A$
 $\implies (\lambda x\ y. \text{list_all2}\ A\ x\ y \wedge \text{length}\ x = n))\ \text{list_update list_update}$
 $\langle \text{proof} \rangle$

lemma $\text{list_update_transfer}'''$:

fixes $A :: 'a \Rightarrow 'b \Rightarrow \text{bool and } n$
shows $((\lambda x\ y. \text{list_all2}\ A\ x\ y \wedge \text{length}\ x = n) \implies (=) \implies A$
 $\implies (\lambda x\ y. \text{list_all2}\ A\ x\ y \wedge \text{length}\ x = n))\ \text{list_update list_update}$
 $\langle \text{proof} \rangle$

lemma *fold_transfer_strong*:

fixes $A B$

assumes $\bigwedge x y. A1\ x\ y \implies A\ x\ y \bigwedge x y. B1\ x\ y \implies B\ x\ y \bigwedge x y. B\ x\ y \implies B2\ x\ y$

$\bigwedge x y. B\ x\ y \implies B3\ x\ y$

shows $((A \implies B2 \implies B1) \implies list_all2\ A1 \implies B \implies B3)\ fold\ fold$

$\langle proof \rangle$

lemma *bval_bval_transfer[transfer_rule]*:

$(state_rel \implies eq_onp\ valid_check \implies (=))\ (\lambda s. bval\ (the\ o\ s))\ bvali$

$\langle proof \rangle$

lemma *mk_upds_mk_updsi_transfer[transfer_rule]*:

$(state_rel \implies list_all2\ (eq_onp\ valid_upd) \implies state_rel)\ mk_upds\ mk_updsi$

$\langle proof \rangle$

lemma *check_bounded_transfer[transfer_rule]*:

$(state_rel \implies (=))\ check_bounded\ check_boundedi$

$\langle proof \rangle$

lemma *trans_map_transfer*:

$(eq_onp\ (\lambda i. i < n_ps) \implies (=) \implies$

$list_all2\ ($

$eq_onp\ valid_check \times_R (=) \times_R eq_onp\ (pred_act\ (\lambda x. x < num_actions))$

$\times_R list_all2\ (eq_onp\ valid_upd) \times_R (=)$

$))\ trans_map\ trans_map$

$\langle proof \rangle$

lemma *trans_map_transfer'*:

$(eq_onp\ (\lambda i. i < n_ps) \implies (=) \implies$

$list_all2\ (eq_onp\ valid_check \times_R (=) \times_R (=) \times_R list_all2\ (eq_onp\ valid_upd) \times_R (=))$

$)\ trans_map\ trans_map$

$\langle proof \rangle$

lemma *map_filter_transfer[transfer_rule]*:

$((S \implies rel_option\ R) \implies list_all2\ S \implies list_all2\ R)\ List.map_filter\ List.map_filter$

$\langle proof \rangle$

lemma *trans_i_map_transfer[transfer_rule]*:

$(eq_onp\ (\lambda i. i < n_ps) \implies (=) \implies$

$list_all2\ (eq_onp\ valid_check \times_R (=) \times_R (=) \times_R list_all2\ (eq_onp\ valid_upd) \times_R (=))$

$)\ trans_i_map\ trans_i_map$

$\langle proof \rangle$

lemma *int_trans_from_loc_transfer[transfer_rule]*:

$(eq_onp\ (\lambda i. i < n_ps) \implies (=) \implies (\lambda x y. list_all2\ (=)\ x\ y \wedge length\ x = n_ps) \implies$

$state_rel$

$\implies list_all2((=) \times_R (=) \times_R (=) \times_R (\lambda x y. list_all2\ (=)\ x\ y \wedge length\ x = n_ps) \times_R$

$state_rel))$

$int_trans_from_loc\ int_trans_from_loc_impl$

$\langle proof \rangle$

lemma *n_ps_transfer*:

$eq_onp\ (\lambda x. x = n_ps)\ n_ps\ n_ps$

$\langle proof \rangle$

lemma *zero_nat_transfer*:

(=) 0 (0::nat)
 ⟨proof⟩

lemma *int_trans_from_all_transfer*[*transfer_rule*]:

((λx y. list_all2 (=) x y ∧ length x = n_ps) ==> state_rel
 ==> list_all2((=) ×_R (=) ×_R (=) ×_R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R
 state_rel))
 int_trans_from_all int_trans_from_all_impl
 ⟨proof⟩

lemma *int_trans_from_vec_transfer*[*transfer_rule*]:

(list_all2 (eq_onp (λx. x < n_ps) ×_R (=)) ==> (λx y. list_all2 (=) x y ∧ length x = n_ps)
 ==> state_rel
 ==> list_all2((=) ×_R (=) ×_R (=) ×_R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R
 state_rel))
 int_trans_from_vec int_trans_from_vec_impl
 ⟨proof⟩ **definition** *R* **where** *R* ≡ (λx y. list_all2 (=) x y ∧ length x = n_ps)

lemma *get_committed_transfer*[*transfer_rule*]:

((λx y. list_all2 (=) x y ∧ length x = n_ps) ==> list_all2 (eq_onp (λx. x < n_ps) ×_R
 (=)))
 get_committed get_committed
 ⟨proof⟩

lemma *eq_transfer*:

(list_all2 (eq_onp (λx. x < n_ps) ×_R (=)) ==> list_all2 (=) ==> (=)) (=) (=)
 ⟨proof⟩

lemma *int_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)))
 int_trans_from int_trans_impl
 ⟨proof⟩

lemma *pairs_by_action_transfer*[*transfer_rule*]:

((λx y. list_all2 (=) x y ∧ length x = n) ==> state_rel ==>
 list_all2 ((=) ×_R eq_onp valid_check ×_R (=) ×_R (=) ×_R list_all2 (eq_onp valid_upd) ×_R
 (=) ×_R (=))
 ==>
 list_all2 ((=) ×_R eq_onp valid_check ×_R (=) ×_R (=) ×_R list_all2 (eq_onp valid_upd) ×_R
 (=) ×_R (=))
 ==>
 list_all2 ((=) ×_R (=) ×_R (=) ×_R (λx y. list_all2 (=) x y ∧ length x = n) ×_R state_rel))
 pairs_by_action pairs_by_action_impl
 ⟨proof⟩

lemmas *rel_elims* =

rel_prod.cases
 rel_funD

lemmas *rel_intros* =

rel_funI

lemma *pairs_by_action_transfer'*:

(($\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n$) $\implies$ state_rel) $\implies$
 $\text{list_all2 } (B \times_R \text{eq_onp valid_check} \times_R C \times_R D \times_R \text{list_all2 } (\text{eq_onp valid_upd}) \times_R E$
 $\times_R F) \implies$
 $\text{list_all2 } (B \times_R \text{eq_onp valid_check} \times_R C \times_R D \times_R \text{list_all2 } (\text{eq_onp valid_upd}) \times_R E$
 $\times_R F) \implies$
 $\text{list_all2 } ((=) \times_R (=) \times_R (=) \times_R (\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n) \times_R \text{state_rel}))$
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 $B C D E F$
 $\langle \text{proof} \rangle$

lemma *trans_in_map_transfer*[*transfer_rule*]:

($\text{eq_onp } (\lambda i. i < n_ps) \implies (=)$)
 $\implies \text{list_all2 } ($
 $\text{eq_onp valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda a. a < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp}$
 $\text{valid_upd}) \times_R (=))$
 $\rangle \text{trans_in_map trans_in_map}$
 $\langle \text{proof} \rangle$

lemma *trans_in_map_transfer*[*transfer_rule*]:

($\text{eq_onp } (\lambda i. i < n_ps) \implies (=)$)
 $\implies \text{list_all2 } (\text{eq_onp valid_check} \times_R (=) \times_R (=) \times_R \text{list_all2 } (\text{eq_onp valid_upd}) \times_R$
 $(=))$
 $\rangle \text{trans_in_map trans_in_map}$
 $\langle \text{proof} \rangle$

lemma *trans_out_map_transfer*[*transfer_rule*]:

($\text{eq_onp } (\lambda i. i < n_ps) \implies (=)$)
 $\implies \text{list_all2 } ($
 $\text{eq_onp valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda a. a < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp}$
 $\text{valid_upd}) \times_R (=)$
 $\rangle \text{trans_out_map trans_out_map}$
 $\langle \text{proof} \rangle$

lemma *trans_out_map_transfer*[*transfer_rule*]:

($\text{eq_onp } (\lambda i. i < n_ps) \implies (=) \implies \text{list_all2 } ($
 $\text{eq_onp valid_check} \times_R (=) \times_R (=) \times_R \text{list_all2 } (\text{eq_onp valid_upd}) \times_R (=))$
 $\text{trans_out_map trans_out_map}$
 $\langle \text{proof} \rangle$

lemma *actions_by_state_transfer*[*transfer_rule*]:

($\text{eq_onp } (\lambda i. i < n_ps) \implies$
 $\text{list_all2 } ((=) \times_R (=) \times_R \text{eq_onp } (\lambda i. i < n) \times_R (=)) \implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } (\text{eq_onp } (\lambda i. i < n_ps) \times_R (=) \times_R (=) \times_R \text{eq_onp } (\lambda x. x < n)$
 $\times_R (=))) x y \wedge \text{length } x = n) \implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } (\text{eq_onp } (\lambda i. i < n_ps) \times_R (=) \times_R (=) \times_R \text{eq_onp } (\lambda x. x < n)$
 $\times_R (=))) x y \wedge \text{length } x = n)$
 $\rangle$
actions_by_state actions_by_state for n

$\langle \text{proof} \rangle$

lemma *actions_by_state_transfer'*[transfer_rule]:

(
 $\text{eq_onp } (\lambda i. i < n_ps) \implies$
 $\text{list_all2 } (\text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda i. i < n) \times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd})$
 $\times_R (=)) \implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } ($
 $\text{eq_onp } (\lambda i. i < n_ps) \times_R \text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda x. x < n)$
 $\times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=))) x y \wedge \text{length } x = n) \implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } ($
 $\text{eq_onp } (\lambda i. i < n_ps) \times_R \text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda x. x < n)$
 $\times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=))) x y \wedge \text{length } x = n)$
 $)$
actions_by_state actions_by_state
for *n*
 $\langle \text{proof} \rangle$

lemma *transfer_consts*:

$(\text{eq_onp } (\lambda x. x = \text{num_actions})) \text{ num_actions num_actions } (\text{eq_onp } (\lambda x. x = 0)) (0::\text{nat}) 0$
 $(\text{eq_onp } (\lambda x. x = n_ps)) n_ps n_ps$
 $\langle \text{proof} \rangle$

lemma *all_actions_by_state_transfer*[transfer_rule]:

(
 $(\text{eq_onp } (\lambda i. i < n_ps) \implies (=) \implies \text{list_all2 } (\text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda i. i < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=)))$
 $\implies$
 $(\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps)$
 $\implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } (\text{eq_onp } (\lambda i. i < n_ps) \times_R \text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda i. i < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=))) x y \wedge \text{length } x = \text{num_actions})$
 $)$
all_actions_by_state all_actions_by_state
 $\langle \text{proof} \rangle$

lemma *all_actions_from_vec_transfer*[transfer_rule]:

(
 $(\text{eq_onp } (\lambda i. i < n_ps) \implies (=) \implies \text{list_all2 } (\text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda i. i < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=)))$
 $\implies$
 $\text{list_all2 } (\text{eq_onp } (\lambda i. i < n_ps) \times_R (=))$
 $\implies$
 $(\lambda x y. \text{list_all2 } (\text{list_all2 } (\text{eq_onp } (\lambda i. i < n_ps) \times_R \text{eq_onp } \text{valid_check} \times_R (=) \times_R \text{eq_onp } (\lambda i. i < \text{num_actions}) \times_R \text{list_all2 } (\text{eq_onp } \text{valid_upd}) \times_R (=))) x y \wedge \text{length } x = \text{num_actions})$
 $)$
all_actions_from_vec all_actions_from_vec
 $\langle \text{proof} \rangle$

lemma *bin_actions_transfer*[transfer_rule]:

$(\text{list_all2 } (\text{eq_onp } (\lambda x. x < \text{num_actions}))) \text{ bin_actions bin_actions}$
 $\langle \text{proof} \rangle$

lemma *Let_transfer_bin_aux:*

$((\lambda x y. \text{list_all2 } (\text{list_all2 } (eq_onp (\lambda i. i < n_ps) \times_R eq_onp \text{valid_check} \times_R \text{list_all2 } (rel_acconstraint (=) (=)) \times_R eq_onp (\lambda i. i < num_actions) \times_R \text{list_all2 } (eq_onp \text{valid_upd}) \times_R \text{list_all2 } (=) \times_R (=))) x y \wedge \text{length } x = num_actions) ==>$
 $((\lambda x y. \text{list_all2 } (\text{list_all2 } ((=) \times_R eq_onp \text{valid_check} \times_R \text{list_all2 } (rel_acconstraint (=) (=)) \times_R (=) \times_R \text{list_all2 } (eq_onp \text{valid_upd}) \times_R \text{list_all2 } (=) \times_R (=))) x y \wedge \text{length } x = num_actions) ==>$
 $\text{list_all2 } (rel_acconstraint (=) (=)) \times_R rel_label (=) \times_R \text{list_all2 } (=) \times_R (\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel})) ==>$
 $\text{list_all2 } (rel_acconstraint (=) (=)) \times_R rel_label (=) \times_R \text{list_all2 } (=) \times_R (\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel}))$
Let Let
<proof>

lemma *bin_trans_from_transfer:*

$((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel} ==> \text{list_all2 } ((=) \times_R (=) \times_R (=) \times_R ((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel})))$
 $\text{bin_trans_from } \text{bin_trans_from_impl}$
<proof>

lemma *trans_map_transfer'':*

$((=) ==> (=) ==>$
 $\text{list_all2 } (eq_onp \text{valid_check} \times_R (=) \times_R eq_onp (\text{pred_act } (\lambda x. x < num_actions))) \times_R \text{list_all2 } (eq_onp \text{valid_upd}) \times_R (=))$
 $\text{trans_map } \text{trans_map}$
<proof>

lemma *compute_upds_transfer:*

$((\text{list_all2 } (=) \times_R (=) \times_R \text{list_all2 } (=) \times_R (\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n) \times_R \text{state_rel}) ==> \text{list_all2 } (\text{list_all2 } ((=) \times_R eq_onp \text{valid_check} \times_R \text{list_all2 } (=) \times_R (=) \times_R \text{list_all2 } (eq_onp \text{valid_upd}) \times_R \text{list_all2 } (=) \times_R (=))) ==>$
 $\text{list_all2 } (\text{list_all2 } (=) \times_R (=) \times_R \text{list_all2 } (=) \times_R (\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n) \times_R \text{state_rel}))$
 $\text{compute_upds } \text{compute_upds_impl}$
<proof>

lemma *in_transfer:*

$(eq_onp (\lambda x. x < num_actions) ==> (=) ==> (=)) (\in) (\in)$
<proof>

lemma *trans_in_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_in_broad_map *trans_in_broad_map*
 <proof>

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*
 <proof>

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'*
 <proof>

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*
 <proof>

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*
 <proof>

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>

```

lemma broad_trans_from_alt_def2:
  broad_trans_from = ( $\lambda(L, s).$ 
    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  $\circ$  s) b))) In;
      Out = map (map (filter ( $\lambda(b, \_).$  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 ( $\lambda x.$  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 ( $\lambda(p, \_).$  if In ! p ! a  $\neq$  [] then Some p else None) pairs;
              always_committed = (length ins_committed > 1)
            in
              concat (map ( $\lambda p.$ 
                let
                  outs = Out ! p ! a
                  in if outs = [] then []
                  else if
                     $\neg$  always_committed  $\wedge$  (ins_committed = [p]  $\vee$  ins_committed = [])
                     $\wedge$   $\neg$  list_ex ( $\lambda(q, \_).$  q = p) pairs
                  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));

```

$$\begin{aligned} & \text{init} = ([], \text{Broad } a, [], (L, s)) \\ & \text{in} \\ & \text{compute_upds init combs} \\ &) \\ & [0..<n_ps]) \\ &) \\ & [0..<num_actions]) \\ &) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *concat_length_transfer*:

$$((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n) ==> \text{list_all2 } A) \text{ concat concat for } A n$$

$$\langle \text{proof} \rangle$$

lemma *broad_trans_from_transfer*:

$$\begin{aligned} & ((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel} \\ & ==> \text{list_all2 } ((=) \times_R (=) \times_R (=) \times_R ((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \\ & \text{state_rel})) \\ & \text{broad_trans_from broad_trans_from_impl} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *trans_from_transfer*:

$$\begin{aligned} & ((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \text{state_rel} \\ & ==> \text{list_all2 } ((=) \times_R (=) \times_R (=) \times_R ((\lambda x y. \text{list_all2 } (=) x y \wedge \text{length } x = n_ps) \times_R \\ & \text{state_rel})) \\ & \text{trans_from trans_impl} \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *list_all2_swap*:

$$\text{list_all2 } S \text{ ys xs if list_all2 } R \text{ xs ys } S = (\lambda x y. R \text{ y } x) \text{ for } R S$$

$$\langle \text{proof} \rangle$$

lemma *swap_rel_prod*: $(\lambda x y. (R \times_R S) \text{ y } x) = (\lambda x y. R \text{ y } x) \times_R (\lambda x y. S \text{ y } x) \text{ for } R S$

$$\langle \text{proof} \rangle$$

lemma *swap_eq*:

$$(\lambda x y. y = x) = (=)$$

$$\langle \text{proof} \rangle$$

lemma *trans_from_refine*:

$$(\text{trans_impl}, \text{trans_from}) \in \text{fun_rel_syn loc_rel } (\text{list_rel } (Id \times_r Id \times_r Id \times_r \text{loc_rel}))$$

$$\langle \text{proof} \rangle$$

lemma *trans_from_correct*:

$$(\text{trans_from}, \text{trans_prod}) \in \text{transition_rel states'}$$

$$\langle \text{proof} \rangle$$

lemma *states'_alt_def*:

$$\text{states}' = \{(L, s). L \in \text{states} \wedge \text{bounded bounds } s\}$$

$$\langle \text{proof} \rangle$$

lemma *states'_alt_def2*:

$$\text{states}' = \{(L, s). L \in \text{states} \wedge \text{dom } s = \{0..<n_vs\} \wedge \text{check_bounded } s\}$$

$\langle proof \rangle$

lemma *trans_prod_states'_inv*:

$l' \in states' \text{ if } (l, g, a, r, l') \in trans_prod \ l \in states'$

$\langle proof \rangle$

lemma *prod_ta_states'_inv*:

$l' \in states' \text{ if } prod_ta \vdash l \longrightarrow^{g,a,r} l' \ l \in states'$

$\langle proof \rangle$

lemma *dom_eq_transfer [transfer_rule]*:

$(state_rel ==> (=)) (\lambda s. dom\ s = \{0..<n_vs\}) (\lambda s. length\ s = n_vs)$

$\langle proof \rangle$

lemma *states'_memi_correct*:

$(states'_memi, (\lambda l. l \in states')) \in loc_rel \rightarrow bool_rel$

$\langle proof \rangle$

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'* ::

$('a, 's, 'c, 't :: time, 'x, 'v :: linorder) \ nta \Rightarrow 's \ list \Rightarrow ('x \rightarrow 'v) \Rightarrow ('c, 't) \ cval$

$\Rightarrow 's \ list \Rightarrow ('x \rightarrow 'v) \Rightarrow ('c, 't) \ cval \Rightarrow bool$

$(_ \vdash \langle _, _, _ \rangle \rightarrow \langle _, _, _ \rangle [61,61,61,61] \ 61)$ **where**

$A \vdash \langle L, s, u \rangle \rightarrow \langle L'', s'', u'' \rangle \text{ if }$

$A \vdash \langle L, s, u \rangle \rightarrow_{Del} \langle L', s', u' \rangle$

$a \neq Del$

$A \vdash \langle L', s', u' \rangle \rightarrow_a \langle L'', s'', u'' \rangle$

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 $A :: 'a \Rightarrow 'a \Rightarrow bool$

and $P :: 'a \Rightarrow bool$

and $B :: 'a \Rightarrow 'a \Rightarrow bool$

assumes $\bigwedge a\ b. A\ a\ b \Longrightarrow P\ a \Longrightarrow B\ a\ b$

```

    and  $\bigwedge a\ b. B\ a\ b \implies P\ a \implies A\ a\ b$ 
    and  $\bigwedge a\ b. P\ a \implies A\ a\ b \implies P\ b$ 
  shows Bisimulation_Invariant  $A\ B\ (=)\ P\ P$ 
  <proof>

context Prod_TA_Defs
begin

definition
  all_prop  $L\ s = (L \in \text{states} \wedge \text{bounded bounds } s)$ 

lemma all_prop_boundedD[dest]:
  bounded bounds  $s$  if all_prop  $L\ s$ 
  <proof>

lemma all_prop_statesD[dest]:
   $L \in \text{states}$  if all_prop  $L\ s$ 
  <proof>

end

context Prod_TA
begin

lemma prod_action_step_not_eq_delay:
   $a \neq \text{Del}$  if prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow_a \langle (L', s'), u' \rangle$ 
  <proof>

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:
  all_prop  $L'\ s'$  if all_prop  $L\ s$  prod_ta  $\vdash \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  <proof>

lemma prod_all_prop_inv':
  all_prop  $L'\ s'$  if all_prop  $L\ s$  prod_ta  $\vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle$ 
  <proof>

interpretation prod_bisim:
  Bisimulation_Invariant
   $(\lambda (L, s, u). \langle (L, s), u \rangle \vdash' \langle (L', s'), u' \rangle)$ 
   $(\lambda (L, s, u). \langle (L, s), u \rangle \vdash \langle (L', s'), u' \rangle)$ 
  (=)
   $(\lambda (L, s, u). \text{all\_prop } L\ s)$ 
   $(\lambda (L, s, u). \text{all\_prop } L\ s)$ 
  <proof>

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
not ('n, 's, 'a, 'b) *sexp* | *and* ('n, 's, 'a, 'b) *sexp* ('n, 's, 'a, 'b) *sexp* |
or ('n, 's, 'a, 'b) *sexp* ('n, 's, 'a, 'b) *sexp* | *imply* ('n, 's, 'a, 'b) *sexp* ('n, 's, 'a, 'b) *sexp* |
 — Does var *a* equal *x*?
eq 'a 'b |
le 'a 'b |
lt 'a 'b |
ge 'a 'b |
gt 'a 'b |
 — Is procces *i* in location *l*?
loc 'n 's

datatype ('n, 's, 'a, 'b) *formula* =
EX ('n, 's, 'a, 'b) *sexp* | *EG* ('n, 's, 'a, 'b) *sexp*
| *AX* ('n, 's, 'a, 'b) *sexp* | *AG* ('n, 's, 'a, 'b) *sexp*
| *Leadsto* ('n, 's, 'a, 'b) *sexp* ('n, 's, 'a, 'b) *sexp*

fun *check_sexp* :: (nat, 's, 'a, 'b :: linorder) *sexp* $\Rightarrow$'s list $\Rightarrow$ ('a $\Rightarrow$ 'b) $\Rightarrow$ bool **where**
check_sexp *sexp.true* _ _ $\longleftrightarrow$ True |
check_sexp (*not* *e*) *L* *s* $\longleftrightarrow$ $\neg$ *check_sexp* *e* *L* *s* |
check_sexp (*and* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexp* *e1* *L* *s* $\wedge$ *check_sexp* *e2* *L* *s* |
check_sexp (*sexp.or* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexp* *e1* *L* *s* $\vee$ *check_sexp* *e2* *L* *s* |
check_sexp (*imply* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexp* *e1* *L* *s* $\longrightarrow$ *check_sexp* *e2* *L* *s* |
check_sexp (*eq* *i* *x*) *L* *s* $\longleftrightarrow$ *s* *i* = *x* |
check_sexp (*le* *i* *x*) *L* *s* $\longleftrightarrow$ *s* *i* $\leq$ *x* |
check_sexp (*lt* *i* *x*) *L* *s* $\longleftrightarrow$ *s* *i* < *x* |
check_sexp (*ge* *i* *x*) *L* *s* $\longleftrightarrow$ *s* *i* $\geq$ *x* |
check_sexp (*gt* *i* *x*) *L* *s* $\longleftrightarrow$ *s* *i* > *x* |
check_sexp (*loc* *i* *x*) *L* *s* $\longleftrightarrow$ *L* ! *i* = *x*

fun *locs_of_sexp* :: ('n, 's, 'a, 'b) *sexp* $\Rightarrow$ 'n set **where**
locs_of_sexp (*not* *e*) = *locs_of_sexp* *e* |
locs_of_sexp (*and* *e1* *e2*) = *locs_of_sexp* *e1* $\cup$ *locs_of_sexp* *e2* |
locs_of_sexp (*sexp.or* *e1* *e2*) = *locs_of_sexp* *e1* $\cup$ *locs_of_sexp* *e2* |
locs_of_sexp (*imply* *e1* *e2*) = *locs_of_sexp* *e1* $\cup$ *locs_of_sexp* *e2* |
locs_of_sexp (*loc* *i* *x*) = {*i*} |
locs_of_sexp _ = {}

fun *vars_of_sexp* :: ('n, 's, 'a, 'b) *sexp* $\Rightarrow$ 'a set **where**
vars_of_sexp (*not* *e*) = *vars_of_sexp* *e* |
vars_of_sexp (*and* *e1* *e2*) = *vars_of_sexp* *e1* $\cup$ *vars_of_sexp* *e2* |
vars_of_sexp (*sexp.or* *e1* *e2*) = *vars_of_sexp* *e1* $\cup$ *vars_of_sexp* *e2* |
vars_of_sexp (*imply* *e1* *e2*) = *vars_of_sexp* *e1* $\cup$ *vars_of_sexp* *e2* |
vars_of_sexp (*eq* *i* *x*) = {*i*} |
vars_of_sexp (*lt* *i* *x*) = {*i*} |

```

vars_of_sexp (le i x) = {i} |
vars_of_sexp (ge i x) = {i} |
vars_of_sexp (gt i x) = {i} |
vars_of_sexp _ = {}

```

```

fun locs_of_formula :: ('n, 's, 'a, 'b) formula  $\Rightarrow$  'n set where
  locs_of_formula (formula.EX  $\varphi$ ) = locs_of_sexp  $\varphi$  |
  locs_of_formula (EG  $\varphi$ ) = locs_of_sexp  $\varphi$  |
  locs_of_formula (AX  $\varphi$ ) = locs_of_sexp  $\varphi$  |
  locs_of_formula (AG  $\varphi$ ) = locs_of_sexp  $\varphi$  |
  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  $\varphi$  |
  vars_of_formula (EG  $\varphi$ ) = vars_of_sexp  $\varphi$  |
  vars_of_formula (AX  $\varphi$ ) = vars_of_sexp  $\varphi$  |
  vars_of_formula (AG  $\varphi$ ) = vars_of_sexp  $\varphi$  |
  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 |
  hd_of_formula (EG  $\varphi$ ) L s = check_sexp  $\varphi$  L s |
  hd_of_formula (AX  $\varphi$ ) L s = Not (check_sexp  $\varphi$  L s) |
  hd_of_formula (AG  $\varphi$ ) L s = Not (check_sexp  $\varphi$  L s) |
  hd_of_formula (Leadsto  $\varphi$  _) L s = check_sexp  $\varphi$  L s

```

```

definition models (_, _  $\models$  _ [61,61] 61) where
  A, a0  $\models$   $\Phi$   $\equiv$  (case  $\Phi$  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))
    | 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))
    | 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))
    | 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))
    | 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))
  ) a0

```

```

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* φ) = *ctl_formula.EX* (*prop_of* φ)
| *ctl_of* (*formula.EG* φ) = *ctl_formula.EG* (*prop_of* φ)
| *ctl_of* (*formula.AX* φ) = *ctl_formula.AX* (*prop_of* φ)
| *ctl_of* (*formula.AG* φ) = *ctl_formula.AG* (*prop_of* φ)
| *ctl_of* (*formula.Leadsto* $\varphi\ \psi$) =
ctl_formula.AG (*ctl_formula.ImpliesC* (*prop_of* φ) (*ctl_formula.AX* (*prop_of* ψ)))

context

fixes $A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder)\ nta$

begin

interpretation *Graph_Defs* $\lambda\ (L, s, u)\ (L', s', u').\ A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle\ \langle proof \rangle$

lemma *models_ctl_iff*:

$A, a_0 \models \Phi \longleftrightarrow models_ctl\ (ctl_of\ \Phi)\ a_0$
 $\langle proof \rangle$

end

fun *check_sexpi* :: (nat, 's, nat, int) *sexp* $\Rightarrow$'s list $\Rightarrow$ int list $\Rightarrow$ bool **where**

check_sexpi *sexp.true* $_ _ \longleftrightarrow True$ |
check_sexpi (*not* e) $L\ s \longleftrightarrow \neg\ check_sexpi\ e\ L\ s$ |
check_sexpi (*and* $e1\ e2$) $L\ s \longleftrightarrow check_sexpi\ e1\ L\ s \wedge check_sexpi\ e2\ L\ s$ |
check_sexpi (*sexp.or* $e1\ e2$) $L\ s \longleftrightarrow check_sexpi\ e1\ L\ s \vee check_sexpi\ e2\ L\ s$ |
check_sexpi (*imply* $e1\ e2$) $L\ s \longleftrightarrow check_sexpi\ e1\ L\ s \longrightarrow check_sexpi\ e2\ L\ s$ |
check_sexpi (*eq* $i\ x$) $L\ s \longleftrightarrow s!\ i = x$ |
check_sexpi (*le* $i\ x$) $L\ s \longleftrightarrow s!\ i \leq x$ |
check_sexpi (*lt* $i\ x$) $L\ s \longleftrightarrow s!\ i < x$ |
check_sexpi (*ge* $i\ x$) $L\ s \longleftrightarrow s!\ i \geq x$ |
check_sexpi (*gt* $i\ x$) $L\ s \longleftrightarrow s!\ i > x$ |
check_sexpi (*loc* $i\ x$) $L\ s \longleftrightarrow L!\ i = x$

fun *hd_of_formulai* :: (nat, 's, nat, int) *formula* $\Rightarrow$'s list $\Rightarrow$ int list $\Rightarrow$ bool **where**

hd_of_formulai (*formula.EX* φ) $L\ s = check_sexpi\ \varphi\ L\ s$ |
hd_of_formulai (*EG* φ) $L\ s = check_sexpi\ \varphi\ L\ s$ |
hd_of_formulai (*AX* φ) $L\ s = Not\ (check_sexpi\ \varphi\ L\ s)$ |
hd_of_formulai (*AG* φ) $L\ s = Not\ (check_sexpi\ \varphi\ L\ s)$ |
hd_of_formulai (*Leadsto* $\varphi\ _$) $L\ s = check_sexpi\ \varphi\ L\ s$

lemma *all_prop_weak_cong*[*cong*]:

Prod_TA_Defs.all_prop $A = Prod_TA_Defs.all_prop\ A$ **for** A
 $\langle proof \rangle$

lemma *equiv'_weak_cong*[*cong*]:

Bisimulation_Invariant.equiv' $R\ P\ Q = Bisimulation_Invariant.equiv'\ R\ P\ Q$ **for** $R\ P\ Q$
 $\langle proof \rangle$

lemma *equiv'_weak_cong2*[*cong*]:

Simulation_Invariant.equiv' $R\ P\ Q = Simulation_Invariant.equiv'\ R\ P\ Q$ **for** $R\ P\ Q$
 $\langle proof \rangle$

```

lemma models_ctl_weak_cong[cong]:
  Graph_Defs.models_ctl E = Graph_Defs.models_ctl E for E
  <proof>
lemma models_path_weak_cong[cong]:
  Graph_Defs.models_path E = Graph_Defs.models_path E for E
  <proof>
lemma models_state_weak_cong[cong]:
  Graph_Defs.models_state E = Graph_Defs.models_state E for E
  <proof>
lemma models_ltl_weak_cong[cong]:
  Graph_Defs.models_ltlc E = Graph_Defs.models_ltlc E for E
  <proof>
lemma models_weak_cong[cong]:
  models E = models E
  for E <proof>

```

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:
     $\forall i < n\_ps. \forall (l, g) \in \text{set} ((snd \circ snd \circ snd) (automata ! i)).$ 
     $\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x)$ 
     $\forall i < n\_ps. \forall (l, \_, g, \_) \in \text{set} ((fst \circ snd \circ snd) (automata ! i)).$ 
     $(\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x))$ 
  and k_resets:
     $\forall i < n\_ps. \forall (l, b, g, a, upd, r, l') \in \text{set} ((fst \circ snd \circ snd) (automata ! i)).$ 
     $\forall c \in \{0..m+1\} - \text{set } r. k ! i ! l' ! c \leq k ! i ! l ! c$ 
  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:
     $\forall (\_, \_, \_, \text{inv}) \in \text{set automata}. \text{distinct } (\text{map } fst \text{ inv})$ 
  and s0_bounded: bounded bounds (map_of s0)
  and L0_len: length L0 = n_ps
  and L0_has_trans:  $\forall i < n\_ps. L_0 ! i \in \text{fst 'set } ((fst \circ snd \circ snd) (automata ! i))$ 
  and vars_of_formula: vars_of_formula formula ⊆ {0..n_vs}

```

```

begin

```

The ceiling k is correct for each individual automaton in the network. We now construct a ceiling for the product automaton:

definition

$k_fun\ l\ c \equiv$
 $if\ (c > 0 \wedge c \leq m) \wedge fst\ l \in states\ then\ Max\ \{k!\ i!\ (fst\ l!\ i)!\ c \mid i.\ i < n_ps\}\ else\ 0$

lemma (in $-$) *default_map_of_distinct*:

$(k, default_map_of\ x\ xs\ k) \in set\ xs \cup \{(k, x)\}$ **if** *distinct* (*map fst xs*)
 $\langle proof \rangle$

lemma *N_inv*:

$(L!\ i, inv\ (N\ i)\ (L!\ i)) \in set\ ((snd\ o\ snd\ o\ snd)\ (automata!\ i)) \cup \{(L!\ i, [])\}$
if $i < n_ps$
 $\langle proof \rangle$

lemma (in $-$) *subset_nat_0_atLeastLessThan_conv*:

$S \subseteq \{0..<n::nat\} \longleftrightarrow (\forall\ x \in S.\ x < n)$
 $\langle proof \rangle$

lemma *k_ceiling_rule*:

$m \leq int\ (k!\ i!\ l!\ x)$
if $i < n_ps$ $(l, b, g, xx) \in set\ ((fst\ o\ snd\ o\ snd)\ (automata!\ i))$
 $(x, m) \in collect_clock_pairs\ g$
for $i\ l\ x\ g\ xx$
 $\langle proof \rangle$

lemma *k_ceiling_1*:

$\forall s.\ \forall L \in states.\ \forall (x, m) \in clkp_set\ prod_ta\ (L, s).\ m \leq k_fun\ (L, s)\ x$
 $\langle proof \rangle$

lemma *k_fun_mono'*:

$k_fun\ (L, s)\ c \leq k_fun\ (L', s')\ c$ **if**
 $\forall i < n_ps.\ k!\ i!\ (L!\ i)!\ c \leq k!\ i!\ (L'!\ i)!\ c\ L \in states\ L' \in states$
 $\langle proof \rangle$

lemma *k_fun_mono*:

$\langle Max\ \{k!\ i!\ (L!\ i)!\ c \mid i.\ i < n_ps\} \leq Max\ \{k!\ i!\ (L'!\ i)!\ c \mid i.\ i < n_ps\} \rangle$
if $\langle \forall i < n_ps.\ k!\ i!\ (L!\ i)!\ c \leq k!\ i!\ (L'!\ i)!\ c \rangle$
 $\langle proof \rangle$

lemma (in $-$) *fold_upds_aux1*:

$fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs!\ i = xs!\ i$ **if** $\langle i \notin set\ ps \rangle$
 $\langle proof \rangle$

lemma (in $-$) *fold_upds_aux2*:

$fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs!\ i = g\ i$ **if** $\langle distinct\ ps \rangle\ \langle i \in set\ ps \rangle\ \langle i < length\ xs \rangle$
 $\langle proof \rangle$

lemma (in $-$) *fold_upds_aux_length*:

$length\ (fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs) = length\ xs$
 $\langle proof \rangle$

lemma *prod_ta_step_statesD*:

assumes $\text{prod_ta} \vdash (L, s) \longrightarrow^{g,a,r} (L', s')$
shows $L \in \text{states } L' \in \text{states}$
 $\langle \text{proof} \rangle$

lemma $k_ceiling_2$:

$\forall l \ g \ a \ r \ l'. \forall \ c \leq m. \text{prod_ta} \vdash l \longrightarrow^{g,a,r} l' \wedge c \notin \text{set } r \longrightarrow k_fun \ l' \ c \leq k_fun \ l \ c$
 $\langle \text{proof} \rangle$

abbreviation F **where** $F \equiv \lambda(L, s). \text{hd_of_formula } formula \ L \ (the \ o \ s)$

abbreviation $Fi \equiv \lambda(L, s). \text{hd_of_formulai } formula \ L \ s$

lemma (**in** $\text{Simple_Network_Impl_nat}$) $\text{check_sexp_check_sexpi}$:

$\text{check_sexp } e \ L \ (the \ o \ s) \longleftrightarrow \text{check_sexpi } e \ L \ s'$

if $\text{state_rel } s \ s' \ \text{vars_of_sexp } e \subseteq \{0..<n_vs\}$

$\langle \text{proof} \rangle$

lemma (**in** $\text{Simple_Network_Impl_nat}$) $\text{hd_of_formula_hd_of_formulai}$:

$\text{hd_of_formula } \varphi \ L \ (the \ o \ s) \longleftrightarrow \text{hd_of_formulai } \varphi \ L \ s'$

if $\text{state_rel } s \ s' \ \text{vars_of_formula } \varphi \subseteq \{0..<n_vs\}$

$\langle \text{proof} \rangle$

lemma F_Fi :

$F \ l \longleftrightarrow Fi \ l' \ \text{if } (l', l) \in \text{loc_rel}$

$\langle \text{proof} \rangle$

abbreviation $l_0 \equiv (L_0, \text{map_of } s_0)$

abbreviation $s_0 i \equiv \text{map } (the \ o \ \text{map_of } s_0) \ [0..<n_vs]$

abbreviation $l_0 i \equiv (L_0, s_0 i)$

lemma state_rel_start :

$\text{state_rel } (\text{map_of } s_0) \ s_0 i$

$\langle \text{proof} \rangle$

lemma statesI :

$L \in \text{states} \ \text{if } \text{length } L = n_ps \ \forall i < n_ps. L ! i \in \text{fst } ' \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! i))))$

$\langle \text{proof} \rangle$

lemma $L_0_states[\text{simp}, \text{intro}]$:

$L_0 \in \text{states}$

$\langle \text{proof} \rangle$

lemma $l_0_states'[\text{simp}, \text{intro}]$:

$l_0 \in \text{states}'$

$\langle \text{proof} \rangle$

sublocale $\text{reach: Reachability_Problem_Defs}$

prod_ta

l_0

m

k_fun

F

$\langle proof \rangle$

lemma (in $-$) *collect_clkt_state_setI*:
 assumes $(x, d) \in \text{Closure.collect_clkt } (\text{trans_of } A) \ l$
 shows $l \in \text{state_set } (\text{trans_of } A) \ l \in \text{Simulation_Graphs_TA.state_set } A$
 $\langle proof \rangle$

lemma *clkp_set_statesD*:
 fixes $x \ d$
 assumes $(x, d) \in \text{Closure.clkp_set prod_ta } (L, s)$
 shows $L \in \text{states}$
 $\langle proof \rangle$

sublocale *reach1: Reachability_Problem*
 prod_ta
 l_0
 m
 k_fun
 F
 $\langle proof \rangle$

lemma *states'_superset*:
 $\{l_0\} \cup \text{Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod} \subseteq \text{states}'$
 (**is** $\{l_0\} \cup ?S \subseteq \text{states}'$)
 $\langle proof \rangle$

definition $k_i \equiv \text{IArray } (\text{map } (\text{IArray } o (\text{map } (\text{IArray } o \text{ map int}))) \ k)$

definition
 $k_impl \equiv \lambda(l, _). \text{IArray } (\text{map } (\lambda \ c. \text{Max } \{k_i \ !! \ i \ !! \ (l \ ! \ i) \ !! \ c \mid i. i < n_ps\}) \ [0..<m+1])$

lemma *Max_int_commute*:
 $\text{int } (\text{Max } S) = \text{Max } (\text{int } ' S) \text{ if finite } S \ S \neq \{\}$
 $\langle proof \rangle$

lemma (in *Simple_Network_Impl_nat*) $n_ps_gt_0: n_ps > 0$
 $\langle proof \rangle$

lemma *statesD*:
 $L \ ! \ i \in \text{fst } ' \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! \ i))))$
 $\vee L \ ! \ i \in (\text{snd } o \text{ snd } o \text{ snd } o \text{ snd } o \text{ snd } o \text{ snd}) \ ' \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } ! \ i))))$
 if $L \in \text{states}$ $\text{length } L = n_ps$ $i < n_ps$
 $\langle proof \rangle$

lemma *k_impl_k_fun*:
 $k_impl \ (L, s) = \text{IArray } (\text{map } (k_fun \ (L, s)) \ [0..<m+1]) \text{ if } L \in \text{states}$
 $\langle proof \rangle$

sublocale *impl: Reachability_Problem_Impl*
 where $\text{trans_fun} = \text{trans_from}$

```

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
⟨proof⟩

end

no_notation UPPAAL_Model_Checking.models (_, _  $\models$  _ [61,61] 61)

context Reachability_Problem_Impl
begin

lemma F_reachable_correct:
  op.F_reachable
   $\longleftrightarrow (\exists l'. \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. \text{conv\_}A \ A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l'))$ 
  ⟨proof⟩

lemma E_op''_F_reachable_correct:
  E_op''.F_reachable
   $\longleftrightarrow (\exists l'. \forall u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \longrightarrow (\exists u'. \text{conv\_}A \ A \vdash' \langle l_0, u_0 \rangle \rightarrow^* \langle l', u' \rangle \wedge F \ l'))$ 
  ⟨proof⟩

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⟩

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')
  ⟨proof⟩

```

abbreviation $A \equiv (\text{set broadcast, map automaton_of automata, map_of bounds'})$

interpretation *conv_eq_bisim*:

Bisimulation_Invariant

$(\lambda(l, u) (l', u'). \text{conv_A prod_ta} \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda(L, s, u) (L', s', u'). \text{conv.prod_ta} \vdash' \langle (L, s), u \rangle \rightarrow \langle (L', s'), u' \rangle)$
 $(\lambda((L, s), u) (L', s', u'). L = L' \wedge u = u' \wedge s = s')$
 $(\lambda((L, s), u). \text{conv.all_prop } L \ s)$
 $(\lambda(L, s, u). \text{conv.all_prop } L \ s)$

$\langle \text{proof} \rangle$

interpretation *Bisimulation_Invariant*

$(\lambda(l, u) (l', u'). \text{conv_A prod_ta} \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $(\lambda(L, s, u) (L', s', u'). \text{Simple_Network_Language.conv } A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle)$
 $(\lambda((L, s), u) (L', s', u'). L = L' \wedge u = u' \wedge s = s')$
 $(\lambda((L, s), u). \text{conv.all_prop } L \ s)$
 $(\lambda(L, s, u). \text{conv.all_prop } L \ s)$

$\langle \text{proof} \rangle$

lemmas *prod_bisim = Bisimulation_Invariant_axioms*

lemmas *deadlock_iff = deadlock_iff*

lemma *conv_all_prop*:

conv.all_prop = all_prop

$\langle \text{proof} \rangle$

definition *prop_of2 where*

prop_of2 $\varphi = \text{PropC } (\lambda((L, s), _). \text{check_sexp } \varphi \ L \ (\text{the } o \ s))$

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* $\varphi \ \psi$) =
ctl_formula.AG (*ctl_formula.ImpliesC* (*prop_of2* φ) (*ctl_formula.AX* (*prop_of2* ψ)))

lemma *models_correct*:

Simple_Network_Language.conv $A, (L_0, s_0, u_0) \models \Phi = (\text{case } \Phi \text{ of}$
formula.EX $\varphi \Rightarrow$

Graph_Defs.Ex_ev

$(\lambda(l, u) (l', u'). \text{conv_A prod_ta} \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$

$(\lambda((L, s), _). \text{check_sexp } \varphi \ L \ (\text{the } o \ s))$

| *formula.EG* $\varphi \Rightarrow$

Not o Graph_Defs.Alw_ev

$(\lambda(l, u) (l', u'). \text{conv_A prod_ta} \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$

$(\lambda((L, s), _). \neg \text{check_sexp } \varphi \ L \ (\text{the } o \ s))$

| *formula.AX* $\varphi \Rightarrow$

Graph_Defs.Alw_ev

$(\lambda(l, u) (l', u'). \text{conv_A prod_ta} \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)$

$(\lambda((L, s), _). \text{check_sexp } \varphi \ L \ (\text{the } o \ s))$

| *formula.AG* $\varphi \Rightarrow$

```

    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⟩

```

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 ∘ fst)
  impl.succs_impl' impl.emptyness_check_impl impl.F_impl
  (impl.Q_impl (λ (L, s). ¬ check_sexp ψ L s))
  impl.tracei;
  return (¬ r)
}

```

definition

```

reachability_checker ≡
  pw_impl
  (return ∘ fst) impl.state_copy_impl impl.tracei impl.subsumes_impl impl.a₀_impl impl.F_impl
  impl.succs_impl impl.emptyness_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 L_0 (*map_of* s_0)
<proof>

lemma *deadlock_start_iff*:
Graph_Defs.deadlock
 $(\lambda(L, s, u) (L', s', u'). \text{Simple_Network_Language.conv } A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle) (L_0, (\text{map_of } s_0), u_0) \longleftrightarrow \text{reach.deadlock } (l_0, u_0)$
<proof>

lemma *F_Fi'*:
check_sexp ψ L (*the o s*) $\longleftrightarrow$ *check_sexp* ψ $L' s'$
if $((L', s'), (L, s)) \in \text{loc_rel}$ *formula* = *Leadsto* φ ψ
<proof>

theorem *model_check'*:
(*uncurry0* *model_checker*,
uncurry0 (
SPEC ($\lambda r.$
 $\neg \text{Graph_Defs.deadlock}$
 $(\lambda(L, s, u) (L', s', u').$
 $\text{Simple_Network_Language.conv } A \vdash \langle L, s, u \rangle \rightarrow \langle L', s', u' \rangle) (L_0, (\text{map_of } s_0), u_0)$
 $\longrightarrow r = (\text{Simple_Network_Language.conv } A, (L_0, (\text{map_of } s_0), u_0) \models \text{formula})$
 $)$
 $)$
 $)$
 $\in \text{unit_assn}^k \rightarrow_a \text{bool_assn}$
<proof>

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.emptiness_check_impl*;
let *trace* = *impl.tracei*;
pw_impl *key* *copy* *trace* *sub* *start* *final* *succs* *empty*

```

};
_ ← return ();
return x
}
⟨proof⟩

```

lemma *Alw_ev_checker_alt_def'*:

```

Alw_ev_checker ≡
do {
  x ← let
    key = return ∘ 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;
  _ ← return ();
  return x
}
⟨proof⟩

```

lemma *leadsto_checker_alt_def'*:

```

leadsto_checker ψ ≡
do {
  r ← let
    key = return ∘ fst;
    sub = impl.subsumes_impl;
    copy = impl.state_copy_impl;
    start = impl.a0_impl;
    final = impl.F_impl;
    final' = (impl.Q_impl (λ(L, s). ¬ check_sexpi ψ 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)
}
⟨proof⟩

```

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⟩

```

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
⟨proof⟩

```

schematic_goal *succs_impl_alt_def*:

```

impl.succs_impl ≡ ?impl
⟨proof⟩

schematic_goal succs_P_impl_alt_def:
  impl.succs_P_impl Pi ≡ ?impl
  if (Pi, P) ∈ inv_rel loc_rel states'
  for P Pi
  ⟨proof⟩

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 ≡ ?impl
  ⟨proof⟩

lemma ψ_compatibleI:
  (λ(L, s). ¬ check_sexp_i ψ L s, λ(L, s). ¬ check_sexp ψ L (the ∘ s)) ∈ inv_rel loc_rel states'
  if formula = Leadsto ϕ ψ
  ⟨proof⟩

lemma Q_impl_alt_def:
  impl.Q_impl (λ(L, s). ¬ check_sexp_i ψ L s) ≡
  λxi. return (case xi of (a1, a2) ⇒ (λ(L, s). ¬ check_sexp_i ψ L s) a1)
  if formula = Leadsto ϕ ψ
  ⟨proof⟩

schematic_goal leadsto_checker_alt_def:
  leadsto_checker ψ ≡ ?impl if formula = Leadsto ϕ ψ
  ⟨proof⟩

schematic_goal reachability_checker_alt_def:
  reachability_checker ≡ ?impl
  ⟨proof⟩

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 ⇒ reachability_checker

```

```

| Simple_Network_Language_Model_Checking.formula.EG xa ⇒
  if PR_CONST F l0 then Alw_ev_checker else return False
| Simple_Network_Language_Model_Checking.formula.AX xa ⇒
  (if PR_CONST F l0 then Alw_ev_checker else return False) ≫= (λr. return (¬ r))
| Simple_Network_Language_Model_Checking.formula.AG xa ⇒
  reachability_checker ≫= (λr. return (¬ r))
| Simple_Network_Language_Model_Checking.formula.Leadsto φ ψ ⇒
  Simple_Network_Language_Model_Checking.leadsto_checker
  broadcast bounds' automata m num_states num_actions k L0 s0 formula ψ show_clock
  show_state)

```

⟨proof⟩

```

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
≡
  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_mc
  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  $\Rightarrow$   $\uparrow(\neg \text{Simple\_Network\_Impl\_nat\_ceiling\_start\_state}$ 
    broadcast bounds automata m num_states num_actions k L0 s0 formula)
 $\rangle_t$ 
<proof>

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 $\text{zone_time_pre} :: ('c, ('t::time)) \text{zone} \Rightarrow ('c, 't) \text{zone}$
 $(_ \downarrow [71] \ 71)$

where

$$Z^\downarrow = \{u \mid u.d. (u \oplus d) \in Z \wedge d \geq (0::'t)\}$$

definition $\text{zone_set_pre} :: ('c, 't::time) \text{zone} \Rightarrow 'c \text{ list} \Rightarrow ('c, 't) \text{zone}$

where

$$\text{zone_set_pre } Z \ r = \{u \mid ([r \rightarrow (0::'t)]u) \in Z\}$$

definition $\text{zone_pre} :: ('c, 't::time) \text{zone} \Rightarrow 'c \text{ list} \Rightarrow ('c, 't) \text{zone}$

where

$$\text{zone_pre } Z \ r = (\text{zone_set_pre } Z \ r)^\downarrow$$

lemma $\text{zone_time_pre_mono}$:

$$A^\downarrow \subseteq B^\downarrow \text{ if } A \subseteq B$$

<proof>

lemma *clock_set_split*:

$P ([r \rightarrow 0]u) x \longleftrightarrow (x \notin \text{set } r \longrightarrow P (u\ x)) \wedge (x \in \text{set } r \longrightarrow P\ 0)$
 $\langle \text{proof} \rangle$

context *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** $x \in Z^\downarrow$ $x \in V$
 $\langle \text{proof} \rangle$

lemma *check_deadlock_alt_def*:

$\text{check_deadlock } l\ Z = (Z \subseteq \bigcup \{$
 $(\text{zone_set_pre } (\{u. u \vdash \text{inv_of } A\ l'\} \cap V) \ r \cap \{u. \forall x \in \text{set } r. u\ x \geq 0\}$
 $\cap \{u. u \vdash g\} \cap \{u. u \vdash \text{inv_of } A\ l\})^\downarrow \cap V$
 $\mid g\ a\ r\ l'. A \vdash l \longrightarrow^{g,a,r} l' \})$ **(is** $_ = (?L \subseteq ?R))$ **if** $Z \subseteq V$
 $\langle \text{proof} \rangle$

lemma *step_trans1*:

assumes $A \vdash_t \langle l, u \rangle \rightarrow_{(g,a,r)} \langle l', u' \rangle$
shows $u \in \text{zone_set_pre } \{u. u \vdash \text{inv_of } A\ l'\} \ r \cap \{u. u \vdash g\} \ A \vdash l \longrightarrow^{g,a,r} l'$
 $\langle \text{proof} \rangle$

lemma *step_trans2*:

assumes $u \in \text{zone_set_pre } \{u. u \vdash \text{inv_of } A\ l'\} \ r \cap \{u. u \vdash g\} \ A \vdash l \longrightarrow^{g,a,r} l'$
shows $\exists u'. A \vdash_t \langle l, u \rangle \rightarrow_{(g,a,r)} \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *time_pre_zone*:

$u \in (Z \cap \{u. u \vdash \text{inv_of } A\ l\})^\downarrow$ **if** $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \ u' \in Z$
 $\langle \text{proof} \rangle$

lemma *time_pre_zone'*:

$\exists d\ u'. u' \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$ **if** $u \in (Z \cap \{u. u \vdash \text{inv_of } A\ l\})^\downarrow$
 $\langle \text{proof} \rangle$

lemma *step_trans3*:

assumes $A \vdash' \langle l, u \rangle \rightarrow_{(g,a,r)} \langle l', u' \rangle$
shows $u \in (\text{zone_set_pre } \{u. u \vdash \text{inv_of } A\ l'\} \ r \cap \{u. u \vdash g\} \cap \{u. u \vdash \text{inv_of } A\ l\})^\downarrow$
 $A \vdash l \longrightarrow^{g,a,r} l'$
 $\langle \text{proof} \rangle$

lemma *step_trans4*:

assumes $u \in (\text{zone_set_pre } \{u. u \vdash \text{inv_of } A \ l^\wedge\} \ r \cap \{u. u \vdash g\} \cap \{u. u \vdash \text{inv_of } A \ l\})^\downarrow$
 $A \vdash l \longrightarrow^{g,a,r} l'$
shows $\exists u'. A \vdash' \langle l, u \rangle \rightarrow^{(g,a,r)} \langle l', u^\wedge \rangle$
 $\langle \text{proof} \rangle$

lemma *check_deadlock_correct*:

$\text{check_deadlock } l \ Z \longleftrightarrow (\forall u \in Z. \exists l' u' g \ a \ r. A \vdash' \langle l, u \rangle \rightarrow^{(g,a,r)} \langle l', u^\wedge \rangle)$
 $\langle \text{proof} \rangle$

lemma *step'_step_trans'_iff*:

$A \vdash' \langle l, u \rangle \rightarrow \langle l', u^\wedge \rangle \longleftrightarrow (\exists g \ a \ r. A \vdash' \langle l, u \rangle \rightarrow^{(g,a,r)} \langle l', u^\wedge \rangle)$
 $\langle \text{proof} \rangle$

lemma *check_deadlock_correct_step'*:

$\text{check_deadlock } l \ Z \longleftrightarrow (\forall u \in Z. \exists l' u'. A \vdash' \langle l, u \rangle \rightarrow \langle l', u^\wedge \rangle)$
 $\langle \text{proof} \rangle$

Unused lemma *delay_step_zone*:

$u' \in Z^\uparrow \cap \{u. u \vdash \text{inv_of } A \ l\}$ **if** $A \vdash \langle l, u \rangle \rightarrow^d \langle l, u^\wedge \rangle \ u \in Z$
 $\langle \text{proof} \rangle$

lemma *delay_step_zone'*:

$\exists d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u^\wedge \rangle$ **if** $u' \in Z^\uparrow \cap \{u. u \vdash \text{inv_of } A \ l\}$
 $\langle \text{proof} \rangle$

lemma *delay_step_zone''*:

$(\exists d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u^\wedge \rangle) \longleftrightarrow u' \in Z^\uparrow \cap \{u. u \vdash \text{inv_of } A \ l\}$
 $\langle \text{proof} \rangle$

lemma *delay_step_zone'''*:

$\{u' \mid u' d \ u. u \in Z \wedge A \vdash \langle l, u \rangle \rightarrow^d \langle l, u^\wedge \rangle\} = Z^\uparrow \cap \{u. u \vdash \text{inv_of } A \ l\}$
 $\langle \text{proof} \rangle$

end

context *Regions_TA_Start_State*

begin

lemma *check_deadlock_deadlocked*:

$\neg \text{check_deadlock } l \ Z \longleftrightarrow (\exists u \in Z. \text{sim.sim.deadlocked } (l, u))$
 $\langle \text{proof} \rangle$

lemma *deadlock_check'*:

$(\exists x_0 \in a_0. \exists l \ u. \text{sim.sim.reaches } x_0 \ (l, u) \wedge \text{sim.sim.deadlocked } (l, u)) \longleftrightarrow$
 $(\exists l \ Z. \text{reaches } (l_0, Z_0) \ (l, Z) \wedge \neg \text{check_deadlock } l \ Z)$
 $\langle \text{proof} \rangle$

lemma *deadlock_check*:

$(\exists x_0 \in a_0. \text{sim.sim.deadlock } x_0) \longleftrightarrow (\exists l \ Z. \text{reaches } (l_0, Z_0) \ (l, Z) \wedge \neg \text{check_deadlock } l \ Z)$
 $\langle \text{proof} \rangle$

end

10.3 Operations

Subset inclusion check for federations on DBMs lemma

$S \subseteq R \longleftrightarrow S \cap -R = \{\}$
 $\langle proof \rangle$

lemma

$A \subseteq B \cup C \longleftrightarrow A \cap -B \cap -C = \{\}$
 $\langle proof \rangle$

lemma

$(A \cup B) \cap (C \cup D) = A \cap C \cup A \cap D \cup B \cap C \cup B \cap D$
 $\langle proof \rangle$

lemma *Le_le_inf[intro]:*

$Le (x :: _ :: linordered_cancel_ab_monoid_add) \preceq \infty$
 $\langle proof \rangle$

lemma *dbm_entry_val_mono:*

$dbm_entry_val\ u\ a\ b\ e'\ \text{if}\ dbm_entry_val\ u\ a\ b\ e\ e \leq e'$
 $\langle proof \rangle$

definition *and_entry ::*

$nat \Rightarrow nat \Rightarrow ('t :: \{linordered_cancel_ab_monoid_add, uminus\})\ DBMEntry \Rightarrow 't\ DBM \Rightarrow 't\ DBM$ **where**
 $and_entry\ a\ b\ e\ M = (\lambda i\ j. \text{if } i = a \wedge j = b \text{ then } \min (M\ i\ j)\ e \text{ else } M\ i\ j)$

lemma *and_entry_mono:*

$and_entry\ a\ b\ e\ M\ i\ j \leq M\ i\ j$
 $\langle proof \rangle$

abbreviation *clock_to_option* $a \equiv (\text{if } a > 0 \text{ then } Some\ a \text{ else } None)$

definition

$dbm_entry_val'\ u\ a\ b\ e \equiv dbm_entry_val\ u\ (clock_to_option\ a)\ (clock_to_option\ b)\ e$

definition

$dbm_minus\ n\ xs\ m = concat\ (map\ (\lambda(i, j). map\ (\lambda M. and_entry\ j\ i\ (neg_dbm_entry\ (m\ i\ j))\ M)\ xs)$
 $[(i, j). i \leftarrow [0..<Suc\ n], j \leftarrow [0..<Suc\ n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty])$

locale *Default_Nat_Clock_Numbering =*

fixes $n :: nat$ **and** $v :: nat \Rightarrow nat$

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:*

$v\ c = c\ \text{if}\ v\ c \leq n$
 $\langle proof \rangle$

lemma *le_n:*

$c \leq n\ \text{if}\ v\ c \leq n$

$\langle proof \rangle$

lemma *gt_0*:

$c > 0$ **if** $v\ c \leq n$

$\langle proof \rangle$

lemma *le_n_iff*:

$v\ c \leq n \longleftrightarrow c \leq n \wedge c > 0$

$\langle proof \rangle$

lemma *v_0*:

$v\ c = 0 \longleftrightarrow False$

$\langle proof \rangle$

lemma *surj_on*: $\forall\ k \leq n. k > 0 \longrightarrow (\exists\ c. v\ c = k)$

$\langle proof \rangle$

abbreviation *zone_of* ($\llbracket _ \rrbracket$) **where** *zone_of* $M \equiv [M]_{v,n}$

abbreviation

dbm_fed $S \equiv \bigcup\ m \in S. \llbracket m \rrbracket$

abbreviation

dbm_list $xs \equiv dbm_fed\ (set\ xs)$

lemma *dbm_fed_singleton*:

dbm_fed $\{m\} = [m]_{v,n}$

$\langle proof \rangle$

lemma *dbm_list_single*:

dbm_list $xs = [m]_{v,n}$ **if** $set\ xs = \{m\}$

$\langle proof \rangle$

lemma *dbm_fed_superset_fold*:

$S \subseteq dbm_list\ xs \longleftrightarrow fold\ (\lambda m\ S. S \cap - ([m]_{v,n}))\ xs\ S = \{\}$

$\langle proof \rangle$

lemma *dbm_fed_superset_fold'*:

dbm_fed $S \subseteq dbm_list\ xs \longleftrightarrow dbm_fed\ (fold\ f\ xs\ S) = \{\}$ **if**

$\bigwedge\ m\ S. m \in set\ xs \implies dbm_fed\ (f\ m\ S) = dbm_fed\ S \cap - ([m]_{v,n})$

$\langle proof \rangle$

lemma *dbm_fed_superset_fold''*:

dbm_list $S \subseteq dbm_list\ xs \longleftrightarrow dbm_list\ (fold\ f\ xs\ S) = \{\}$ **if**

$\bigwedge\ m\ S. m \in set\ xs \implies dbm_list\ (f\ m\ S) = dbm_list\ S \cap - ([m]_{v,n})$

$\langle proof \rangle$

lemma *neg_inf*:

$\{u. \neg dbm_entry_val\ u\ a\ b\ e\} = \{\}$ **if** $e = (\infty :: _ DBMEntry)$

$\langle proof \rangle$

lemma *dbm_entry_val'_diff_shift*:

dbm_entry_val' $(u \oplus d)\ c1\ c2\ (M\ c1\ c2)$ **if** *dbm_entry_val'* $u\ c1\ c2\ (M\ c1\ c2)$ $0 < c1\ 0 < c2$

$\langle proof \rangle$

lemma *dbm_entry_val_iff_bounded_Le1*:
 $dbm_entry_val\ u\ (Some\ c1)\ None\ e \longleftrightarrow Le\ (u\ c1) \leq e$
 $\langle proof \rangle$

lemma *dbm_entry_val_iff_bounded_Le2*:
 $dbm_entry_val\ u\ None\ (Some\ c2)\ e \longleftrightarrow Le\ (-\ u\ c2) \leq e$
 $\langle proof \rangle$

lemma *dbm_entry_val_iff_bounded_Le3*:
 $dbm_entry_val\ u\ (Some\ c1)\ (Some\ c2)\ e \longleftrightarrow Le\ (u\ c1 - u\ c2) \leq e$
 $\langle proof \rangle$

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 $c1 > 0 \vee c2 > 0$
 $\langle proof \rangle$

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 $e \neq (\infty :: _ DBMEntry)\ a > 0 \vee b > 0$
 $\langle proof \rangle$

lemma *neg_unbounded*:
 $\{u. \neg dbm_entry_val'\ u\ i\ j\ e\} = \{\}$ **if** $e = (\infty :: _ DBMEntry)$
 $\langle proof \rangle$

lemma *and_entry_sound*:
 $u \vdash_{v,n} and_entry\ a\ b\ e\ M$ **if** $dbm_entry_val'\ u\ a\ b\ e\ u \vdash_{v,n} M$
 $\langle proof \rangle$

lemma *DBM_val_bounded_mono*:
 $u \vdash_{v,n} M$ **if** $u \vdash_{v,n} M' \forall\ i \leq n. \forall\ j \leq n. M'\ i\ j \leq M\ i\ j$
 $\langle proof \rangle$

lemma *and_entry_entry*:
 $dbm_entry_val'\ u\ a\ b\ e$ **if** $u \vdash_{v,n} and_entry\ a\ b\ e\ M\ a \leq n\ b \leq n\ a > 0 \vee b > 0$
 $\langle proof \rangle$

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 $a \leq n\ b \leq n\ a > 0 \vee b > 0$
 $\langle proof \rangle$

lemma *dbm_list_Int_entry_iff_map*:
 $dbm_list\ xs \cap \{u. dbm_entry_val'\ u\ i\ j\ e\} = dbm_list\ (map\ (\lambda\ m. and_entry\ i\ j\ e\ m)\ xs)$
if $i \leq n\ j \leq n\ i > 0 \vee j > 0$
 $\langle proof \rangle$

context

fixes $m :: \text{nat} \Rightarrow \text{nat} \Rightarrow ('a :: \{\text{time}\}) \text{DBMEntry}$

assumes $Le\ 0 \preceq m\ 0\ 0$

begin

private lemma *A*:

$- ([m]_{v,n}) =$
 $(\bigcup (i, j) \in \{(i, j). i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$
 $\{u. \neg \text{dbm_entry_val } u\ (\text{Some } i)\ (\text{Some } j)\ (m\ i\ j)\})$
 $\cup (\bigcup i \in \{i. i > 0 \wedge i \leq n\}. \{u. \neg \text{dbm_entry_val } u\ (\text{Some } i)\ \text{None}\ (m\ i\ 0)\})$
 $\cup (\bigcup j \in \{j. j > 0 \wedge j \leq n\}. \{u. \neg \text{dbm_entry_val } u\ \text{None}\ (\text{Some } j)\ (m\ 0\ j)\})$
 $\langle \text{proof} \rangle$ **lemma** *B*:
 $S \cap - ([m]_{v,n}) =$
 $(\bigcup (i, j) \in \{(i, j). i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$
 $S \cap \{u. \neg \text{dbm_entry_val } u\ (\text{Some } i)\ (\text{Some } j)\ (m\ i\ j)\})$
 $\cup (\bigcup i \in \{i. i > 0 \wedge i \leq n\}. S \cap \{u. \neg \text{dbm_entry_val } u\ (\text{Some } i)\ \text{None}\ (m\ i\ 0)\})$
 $\cup (\bigcup j \in \{j. j > 0 \wedge j \leq n\}. S \cap \{u. \neg \text{dbm_entry_val } u\ \text{None}\ (\text{Some } j)\ (m\ 0\ j)\})$
 $\langle \text{proof} \rangle$ **lemma** *UNION_cong*:
 $(\bigcup x \in S. f\ x) = (\bigcup x \in T. g\ x) \text{ if } S = T \wedge x. x \in T \implies f\ x = g\ x$
 $\langle \text{proof} \rangle$ **lemma** *1*:
 $S \cap - ([m]_{v,n}) =$
 $(\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}. S \cap \{u. \neg \text{dbm_entry_val}'\ u\ i\ j\ (m\ i\ j)\})$

$\langle \text{proof} \rangle$ **lemma** *UNION_remove*:

$(\bigcup x \in S. f\ x) = (\bigcup x \in T. g\ x)$
if $T \subseteq S \wedge x. x \in T \implies f\ x = g\ x \wedge x. x \in S - T \implies f\ x = \{\}$
 $\langle \text{proof} \rangle$ **lemma** *2*:
 $(\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}. S \cap \{u. \neg \text{dbm_entry_val}'\ u\ i\ j\ (m\ i\ j)\})$
 $= (\bigcup (i, j) \in \{(i, j). (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty\}.$
 $S \cap \{u. \text{dbm_entry_val}'\ u\ j\ i\ (\text{neg_dbm_entry}\ (m\ i\ j))\})$
 $\langle \text{proof} \rangle$

lemma *dbm_list_subtract*:

$\text{dbm_list } xs \cap - ([m]_{v,n}) = \text{dbm_list } (\text{dbm_minus } n\ xs\ m)$
 $\langle \text{proof} \rangle$

end — Context for fixed DBM

end — Simplifier setup

lemma *dbm_list_empty_check*:

$\text{dbm_list } xs = \{\} \longleftrightarrow \text{list_all } (\lambda m. [m]_{v,n} = \{\})\ xs$
 $\langle \text{proof} \rangle$

lemmas *dbm_list_superset_op* =

$\text{dbm_fed_superset_fold''}[\text{OF dbm_list_subtract}[\text{symmetric}], \text{unfolded dbm_list_empty_check}]$

end

context *TA_Start_No_Ceiling*

begin

sublocale *dbm*: *Default_Nat_Clock_Numbering* *n v*
 ⟨*proof*⟩

end

Down

Auxiliary lemma *dbm_entry_le_iff*:

$Le\ a \leq Le\ b \longleftrightarrow a \leq b$
 $Le\ a \leq Lt\ b \longleftrightarrow a < b$
 $Lt\ a \leq Le\ b \longleftrightarrow a \leq b$
 $Lt\ a \leq Lt\ b \longleftrightarrow a < b$
 $0 \leq Le\ a \longleftrightarrow 0 \leq a$
 $0 \leq Lt\ a \longleftrightarrow 0 < a$
 $Le\ a \leq 0 \longleftrightarrow a \leq 0$
 $Lt\ a \leq 0 \longleftrightarrow a \leq 0$
 $\infty \leq x \longleftrightarrow x = \infty$
 $x \leq \infty \longleftrightarrow True$

⟨*proof*⟩

lemma *dbm_entry_lt_iff*:

$Le\ a < Le\ b \longleftrightarrow a < b$
 $Le\ a < Lt\ b \longleftrightarrow a < b$
 $Lt\ a < Le\ b \longleftrightarrow a \leq b$
 $Lt\ a < Lt\ b \longleftrightarrow a < b$
 $0 < Le\ a \longleftrightarrow 0 < a$
 $0 < Lt\ a \longleftrightarrow 0 < a$
 $Le\ a < 0 \longleftrightarrow a < 0$
 $Lt\ a < 0 \longleftrightarrow a \leq 0$
 $x < \infty \longleftrightarrow x \neq \infty$
 $\infty < x \longleftrightarrow False$

⟨*proof*⟩

lemmas [*dbm_entry_simps*] = *dbm_entry_le_iff*(1-9) *dbm_entry_lt_iff*(1-9)

lemma *Le_le_sum_iff*:

$Le\ (y :: _ :: time) \leq e \longleftrightarrow 0 \leq e + Le\ (-\ y)$
 ⟨*proof*⟩

lemma *dense'*:

$\exists c \geq a. c \leq b$ **if** $a \leq b$ **for** $a :: _ :: time$
 ⟨*proof*⟩

lemma *aux1*:

$-c \leq (a :: _ :: time)$ **if** $a \geq 0$ $c \geq 0$
 ⟨*proof*⟩

lemma *dbm_entries_dense*:

$\exists d \geq 0. Le\ (-\ d) \leq l \wedge Le\ (d :: _ :: time) \leq r$ **if** $0 \leq l$ $l \leq r$
 ⟨*proof*⟩

lemma *dbm_entries_dense'_aux*:

$\exists d \geq 0. l + Le\ d \geq 0 \wedge 0 \leq r + Le\ (-\ d :: _ :: time)$ **if** $l \leq 0$ $l + r \geq 0$ $r \geq 0$

$\langle proof \rangle$

lemma *dbm_entries_dense'*:

$\exists d \geq 0. l + Le\ d \geq 0 \wedge 0 \leq r + Le\ (-\ d :: _ :: time) \text{ if } l \leq 0 \wedge l + r \geq 0$

$\langle proof \rangle$

lemma (*in time*) *non_trivial_pos*: $\exists x. x > 0$

$\langle proof \rangle$

lemma *dbm_entries_dense_pos*:

$\exists d > 0. Le\ (d :: _ :: time) \leq e \text{ if } e > 0$

$\langle proof \rangle$

lemma *le_minus_iff*:

$-x \leq (y :: _ :: time) \longleftrightarrow 0 \leq y + x$

$\langle proof \rangle$

lemma *lt_minus_iff*:

$-x < (y :: _ :: time) \longleftrightarrow 0 < y + x$

$\langle proof \rangle$

context *Default_Nat_Clock_Numbering*

begin

lemma *DBM_val_bounded_alt_def1*:

$u \vdash_{v,n} m \equiv$

$Le\ 0 \leq m\ 0\ 0 \wedge$

$(\forall c. c > 0 \wedge c \leq n \longrightarrow$

$dbm_entry_val\ u\ None\ (Some\ c)\ (m\ 0\ c) \wedge$

$dbm_entry_val\ u\ (Some\ c)\ None\ (m\ c\ 0)) \wedge$

$(\forall c1\ c2. c1 > 0 \wedge c1 \leq n \wedge c2 > 0 \wedge c2 \leq n \longrightarrow dbm_entry_val\ u\ (Some\ c1)\ (Some\ c2)$

$(m\ c1\ c2))$

$\langle proof \rangle$

lemma *DBM_val_bounded_alt_def2*:

$u \vdash_{v,n} m \equiv$

$Le\ 0 \leq m\ 0\ 0 \wedge$

$(\forall c1\ c2. (c1 \neq 0 \vee c2 \neq 0) \wedge c1 \leq n \wedge c2 \leq n \longrightarrow dbm_entry_val'\ u\ c1\ c2\ (m\ c1\ c2))$

$\langle proof \rangle$

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

$u \in \llbracket m \rrbracket$

$\langle proof \rangle$

lemma *dbm_entry_val'_delay1*:

$dbm_entry_val'\ u\ c1\ c2\ (m\ c1\ c2) \text{ if } dbm_entry_val'\ (u \oplus d)\ c1\ c2\ (m\ c1\ c2)\ d \geq 0 \wedge c1 > 0$

$\langle proof \rangle$

lemma *dbm_entry_val'_delay2*:

$dbm_entry_val'\ u\ (0 :: nat)\ c2\ (m\ c1\ c2) \text{ if}$

$dbm_entry_val' (u \oplus d) \ c1 \ c2 \ (m \ c1 \ c2) \ d \geq 0$
 $c1 > 0 \ c2 > 0 \ c1 \leq n \ c2 \leq n$
 $\forall \ c \leq n. \ c > 0 \longrightarrow u \ c \geq 0$
 $\langle proof \rangle$

lemma *dbm_entry_val'_nonneg_bound*:
 $dbm_entry_val' \ u \ (0 :: nat) \ c \ (Le \ 0) \ \text{if} \ u \ c \geq 0 \ c > 0$
 $\langle proof \rangle$

lemma *neg_diag_empty_spec*:
 $\llbracket M \rrbracket = \{\}$ **if** $i \leq n \ M \ i \ i < 0$
 $\langle proof \rangle$

lemma *in_DBM_D*:
 $dbm_entry_val' \ u \ c1 \ c2 \ (M \ c1 \ c2) \ \text{if} \ u \in \llbracket M \rrbracket \ c1 \neq 0 \vee c2 \neq 0 \ c1 \leq n \ c2 \leq n$
 $\langle proof \rangle$

context
fixes $M :: ('t::time) \ DBM$
assumes $\llbracket M \rrbracket \neq \{\}$
begin

lemma *non_empty_diag_0_0*: $M \ 0 \ 0 \geq 0$
 $\langle proof \rangle$

lemma *M_k_0*: $M \ k \ 0 \geq 0 \ \text{if} \ \forall \ u \in \llbracket M \rrbracket. \ \forall \ c \leq n. \ c > 0 \longrightarrow u \ c \geq 0 \ k \leq n$
 $\langle proof \rangle$

lemma *non_empty_cycle_free*:
 $cycle_free \ M \ n$
 $\langle proof \rangle$

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** $u \in \llbracket M \rrbracket \ u \ c = - \ r$
 $\langle proof \rangle$

lemma *M_0_k*: $M \ 0 \ k \leq 0$
if $canonical \ M \ n \ M \ 0 \ 0 \leq 0 \ \forall \ u \in \llbracket M \rrbracket. \ \forall \ c \leq n. \ c > 0 \longrightarrow u \ c \geq 0 \ k \leq n$
 $\langle proof \rangle$

end

end

Definition **definition**

$down :: nat \Rightarrow ('t::linordered_cancel_ab_monoid_add) \ DBM \Rightarrow 't \ DBM$
where

$down\ n\ M \equiv$
 $\lambda\ i\ j. \text{ if } i = 0 \wedge j > 0 \text{ then } Min\ (\{Le\ 0\} \cup \{M\ k\ j \mid k. 1 \leq k \wedge k \leq n\}) \text{ else } M\ i\ j$

Correctness **context** *Default_Nat_Clock_Numbering*
begin

sublocale *Alpha_defs* $\{1..n\}$ $\langle proof \rangle$

context
fixes $M :: ('t::time)\ DBM$
begin

lemma *down_complete*: $u \in \llbracket down\ n\ M \rrbracket$ **if** $u \in \llbracket M \rrbracket^\downarrow \forall\ c \leq n. c > 0 \longrightarrow u\ c \geq 0$
 $\langle proof \rangle$

lemma *down_sound*: $u \in \llbracket M \rrbracket^\downarrow$ **if** $u \in \llbracket down\ n\ M \rrbracket$ *canonical* $M\ n$
 $\langle proof \rangle$

lemma *down_canonical*:
canonical (*down* $n\ M$) n
if *assms*: *canonical* $M\ n$ $\llbracket M \rrbracket \neq \{\}$ $\forall\ u \in \llbracket M \rrbracket. \forall\ c \leq n. c > 0 \longrightarrow u\ c \geq 0$ $M\ 0\ 0 \leq 0$
 $\langle proof \rangle$

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 $n\ a\ b\ e\ M \equiv \text{repair_pair}\ n\ (\text{and_entry}\ a\ b\ e\ M)\ a\ b$

definition

restrict_zero $n\ M\ x \equiv$
let
 $M1 = \text{and_entry}\ x\ 0\ (Le\ 0)\ M;$
 $M2 = \text{and_entry}\ 0\ x\ (Le\ 0)\ M1$
in *repair_pair* $n\ M2\ x\ 0$

definition

pre_reset $n\ M\ x \equiv \text{free}\ n\ (\text{restrict_zero}\ n\ M\ x)\ x$

definition

pre_reset_list $n\ M\ r \equiv \text{fold}\ (\lambda\ x\ M. \text{pre_reset}\ n\ M\ x)\ r\ M$

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*) $\vee (\exists i \leq n. \text{repair_pair } n \ M \ a \ b \ i \ i < 0)$
 $\langle \text{proof} \rangle$

lemma *repair_pair_mono*:

assumes $i \leq n$
and $j \leq n$
shows *repair_pair* *n M a b i j* $\leq M \ i \ j$
 $\langle \text{proof} \rangle$

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** $a \leq n \ b \leq n$
 $\langle \text{proof} \rangle$

context

fixes *c1 c2 c x :: nat*
notes [*simp*] = *dbm_entry_val'_iff_bounded dbm_entry_simps DBM.add algebra_simps*
begin

lemma *dbm_entry_val'_diag_iff*: *dbm_entry_val'* *u c c e* $\longleftrightarrow e \geq 0$ **if** $c > 0$
 $\langle \text{proof} \rangle$

lemma *dbm_entry_val'_inf*: *dbm_entry_val'* *u c1 c2* ∞ $\longleftrightarrow \text{True}$
 $\langle \text{proof} \rangle$

lemma *dbm_entry_val'_reset_1*:

dbm_entry_val' (*u*($x := d$)) *x c e* $\longleftrightarrow \text{dbm_entry_val}' \ u \ 0 \ c \ (e + Le \ (-d))$
if $d \geq 0 \ c \neq x \ c > 0 \ x > 0$
 $\langle \text{proof} \rangle$

lemma *dbm_entry_val'_reset_2*:

dbm_entry_val' (*u*($x := d$)) *c x e* $\longleftrightarrow \text{dbm_entry_val}' \ u \ c \ (0 :: nat) \ (e + Le \ d)$
if $d \geq 0 \ c \neq x \ c > 0 \ x > 0$
 $\langle \text{proof} \rangle$

lemma *dbm_entry_val'_reset_2'*:

dbm_entry_val' (*u*($x := d$)) $0 \ x \ e$ $\longleftrightarrow Le \ (-d) \leq e$ **if** $d \geq 0 \ x > 0$
 $\langle \text{proof} \rangle$

lemma *dbm_entry_val'_reset_3*:

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*
 $\langle \text{proof} \rangle$

end

Correctness context

fixes $M :: ('t::time) DBM$

begin

lemma *free_complete*: $u(x := d) \in \llbracket free\ n\ M\ x \rrbracket$
if *assms*: $u \in \llbracket M \rrbracket\ d \geq 0\ x > 0\ \forall c \leq n. M\ c\ c \geq 0$
<proof>

lemma *free_sound*: $\exists d \geq 0. u(x := d) \in \llbracket M \rrbracket\ u\ x \geq 0$
if *assms*: $u \in \llbracket free\ n\ M\ x \rrbracket\ x > 0\ x \leq n\ canonical\ M\ n\ M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$
<proof>

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 $x > 0\ x \leq n\ \forall c \leq n. M\ c\ c \geq 0\ \forall u \in \llbracket M \rrbracket. u\ x \geq 0\ canonical\ M\ n$
 $M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$
<proof>

lemma *pre_reset_correct_aux*:
 $\{u. (u(x := (0::'t))) \in \llbracket M \rrbracket\} \cap \{u. u\ x \geq 0\} = \{u(x := d) \mid u\ d. u \in \llbracket M \rrbracket \wedge u\ x = 0 \wedge d \geq 0\}$
<proof>

lemma *restrict_zero_correct*:
 $\llbracket restrict_zero\ n\ M\ x \rrbracket = \{u. u \in \llbracket M \rrbracket \wedge u\ x = 0\}$ if $0 < x\ x \leq n$
<proof>

lemma *restrict_zero_canonical*:
 $canonical\ (restrict_zero\ n\ M\ x)\ n \vee check_diag\ n\ (uncurry\ (restrict_zero\ n\ M\ x))$
if $canonical\ M\ n\ x \leq n$
<proof>

end

10.4 Structural Properties

lemma *free_canonical*:
 $canonical\ (free\ n\ M\ x)\ n$ if $canonical\ M\ n\ M\ x\ x \geq 0$
<proof>

lemma *free_diag*:
 $free\ n\ M\ x\ i\ i = M\ i\ i$
<proof>

lemma *check_diag_free*:
 $check_diag\ n\ (uncurry\ (free\ n\ M\ x))$ if $check_diag\ n\ (uncurry\ M)$
<proof>

lemma
 $\forall i \leq n. (free\ n\ M\ x)\ i\ i \leq 0$ if $\forall i \leq n. M\ i\ i \leq 0$
<proof>

lemma *canonical_nonneg_diag_non_empty*:
assumes $canonical\ M\ n\ \forall i \leq n. 0 \leq M\ i\ i$
shows $[M]_{v,n} \neq \{\}$

$\langle \text{proof} \rangle$

lemma *V_structuralI*:

$\llbracket M \rrbracket \subseteq V$ **if** $\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0$

$\langle \text{proof} \rangle$

lemma *canonical_V_non_empty_iff*:

assumes *canonical* $M\ n\ M\ 0\ 0 \leq 0$

shows $\llbracket M \rrbracket \subseteq V \wedge \llbracket M \rrbracket \neq \{\}$ $\longleftrightarrow (\forall i \leq n. i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall i \leq n. M\ i\ i \geq 0)$

$\langle \text{proof} \rangle$

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 $(\forall i \leq n. i > 0 \longrightarrow \text{free}\ n\ M\ x\ 0\ i \leq 0) \wedge (\forall i \leq n. \text{free}\ n\ M\ x\ i\ i \geq 0)$

$\langle \text{proof} \rangle$

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 $\llbracket f\ M \rrbracket \subseteq V$

$\langle \text{proof} \rangle$

lemma

$\llbracket \text{free}\ n\ M\ x \rrbracket \subseteq V$ **if** *assms*: $x > 0$ *canonical* $M\ n\ M\ 0\ 0 \leq 0\ 0 \leq M\ x\ x\ \llbracket M \rrbracket \subseteq V$

$\langle \text{proof} \rangle$

lemma

down $n\ M\ i\ i = M\ i\ i$

$\langle \text{proof} \rangle$

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 $(\forall i \leq n. i > 0 \longrightarrow \text{down}\ n\ M\ 0\ i \leq 0) \wedge (\forall i \leq n. \text{down}\ n\ M\ i\ i \geq 0)$

$\langle \text{proof} \rangle$

lemma *check_diag_empty*:

$\llbracket M \rrbracket = \{\}$ **if** *check_diag* $n\ (\text{uncurry}\ M)$

$\langle \text{proof} \rangle$

lemma *restrict_zero_mono*:

restrict_zero $n\ M\ x\ i\ j \leq M\ i\ j$ **if** $i \leq n\ j \leq n$

$\langle \text{proof} \rangle$

lemma *restrict_zero_diag*:

check_diag $n\ (\text{uncurry}\ (\text{restrict_zero}\ n\ M\ x))$ **if** *check_diag* $n\ (\text{uncurry}\ M)$

$\langle \text{proof} \rangle$

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* $M\ n \vee \text{check_diag}\ n\ (\text{uncurry}\ M) \ M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$

$\langle \text{proof} \rangle$

lemma *zone_set_pre_Cons*:

$\text{zone_set_pre } \llbracket M \rrbracket (x \# r) = \text{zone_set_pre } \{u. (u(x := (0::'t::\text{time}))) \in \llbracket M \rrbracket\} r$

$\langle \text{proof} \rangle$

lemma *pre_reset_list_Cons*:

$\text{pre_reset_list } n M (x \# r) = \text{pre_reset_list } n (\text{pre_reset } n M x) r$

$\langle \text{proof} \rangle$

lemma *pre_reset_diag*:

$\text{check_diag } n (\text{uncurry } (\text{pre_reset } n M x)) \text{ if } \text{check_diag } n (\text{uncurry } M)$

$\langle \text{proof} \rangle$

lemma *free_canonical'*:

$\text{canonical } (\text{free } n (M :: (_ :: \text{time}) \text{ DBM}) x) n \vee \text{check_diag } n (\text{uncurry } (\text{free } n M x))$

if $\text{canonical } M n \vee \text{check_diag } n (\text{uncurry } M) x \leq n$

$\langle \text{proof} \rangle$

lemma *pre_reset_canonical'*:

$\text{canonical } (\text{pre_reset } n (M :: (_ :: \text{time}) \text{ DBM}) x) n \vee \text{check_diag } n (\text{uncurry } (\text{pre_reset } n M x))$

if $\text{canonical } M n \vee \text{check_diag } n (\text{uncurry } M) x \leq n$

$\langle \text{proof} \rangle$

lemma *pre_reset_list_correct*:

$\llbracket \text{pre_reset_list } n M r \rrbracket = \text{zone_set_pre } \llbracket M \rrbracket r \cap \{u. \forall x \in \text{set } r. u x \geq 0\}$

if $\forall x \in \text{set } r. x > 0 \wedge x \leq n$

$\text{canonical } M n \vee \text{check_diag } n (\text{uncurry } M) \forall x \in \text{set } r. M 0 x \leq 0 \wedge M 0 0 \leq 0$

$\langle \text{proof} \rangle$

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

$\text{dbm_minus_canonical } n xs M =$

$[\text{and_entry_repair } n j i (\text{neg_dbm_entry } (M i j)) M'.$

$(i, j) \leftarrow [(i, j).$

$i \leftarrow [0..<\text{Suc } n], j \leftarrow [0..<\text{Suc } n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge M i j \neq \infty],$

$M' \leftarrow xs$

$]$

Same as $\text{dbm_minus_canonical}$ but filters out empty DBMs.

definition

$\text{dbm_minus_canonical_check } n xs M =$

$\text{filter } (\lambda M. \neg \text{check_diag } n (\text{uncurry } M)) (\text{dbm_minus_canonical } n xs M)$

Checks whether $\llbracket M \rrbracket - \text{dbm_list } xs = \{\}$.

definition

$\text{dbm_subset_fed } n M xs \equiv$

$\text{let } xs = \text{filter } (\lambda M. \neg \text{check_diag } n (\text{uncurry } M)) xs \text{ in}$

$\text{list_all } (\lambda M. \text{check_diag } n (\text{uncurry } M)) (\text{fold } (\lambda m S. \text{dbm_minus_canonical } n S m) xs [M])$

definition

$dbm_subset_fed_check\ n\ M\ xs \equiv$
 let
 $\quad xs = \text{filter } (\lambda M. \neg \text{check_diag } n\ (\text{uncurry } M))\ xs;$
 $\quad is_direct_subset = \text{list_ex } (\lambda M'. dbm_subset'\ n\ (\text{uncurry } M)\ (\text{uncurry } M'))\ xs$
 $\text{in } is_direct_subset \vee$
 $\quad \text{list_all } (\lambda M. \text{check_diag } n\ (\text{uncurry } M))\ (\text{fold } (\lambda m\ S. dbm_minus_canonical_check\ n\ S$
 $m)\ xs\ [M])$

definition $canonical'\ n\ M \equiv canonical\ M\ n \vee \text{check_diag } n\ (\text{uncurry } M)$

lemma $canonical'I$:

$canonical'\ n\ (f\ M)\ \mathbf{if}$
 $canonical'\ n\ M$
 $canonical\ M\ n \implies canonical'\ n\ (f\ M)\ \text{check_diag } n\ (\text{uncurry } M) \implies \text{check_diag } n\ (\text{uncurry } (f\ M))$
 $\langle proof \rangle$

lemma $check_diag_repair_pair$:

$\mathbf{assumes}$ $check_diag\ n\ (\text{uncurry } M)$
 $\mathbf{shows}$ $check_diag\ n\ (\text{uncurry } (\text{repair_pair } n\ M\ i\ j))$
 $\langle proof \rangle$

lemma $check_diag_and_entry$:

$\mathbf{assumes}$ $check_diag\ n\ (\text{uncurry } M)$
 $\mathbf{shows}$ $check_diag\ n\ (\text{uncurry } (\text{and_entry } a\ b\ e\ M))$
 $\langle proof \rangle$

lemma $canonical'_and_entry_repair$:

$canonical'\ n\ (\text{and_entry_repair } n\ i\ j\ e\ M)\ \mathbf{if}\ canonical'\ n\ M\ i \leq n\ j \leq n$
 $\langle proof \rangle$

lemma $dbm_minus_canonical_canonical'$:

$\forall M \in \text{set } (dbm_minus_canonical\ n\ xs\ m). canonical'\ n\ M\ \mathbf{if}\ \forall M \in \text{set } xs. canonical'\ n\ M$
 $\langle proof \rangle$

lemma $dbm_minus_canonical_check_canonical'$:

$\forall M \in \text{set } (dbm_minus_canonical_check\ n\ xs\ m). canonical'\ n\ M\ \mathbf{if}\ \forall M \in \text{set } xs. canonical'\ n\ M$
 $\langle proof \rangle$

10.5 Correctness of dbm_subset_fed

Misc lemma $list_all_iffI$:

$\mathbf{assumes}$ $\forall x \in \text{set } xs. \exists y \in \text{set } ys. P\ x \longleftrightarrow Q\ y$
 $\mathbf{and}$ $\forall y \in \text{set } ys. \exists x \in \text{set } xs. P\ x \longleftrightarrow Q\ y$
 $\mathbf{shows}$ $list_all\ P\ xs \longleftrightarrow list_all\ Q\ ys$
 $\langle proof \rangle$

lemma $list_all_iff_list_all2I$:

$\mathbf{assumes}$ $list_all2\ (\lambda x\ y. P\ x \longleftrightarrow Q\ y)\ xs\ ys$
 $\mathbf{shows}$ $list_all\ P\ xs \longleftrightarrow list_all\ Q\ ys$
 $\langle proof \rangle$

lemma *list_all2_mapI*:

assumes *list_all2* $(\lambda x y. P (f x) (g y))$ *xs ys*

shows *list_all2* *P* $(\text{map } f \text{ } xs)$ $(\text{map } g \text{ } ys)$

$\langle \text{proof} \rangle$

context *Default_Nat_Clock_Numbering*

begin

lemma *canonical_empty_zone*:

$\llbracket M \rrbracket_{v,n} = \{\}$ $\longleftrightarrow (\exists i \leq n. M \ i \ i < 0)$ **if** *canonical* *M n*

$\langle \text{proof} \rangle$

lemma *check_diag_iff_empty*:

check_diag *n* $(\text{uncurry } M)$ $\longleftrightarrow \llbracket M \rrbracket = \{\}$ **if** *canonical'* *n M*

$\langle \text{proof} \rangle$

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** $a \leq n \ b \leq n$

$\langle \text{proof} \rangle$

lemma *dbm_minus_rel*:

assumes *list_all2* $(\lambda x y. \llbracket x \rrbracket = \llbracket y \rrbracket)$ *ms ms'*

shows *list_all2* $(\lambda x y. \llbracket x \rrbracket = \llbracket y \rrbracket)$ $(\text{dbm_minus } n \ ms \ m)$ $(\text{dbm_minus_canonical } n \ ms' \ m)$

$\langle \text{proof} \rangle$

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* :: $(t :: \text{time}) \text{ DBM list}$

$\langle \text{proof} \rangle$

lemma *not_check_diag_nonnegD*:

$M \ i \ i \geq 0$ **if** $\neg \text{check_diag } n \ (\text{uncurry } M) \ i \leq n$

$\langle \text{proof} \rangle$

theorem *dbm_subset_fed_correct*:

fixes *xs* :: $(\text{nat} \Rightarrow \text{nat} \Rightarrow (t :: \text{time}) \text{ DBMEntry}) \text{ list}$

and *S* :: $(\text{nat} \Rightarrow \text{nat} \Rightarrow t \text{ DBMEntry}) \text{ list}$

assumes *canonical'* *n M*

shows $\llbracket M \rrbracket \subseteq (\bigcup m \in \text{set } xs. \llbracket m \rrbracket) \longleftrightarrow \text{dbm_subset_fed } n \ M \ xs$

$\langle \text{proof} \rangle$

lemma *dbm_minus_canonical_check_fed_equiv*:

dbm_list $(\text{dbm_minus_canonical_check } n \ S \ m) = \text{dbm_list } (\text{dbm_minus_canonical } n \ S \ m)$

$\langle \text{proof} \rangle$

lemma *dbm_minus_canonical_dbm_minus*:

dbm_list $(\text{dbm_minus_canonical } n \ xs \ m) = \text{dbm_list } (\text{dbm_minus } n \ xs \ m)$

$\langle \text{proof} \rangle$

lemma *dbm_minus_canonical_fed_equiv*:

dbm_list $(\text{dbm_minus_canonical } n \ xs \ m) = \text{dbm_list } (\text{dbm_minus_canonical } n \ xs' \ m)$

if *dbm_list* *xs* = *dbm_list* *xs'* $0 \leq m \ 0 \ 0$

$\langle \text{proof} \rangle$

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 $\llbracket M \rrbracket \subseteq (\bigcup_{m \in set\ xs} \llbracket m \rrbracket) \longleftrightarrow$
 $let\ xs = filter\ (\lambda M. \neg check_diag\ n\ (uncurry\ M))\ xs\ in$
 $list_all\ (\lambda M. check_diag\ n\ (uncurry\ M))\ (fold\ (\lambda m\ S. dbm_minus_canonical_check\ n\ S\ m)$
 $xs\ [M]))$
 $\langle proof \rangle$

lemma *subset_if_pointwise_le*:
 $\llbracket M \rrbracket \subseteq \llbracket M' \rrbracket$ **if** *pointwise_cmp* $(\leq)\ n\ M\ M'$
 $\langle proof \rangle$

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 $\llbracket M \rrbracket \subseteq (\bigcup_{m \in set\ xs} \llbracket m \rrbracket) \longleftrightarrow dbm_subset_fed_check\ n\ M\ xs$
 $\langle proof \rangle$

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* ::
 $nat \Rightarrow nat \Rightarrow int\ DBMEntry \Rightarrow int\ DBM' \Rightarrow int\ DBM'$ **where**
 $and_entry_upd\ a\ b\ e\ M = M((a,b) := \min\ (M\ (a,\ b))\ e)$

definition
 $and_entry_repair_upd\ n\ a\ b\ e\ M \equiv$
 $Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair\ n\ (and_entry_upd\ a\ b\ e\ M)\ a\ b$

definition
 $dbm_minus_canonical_upd\ n\ xs\ m =$
 $concat\ (map\ (\lambda(i,j). map\ (\lambda M. and_entry_repair_upd\ n\ j\ i\ (neg_dbm_entry\ (m\ (i,\ j))))\ M)$
 $xs)$
 $[(i,j). i \leftarrow [0..<Suc\ n], j \leftarrow [0..<Suc\ n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ (i,\ j) \neq \infty])$

definition
 $dbm_minus_canonical_check_upd\ n\ xs\ M =$
 $filter\ (\lambda M. \neg check_diag\ n\ M)\ (dbm_minus_canonical_upd\ n\ xs\ M)$

definition
 $dbm_subset_fed_upd\ n\ M\ xs \equiv$
 $let\ xs = filter\ (\lambda M. \neg check_diag\ n\ M)\ xs;$
 $is_direct_subset = list_ex\ (\lambda M'. dbm_subset'\ n\ M\ M')\ xs$
 $in\ is_direct_subset \vee$
 $list_all\ (\lambda M. check_diag\ n\ M)\ (fold\ (\lambda m\ S. dbm_minus_canonical_check_upd\ n\ S\ m)\ xs$

[M])

lemma *list_all_filter_neg*:

list_all P (*filter* $(\lambda x. \neg P x)$ *xs*) $\longleftrightarrow$ (*filter* $(\lambda x. \neg P x)$ *xs*) = []
 ⟨proof⟩

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] = []
 ⟨proof⟩

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 :: $\text{nat} \Rightarrow _ DBM' \Rightarrow _ DBM'$

where

down_upd n $M \equiv \lambda(i, j).$
 if $i = 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n$ then *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
 $M1 = \text{and_entry_upd } x\ 0\ (Le\ 0)\ M;$
 $M2 = \text{and_entry_upd } 0\ x\ (Le\ 0)\ M1$
 in *Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair* n $M2$ $x\ 0$

definition

free_upd :: $\text{nat} \Rightarrow _ DBM' \Rightarrow \text{nat} \Rightarrow _ DBM'$

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 ∞ 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*]:

$(\text{rel_DBMEntry } ri ==> \text{rel_DBMEntry } ri) \text{ neg_dbm_entry neg_dbm_entry}$
 ⟨proof⟩

lemma *fold_min_transfer*:
 $((\text{list_all2 } (\text{rel_DBMEntry } ri)) \implies \text{rel_DBMEntry } ri \implies \text{rel_DBMEntry } ri) \text{ (fold min)}$
 $\langle \text{proof} \rangle$

context
fixes $n :: \text{nat}$
begin

definition
 $RI2 \ M \ D \equiv RI \ n \ (\text{uncurry } M) \ D$

lemma *and_entry_transfer[transfer_rule]*:
 $((=) \implies (=) \implies \text{rel_DBMEntry } ri \implies RI2 \implies RI2) \text{ and_entry and_entry_upd}$
 $\langle \text{proof} \rangle$

lemma *FWI_transfer[transfer_rule]*:
 $(RI2 \implies \text{eq_onp } (\lambda x. x = n) \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies RI2) \text{ FWI FWI' (is ?A)}$
and *FW_transfer[transfer_rule]*:
 $(RI2 \implies \text{eq_onp } (\lambda x. x = n) \implies RI2) \text{ FW FW' (is ?B)}$
 $\langle \text{proof} \rangle$

lemma *repair_pair_transfer[transfer_rule]*:
 $(\text{eq_onp } (\lambda x. x = n) \implies RI2 \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies RI2)$
 $\text{repair_pair Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair}$
 $\langle \text{proof} \rangle$

lemma *and_entry_transfer_weak*:
 $(\text{eq_onp } (\lambda x. x < \text{Suc } n) \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies \text{rel_DBMEntry } ri \implies RI2 \implies RI2)$
 $\text{and_entry and_entry_upd}$
 $\langle \text{proof} \rangle$

lemma *and_entry_repair_transfer[transfer_rule]*:
 $(\text{eq_onp } (\lambda x. x = n) \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies \text{eq_onp } (\lambda x. x < \text{Suc } n) \implies \text{rel_DBMEntry } ri \implies RI2 \implies RI2) \text{ and_entry_repair and_entry_repair_upd}$
 $\langle \text{proof} \rangle$

lemma *dbm_minus_canonical_transfer[transfer_rule]*:
 $(\text{eq_onp } (\lambda x. x = n) \implies \text{list_all2 } RI2 \implies RI2 \implies \text{list_all2 } RI2)$
 $\text{dbm_minus_canonical dbm_minus_canonical_upd}$
 $\langle \text{proof} \rangle$

lemma *check_diag_transfer[transfer_rule]*:
 $(\text{eq_onp } (\lambda x. x = n) \implies RI2 \implies (=)) (\lambda n \ M. \text{check_diag } n \ (\text{uncurry } M)) \text{ check_diag}$
 $\langle \text{proof} \rangle$

lemma *dbm_minus_canonical_check_transfer[transfer_rule]*:
 $(\text{eq_onp } (\lambda x. x = n) \implies \text{list_all2 } RI2 \implies RI2 \implies \text{list_all2 } RI2)$

dbm_minus_canonical_check dbm_minus_canonical_check_upd

<proof>

lemma *le_rel_DBMEntry_iff*:

$a \leq b \iff x \leq y$ **if** *rel_DBMEntry ri a x rel_DBMEntry ri b y*

<proof>

lemma *dbm_subset'_transfer[transfer_rule]*:

$(eq_onp (\lambda x. x = n) ==> RI2 ==> RI2 ==> (=))$
 $(\lambda n M M'. dbm_subset' n (uncurry M) (uncurry M')) dbm_subset'$

<proof>

lemma *dbm_subset_fed_transfer*:

$(eq_onp (\lambda x. x = n) ==> RI2 ==> list_all2 RI2 ==> (=)) dbm_subset_fed_check$
dbm_subset_fed_upd

<proof>

lemma *V_dbm_transfer[transfer_rule]*:

$RI2 V_dbm (V_dbm' n)$

<proof>

lemma *down_transfer[transfer_rule]*:

$(eq_onp (\lambda x. x = n) ==> RI2 ==> RI2) down down_upd$

<proof>

lemma *free_upd[transfer_rule]*:

$(eq_onp (\lambda x. x = n) ==> RI2 ==> (=) ==> RI2) free free_upd$

<proof>

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>

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

<proof>

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

<proof>

lemma *diag_unbounded_dbm*:

unbounded_dbm i i = 0

<proof>

lemma *down_diag*:
 $down\ n\ M\ i\ i = M\ i\ i$
 $\langle proof \rangle$

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*:
 $abstr\ cc\ M\ v\ i\ j \leq M\ i\ j$
 $\langle proof \rangle$

lemma *abstr_FW_mono*:
 $abstr_FW\ n\ cc\ M\ v\ i\ j \leq M\ i\ j$ **if** $i \leq n \wedge j \leq n$
 $\langle proof \rangle$

lemma *abstr_FW_diag_preservation*:
 $\forall k \leq n. abstr_FW\ n\ cc\ M\ v\ k\ k \leq 0$ **if** $\forall k \leq n. M\ k\ k \leq 0$
 $\langle proof \rangle$

lemma *FW_canonical*:
 $canonical'\ n\ (FW\ M\ n)$
 $\langle proof \rangle$

lemma *abstr_FW_canonical*:
 $canonical'\ n\ (abstr_FW\ n\ cc\ M\ v)$
 $\langle proof \rangle$

lemma *down_check_diag*:
 $check_diag\ n\ (uncurry\ (down\ n\ M))$ **if** $check_diag\ n\ (uncurry\ M)$
 $\langle proof \rangle$

context *Default_Nat_Clock_Numbering*
begin

lemma *clock_numbering*:
 $\forall\ c. v\ c > 0 \wedge (\forall x. \forall y. v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)$
 $\langle proof \rangle$

lemma *V_dbm_correct*:
 $\llbracket V_dbm \rrbracket = V$
 $\langle proof \rangle$

lemma *abstr_correct*:
 $\llbracket abstr\ cc\ M\ v \rrbracket = \llbracket M \rrbracket \cap \{u. u \vdash cc\}$ **if** $\forall c \in collect_clks\ cc. 0 < c \wedge c \leq n$
 $\langle proof \rangle$

lemma *unbounded_dbm_correct*:
 $\llbracket unbounded_dbm \rrbracket = UNIV$
 $\langle proof \rangle$

lemma *abstr_correct'*:

$\llbracket \text{abstr } cc \text{ unbounded_dbm } v \rrbracket = \{u. u \vdash cc\}$ **if** $\forall c \in \text{collect_clks } cc. 0 < c \wedge c \leq n$
 $\langle \text{proof} \rangle$

lemma *abstr_FW_correct*:

$\llbracket \text{abstr_FW } n \text{ cc } M \text{ } v \rrbracket = \llbracket M \rrbracket \cap \{u. u \vdash cc\}$ **if** $\forall c \in \text{collect_clks } cc. 0 < c \wedge c \leq n$
 $\langle \text{proof} \rangle$

lemma *abstr_FW_correct'*:

$\llbracket \text{abstr_FW } n \text{ cc unbounded_dbm } v \rrbracket = \{u. u \vdash cc\}$ **if** $\forall c \in \text{collect_clks } cc. 0 < c \wedge c \leq n$
 $\langle \text{proof} \rangle$

lemma *down_V*:

$\llbracket \text{down } n \text{ } M \rrbracket \subseteq V$
 $\langle \text{proof} \rangle$

lemma *down_correct'*:

$\llbracket \text{down } n \text{ } M \rrbracket = \llbracket M \rrbracket^\downarrow \cap V$ **if** *canonical* $M \text{ } n$
 $\langle \text{proof} \rangle$

lemma *down_correct*:

$\llbracket \text{down } n \text{ } M \rrbracket = \llbracket M \rrbracket^\downarrow \cap V$ **if** *canonical'* $n \text{ } M$
 $\langle \text{proof} \rangle$

lemma *pre_reset_diag_preservation*:

$\text{pre_reset } n \text{ } M \text{ } x \text{ } i \leq M \text{ } i \text{ } i$ **if** $i \leq n$
 $\langle \text{proof} \rangle$

lemma *pre_reset_list_diag*:

$\text{pre_reset_list } n \text{ } M \text{ } r \text{ } i \text{ } i \leq M \text{ } i \text{ } i$ **if** $i \leq n$
 $\langle \text{proof} \rangle$

context

includes *lifting_syntax*

begin

lemma *abstra_upd_abstra*:

$\text{abstra_upd } ac \text{ } M \text{ } (i, j) = \text{abstra } ac \text{ } (\text{curry } M) \text{ } v \text{ } i \text{ } j$ **if** $0 < \text{constraint_clk } ac \text{ } i \leq n \text{ } j \leq n$
 $\langle \text{proof} \rangle$

lemma *abstra_transfer[transfer_rule]*:

$(\text{rel_acconstraint } (\text{eq_onp } (\lambda x. 0 < x \wedge x < \text{Suc } n)) \text{ } ri) \implies \text{RI2 } n \implies \text{RI2 } n$
 $(\lambda cc \text{ } M. \text{abstra } cc \text{ } M \text{ } v) \text{ } \text{abstra_upd}$
 $\langle \text{proof} \rangle$

lemma *abstr_transfer[transfer_rule]*:

$(\text{list_all2 } (\text{rel_acconstraint } (\text{eq_onp } (\lambda x. 0 < x \wedge x < \text{Suc } n)) \text{ } ri) \implies \text{RI2 } n \implies \text{RI2 } n$
 $n)$
 $(\lambda cc \text{ } M. \text{abstr } cc \text{ } M \text{ } v) \text{ } \text{abstr_upd}$
 $\langle \text{proof} \rangle$

lemma *abstr_FW_transfer[transfer_rule]*:

```

  (list_all2 (rel_acconstraint (eq_onp ( $\lambda x. 0 < x \wedge x < \text{Suc } n$ )) ri) ==> RI2 n ==> RI2
n)
  ( $\lambda cc M. \text{abstr\_FW } n \text{ cc } M \text{ v}$ ) ( $\text{abstr\_FW\_upd } n$ )
<proof>

```

end

end

```

lemma RI2_trivial_transfer[transfer_rule]: (RI2 n) (curry (conv_M M)) M
<proof>

```

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
<proof>

```

```

lemma constraint_clk_conv_cc:
  collect_clks (conv_cc cc) = collect_clks cc
<proof>

```

```

lemma atLeastLessThan_alt_def:
  {a.. $<b$ } = {k.  $a \leq k \wedge k < b$ }
<proof>

```

```

lemma atLeastLessThan_Suc_alt_def:
  {a.. $<\text{Suc } b$ } = {k.  $a \leq k \wedge k \leq b$ }
<proof>

```

```

lemma (in Graph_Defs) deadlock_if_deadlocked:
  deadlock y if deadlocked y
<proof>

```

10.8 Functional Refinement

```

Elementary list operations lemma map_conv_rev_fold:
  map f xs = rev (fold ( $\lambda a \text{ xs}. f a \# \text{xs}$ ) xs [])
<proof>

```

```

lemma concat_map_conv_rev_fold:
  concat (map f xs) = rev (fold ( $\lambda \text{xs } \text{ys}. \text{rev } (f \text{xs}) @ \text{ys}$ ) xs [])
<proof>

```

```

lemma concat_conv_fold_rev:
  concat xss = fold (@) (rev xss) []
<proof>

```

```

lemma filter_conv_rev_fold:
  filter P xs = rev (fold ( $\lambda x \text{ xs}. \text{if } P x \text{ then } x \# \text{xs} \text{ else } \text{xs}$ ) xs [])
<proof>

```

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.. $\text{Suc } n$ ] M;
    M2 = fold (λi M. if i ≠ c then M((c,i) := ∞) else M) [0.. $\text{Suc } n$ ] 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
⟨proof⟩

```

lemma upd_pairs_Cons:

```

upd_pairs ((p, q) # xs) f = upd_pairs xs (f(p:=q))
⟨proof⟩

```

lemma upd_pairs_no_upd:

```

assumes p ∉ fst ` set xs
shows upd_pairs xs f p = f p
⟨proof⟩

```

lemma upd_pairs_upd:

```

assumes (p, y) ∈ set xs distinct (map fst xs)
shows upd_pairs xs f p = y
⟨proof⟩

```

lemma upd_pairs_append:

```

upd_pairs (xs @ ys) = upd_pairs ys o upd_pairs xs
⟨proof⟩

```

lemma upd_pairs_commute_single:

```

upd_pairs xs (f(a := b)) = (upd_pairs xs f)(a := b) if a ∉ fst ` set xs
⟨proof⟩

```

lemma upd_pairs_append':

```

upd_pairs (xs @ ys) = upd_pairs xs o upd_pairs ys if fst ` set xs ∩ fst ` set ys = {}
⟨proof⟩

```

definition

```
upd_pairs' xs = fold (λ(p,q) f. f(p:=f q)) xs
```

lemma upd_pairs'_Nil:

```
upd_pairs' [] f = f
```

$\langle \text{proof} \rangle$

lemma *upd_pairs'_Cons*:

upd_pairs' ((p, q) # xs) f = upd_pairs' xs (f(p:=f q))

$\langle \text{proof} \rangle$

lemma *upd_pairs'_upd_pairs*:

assumes *fst* ' set *xs* $\cap$ *snd* ' set *xs* = {}

shows *upd_pairs' xs f = upd_pairs (map ($\lambda(p, q). (p, f q)$) xs) f*

$\langle \text{proof} \rangle$

lemma *upd_pairs_map*:

upd_pairs (map f xs) = fold ($\lambda pq g. \text{let } (p, q) = f pq \text{ in } g(p:=q)$) xs

$\langle \text{proof} \rangle$

lemma *down_upd_alt_def*:

down_upd n M =

upd_pairs ([((0, j), fold min [M (k, j). k $\leftarrow$ [1..\leftarrow [1..

$\langle \text{proof} \rangle$

lemma *down_upd_alt_def1*:

down_upd n M =

fold ($\lambda j M. \text{let } (p, q) = ((0, j), \text{fold min [M (k, j). k $\leftarrow$ [1..) [1..$

$\langle \text{proof} \rangle$

lemma

free_upd n M c =

upd_pairs ([((c, i), ∞). i $\leftarrow$ [0..\neq c] @ [((i, c), M(i, 0)). i $\leftarrow$ [0..\neq c]) M

if *c* $\leq$ *n*

$\langle \text{proof} \rangle$

lemma *free_upd_alt_def1*:

free_upd n M c = (let

M1 = upd_pairs' ([((i, c), (i, 0)). i $\leftarrow$ [0..\neq c]) M;

M2 = upd_pairs ([((c, i), ∞). i $\leftarrow$ [0..\neq c]) M1

in

M2

) (is _ = ?r)

if *0 < c* *c* $\leq$ *n*

$\langle \text{proof} \rangle$

lemma *free_upd_free_upd1*:

free_upd n M c = free_upd1 n M c if c > 0 c $\leq$ n

$\langle \text{proof} \rangle$

lemma *free_upd_alt_def*:

free_upd n M c =

fold ($\lambda i M. \text{if } i \neq c \text{ then } (M((c, i) := \infty, (i, c) := M(i, 0))) \text{ else } M$) [0..

if *c* $\leq$ *n*

$\langle \text{proof} \rangle$

lemma *pre_reset_upd_pre_reset_upd1*:

pre_reset_upd n M c = pre_reset_upd1 n M c if c > 0 c $\leq$ n

$\langle \text{proof} \rangle$

lemma *pre_reset_list_upd_pre_reset_list_upd1*:
 $\text{pre_reset_list_upd } n \ M \ cs = \text{pre_reset_list_upd1 } n \ M \ cs$ **if** $\forall c \in \text{set } cs. 0 < c \wedge c \leq n$
 $\langle \text{proof} \rangle$

lemma *pre_reset_list_transfer'*[*transfer_rule*]:
includes *lifting_syntax* **shows**
 $(\text{eq_onp } (\lambda x. x = n) ==> \text{RI2 } n ==> \text{list_all2 } (\text{eq_onp } (\lambda x. 0 < x \wedge x < \text{Suc } n)) ==> \text{RI2 } n)$
 $\text{pre_reset_list } \text{pre_reset_list_upd1}$
 $\langle \text{proof} \rangle$

10.9 Imperative Refinement

context
fixes $n :: \text{nat}$
notes [*id_rules*] = *itypeI*[of $n \ \text{TYPE } (\text{nat})$]
and [*sepref_import_param*] = *IdI*[of n]
begin

interpretation *DBM_Impl* $n \ \langle \text{proof} \rangle$

sepref_definition *down_impl* **is**
 $\text{RETURN } o \ \text{PR_CONST } (\text{down_upd } n) :: \text{mtx_assn}^d \rightarrow_a \text{mtx_assn}$
 $\langle \text{proof} \rangle$

context
notes [*map_type_eqs*] = *map_type_eqI*[of $\text{TYPE } (\text{nat} * \text{nat} \Rightarrow 'e) \ \text{TYPE } ('e \ i_mtx)$]
begin

sepref_register
 $\text{abstr_upd } \text{FW}'' \ n \ \text{Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair } n$
 $\text{and_entry_upd } \text{free_upd1 } n \ \text{restrict_zero_upd } n \ \text{pre_reset_upd1 } n$

end

lemmas [*sepref_fr_rules*] = $\text{abstr_upd_impl.refine } \text{fw_impl_refine_FW}''$

sepref_definition *abstr_FW_impl* **is**
 $\text{uncurry } (\text{RETURN } oo \ \text{PR_CONST } (\text{abstr_FW_upd } n)) ::$
 $(\text{list_assn } (\text{acconstraint_assn } \text{clock_assn } \text{id_assn}))^k *_a \text{mtx_assn}^d \rightarrow_a \text{mtx_assn}$
 $\langle \text{proof} \rangle$

sepref_definition *free_impl* **is**
 $\text{uncurry } (\text{RETURN } oo \ \text{PR_CONST } (\text{free_upd1 } n)) ::$
 $[\lambda(_, i). i \leq n]_a \text{mtx_assn}^d *_a \text{nat_assn}^k \rightarrow \text{mtx_assn}$
 $\langle \text{proof} \rangle$

sepref_definition *and_entry_impl* **is**
 $\text{uncurry2 } (\text{uncurry } (\lambda x. \text{RETURN } ooo \ \text{and_entry_upd } x)) ::$
 $[\lambda(((i, j), _), _). i \leq n \wedge j \leq n]_a \text{nat_assn}^k *_a \text{nat_assn}^k *_a \text{id_assn}^k *_a \text{mtx_assn}^d \rightarrow \text{mtx_assn}$
 $\langle \text{proof} \rangle$

lemmas [sepref_fr_rules] = and_entry_impl.refine

sepref_definition restrict_zero_impl is

uncurry (RETURN oo PR_CONST (restrict_zero_upd n)) ::
 $[\lambda(_, i). i \leq n]_a \text{mtx_assn}^d *_{\alpha} \text{nat_assn}^k \rightarrow \text{mtx_assn}$
 ⟨proof⟩

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)) ::
 $[\lambda(_, i). i \leq n]_a \text{mtx_assn}^d *_{\alpha} (\text{clock_assn})^k \rightarrow \text{mtx_assn}$
 ⟨proof⟩

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)) ::
 $\text{mtx_assn}^d *_{\alpha} (\text{list_assn } (\text{clock_assn}))^k \rightarrow_a \text{mtx_assn}$
 ⟨proof⟩

sepref_definition and_entry_repair_impl is

uncurry2 (uncurry ($\lambda x. \text{RETURN } \text{ooo } \text{PR_CONST } (\text{and_entry_repair_upd } n) x$)) ::
 $[\lambda(((i, j), _), _). i \leq n \wedge j \leq n]_a \text{nat_assn}^k *_{\alpha} \text{nat_assn}^k *_{\alpha} \text{id_assn}^k *_{\alpha} \text{mtx_assn}^d \rightarrow \text{mtx_assn}$
 ⟨proof⟩ **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 $V_dbm' \text{ alt_def}$:

$V_dbm' n = \text{op_amtx_new } (\text{Suc } n) (\text{Suc } n) (V_dbm'')$
 ⟨proof⟩

notation $\text{fun_rel_syn } (\text{infixr } \rightarrow 60)$

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''))))
 $\in \text{unit_assn}^k \rightarrow_a \text{pure } (\text{nat_rel } \times_r \text{ nat_rel } \rightarrow \text{Id})$
 ⟨proof⟩

sepref_register $V_dbm'' :: \text{nat} \times \text{nat} \Rightarrow _ \text{DBMEntry}$

Necessary to solve side conditions of op_amtx_new

lemma $V_dbm'' \text{ bounded}$:

$\text{mtx_nonzero } V_dbm'' \subseteq \{0..<\text{Suc } n\} \times \{0..<\text{Suc } n\}$
 ⟨proof⟩

We need to pre-process the lemmas due to a failure of *TRADE*

lemma $V_dbm'' \text{ bounded_1}$:

$(a, b) \in \text{mtx_nonzero } V_dbm'' \implies a < \text{Suc } n$
 ⟨proof⟩

lemma $V_dbm''_bounded_2$:
 $(a, b) \in mtx_nonzero\ V_dbm'' \implies b < Suc\ n$
 $\langle proof \rangle$

lemmas $[sepref_opt_simps] = V_dbm''_def$

sepref_definition V_dbm_impl is
 $uncurry0\ (RETURN\ (PR_CONST\ (V_dbm'\ n))) :: unit_assn^k \rightarrow_a mtx_assn$
 $\langle proof \rangle$

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 $make_clock[sepref_import_param]$:
 $(make_clock, make_clock) \in [\lambda i. i \leq n]_f\ nat_rel \rightarrow nb_rel\ (Suc\ n)$
 $\langle proof \rangle$ **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 $get_entries_alt_def$:
 $get_entries\ m = [(make_clock\ i, make_clock\ j).$
 $i \leftarrow [0..<Suc\ n], j \leftarrow [0..<Suc\ n], (i > 0 \vee j > 0) \wedge m\ (i, j) \neq \infty]$
 $\langle proof \rangle$ **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
 $upd_entries\ i\ j\ m = map\ (\lambda\ M. upd_entry\ i\ j\ M\ m)$

context
notes $[map_type_eqs] = map_type_eqI[of\ TYPE(nat * nat \Rightarrow 'e)\ TYPE('e\ i_mtx)]$
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 $[sepref_import_param]$: $(neg_dbm_entry, neg_dbm_entry) \in Id \rightarrow Id\ \langle proof \rangle$

lemmas $[sepref_fr_rules] = and_entry_repair_impl.refine$

sepref_definition upd_entry_impl is
 $uncurry2\ (uncurry\ (\lambda x. RETURN\ ooo\ PR_CONST\ upd_entry\ x)) ::$
 $[\lambda(((i, j), _), _). i \leq n \wedge j \leq n]_a$
 $nat_assn^k *_{\alpha} nat_assn^k *_{\alpha} mtx_assn^k *_{\alpha} mtx_assn^k \rightarrow mtx_assn$
 $\langle proof \rangle$

This is to ensure that the refinement initially infers the right 'type' for list arguments.

lemma $[intf_of_assn]$:
 $intf_of_assn\ AA\ TYPE('aa) \implies intf_of_assn\ (list_assn(AA))\ TYPE('aa\ list)$

$\langle \text{proof} \rangle$

lemmas [sepref_fr_rules] = upd_entry_impl.refine

sepref_definition upd_entries_impl is

uncurry2 (uncurry ($\lambda x. \text{RETURN } \text{ooo } \text{PR_CONST } \text{upd_entries } x$)) ::
 $[\lambda((i, j), _). i \leq n \wedge j \leq n]_a$
 $\text{nat_assn}^k *_a \text{nat_assn}^k *_a \text{mtx_assn}^k *_a (\text{list_assn } \text{mtx_assn})^k \rightarrow \text{list_assn } \text{mtx_assn}$
 $\langle \text{proof} \rangle$

lemma [sepref_import_param]: ((=), (=)) $\in \text{Id} \rightarrow \text{Id} \rightarrow \text{Id}$ $\langle \text{proof} \rangle$

sepref_definition get_entries_impl is

$\text{RETURN } o \text{ PR_CONST } \text{get_entries} ::$
 $\text{mtx_assn}^k \rightarrow_a \text{list_assn } ((\text{clock_assn}) \times_a (\text{clock_assn}))$
 $\langle \text{proof} \rangle$

lemmas [sepref_fr_rules] = upd_entries_impl.refine get_entries_impl.refine

private lemma dbm_minus_canonical_upd_alt_def:

$\text{dbm_minus_canonical_upd } n \text{ xs } m =$
 $\text{concat } (\text{map } (\lambda ij. \text{map } (\lambda M.$
 $\text{and_entry_repair_upd } n (\text{snd } ij) (\text{fst } ij) (\text{neg_dbm_entry } (m (\text{fst } ij, \text{snd } ij))) (\text{op_mtx_copy}$
 $M))$
 $\text{xs}) (\text{get_entries } m))$
 $\langle \text{proof} \rangle$

sepref_definition dbm_minus_canonical_impl is

uncurry ($\text{RETURN } \text{oo } \text{PR_CONST } (\text{dbm_minus_canonical_check_upd } n)$) ::
 $(\text{list_assn } \text{mtx_assn})^k *_a \text{mtx_assn}^k \rightarrow_a \text{list_assn } \text{mtx_assn}$
 $\langle \text{proof} \rangle$

lemmas [sepref_fr_rules] = dbm_minus_canonical_impl.refine

sepref_definition dbm_subset_fed_impl is

uncurry ($\text{RETURN } \text{oo } \text{PR_CONST } (\text{dbm_subset_fed_upd } n)$) ::
 $\text{mtx_assn}^d *_a (\text{list_assn } \text{mtx_assn})^d \rightarrow_a \text{bool_assn}$
 $\langle \text{proof} \rangle$

end

10.10 Implementation for a Concrete Automaton

context TA_Impl

begin

sublocale TA_Impl_Op

where $\text{loc_rel} = \text{loc_rel}$ **and** $f = \text{PR_CONST } E_{\text{op}}''$ **and** $\text{op_impl} = E_{\text{op}}''_{\text{impl}}$
 $\langle \text{proof} \rangle$

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 $\equiv$ [curry (conv_M M)]_{v,n}

theorem check_deadlock_dbm_correct:

```

TA.check_deadlock l  $\llbracket M \rrbracket$  = check_deadlock_dbm l M if
 $\llbracket M \rrbracket \subseteq V \mid \in \text{states}' \text{Deadlock.canonical}' n (\text{curry } (\text{conv\_M } M))$ 
<proof> include lifting_syntax
<proof>

```

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 →a bool_assn
<proof>

```

end

10.11 Checking a Property on the Reachable Set

locale Worklist_Map2_Impl_check =

```

Worklist_Map2_Impl_finite +
fixes Q :: 'a ⇒ bool
fixes Qi
assumes Q_refine: (Qi, RETURN o PR_CONST Q) ∈ Ad →a bool_assn
and F_False: F = (λ _. 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) ← pw_algo_map2;
  ASSERT (finite (dom passed));
  passed ← ran_of_map passed;
  r ← nfoldli passed (λ b. ¬b)
  (λ passed' _.
    do {
      passed' ← SPEC (λ l. set l = passed');
      nfoldli passed' (λ b. ¬b)
      (λ v' _.
        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⟩

schematic_goal pw_algo_map2_refine:
  (?x, uncurry0 (PR_CONST pw_algo_map2)) ∈
  unit_assnk →a bool_assn ×a hm.hms_assn' K (lso_assn A)
  ⟨proof⟩

sepref_register pw_algo_map2

sepref_register PR_CONST Q

sepref_thm check_passed_impl is
  uncurry0 check_passed :: unit_assnk →a bool_assn
  ⟨proof⟩

concrete_definition (in -) check_passed_impl
  uses Worklist_Map2_Impl_check.check_passed_impl.refine_raw is (uncurry0 ?f,_) ∈

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
  ⟨proof⟩

end

## 10.12 Complete Deadlock Checker

Miscellaneous Properties context E_From_Op_Bisim
begin

More general variant of ?F_rel = (λ(l, D). ?F l ∧ ¬ check_diag n D) ⇒ (∃ l' D'. E** a0 (l', D') ∧ ?F_rel (l', D')) = (∃ l' D'. E_from_op** a0 (l', D') ∧ ?F_rel (l', D')) *)

theorem E_from_op_reachability_check:
  assumes ∧a b. P a ⇒ a ~ b ⇒ wf_state a ⇒ wf_state b ⇒ P b
  shows
  (∃ l' D'. E** a0 (l', D') ∧ P (l', D')) ↔ (∃ l' D'. E_from_op** a0 (l', D') ∧ P (l', D'))
  ⟨proof⟩

end

context Regions_TA
begin

lemma check_deadlock_anti_mono:
  check_deadlock l Z if check_deadlock l Z' Z ⊆ Z'
  ⟨proof⟩

```

lemma *global_clock_numbering*:

global_clock_numbering A v n

$\langle \text{proof} \rangle$

Variant of $\llbracket (\forall c. 0 < v\ c \wedge (\forall x\ y. v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)) \wedge (\forall c \in \text{clk_set } ?A. v\ c \leq n) \wedge (\forall k \leq n. 0 < k \longrightarrow (\exists c. v\ c = k)) \rrbracket$; *Closure.valid_abstraction* $?A\ X\ (\lambda x\ xa. \text{real } (k\ x\ xa)) \rrbracket \implies \text{Bisimulation_Invariant } (\lambda(l, Z)\ (l', Z'). ?A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\})\ (\lambda(l, D)\ (l', D'). \exists a. ?A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}a} \langle l', D' \rangle \wedge [D]_{v,n} \neq \{\})\ (\lambda(l, Z)\ (l', D). l = l' \wedge Z = [D]_{v,n})\ (\lambda_. \text{True})\ (\lambda(l, y). \text{local.valid_dbm } y)$ without emptiness.

lemma *bisim*:

Bisimulation_Invariant

$(\lambda(l, Z)\ (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle)$

$(\lambda(l, D)\ (l', D'). \exists a. A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle)$

$(\lambda(l, Z)\ (l', D). l = l' \wedge Z = [D]_{v,n})$

$(\lambda_. \text{True})\ (\lambda(l, D). \text{valid_dbm } D)$

$\langle \text{proof} \rangle$

lemma *steps_z_beta_reaches*:

reaches $(l, Z)\ (l', Z')$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle\ Z' \neq \{\}$

$\langle \text{proof} \rangle$

including *graph_automation_aggressive* $\langle \text{proof} \rangle$

lemma *reaches_steps_z_beta_iff*:

reaches $(l, Z)\ (l', Z') \iff A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle \wedge Z' \neq \{\}$ **if** $Z \neq \{\}$

$\langle \text{proof} \rangle$

including *graph_automation_aggressive* $\langle \text{proof} \rangle$

end

lemma *dbm_int_init_dbm*:

dbm_int (*curry* *init_dbm*) n

$\langle \text{proof} \rangle$

context *TA_Start*

begin

lemma *wf_dbm_canonical'D*:

Deadlock.canonical' n (*curry* (*conv_M* D)) **if** *wf_dbm* D

$\langle \text{proof} \rangle$

lemma *subsumes_dbm_subsetD*:

dbm_subset $n\ M\ M'$ **if** *subsumes* $n\ (l, M)\ (l', M') \multimap \text{check_diag } n\ M$

$\langle \text{proof} \rangle$

lemma *subsumes_loc_eqD*:

$l = l'$ **if** *subsumes* $n\ (l, M)\ (l', M') \multimap \text{check_diag } n\ M$

$\langle \text{proof} \rangle$

lemma *init_dbm_zone*:

$[\text{curry } (\text{init_dbm} :: \text{real } \text{DBM}')]_{v,n} = \{u. \forall c \in \{1..n\}. u\ c = 0\}$

$\langle \text{proof} \rangle$

lemma *not_check_deadlock_mono*:

$(\text{case } a \text{ of } (l, M) \Rightarrow \neg \text{TA.check_deadlock } l \text{ (dbm.zone_of (curry (conv_M M)))) \Rightarrow$
 $a \sim b \Rightarrow \text{wf_state } a \Rightarrow \text{wf_state } b \Rightarrow$
 $\text{case } b \text{ of } (l, M) \Rightarrow \neg \text{TA.check_deadlock } l \text{ (dbm.zone_of (curry (conv_M M))))$
 $\langle \text{proof} \rangle$

lemma *not_check_deadlock_non_empty*:
 $Z \neq \{\}$ **if** $\neg \text{TA.check_deadlock } l \text{ } Z$
 $\langle \text{proof} \rangle$

end

context *TA_Impl*
begin

lemma *op_E_from_op_iff*:
 $\text{op.E_from_op} = \text{E_op''}.E_from_op$
 $\langle \text{proof} \rangle$

lemmas *reachable_states* = *reachable_states*[*unfolded op_E_from_op_iff*]

lemma *E_op''_states*:
 $l' \in \text{states}$ **if** $\text{E_op''}.E_from_op \text{ (} l, M \text{) (} l', M' \text{) } l \in \text{states}$
 $\langle \text{proof} \rangle$

Instantiating the Checker Algorithm **corollary** *check_deadlock_dbm_correct'*:
assumes $l \in \text{states'}$ $\text{wf_state } (l, M)$
shows $\text{TA.check_deadlock } l \text{ (dbm.zone_of (curry (conv_M M)))} = \text{check_deadlock_dbm } l \text{ } M$
 $\langle \text{proof} \rangle$

corollary *check_deadlock_dbm_correct''*:
assumes $\text{E_op''}.E_from_op^{**} a_0 \text{ (} l, M \text{)}$
shows $\text{TA.check_deadlock } l \text{ (dbm.zone_of (curry (conv_M M)))} = \text{check_deadlock_dbm } l \text{ } M$
 $\langle \text{proof} \rangle$

lemma *not_check_deadlock_non_empty*:
 $Z \neq \{\}$ **if** $\neg \text{TA.check_deadlock } l \text{ } Z$
 $\langle \text{proof} \rangle$

sempref_register *check_deadlock_dbm* :: 's $\Rightarrow$ int DBMEntry i_mtx $\Rightarrow$ bool

sempref_definition *check_deadlock_neg_impl* **is**
 $\text{RETURN } o \text{ (} \lambda(l, M). \neg \text{check_deadlock_dbm } l \text{ } M \text{) :: (location_assn } \times_a \text{ (mtx_assn } n \text{))}^d \rightarrow_a$
 bool_assn
 $\langle \text{proof} \rangle$

lemma *not_check_deadlock_compatible*:
assumes
 $(\text{case } a \text{ of } (l, Z) \Rightarrow \lambda(l', D'). l' = l \wedge \text{dbm.zone_of (curry (conv_M } D')) = Z) \text{ } b$
 $\text{case } b \text{ of } (l, M) \Rightarrow l \in \text{states'} \wedge \text{wf_state } (l, M)$
shows
 $(\text{case } a \text{ of } (l, Z) \Rightarrow \neg \text{TA.check_deadlock } l \text{ } Z) = (\text{case } b \text{ of } (l, M) \Rightarrow \neg \text{check_deadlock_dbm } l \text{ } M)$
 $\langle \text{proof} \rangle$

lemma *deadlock_check_diag*:
 $\neg \text{check_diag } n \ M \text{ if } \neg \text{check_deadlock_dbm } l \ M \ E_op''.E_from_op^{**} \ a_0 \ (l, \ M)$
 $\langle \text{proof} \rangle$

lemma *norm_final_bisim*:
Bisimulation_Invariant
 $(\lambda(l, \ D) \ (l', \ D'). \exists a. \text{step_z_norm}'' \ (\text{conv_A } A) \ l \ D \ a \ l' \ D')$
 $E_op''.E_from_op$
 $(\lambda \ (l, \ M) \ (l', \ D'). l' = l \wedge [\text{curry } (\text{conv_M } D')]_{v,n} = [M]_{v,n})$
 $(\lambda(l, \ y). \text{valid_dbm } y) \ \text{wf_state}$
 $\langle \text{proof} \rangle$

interpretation *bisim*:
Bisimulation_Invariant
 $\lambda(l, \ Z) \ (l', \ Z'). \text{step_z_beta}' \ (\text{conv_A } A) \ l \ Z \ l' \ Z'$
 $\lambda a \ b. \text{op}.E_from_op \ a \ b$
 $\lambda(l, \ Z) \ (l', \ D'). l' = l \wedge [\text{curry } (\text{conv_M } D')]_{v,n} = Z$
 $\lambda_ . \text{True } \lambda(l, \ M). l \in \text{states} \wedge \text{wf_state } (l, \ M)$
 $\langle \text{proof} \rangle$

lemmas *beta_final_bisim* = *bisim.Bisimulation_Invariant_axioms*

definition
 $\text{is_start_in_states} = (\text{trans_fun } l_0 \neq [])$

lemma *is_start_in_states*:
 $l_0 \in \text{Simulation_Graphs_TA.state_set } A \text{ if } \text{is_start_in_states}$
 $\langle \text{proof} \rangle$

lemma *deadlocked_if_not_is_start_in_states*:
 $\text{deadlocked } (l_0, \ Z_0) \text{ if } \neg \text{is_start_in_states}$
 $\langle \text{proof} \rangle$

lemma *deadlock_if_not_is_start_in_states*:
 $\text{deadlock } (l_0, \ Z_0) \text{ if } \neg \text{is_start_in_states}$
 $\langle \text{proof} \rangle$

sempref_definition *is_start_in_states_impl* is
 $\text{uncurry0 } (\text{RETURN } \text{is_start_in_states}) :: \text{unit_assn}^k \rightarrow_a \text{bool_assn}$
 $\langle \text{proof} \rangle$

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*:
 $\langle \text{emp} \rangle \text{is_start_in_states_impl}$
 $\langle \lambda r. \uparrow ((r \longrightarrow l_0 \in \text{Simulation_Graphs_TA.state_set } A)$
 $\quad \wedge (\neg r \longrightarrow \text{deadlocked } (l_0, \ \lambda_ . 0)))$
 $\rangle_{>t}$
 $\langle \text{proof} \rangle$

lemma *deadlock_if_deadlocked*:
 $\text{deadlock } y \text{ if } \text{deadlocked } y$

$\langle \text{proof} \rangle$

lemma *is_start_in_states_impl_hoare'*:

<emp> is_start_in_states_impl
 $\langle \lambda r. \uparrow ((r \longrightarrow l_0 \in \text{Simulation_Graphs_TA.state_set } A)$
 $\quad \wedge (\neg r \longrightarrow (\exists u_0. (\forall c \in \{1..n\}. u_0 \text{ } c = 0) \wedge \text{deadlock } (l_0, u_0)))$
 $\quad \rangle)_{>t}$
 $\langle \text{proof} \rangle$

end

context *Reachability_Problem_Impl*

begin

context

assumes $l_0_in_A: l_0 \in \text{Simulation_Graphs_TA.state_set } A$

begin

interpretation *TA*:

Regions_TA_Start_State $v \ n \ Suc \ n \ \{1..<Suc \ n\} \ k \ conv_A \ A \ l_0 \ \{u. \forall c \in \{1..n\}. u \text{ } c = 0\}$
 $\langle \text{proof} \rangle$

interpretation *bisim*:

Bisimulation_Invariant
 $\lambda(l, Z) \ (l', Z'). \text{step_z_beta}' (conv_A \ A) \ l \ Z \ l' \ Z'$
 $\lambda a \ b. \text{op.E_from_op } a \ b$
 $\lambda(l, Z) \ (l', D'). \ l' = l \wedge [\text{curry } (conv_M \ D')]_{v,n} = Z$
 $\lambda_ . \text{True } \lambda(l, M). \ l \in \text{states} \wedge \text{wf_state } (l, M)$
 $\langle \text{proof} \rangle$

lemma *check_deadlock*:

$(\exists a. \text{E_op}'' . \text{E_from_op}^{**} \ a_0 \ a \wedge$
 $\quad \neg (\text{case } a \text{ of } (l, M) \Rightarrow \text{check_diag } n \ M) \wedge (\text{case } a \text{ of } (l, M) \Rightarrow \neg \text{check_deadlock_dbm } l \ M))$
 $\longleftrightarrow (\exists u_0. (\forall c \in \{1..n\}. u_0 \text{ } c = 0) \wedge \text{deadlock } (l_0, u_0)) \ (\text{is } ?l \longleftrightarrow ?r)$
 $\langle \text{proof} \rangle$

lemma *check_deadlock'*:

$(\nexists a. \text{E_op}'' . \text{E_from_op}^{**} \ a_0 \ a \wedge$
 $\quad \neg (\text{case } a \text{ of } (l, M) \Rightarrow \text{check_diag } n \ M) \wedge (\text{case } a \text{ of } (l, M) \Rightarrow \neg \text{check_deadlock_dbm } l \ M))$
 $\longleftrightarrow (\forall u_0. (\forall c \in \{1..n\}. u_0 \text{ } c = 0) \longrightarrow \neg \text{deadlock } (l_0, u_0)) \ (\text{is } ?l \longleftrightarrow ?r)$
 $\langle \text{proof} \rangle$

context

assumes $F = (\lambda _. \text{False})$

begin

interpretation *Worklist_Map2_Impl_check*

op.E_from_op $a_0 \ F_rel \text{subsumes } n \ \text{succs } \lambda \ (l, M). \text{check_diag } n \ M \text{subsumes}'$
 $\lambda \ (l, M). \ F \ l \ \text{state_assn}'$
 $\text{succs_impl } a_0 \ \text{impl } F \ \text{impl } \text{subsumes_impl } \text{emptiness_check_impl } \text{fst return o fst state_copy_impl}$
 $\text{tracei location_assn } \lambda(l, M). \neg \text{check_deadlock_dbm } l \ M \ \text{check_deadlock_neg_impl}$
 $\langle \text{proof} \rangle$

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  $\equiv$  do {
  r1  $\leftarrow$  is_start_in_states_impl;
  if r1 then do {
    r2  $\leftarrow$  check_passed_impl succs_impl a0_impl F_impl subsumes_impl emptiness_check_impl
      (return  $\circ$  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 ( $\exists u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \wedge \text{deadlock } (l_0, u_0)))$ )
)  $\in$  unit_assnk  $\rightarrow_a$  bool_assn if F = ( $\lambda\_.$  False)
<proof>

```

end

context *Reachability_Problem_precompiled*

begin

lemma *F_is_False_iff*:

```

PR_CONST F = ( $\lambda\_.$  False)  $\longleftrightarrow$  final = []
<proof>

```

lemma *F_impl_False*: *F_impl* = ($\lambda_.$ return *False*) **if** *final* = []

<proof>

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 ( $\exists u_0. (\forall c \in \{1..m\}. u_0 \ c = 0) \wedge \text{deadlock}$ 
    ( $0, u_0$ ))))
     $\in \text{unit\_assn}^k \rightarrow_a \text{bool\_assn}$ 
    <proof>

schematic_goal deadlock_checker_alt_def:
  deadlock_checker  $\equiv$  ?impl
  <proof>

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  $m \ k \ I \ T \gg= (\lambda x. \text{return } (\text{Some } x))$ 
    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
             ( $\lambda(l, u) \ (l', u').$ 
              ( $\text{conv\_A } (\text{Reachability\_Problem\_precompiled\_defs.A } n \ I \ T) \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
              ( $0, u_0$ )
            then True
            else False
          )
        else None
      )
    )
  )  $\in \text{unit\_assn}^k \rightarrow_a \text{id\_assn}$ 
  <proof>

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*

```

( $\lambda(l, u) (l', u'). \text{conv\_}A \ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
( $\lambda(l, u) (l', u'). \text{conv\_}A \ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ )
( $\lambda(l, u) (l', u'). l' = l \wedge (\forall c. c \in \text{clk\_set} (\text{conv\_}A \ A) \longrightarrow u \ c = u' \ c)$ )
( $\lambda\_.$  True) ( $\lambda\_.$  True)
<proof>

```

lemma *deadlock_zero_clock_val_iff*:

```

( $\exists u_0. (\forall c \in \{1..n\}. u_0 \ c = 0) \wedge \text{deadlock} (l_0, u_0) \longleftrightarrow \text{deadlock} (l_0, \lambda\_.$  0))
<proof>

```

context

assumes *F_fun_False*: $\bigwedge x. F_fun \ x = \text{False}$ **and** *F_False*: *F* = ($\lambda_.$ *False*)

begin

lemma *F_impl_is_False*:

```

F_impl = ( $\lambda\_.$  return False)
<proof>

```

lemma *deadlock_checker_correct*:

```

(uncurry0 deadlock_checker, uncurry0 (Refine_Basic.RETURN (deadlock (l0,  $\lambda\_.$  0))))
 $\in \text{unit\_assn}^k \rightarrow_a \text{bool\_assn}$ 
<proof>

```

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 ( $\lambda(x, y). F\ x\ y$ ) = ( $\lambda_. False$ )
   $\langle proof \rangle$ 

lemma final_fun_is_False:
  final_fun a = False
   $\langle proof \rangle$ 

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
   $\langle proof \rangle$ 

schematic_goal deadlock_checker_alt_def_refined:
  impl.deadlock_checker  $\equiv$  ?impl
   $\langle proof \rangle$ 

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
     $\gg (\lambda x. \text{return } (\text{Some } x))$ 
  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>

```

export_code *precond_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* ($_ \sqcup _$ [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* $\implies$ *y*

shows *TAG t y*

<proof>

lemma *TAG_mp'*:

assumes *TAG t x x* $\implies$ *y*

shows *y*

<proof>

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*

<proof>

lemma *Ball_mono*:

assumes $\forall x \in S. P\ x \wedge x \in S \implies P\ x \implies Q\ x$

shows $\forall x \in S. Q\ x$

<proof>

```

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 ≡ λ(C, U, T, I).
    (C, {}, (λ(l, b, g, a, f, r, l'). (l, b, g, a, f, urge # r, l')) ' T, I)

definition
  add_inv ≡ λ(C, U, T, I). (C, U, T,
    (λl. if l ∈ U then acconstraint.LE urge 0 # I l else I l))

definition A_urge ≡
  (fst A, map add_reset (map add_inv (fst (snd A))), snd (snd A))

definition
  N_urge i ≡ 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:
    ∀(_, _, _, I) ∈ set (fst (snd A)). ∀l. urge ∉ constraint_clk ' set (I l)
  assumes urge_not_in_guards:
    ∀(_, _, T, _) ∈ set (fst (snd A)). ∀(l, b, g, a, f, r, l') ∈ T. urge ∉ constraint_clk ' set g
  assumes urge_not_in_resets:
    ∀(_, _, T, _) ∈ set (fst (snd A)). ∀(l, b, g, a, f, r, l') ∈ T. urge ∉ set r
begin

lemma N_urge_simp:
  N_urge i = add_reset (add_inv (N i)) if i < n_ps
  ⟨proof⟩

lemma inv_add_reset:
  inv (add_reset A') = inv A'
  ⟨proof⟩

lemma inv_add_inv:
  inv (add_inv (C, U, T, I)) = (λl. if l ∈ U then acconstraint.LE urge 0 # I l else I l)
  ⟨proof⟩

definition
  is_urgent L ≡ ∃ p < n_ps. L ! p ∈ urgent (N p)

lemma map_nth':
  map (!) xs [0.. $n$ ] = xs if n = length xs
  ⟨proof⟩

```

lemma *A_urge_split*:

$A_urge = (\text{broadcast}, \text{map } N_urge [0..<n_ps], \text{bounds})$
 $\langle \text{proof} \rangle$

lemma *inv_add_inv'*:

$\text{inv } (\text{add_inv } A') =$
 $(\lambda l. \text{ if } l \in \text{urgent } A' \text{ then } \text{acconstraint.LE } \text{urge } 0 \# \text{ inv } A' l \text{ else } \text{inv } A' l)$
 $\langle \text{proof} \rangle$

lemma *A_split'*:

$A = (\text{fst } A, \text{fst } (\text{snd } A), \text{snd } (\text{snd } A))$
 $\langle \text{proof} \rangle$

lemma *is_urgent_simp*:

$(\exists p < n_ps. L ! p \in \text{urgent } (\text{map } N [0..<n_ps] ! p)) \longleftrightarrow \text{is_urgent } L$
 $\langle \text{proof} \rangle$

lemma *urgent_N_urge*:

$\text{urgent } (N_urge p) = \{\}$ **if** $p < n_ps$
 $\langle \text{proof} \rangle$

lemma *committed_N_urge*:

$\text{committed } (N_urge p) = \text{committed } (N p)$ **if** $p < n_ps$
 $\langle \text{proof} \rangle$

lemma *is_urgent_simp2*:

$(\exists p < n_ps. L ! p \in \text{urgent } (\text{map } N_urge [0..<n_ps] ! p)) \longleftrightarrow \text{False}$
 $\langle \text{proof} \rangle$

lemma *inv_add_invI1*:

$u \vdash \text{inv } (\text{add_inv } (N p)) (L ! p)$ **if** $u \vdash \text{inv } (N p) (L ! p) \neg \text{is_urgent } L p < n_ps$
 $\langle \text{proof} \rangle$

lemma *inv_add_invI2*:

$u \vdash \text{inv } (\text{add_inv } (N p)) (L ! p)$ **if** $u \vdash \text{inv } (N p) (L ! p) u \text{urge} \leq 0 p < n_ps$
 $\langle \text{proof} \rangle$

lemma *inv_add_invD*:

$u \vdash \text{inv } A' l$ **if** $u \vdash \text{inv } (\text{add_inv } A') l$
 $\langle \text{proof} \rangle$

lemma *inv_N_urge*:

$\text{inv } (N_urge p) = \text{inv } (\text{add_inv } (N p))$ **if** $p < n_ps$
 $\langle \text{proof} \rangle$

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 $i < n_ps$
 $\langle \text{proof} \rangle$

lemma *trans_N_urgeD*:

$(l, b, g, a, f, \text{urge} \# r, l') \in \text{trans } (N_urge p)$
if $(l, b, g, a, f, r, l') \in \text{trans } (N p) p < n_ps$

$\langle proof \rangle$

lemma *trans_N_urgeE*:

assumes $(l, b, g, a, f, r, l') \in \text{trans } (N_urges\ p)\ p < n_ps$

obtains $(l, b, g, a, f, tl\ r, l') \in \text{trans } (N\ p)\ r = urges\ \# \ tl\ r$

$\langle proof \rangle$

lemma *urges_not_in_inv*:

$urges \notin \text{constraint_clk } 'set\ (inv\ (N\ p)\ l)\ \text{if } p < n_ps$

$\langle proof \rangle$

lemma *urges_not_in_guards'*:

assumes $(l, b, g, a, f, r, l') \in \text{trans } (N\ p)\ p < n_ps$

shows $urges \notin \text{constraint_clk } 'set\ g$

$\langle proof \rangle$

lemma *urges_not_in_resets'*:

assumes $(l, b, g, a, f, r, l') \in \text{trans } (N\ p)\ p < n_ps$

shows $urges \notin \text{set } r$

$\langle proof \rangle$

lemma *clk_upd_clock_val_a_simp*:

$u(c := d) \vdash_a ac \longleftrightarrow u \vdash_a ac\ \text{if } c \neq \text{constraint_clk } ac$

$\langle proof \rangle$

lemma *clk_upd_clock_val_simp*:

$u(c := d) \vdash cc \longleftrightarrow u \vdash cc\ \text{if } c \notin \text{constraint_clk } 'set\ cc$

$\langle proof \rangle$

lemma *inv_N_urgesI*:

assumes $u \vdash inv\ (N\ p)\ l\ p < n_ps$

shows $u(urges := 0) \vdash inv\ (N_urges\ p)\ l$

$\langle proof \rangle$

lemma *inv_N_urges_updI*:

assumes $u \vdash inv\ (N\ p)\ l\ p < n_ps$

shows $u(urges := d) \vdash inv\ (N\ p)\ l$

$\langle proof \rangle$

lemma *inv_N_urges_upd_simp*:

assumes $p < n_ps$

shows $u(urges := d) \vdash inv\ (N\ p)\ l \longleftrightarrow u \vdash inv\ (N\ p)\ l$

$\langle proof \rangle$

lemma *inv_N_urges_updD*:

assumes $u(urges := d) \vdash inv\ (N\ p)\ l\ p < n_ps$

shows $u \vdash inv\ (N\ p)\ l$

$\langle proof \rangle$

lemma *inv_N_urgesD*:

assumes $u \vdash inv\ (N_urges\ p)\ l\ p < n_ps$

shows $u(urges := d) \vdash inv\ (N\ p)\ l$

$\langle proof \rangle$

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 $d \leq 0$
 $\langle \text{proof} \rangle$

lemma *trans_urge_upd_iff*:
assumes $(l, b, g, a, f, r, l') \in \text{trans } (N \ p) \ p < n_ps$
shows $u(\text{urge} := d) \vdash g \longleftrightarrow u \vdash g$
 $\langle \text{proof} \rangle$

lemma *cval_add_0[simp]*:
 $u \oplus (0 :: 'tt :: \text{time}) = u$
 $\langle \text{proof} \rangle$

lemma *clock_val_reset_delay*:
 $u(c := 0) \oplus (d :: 'tt :: \text{time}) = (u \oplus d)(c := d)$
 $\langle \text{proof} \rangle$

lemma *clock_set_upd_simp*:
 $[r \rightarrow d]u(c := d') = [r \rightarrow d]u \ \text{if } c \in \text{set } r$
 $\langle \text{proof} \rangle$

lemma *fun_upd_twist2*:
 $f(a := x, b := x) = f(b := x, a := x)$
 $\langle \text{proof} \rangle$

lemma *clock_set_upd_simp*:
 $[c \# r \rightarrow d]u(c := d') = [c \# r \rightarrow d]u$
 $\langle \text{proof} \rangle$

lemma *clock_set_commute_single*:
 $[r \rightarrow d]u(c := d') = ([r \rightarrow d]u)(c := d') \ \text{if } c \notin \text{set } r$
 $\langle \text{proof} \rangle$

lemma *clock_set_upd_simp2*:
 $([xs @ c \# ys \rightarrow d]u)(c := d') = ([xs @ ys \rightarrow d]u)(c := d')$
 $\langle \text{proof} \rangle$

lemma *clock_set_upd_simp3*:
 $([xs \rightarrow d]u)(c := d') = ([\text{filter } (\lambda x. x \neq c) \ xs \rightarrow d]u)(c := d')$
 $\langle \text{proof} \rangle$

interpretation *urge*: *Prod_TA_sem A_urge* $\langle \text{proof} \rangle$

lemma *urge_n_ps_eq*:
 $\text{urge}.n_ps = n_ps$
 $\langle \text{proof} \rangle$

lemma *urge_N_simp[simp]*:
 $\text{urge}.N = N_urge$
 $\langle \text{proof} \rangle$

lemma *urge_states_eq*:
 $\text{urge}.states = states$

$\langle proof \rangle$

interpretation *Bisimulation_Invariant*

$\lambda(L, s, u) (L', s', u'). \text{step_}u' A L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). \text{step_}u' A_urge L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). L' = L \wedge s' = s \wedge u' = u(urge := 0)$
 $\lambda(L, s, u). L \in \text{states} \lambda(L, s, u). \text{True}$

$\langle proof \rangle$

lemmas *urge_bisim = Bisimulation_Invariant_axioms*

end

context *Simple_Network_Impl_Defs*

begin

lemma *dom_bounds: dom bounds = fst 'set bounds'*

$\langle proof \rangle$

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 $i < n_ps$

$\langle proof \rangle$

lemma *length_automata_eq_n_ps:*

$\text{length automata} = n_ps$

$\langle proof \rangle$

lemma *N_p_trans_eq:*

$\text{Simple_Network_Language.trans } (N\ p) = \text{set } (\text{fst } (\text{snd } (\text{snd } (\text{automata } !\ p)))) \text{ if } p < n_ps$

$\langle proof \rangle$

lemma *loc_set_compute:*

$\text{loc_set} = \bigcup ((\lambda(_, _, t, _). \bigcup ((\lambda(l, _, _, _, _, _, l'). \{l, l'\}) \text{ 'set } t)) \text{ 'set automata})$

$\langle proof \rangle$

lemma *var_set_compute:*

$\text{var_set} =$

$(\bigcup S \in (\lambda t. (\text{fst} \circ \text{snd}) \text{ 'set } t) \text{ 'set } ((\lambda(_, _, t, _). t) \text{ 'set automata}). \bigcup b \in S. \text{vars_of_bexp } b)$

$\bigcup$

$(\bigcup S \in (\lambda t. (\text{fst} \circ \text{snd} \circ \text{snd} \circ \text{snd} \circ \text{snd}) \text{ 'set } t) \text{ 'set } ((\lambda(_, _, t, _). t) \text{ 'set automata}).$

$\bigcup f \in S. \bigcup (x, e) \in \text{set } f. \{x\} \cup \text{vars_of_exp } e)$

$\langle proof \rangle$

lemma *states_loc_setD:*

$\text{set } L \subseteq \text{loc_set} \text{ if } L \in \text{states}$

$\langle proof \rangle$

end

context *Simple_Network_Impl*

begin

lemma *sem_bounds_eq*: *sem.bounds = bounds*
 ⟨*proof*⟩

lemma *n_ps_eq*[*simp*]:
sem.n_ps = n_ps
 ⟨*proof*⟩

lemma *sem_loc_set_eq*:
sem.loc_set = loc_set
 ⟨*proof*⟩

lemma *sem_states_eq*:
sem.states = states
 ⟨*proof*⟩

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 ≡ *map* *renum_upd*

definition
renum_reset = *map* *renum_clocks*

definition *renum_automaton* **where**

renum_automaton *i* $\equiv \lambda(\text{committed}, \text{urgent}, \text{trans}, \text{inv}). \text{let}$
 committed' = *map* (*renum_states* *i*) *committed*;
 urgent' = *map* (*renum_states* *i*) *urgent*;
 trans' = *map* ($\lambda(l, b, g, a, \text{upd}, r, l').$
 (*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* ($\lambda(l, g). (\text{renum_states } i \text{ } l, \text{renum_cconstraint } g))$ *inv*
 in (*committed'*, *urgent'*, *trans'*, *inv'*)

definition

vars_inv $\equiv \text{the_inv } \text{renum_vars}$

definition

map_st $\equiv \lambda(L, s). (\text{map_index } \text{renum_states } L, s \text{ o } \text{vars_inv})$

definition

clocks_inv $\equiv \text{the_inv } \text{renum_clocks}$

definition

map_u *u* = *u* o *clocks_inv*

lemma *map_u_add[simp]*:

map_u (*u* $\oplus$ *d*) = *map_u* *u* $\oplus$ *d*
 ⟨*proof*⟩

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 ($\lambda(a, p). (\text{renum_vars } a, p))$ *bounds'*
 ⟨*proof*⟩

lemma *renum_n_ps_simp[simp]*:

renum.n_ps = *n_ps*
 ⟨*proof*⟩

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*
 $\times$ ((*'s* :: *countable*) $\times$ (*'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'
⟨proof⟩

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⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

lemma dom_map_of_map:
  dom (map_of (map (λ (a, b). (f a, g b)) xs)) = f ' fst ' set xs
  ⟨proof⟩

lemma inj_image_eqI:
  S = T if inj f f ' S = f ' T
  ⟨proof⟩

lemmas [finite_intros] = finite_set

lemma map_of_NoneI:
  map_of xs x = None if x ∉ fst ' set xs
  ⟨proof⟩

lemma bij_f_the_inv_f:
  f (the_inv f x) = x if bij f
  ⟨proof⟩

fun map_sexp ::
  (nat ⇒ 's ⇒ 's1) ⇒ ('a ⇒ 'a1) ⇒ ('b ⇒ 'b1) ⇒ (nat, 's, 'a, 'b) sexp
  ⇒ (nat, 's1, 'a1, 'b1) 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 ⇒ 's ⇒ 's1) ⇒ ('a ⇒ 'a1) ⇒ ('b ⇒ 'b1)
⇒ (nat, 's, 'a, 'b) formula ⇒ (nat, 's1, 'a1, 'b1) formula

```

where

```

map_formula f g h (formula.EX φ) = formula.EX (map_sexp f g h φ) |
map_formula f g h (EG φ) = EG (map_sexp f g h φ) |
map_formula f g h (AX φ) = AX (map_sexp f g h φ) |
map_formula f g h (AG φ) = AG (map_sexp f g h φ) |
map_formula f g h (Leadsto φ ψ) = Leadsto (map_sexp f g h φ) (map_sexp f g h ψ)

```

locale Simple_Network_Impl_real =

fixes automata ::

```

('s list × 's list × ('a act, 's, 'c, real, 'x, int) transition list
× ('s × ('c, real) cconstraint) list) list

```

and broadcast :: 'a list

and bounds' :: ('x × (int × int)) list

begin

sublocale Simple_Network_Impl_Defs automata broadcast bounds' ⟨proof⟩

abbreviation sem ≡ (set broadcast, map automaton_of automata, map_of bounds')

sublocale Prod_TA_sem (set broadcast, map automaton_of automata, map_of bounds') ⟨proof⟩

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
⟨proof⟩

```

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
⟨proof⟩

```

lemma sem_var_set_eq:

```

sem.var_set = var_set

```

```

    <proof>

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'
  <proof>

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: ∀ i < n_ps. urgent (N i) = {}
begin

lemma aux_1:
  (λx. case case x of
    (l, g) ⇒ (renum_states p l, renum_cconstraint g) of
    (s, cc) ⇒ (s, map conv_ac cc))
  = (λ (l, g). (renum_states p l, map conv_ac (renum_cconstraint g)))

  <proof>

lemma clocks_inv_inv:
  clocks_inv (renum_clocks a) = a
  <proof>

```

lemma *map_u_renum_constraint_clock_valI*:
 $\text{map_u } u \vdash \text{renum_constraint } cc \text{ if } u \vdash cc$
 $\langle \text{proof} \rangle$

lemma *map_u_renum_constraint_clock_valD*:
 $u \vdash cc \text{ if } \text{map_u } u \vdash \text{renum_constraint } cc$
 $\langle \text{proof} \rangle$

lemma *inj_renum_states*: $\text{inj } (\text{renum_states } p) \text{ if } p < n_ps$
 $\langle \text{proof} \rangle$

lemma *inv_renum_sem_I*:
assumes
 $u \vdash \text{Simple_Network_Language.inv } (N \ p) \ l \ p < n_ps \ l \in \text{loc_set}$
shows
 $\text{map_u } u \vdash \text{Simple_Network_Language.inv } (\text{renum}.N \ p) (\text{renum_states } p \ l)$
 $\langle \text{proof} \rangle$

lemma *inv_renum_sem_D*:
assumes
 $\text{map_u } u \vdash \text{Simple_Network_Language.inv } (\text{renum}.N \ p) (\text{renum_states } p \ l) \ p < n_ps \ l \in \text{loc_set}$
shows
 $u \vdash \text{Simple_Network_Language.inv } (N \ p) \ l$
 $\langle \text{proof} \rangle$

lemma *dom_the_inv_comp*:
 $\text{dom } (m \circ \text{the_inv } f) = f' \text{ dom } m \text{ if } \text{inj } f \text{ range } f = \text{UNIV}$
 $\langle \text{proof} \rangle$

lemma *inj_renum_vars*:
 inj renum_vars
 $\langle \text{proof} \rangle$

lemma *surj_renum_vars*:
 surj renum_vars
 $\langle \text{proof} \rangle$

lemma *map_of_inv_map*:
 $\text{map_of } xs \ (\text{the_inv } f \ x) = \text{map_of } (\text{map } (\lambda (a, b). (f \ a, b)) \ xs) \ x$
if $\text{inj } f \text{ surj } f \text{ the_inv } f \ x \in \text{dom } (\text{map_of } xs)$
 $\langle \text{proof} \rangle$

lemma *dom_vars_invD*:
assumes $x \in \text{dom } (s \circ \text{vars_inv})$
shows $x \in \text{renum_vars} \text{ ' dom } s \text{ (is ?A) and the_inv renum_vars } x \in \text{dom } s \text{ (is ?B)}$
 $\langle \text{proof} \rangle$

lemma *bounded_renumI*:
assumes $\text{bounded } (\text{map_of } \text{bounds'}) \ s$
shows $\text{bounded } (\text{map_of } (\text{map } (\lambda(a, y). (\text{renum_vars } a, y)) \ \text{bounds'})) \ (s \circ \text{vars_inv})$
 $\langle \text{proof} \rangle$

lemma *map_of_renum_vars_simp*:

assumes
 $dom (s \circ vars_inv)$
 $= dom (map_of (map (\lambda(a, y). (renum_vars a, y)) bounds'))$
 $x \in dom s \implies dom s \subseteq var_set$
shows $map_of (map (\lambda(a, y). (renum_vars a, y)) bounds') (renum_vars x) = map_of bounds'$
 x
 $\langle proof \rangle$

lemma *bounded_renumD*:
assumes
 $Simple_Network_Language.bounded$
 $(map_of (map (\lambda(a, y). (renum_vars a, y)) bounds')) (s \circ vars_inv)$
and $dom s \subseteq var_set$
shows $Simple_Network_Language.bounded (map_of bounds') s$
 $\langle proof \rangle$

lemma *dom_bounds_var_set*: $dom bounds = var_set$
 $\langle proof \rangle$

lemma *sem_states_loc_setD*: $L \vdash p \in loc_set$ **if** $p < length\ automata$ $L \in states$ **for** $L \vdash p$
 $\langle proof \rangle$

lemma *trans_N_renumD*:
assumes $(l, b, g, a, f, r, l') \in Simple_Network_Language.trans (N \ p) \ p < n_ps$
shows $(renum_states \ p \ l, renum_bexp \ b, renum_cconstraint \ g, renum_act \ a, renum_upds \ f,$
 $renum_reset \ r, renum_states \ p \ l')$
 $\in Simple_Network_Language.trans (renum.N \ p)$
 $\langle proof \rangle$

lemma *match_assumption2*:
assumes $P \ a1 \ b1 \ a1 = a \ b1 = b$ **shows** $P \ a \ b$
 $\langle proof \rangle$

lemma *inj_pair*:
assumes $inj \ f \ inj \ g$
shows $inj (\lambda(a, b). (f \ a, g \ b))$
 $\langle proof \rangle$

lemma *injective_functions*:
 $inj \ renum_reset \ inj \ renum_upd \ inj \ renum_act \ inj \ renum_cconstraint \ inj \ renum_bexp$
 $\wedge p. \ p < length\ automata \implies inj (renum_states \ p)$
 $\langle proof \rangle$

lemma *trans_N_renumI*:
assumes (
 $renum_states \ p \ l, renum_bexp \ b, renum_cconstraint \ g, renum_act \ a, renum_upds \ f, renum_reset$
 $r,$
 $renum_states \ p \ l')$
 $\in trans (renum.N \ p) \ p < n_ps$
shows $(l, b, g, a, f, r, l') \in trans (N \ p)$
 $\langle proof \rangle$

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
⟨proof⟩

lemma renum_cconstraint_eq_convD:
  assumes renum_cconstraint g = conv_cc g'
  obtains g1 where g = conv_cc g1 g' = renum_cconstraint g1
⟨proof⟩

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,
    renum_states p l')
  ∈ trans (renum.N p) p < n_ps
  shows (l, b, g, a, f, r, l') ∈ trans (N p)
  ⟨proof⟩

lemma trans_sem_N_renumI':
  assumes (renum_states p l, b, g, a, f, r, l') ∈ trans (renum.N p) p < n_ps
  shows ∃ b' g' a' f' r' l1.
    TRANS "orig" ∥ (l, b', g', a', f', r', l1) ∈ Simple_Network_Language.trans (N p)
    ∧ "renum bexp" ∥ b = renum_bexp b' ∧ "renum guard" ∥ g = renum_cconstraint g'
    ∧ "renum action" ∥ a = renum_act a' ∧ "renum upds" ∥ f = renum_upds f'
    ∧ "renum reset" ∥ r = renum_reset r' ∧ "renum loc" ∥ l' = renum_states p l1
  ⟨proof⟩

lemma committed_renum_eq:
  committed (renum.N p) = renum_states p ' committed (N p) if p < n_ps
  ⟨proof⟩

lemma urgent_renum_eq:
  urgent (renum.N p) = renum_states p ' urgent (N p) if p < n_ps
  ⟨proof⟩

lemma renum_urgency_removed:
  ∀ i < n_ps. urgent (renum.N i) = {}
  ⟨proof⟩

lemma check_bexp_renumD:
  check_bexp s b bv ⇒ check_bexp (s o vars_inv) (renum_bexp b) bv
and is_val_renumD:
  is_val s e v ⇒ is_val (s o vars_inv) (renum_exp e) v
  ⟨proof⟩

method solve_case =
  auto
  intro: check_bexp_is_val.intros
  simp: renum_bexp_def renum_exp_def vars_inv_def the_inv_f_f[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 ⇒ 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*
 ⟨*proof*⟩

lemma *renum_reset_map_u*:
 [*renum_reset* *r* $\rightarrow$ 0] *map_u* *u* = *map_u* ([*r* $\rightarrow$ 0] *u*)
 ⟨*proof*⟩

lemma *bounded_renumI'*:
 assumes *bounded* *bounds* *s'*
 shows *bounded* *renum.bounds* (*s'* *o vars_inv*)
 ⟨*proof*⟩

lemma *bounded_renumD'*:
 assumes *bounded* *renum.bounds* (*s'* *o vars_inv*) *dom s' $\subseteq$ var_set*
 shows *bounded* *bounds* *s'*
 ⟨*proof*⟩

lemma *is_upd_renumD*:
 assumes *is_upd* *s* *f* *s'*
 shows *is_upd* (*s* *o vars_inv*) (*renum_upd* *f*) (*s'* *o vars_inv*)
 ⟨*proof*⟩

lemma *is_upds_renumD*:
 assumes *is_upds* *s1* *ps* *s'*
 shows *is_upds* (*s1* *o vars_inv*) (*renum_upds* *ps*) (*s'* *o vars_inv*)
 ⟨*proof*⟩

lemma *is_upds_concat_renumD*:
 assumes *is_upds* *s1* (*concat* *ps*) *s'*
 shows *is_upds* (*s1* *o vars_inv*) (*concat_map* *renum_upds* *ps*) (*s'* *o vars_inv*)
 ⟨*proof*⟩

lemma *Ball_mono*:
 assumes $\forall x \in S. P\ x \bigwedge x. x \in S \implies P\ x \implies Q\ x$
 shows $\forall x \in S. Q\ x$
 ⟨*proof*⟩

lemma *atLeastLessThan_upperD*:
 assumes *S* $\subseteq$ {*l*..*u*} *x* $\in$ *S*
 shows *x* < *u*
 ⟨*proof*⟩

lemma *imp_mono_rule*:
 assumes *P1* $\longrightarrow$ *P2*
 and *Q1* $\implies$ *P1*
 and *Q1* $\implies$ *P2* $\implies$ *Q2*
 shows *Q1* $\longrightarrow$ *Q2*
 ⟨*proof*⟩

lemma *inj_id*:
inj *id*
 ⟨*proof*⟩

lemma *step_single_renumD*:

assumes $\text{step_u sem } L \text{ s u a } L' \text{ s' u' } L \in \text{states dom s} \subseteq \text{var_set}$
shows step_u renum.sem
 $(\text{map_index renum_states } L) (s \text{ o vars_inv}) (\text{map_u } u)$
 $(\text{renum_label } a)$
 $(\text{map_index renum_states } L') (s' \text{ o vars_inv}) (\text{map_u } u')$
 $\langle \text{proof} \rangle$

lemma inj_the_inv :
 $\text{inj } (\text{the_inv } f) \text{ if } \text{bij } f$
 $\langle \text{proof} \rangle$

lemma inj_vars_inv :
 inj vars_inv
 $\langle \text{proof} \rangle$

lemma $\text{comp_vars_inv_upd_commute}$:
 $(s \text{ o vars_inv})(x \mapsto y) = s(\text{vars_inv } x \mapsto y) \text{ o vars_inv}$
 $\langle \text{proof} \rangle$

lemma is_upd_renumI :
assumes $\text{is_upd } (s \text{ o vars_inv}) (\text{renum_upd } f) \text{ s'}$
shows $\text{is_upd } s \text{ f } (s' \text{ o renum_vars})$
 $\langle \text{proof} \rangle$

lemma is_upd_renumI' :
assumes $\text{is_upd } (s \text{ o vars_inv}) (\text{renum_upd } f) \text{ s'}$
obtains $s1$ **where** $\text{is_upd } s \text{ f } s1 \text{ s1} = s' \text{ o renum_vars}$
 $\langle \text{proof} \rangle$

lemma is_upd_renumI'' :
assumes $\text{is_upd } s (\text{renum_upd } f) \text{ s'}$
shows $\text{is_upd } (s \text{ o renum_vars}) f (s' \text{ o renum_vars})$
 $\langle \text{proof} \rangle$

lemma is_upds_renumI :
assumes $\text{is_upds } (s \text{ o vars_inv}) (\text{renum_upds upds}) \text{ s'}$
shows $\exists s1. \text{is_upds } s \text{ upds } s1 \wedge s1 = s' \text{ o renum_vars}$
 $\langle \text{proof} \rangle$

lemma is_upds_renumI' :
assumes $\text{is_upds } (s \text{ o vars_inv}) (\text{renum_upds } f) \text{ s'}$
shows $\text{is_upds } s \text{ f } (s' \text{ o renum_vars})$
 $\langle \text{proof} \rangle$

lemma is_upds_renumI'' :
assumes $\text{is_upds } s (\text{renum_upds ps}) \text{ s'}$
shows $\text{is_upds } (s \text{ o renum_vars}) ps (s' \text{ o renum_vars})$
 $\langle \text{proof} \rangle$

lemma $\text{is_upds_concat_renumI''}$:
assumes $\text{is_upds } s (\text{concat_map renum_upds ps}) \text{ s'}$
shows $\text{is_upds } (s \text{ o renum_vars}) (\text{concat ps}) (s' \text{ o renum_vars})$
 $\langle \text{proof} \rangle$

lemma *bounded_renumD1*:
assumes *bounded_renum.bounds s' dom (s' o renum_vars) ⊆ var_set*
shows *bounded_bounds (s' o renum_vars)*
 $\langle \text{proof} \rangle$

lemma *renum_reset_append*:
 $\text{renum_reset } xs \text{ @ renum_reset } ys = \text{renum_reset } (xs \text{ @ } ys)$
 $\langle \text{proof} \rangle$

lemma *if_eq_distrib*:
 $(\text{if } i = j \text{ then } f \ i \ a \ \text{else } f \ j \ b) = (f \ j \ (\text{if } i = j \text{ then } a \ \text{else } b))$
 $\langle \text{proof} \rangle$

lemma *dom_comp_eq_vimage*:
 $\text{dom } (s \circ f) = f^{-1} \text{ dom } s$
 $\langle \text{proof} \rangle$

lemma *dom_comp_vars_inv_eqD*:
assumes $\text{dom } s' = \text{dom } (s \circ \text{vars_inv})$
shows $\text{dom } (s' \circ \text{renum_vars}) = \text{dom } s$
 $\langle \text{proof} \rangle$

lemma *sem_trans_upd_domD*:
assumes $(L \ ! \ p, b, g', a, f, r, l1) \in \text{trans } (N \ p) \ p < n_ps$
shows $\text{fst } ' \text{ set } f \subseteq \text{var_set}$
 $\langle \text{proof} \rangle$

lemma *SilD*:
fixes *map_action*
assumes $\text{Sil } a = \text{map_act } \text{map_action } a1$
obtains *a'* **where** $a1 = \text{Sil } a' \ a = \text{map_action } a'$
 $\langle \text{proof} \rangle$

lemma *InD*:
fixes *map_action*
assumes $\text{In } a = \text{map_act } \text{map_action } a1$
obtains *a'* **where** $a1 = \text{In } a' \ a = \text{map_action } a'$
 $\langle \text{proof} \rangle$

lemma *OutD*:
fixes *map_action*
assumes $\text{Out } a = \text{map_act } \text{map_action } a1$
obtains *a'* **where** $a1 = \text{Out } a' \ a = \text{map_action } a'$
 $\langle \text{proof} \rangle$

lemma *In_OutD*:
assumes $\text{In } a = \text{renum_act } a1 \ \text{Out } a = \text{renum_act } a2$
obtains *a'* **where** $a = \text{renum_acts } a' \ a1 = \text{In } a' \ a2 = \text{Out } a'$
 $\langle \text{proof} \rangle$

lemma *renum_sem_broadcast_eq*:
 $\text{renum.broadcast} = \text{renum_acts } ' \text{ set broadcast}$
 $\langle \text{proof} \rangle$

lemma *inj_eq_iff*:
 $f\ x = f\ y \longleftrightarrow x = y$ **if** *inj f*
 <proof>

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>

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
 $\forall pa < n_ps.$
 $[\text{renum_reset } r\ @\ \text{concat } (\text{map } rs\ ps) \rightarrow 0] \text{map_u } u$
 $\vdash \text{inv } (\text{renum}.N\ pa)$
 $((\text{fold } (\lambda p\ L. L[p := ls\ p])\ ps\ (\text{map_index } \text{renum_states } L))\ [p := \text{renum_states } p\ l1] !\ pa)$
 $\forall p \in \text{set } ps. rs\ p = \text{renum_reset } (rs'\ p)$
 $\forall p \in \text{set } ps. ls\ p = \text{renum_states } p\ (ls'\ p)$
 $\forall p \in \text{set } ps. ls'\ p \in (\bigcup (l,\ b,\ g,\ a,\ r,\ u,\ l') \in \text{trans } (N\ p). \{l,\ l'\})$
 $l1 \in (\bigcup (l,\ b,\ g,\ a,\ r,\ u,\ l') \in \text{trans } (N\ p). \{l,\ l'\})$
 $L \in \text{states}$
shows
 $\forall pa < n_ps.$
 $[r\ @\ \text{concat } (\text{map } rs'\ ps) \rightarrow 0] u \vdash \text{inv } (N\ pa)\ ((\text{fold } (\lambda p\ L. L[p := ls'\ p])\ ps\ L)\ [p := l1] !\ pa)$
 <proof>

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_cconstraint_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 $(L \mid p, b', g', a, f', r', l1) \in \text{trans } (N \ p) \ \text{dom } s = \text{var_set } p < n_ps$
shows $\forall (x, _) \in \text{set } (\text{renum_upds } f'). \ x \in \text{dom } (s \ o \ \text{vars_inv})$
 <proof>

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'
 $L \in \text{states} \ \text{dom } s \subseteq \text{var_set} \ \text{dom } s = \text{var_set}$
shows $\exists a1 \ L1 \ s1 \ u1. \ \text{step_u sem } L \ s \ u \ a1 \ L1 \ s1 \ u1 \wedge \text{renum_label } a1 = a \wedge$
 $L' = \text{map_index renum_states } L1 \wedge s' = s1 \ o \ \text{vars_inv} \wedge u' = \text{map_u } u1$
 <proof>

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
 <proof>

lemmas step_u_invariants = states_inv step_u_var_set_invariant

interpretation single: Bisimulation_Invariant

$\lambda(L, s, u) \ (L', s', u'). \ \exists a. \ \text{step_u sem } L \ s \ u \ a \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ \exists a. \ \text{step_u renum.sem } L \ s \ u \ a \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ L' = \text{map_index renum_states } L \wedge s' = s \ o \ \text{vars_inv} \wedge u' = \text{map_u } u$
 $\lambda \ (L, s, u). \ L \in \text{states} \wedge \text{dom } s = \text{var_set} \ \lambda_. \ \text{True}$
 <proof>

interpretation Bisimulation_Invariant

$\lambda(L, s, u) \ (L', s', u'). \ \text{step_u' sem } L \ s \ u \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ \text{step_u' renum.sem } L \ s \ u \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ L' = \text{map_index renum_states } L \wedge s' = s \ o \ \text{vars_inv} \wedge u' = \text{map_u } u$
 $\lambda \ (L, s, u). \ L \in \text{states} \wedge \text{dom } s = \text{var_set} \ \lambda_. \ \text{True}$
 <proof>

interpretation Bisimulation_Invariant

$\lambda(L, s, u) \ (L', s', u'). \ \text{step_u' sem } L \ s \ u \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ \text{step_u' renum.sem } L \ s \ u \ L' \ s' \ u'$
 $\lambda(L, s, u) \ (L', s', u'). \ L' = \text{map_index renum_states } L \wedge s' = s \ o \ \text{vars_inv} \wedge u' = \text{map_u } u$
 $\lambda \ (L, s, u). \ L \in \text{states} \wedge \text{dom } s = \text{var_set} \ \lambda_. \ \text{True}$
 <proof>

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

$\times ((\text{'a} :: \text{countable}) \ \text{act}, \text{'s}, \text{'c}, \text{real}, \text{'x} :: \text{countable}, \text{int}) \ \text{transition list}$

$\times ((s :: \text{countable}) \times (c :: \text{countable}, \text{real}) \text{ cconstraint}) \text{ list}) \text{ list} +$
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 Φ' **where**
 $\Phi' = \text{map_formula } \text{renum_states } \text{renum_vars } \text{id } \Phi$

definition a_0 **where**
 $a_0 = (L_0, \text{map_of } s_0, \lambda_. 0)$

definition a_0' **where**
 $a_0' = (\text{map_index } \text{renum_states } L_0, \text{map_of } (\text{map } (\lambda(x, v). (\text{renum_vars } x, v)) s_0), \lambda_. 0)$

context
assumes $L_0_states: L_0 \in \text{states}$
assumes $s_0_dom: \text{fst } ' \text{ set } s_0 = \text{var_set}$ **and** $s_0_distinct: \text{distinct } (\text{map } \text{fst } s_0)$
begin

lemma state_eq_aux :
assumes $x \notin \text{renum_vars } ' \text{ var_set}$
shows $\text{vars_inv } x \notin \text{var_set}$
 $\langle \text{proof} \rangle$

lemma state_eq :
assumes $\text{fst } ' \text{ set } s_0 = \text{var_set } \text{distinct } (\text{map } \text{fst } s_0)$
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 $?l = ?r)$
 $\langle \text{proof} \rangle$

interpretation $\text{Bisimulation_Invariant}$
 $\lambda(L, s, u) (L', s', u'). \text{step_u' sem } L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). \text{step_u' renum.sem } L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). L' = \text{map_index } \text{renum_states } L \wedge s' = s \circ \text{vars_inv} \wedge u' = \text{map_u } u$
 $\lambda(L, s, u). L \in \text{states} \wedge \text{dom } s = \text{var_set } \lambda_. \text{True}$
 $\langle \text{proof} \rangle$

lemma start_equiv :
 $A_B.\text{equiv}' a_0 a_0'$
 $\langle \text{proof} \rangle$

lemma check_sexp_equiv :
assumes $A_B.\text{equiv}' (L, s, u) (L', s', u') \text{ locs_of_sexp } e \subseteq \{0..<n_ps\}$
shows
 $\text{check_sexp } e L (\text{the } \circ s) \longleftrightarrow$
 $\text{check_sexp } (\text{map_sexp } \text{renum_states } \text{renum_vars } \text{id } e) L' (\text{the } \circ s')$
 $\langle \text{proof} \rangle$

lemma models_iff :
 $\text{sem}, a_0 \models \Phi = \text{renum.sem}, a_0' \models \Phi'$ **if** $\text{locs_of_formula } \Phi \subseteq \{0..<n_ps\}$
 $\langle \text{proof} \rangle$

```

lemma has_deadlock_iff:
  has_deadlock sem a0  $\longleftrightarrow$  has_deadlock renum.sem a0'
   $\langle$ proof $\rangle$ 

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
   $\langle$ proof $\rangle$ 

lemma Bisimulation_Invariants_Bisimulation_Invariant_iff:
  Bisimulation_Invariants A B sim PA PA PB PB  $\longleftrightarrow$  Bisimulation_Invariant A B sim PA PB
   $\langle$ proof $\rangle$ 

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  $\bigwedge a b. P a \implies A a b \implies P b$ 
   $\langle$ proof $\rangle$ 

locale Simple_Network_Rename_int =
  Simple_Network_Rename_Defs_int where automata = automata +
  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 +
  assumes urgency_removed:  $\forall$  (_, U, _, _)  $\in$  set 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
   $\langle$ proof $\rangle$ 

lemma var_set_eq1:
  Prod_TA_Defs.var_set
  (set broadcast, map automaton_of (map conv_automaton automata),
    map_of bounds') = var_set
   $\langle$ proof $\rangle$ 

lemma urgency_removed':
   $\forall i < n\_ps. \text{urgent}$ 
  (Prod_TA_Defs.N (set broadcast, map automaton_of (map conv_automaton automata), map_of

```

$\text{bounds}' i)$
 $= \{\}$
 $\langle \text{proof} \rangle$

sublocale *real*: *Simple_Network_Rename_real* **where** *automata* = *map conv_automaton automata*
 $\langle \text{proof} \rangle$

lemma *sem_unfold1*:
 $\text{real.sem} = \text{sem}$
 $\langle \text{proof} \rangle$

lemma *var_set_eq*:
 $\text{real.var_set} = \text{sem.var_set}$
 $\langle \text{proof} \rangle$

lemma *map_acconstraint_conv_ac_commute*:
 $\text{map_acconstraint renum_clocks id (conv_ac ac)} = \text{conv_ac (map_acconstraint renum_clocks id ac)}$
 $\langle \text{proof} \rangle$

lemma *map_ccconstraint_conv_cc_commute*:
 $\text{renum_cconstraint (conv_cc g)} = \text{conv_cc (renum_cconstraint g)}$
 $\langle \text{proof} \rangle$

lemma *rename_conv_automaton_commute*:
 $\text{real.renum_automaton } n \text{ (conv_automaton } x) = \text{conv_automaton (real.renum_automaton } n \text{ } x)$
 $\langle \text{proof} \rangle$

lemma *sem_unfold2*:
 $\text{real.renum.sem} = \text{renum.sem}$
 $\langle \text{proof} \rangle$

sublocale *renum_bisim*: *Bisimulation_Invariant*
 $\lambda(L, s, u) (L', s', u'). \text{step_u' sem } L \text{ } s \text{ } u \text{ } L' \text{ } s' \text{ } u'$
 $\lambda(L, s, u) (L', s', u'). \text{step_u' renum.sem } L \text{ } s \text{ } u \text{ } L' \text{ } s' \text{ } u'$
 $\lambda(L, s, u) (L', s', u'). L' = \text{map_index renum_states } L \wedge s' = s \text{ } o \text{ } \text{vars_inv} \wedge u' = \text{map_u } u$
 $\lambda(L, s, u). L \in \text{sem.states} \wedge \text{dom } s = \text{var_set } \lambda_.$ *True*
 $\langle \text{proof} \rangle$

lemmas *renum_bisim* = *renum_bisim.Bisimulation_Invariant_axioms*

end

locale *Simple_Network_Rename_Start'* =
Simple_Network_Rename' **where** *automata* = *automata*
for *automata* ::
 $('s \text{ list} \times 's \text{ list}$
 $\times (('a :: \text{countable}) \text{ act, 's, 'c, 't, 'x} :: \text{countable, int}) \text{ transition list}$
 $\times (('s :: \text{countable}) \times ('c :: \text{countable, 't}) \text{ cconstraint) list) list} +$
fixes *s₀* :: $('x \times \text{int}) \text{ list}$
and *L₀* :: $'s \text{ list}$
assumes *L₀_states*: *L₀* ∈ *states*
assumes *s₀_dom*: *fst* ' *set* *s₀* = *var_set* **and** *s₀_distinct*: *distinct (map fst s₀)*

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 list × 's list
 × (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list
 × (('s :: countable) × ('c :: countable, int) cconstraint) list) list

begin

definition *a₀* **where**

a₀ = (*L₀*, *map_of s₀*, *λ_. 0*)

definition *a₀'* **where**

a₀' = (*map_index renum_states L₀*, *map_of (map (λ(x, v). (renum_vars x, v)) s₀)*, *λ_. 0*)

lemma *state_eq_aux*:

assumes *x* ∉ *renum_vars* ' *var_set*

shows *vars_inv x* ∉ *var_set*

⟨*proof*⟩

lemma *state_eq*:

assumes *fst* ' *set s₀* = *var_set distinct (map fst s₀)*

shows *map_of (map (λ(x, y). (renum_vars x, y)) s₀)* = (*map_of s₀ ∘ ∘ the_inv_into UNIV*
renum_vars

(**is** ?*l* = ?*r*)

⟨*proof*⟩

lemma *start_equiv*:

renum_bisim.A_B.equiv' a₀ a₀'

⟨*proof*⟩

lemma *check_sexp_equiv*:

assumes *renum_bisim.A_B.equiv' (L, s, u) (L', s', u')* *locs_of_sexp e* ⊆ {0..*n_ps*}

shows

check_sexp e L (the ∘ s) ⟷

check_sexp (map_sexp renum_states renum_vars id e) L' (the ∘ s')

⟨*proof*⟩

lemma *check_sexp_compatible*:

assumes *locs_of_sexp e* ⊆ {0..*n_ps*}

shows *renum_bisim.compatible*

(*λ(L, s, u). check_sexp e L (the ∘ s)*)

(*λ(L', s', u'). check_sexp (map_sexp renum_states renum_vars id e) L' (the ∘ s')*)

⟨*proof*⟩

lemma *has_deadlock_iff*:

has_deadlock sem a₀ ⟷ *has_deadlock renum.sem a₀'*

⟨*proof*⟩

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*
 $(\text{map_state_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \varphi)$
 $(\text{map_state_formula } (\lambda P (L, s, _).$
 $\text{check_sexp } (\text{map_sexp } (\lambda p. \text{renum_states } p) \text{renum_vars id } P) L (\text{the } o s))$
 $\varphi)$ **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*
 $(\text{map_path_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \psi)$
 $(\text{map_path_formula } (\lambda P (L, s, _).$
 $\text{check_sexp } (\text{map_sexp } (\lambda p. \text{renum_states } p) \text{renum_vars id } P) L (\text{the } o s))$
 $\psi)$
 <proof>

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
 $\times ((\text{'a} :: \text{countable}) \text{act}, \text{'s}, \text{'c}, \text{int}, \text{'x} :: \text{countable}, \text{int}) \text{transition list}$
 $\times ((\text{'s} :: \text{countable}) \times (\text{'c} :: \text{countable}, \text{int}) \text{cconstraint}) \text{list}) \text{list} +$
 fixes $\Phi :: (\text{nat}, \text{'s}, \text{'x}, \text{int}) \text{formula}$

begin

definition Φ' **where**

$\Phi' = \text{map_formula } \text{renum_states } \text{renum_vars id } \Phi$

lemma *models_iff*:

$\text{sem}, a_0 \models \Phi = \text{renum.sem}, a_0' \models \Phi' \text{ if } \text{locs_of_formula } \Phi \subseteq \{0..<n_ps\}$
 <proof>

end

lemma *vars_of_bexp_finite*[*finite_intros*]:

finite (*vars_of_bexp* ($b :: (\text{'a}, \text{'b}) \text{bexp}$))

and *vars_of_exp_finite*[*finite_intros*]:

finite (*vars_of_exp* ($e :: (\text{'a}, \text{'b}) \text{exp}$))

<proof>

lemma *set_bexp_vars_of_bexp*:

set_bexp ($b :: (\text{'a}, \text{'b}) \text{bexp}$) = *vars_of_bexp* *b*

and *set_exp_vars_of_exp*:

set_exp ($e :: (\text{'a}, \text{'b}) \text{exp}$) = *vars_of_exp* *e*

<proof>

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*:
act_set =
 $\bigcup ((\lambda(_, _, t, _). \bigcup ((\lambda(l, e, g, a, _). \text{act.set_act } a) \text{ 'set } t)) \text{ 'set automata}) \cup \text{set broadcast}$
 <proof>

lemma *set1_acconstraint_elim*:
assumes $c \in \text{set1_acconstraint } ac$
obtains x **where** $(c, x) = \text{constraint_pair } ac$
 <proof>

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 $\text{clk_set'I1[intro]: } c \in \text{set } r \implies c \in \text{clk_set'}$
and $\text{clk_set'I2[intro]: } ac \in \text{set } g \implies c \in \text{set1_acconstraint } ac \implies c \in \text{clk_set'}$
and $\text{loc_setI1[intro]: } l \in \text{loc_set and loc_setI2[intro]: } l' \in \text{loc_set}$
and $\text{act_setI[intro]: } a' \in \text{set_act } a \implies a' \in \text{act_set}$
and $\text{var_setI1[intro]: } v \in \text{set_bexp } b \implies v \in \text{var_set}$
and $\text{var_setI2[intro]: } (x, e) \in \text{set } f \implies x \in \text{var_set}$
and $\text{var_setI3[intro]: } (x, e) \in \text{set } f \implies v \in \text{set_exp } e \implies v \in \text{var_set}$
 <proof>

lemma (in *Simple_Network_Impl*) *clk_set'I3[intro]*:
assumes $(x1, x2, T, I) \in \text{set automata}$
shows $(l, g') \in \text{set } I \implies ac \in \text{set } g' \implies c \in \text{set1_acconstraint } ac \implies c \in \text{clk_set'}$
 <proof>

definition

find_remove $P = \text{map_option } (\lambda(xs, x, ys). (x, xs @ ys)) \circ \text{List.extract } P$

fun *merge_pairs* :: $('a \times 'b \text{ list}) \text{ list} \Rightarrow ('a \times 'b \text{ list}) \text{ list} \Rightarrow ('a \times 'b \text{ list}) \text{ list}$ **where**
merge_pairs [] $ys = ys$ |
merge_pairs $((k, v) \# xs) \text{ } ys = (\text{case find_remove } (\lambda(k', _). k' = k) \text{ } ys \text{ of}$
 None $\Rightarrow (k, v) \# \text{merge_pairs } xs \text{ } ys$
 Some $((_, v'), ys) \Rightarrow (k, v @ v') \# \text{merge_pairs } xs \text{ } ys$
)

definition

conv_urge $urges \equiv \lambda(\text{committed}, \text{urgent}, \text{trans}, \text{inv}).$
 (committed,
 [],
 map $(\lambda(l, b, g, a, f, r, l'). (l, b, g, a, f, \text{urges} \# r, l')) \text{ } \text{trans},$
 merge_pairs (map $(\lambda l. (l, [\text{acconstraint.LE } \text{urges } 0])) \text{ } \text{urgent}) \text{ } \text{inv}$)

lemma *map_of_distinct_upd2*:
assumes $x \notin \text{set } (\text{map fst } xs)$
shows $\text{map_of } (xs @ (x, y) \# ys) = (\text{map_of } (xs @ ys))(x \mapsto y)$
 <proof>

lemma *map_of_distinct_lookup*:

assumes $x \notin \text{set } (\text{map } \text{fst } xs)$
shows $\text{map_of } (xs @ (x,y) \# ys) \ x = \text{Some } y$
 $\langle \text{proof} \rangle$

lemma *find_remove_SomeD*:

assumes $\text{find_remove } P \ xs = \text{Some } (x, ys)$
shows $\exists as \ bs. \ xs = as @ x \# bs \wedge ys = as @ bs \wedge (\forall x \in \text{set } as. \neg P \ x) \wedge P \ x$
 $\langle \text{proof} \rangle$

lemma *find_map*:

$\text{find } Q \ (\text{map } f \ xs) = \text{map_option } f \ (\text{find } P \ xs) \text{ if } \forall x \in \text{set } xs. \ Q \ (f \ x) \longleftrightarrow P \ x$
 $\langle \text{proof} \rangle$

lemma *find_remove_map*:

$\text{find_remove } Q \ (\text{map } f \ xs) = \text{map_option } (\lambda(x, xs). \ (f \ x, \text{map } f \ xs)) \ (\text{find_remove } P \ xs)$
if $\forall x \in \text{set } xs. \ Q \ (f \ x) \longleftrightarrow P \ x$
 $\langle \text{proof} \rangle$

lemma *map_merge_pairs2*:

$\text{map } (\lambda(k, v). \ (g \ k, \text{map } f \ v)) \ (\text{merge_pairs } xs \ ys)$
 $= \text{merge_pairs } (\text{map } (\lambda(k, v). \ (g \ k, \text{map } f \ v)) \ xs) \ (\text{map } (\lambda(k, v). \ (g \ k, \text{map } f \ v)) \ ys)$
if *inj*: $\text{inj_on } g \ (\text{fst } ` \ (\text{set } xs \cup \text{set } ys))$
 $\langle \text{proof} \rangle$

lemma *map_merge_pairs*:

$\text{map } (\lambda(k, v). \ (k, \text{map } f \ v)) \ (\text{merge_pairs } xs \ ys)$
 $= \text{merge_pairs } (\text{map } (\lambda(k, v). \ (k, \text{map } f \ v)) \ xs) \ (\text{map } (\lambda(k, v). \ (k, \text{map } f \ v)) \ ys)$
 $\langle \text{proof} \rangle$

lemma *map_map_commute*:

$\text{map } f \ (\text{map } g \ xs) = \text{map } g \ (\text{map } f \ xs) \text{ if } \forall x \in \text{set } xs. \ f \ (g \ x) = g \ (f \ x)$
 $\langle \text{proof} \rangle$

lemma *conv_automaton_conv_urge_commute*:

$\text{conv_automaton } (\text{conv_urge } c \ A) = \text{conv_urge } c \ (\text{conv_automaton } A)$
 $\langle \text{proof} \rangle$

lemma *conv_automaton_conv_urge_commute'*:

$\text{conv_automaton } o \ \text{conv_urge } c = \text{conv_urge } c \ o \ \text{conv_automaton}$
 $\langle \text{proof} \rangle$

locale *Simple_Network_Rename_Defs_int_urge* =

Simple_Network_Rename_Defs_int **where** *automata* = *automata* **for** *automata* ::
 $('s \ \text{list} \times 's \ \text{list})$
 $\times (('a :: \text{countable}) \ \text{act}, 's, 'c, \text{int}, 'x :: \text{countable}, \text{int}) \ \text{transition list}$
 $\times (('s :: \text{countable}) \times ('c :: \text{countable}, \text{int}) \ \text{cconstraint}) \ \text{list} \ \text{list}$
+ fixes *urge* :: $'c$

lemma (**in** *Simple_Network_Rename_Defs*) *conv_automaton_of*:

$\text{Simple_Network_Language.conv_A } (\text{automaton_of } A) = \text{automaton_of } (\text{conv_automaton } A)$
 $\langle \text{proof} \rangle$

locale *Simple_Network_Rename* =

Simple_Network_Rename_Defs_int_urge **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 renum_states_inj:
  ∀ i < n_ps. ∀ x ∈ loc_set. ∀ y ∈ loc_set. renum_states i x = renum_states i y ⟶ 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 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: ⋃ ((λg. fst 'set g) 'set (map (snd o snd o snd) automata)) ⊆ loc_set
and loc_set_committed: ⋃ ((set o fst) 'set automata) ⊆ loc_set
and loc_set_urgent: ⋃ ((set o (fst o snd)) 'set automata) ⊆ loc_set
and urge_not_in_clk_set: urge ∉ clk_set'
begin

lemma clk_set'_finite:
  finite clk_set'
  ⟨proof⟩

lemmas [finite_intros] = trans_N_finite

lemma set_act_finite[finite_intros]:
  finite (set_act a)
  ⟨proof⟩

lemma loc_set_finite:
  finite loc_set
  ⟨proof⟩

lemma var_set_finite[finite_intros]:
  finite var_set
  ⟨proof⟩

lemma act_set_finite:
  finite act_set
  ⟨proof⟩

lemma bij_extend_bij_renum_clocks:
  bij (extend_bij renum_clocks (insert urge clk_set'))
  ⟨proof⟩

lemma renum_vars_bij_extends[simp]:
  extend_bij renum_vars var_set x = renum_vars x if x ∈ var_set
  ⟨proof⟩

lemma bij_extend_bij_renum_states:
  bij (extend_bij renum_vars var_set)
  ⟨proof⟩

lemma inj_extend_bij_renum_states: inj (extend_bij (renum_states p) loc_set) if p < n_ps
  ⟨proof⟩

```

lemma *renum_states_extend*:
extend_bij (*renum_states* *p*) *loc_set* *l* = *renum_states* *p* *l* **if** *l* ∈ *loc_set* *p* < *n_ps*
 ⟨*proof*⟩

lemma *renum_acts_bij_extends[simp]*:
extend_bij *renum_acts* *act_set* *x* = *renum_acts* *x* **if** *x* ∈ *act_set*
 ⟨*proof*⟩

lemma *inj_extend_bij_renum_acts*: *inj* (*extend_bij* *renum_acts* *act_set*)
 ⟨*proof*⟩

lemma *constraint_clk_conv_ac*:
constraint_clk (*conv_ac* *ac*) = *constraint_clk* *ac*
 ⟨*proof*⟩

interpretation *urge*: *Prod_TA_sem_urge*
 (*set* *broadcast*, *map* (*Simple_Network_Language.conv_A* *o* *automaton_of*) *automata*, *map_of*
bounds')
urge
 ⟨*proof*⟩

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'*)
 $\lambda p.$ *extend_bij* (*renum_states* *p*) *loc_set*
map (*conv_urge* *urge*) *automata*
 ⟨*proof*⟩

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'*)
 $\lambda p.$ *extend_bij* (*renum_states* *p*) *loc_set*
map (*conv_urge* *urge*) *automata*
 ⟨*proof*⟩

definition
renum_upd' = *map* ($\lambda(x, upd).$ (*renum_vars* *x*, *map_exp* *renum_vars* *upd*))

definition
renum_reset' = *map* *renum_clocks*

definition *renum_automaton'* **where**
renum_automaton' *i* $\equiv \lambda(committed, trans, inv).$ *let*
committed' = *map* (*renum_states* *i*) *committed*;
trans' = *map* ($\lambda(l, g, a, upd, r, l').$
 (*renum_states* *i* *l*,
map_cconstraint *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_cconstraint 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_cconstraint_def renum_act_def
]

```

```

lemma renum_automaton_eq:
  rename.renum_automaton p (automata ! p) = renum_automaton p (automata ! p)
  if p < n_ps
<proof>

```

```

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
<proof>

```

```

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
<proof>

```

```

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
<proof>

```

```

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')

```

<proof>

```

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>

```

```

lemma default_map_of_merge_pairs:
  default_map_of [] (merge_pairs xs ys) = (λx.
  (if x ∈ fst ' set xs then the (map_of xs x) @ default_map_of [] ys x
  else default_map_of [] ys x))
<proof>

```

lemma *automaton_of_conv_urge_commute*:
 $\text{automaton_of } (\text{conv_urge } \text{urges } A) = \text{urges.add_reset } (\text{urges.add_inv } (\text{automaton_of } A))$
 $\langle \text{proof} \rangle$

lemma *urges_commute*:
 $\text{rename.sem} = \text{urges.A_urges}$
 $\langle \text{proof} \rangle$

lemma *conv_automaton_of'*:
 $\text{automaton_of} \circ \text{conv_automaton} = \text{Simple_Network_Language.conv_A } o \text{ automaton_of}$
 $\langle \text{proof} \rangle$

sublocale *urges_bisim: Bisimulation_Invariant*
 $\lambda(L, s, u) (L', s', u'). \text{step_u' sem } L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). \text{step_u' rename.sem } L s u L' s' u'$
 $\lambda(L, s, u) (L', s', u'). L' = L \wedge s' = s \wedge u' = u(\text{urges} := 0)$
 $\lambda(L, s, u). L \in \text{states } \lambda(L, s, u). \text{True}$
 $\langle \text{proof} \rangle$

lemma *check_sexp_equiv*:
assumes $\text{urges_bisim.A_B.equiv' } (L, s, u) (L', s', u')$
shows $\text{check_sexp } e L (\text{the } o s) = \text{check_sexp } e L' (\text{the } o s')$
 $\langle \text{proof} \rangle$

context
fixes $a_0 :: 's \text{ list} \times ('x \Rightarrow \text{int option}) \times ('c \Rightarrow \text{real})$
assumes $\text{start: fst } a_0 \in \text{states snd } (\text{snd } a_0) \text{ urges} = 0$
begin

lemma *start_equiv*: $\text{urges_bisim.A_B.equiv' } a_0 a_0$
 $\langle \text{proof} \rangle$

lemma *urges_models_iff*:
 $\text{sem}, a_0 \models \Phi \iff \text{rename.sem}, a_0 \models \Phi$
 $\langle \text{proof} \rangle$

lemma *urges_has_deadlock_iff*:
 $\text{has_deadlock sem } a_0 \iff \text{has_deadlock rename.sem } a_0$
 $\langle \text{proof} \rangle$

lemma *state_formula_compatible*:
 $(\bigcup x \in \text{set_state_formula } \varphi. \text{locs_of_sexp } x) \subseteq \{0..n_ps\} \implies$
 $\text{rel_state_formula } \text{urges_bisim.compatible}$
 $(\text{map_state_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \varphi)$
 $(\text{map_state_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \varphi) \text{ and path_formula_compatible:}$
 $(\bigcup x \in \text{set_path_formula } \psi. \text{locs_of_sexp } x) \subseteq \{0..n_ps\} \implies$
 $\text{rel_path_formula } \text{urges_bisim.compatible}$
 $(\text{map_path_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \psi)$
 $(\text{map_path_formula } (\lambda P (L, s, _). \text{check_sexp } P L (\text{the } o s)) \psi)$
 $\langle \text{proof} \rangle$

lemmas *urges_models_state_compatible* =
 $\text{urges_bisim.models_state_compatible}[OF \text{state_formula_compatible}]$

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 *s*₀ :: ('x $\times$ int) list
and *L*₀ :: 's list
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)
begin

lemma *rename_n_ps_eq*:
rename.n_ps = *n_ps*
 ⟨proof⟩

lemma *rename_states_eq*:
rename.states = *states*
 ⟨proof⟩

lemma *state_eq*:
 map_of (map ($\lambda(x, y).$ (extend_bij renum_vars var_set *x*, *y*)) *s*₀) =
 map_of (map ($\lambda(x, y).$ (renum_vars *x*, *y*)) *s*₀)
 ⟨proof⟩

lemma *sexp_eq*:
assumes
 ⟨vars_of_sexp *e* $\subseteq$ *var_set*⟩
 ⟨set2_sexp *e* $\subseteq$ *loc_set*⟩
 ⟨locs_of_sexp *e* $\subseteq$ {0..*n_ps*}⟩
shows ⟨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*⟩
 ⟨proof⟩

lemma *L*₀_dom:
 length *L*₀ = *n_ps* set *L*₀ $\subseteq$ *loc_set*
 ⟨proof⟩

definition
*a*₀ = (*L*₀, map_of *s*₀, $\lambda_.$ 0)

definition
*a*₀' = (
 map_index ($\lambda p.$ renum_states *p*) *L*₀,
 map_of (map ($\lambda(x, y).$ (renum_vars *x*, *y*)) *s*₀),
 $\lambda_.$ 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')
 $\lambda p. \text{extend_bij}(\text{renum_states } p) \text{ loc_set}$
 $s_0 \ L_0$
map (conv_urge urge) automata
<proof>

lemma *rename_a0_eq:*

rename.a₀ = a₀
<proof>

lemma *rename_a0'_eq:*

rename.a₀' = a₀'
<proof>

lemma *has_deadlock_iff:*

has_deadlock rename.renum.sem a₀' $\longleftrightarrow$ has_deadlock sem a₀
<proof>

lemma *N_eq_sem:*

Simple_Network_Language_Model_Checking.N broadcast automata bounds' = sem
<proof>

lemma *rename_N_eq_sem':*

Simple_Network_Language_Model_Checking.N
(map renum_acts broadcast)
(map_index renum_automaton automata)
(map ($\lambda(a,p). (\text{renum_vars } a,p)$) bounds')
= renum.sem
<proof>

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 a₀, unfolded a₀_def, simplified, OF L₀_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 × 's list
   × (('a :: countable) act, 's, 'c, int, 'x :: countable, int) transition list
   × (('s :: countable) × ('c :: countable, int) cconstraint) list) list +
fixes Φ :: (nat, 's, 'x, int) formula
assumes formula_dom:
  set2_formula Φ ⊆ loc_set
  locs_of_formula Φ ⊆ {0..n_ps}
  vars_of_formula Φ ⊆ 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')
  λp. extend_bij (renum_states p) loc_set
  s0 L0
  map (conv_urge urge) automata
  ⟨proof⟩

definition
  Φ' = map_formula (λp. renum_states p) renum_vars id Φ

lemma rename_Φ'_eq:
  rename.Φ' = Φ'
  ⟨proof⟩

lemma models_iff1:
  rename.renum.sem, a0' ⊨ Φ' ↔ rename.sem, a0 ⊨ Φ
  ⟨proof⟩

lemma models_iff2:
  rename.sem, a0 ⊨ Φ ↔ sem, a0 ⊨ Φ
  ⟨proof⟩

lemma models_iff:
  rename.renum.sem, a0' ⊨ Φ' ↔ sem, a0 ⊨ Φ
  ⟨proof⟩

lemmas models_iff' =
  models_iff[unfolded rename_N_eq_sem, folded N_eq_sem, unfolded a0_def a0'_def Φ'_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)
   $\langle proof \rangle$ 

lemma final_fun_is_False:
  Fi a = False
   $\langle proof \rangle$ 

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
   $\langle proof \rangle$ 

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
     $\gg= (\lambda x. \text{return } (\text{Some } x))$ 
  else return None

theorem deadlock_check:
   $\langle emp \rangle$  precond_dc
  show_clock show_state broadcast bounds automata m num_states num_actions k L0 s0
  (formula.EX (not sexp.true))
   $\langle \lambda \text{ Some } r \Rightarrow \uparrow($ 
    Simple_Network_Impl_nat_ceiling_start_state
    broadcast bounds automata m num_states num_actions k L0 s0 (formula.EX (not
sexp.true))  $\wedge$ 
     $r = \text{has\_deadlock } (N \text{ broadcast automata bounds}) (L_0, \text{map\_of } s_0, \lambda\_ . 0)$ 
  )
  | None  $\Rightarrow \uparrow(\neg \text{Simple\_Network\_Impl\_nat\_ceiling\_start\_state}$ 
    broadcast bounds automata m num_states num_actions k L0 s0 (formula.EX (not sexp.true)))
   $\rangle_t$ 
   $\langle proof \rangle$ 

```

```

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)
        )
      vs
    )
  sccs
  d
);
d = (
  fold
    (λ vs d.
      let
        dscc =
          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 Φ 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 Φ :: (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:  $\bigcup ((\lambda g. \text{fst } ' \text{ set } g) ' \text{ set } (\text{map } (\text{snd } o \text{ snd } o \text{ snd}) \text{ automata})) \subseteq \text{loc\_set}$ 
and loc_set_broadcast:  $\bigcup ((\text{set } o \text{ fst}) ' \text{ set } \text{ automata}) \subseteq \text{loc\_set}$ 
and loc_set_urgent:  $\bigcup ((\text{set } o (\text{fst } o \text{ snd})) ' \text{ set } \text{ automata}) \subseteq \text{loc\_set}$ 
and urge_not_in_clk_set: urge  $\notin$  clk_set'
assumes L0_states: L0  $\in$  states
  and s0_dom: fst ' set s0 = var_set and s0_distinct: distinct (map fst s0)
assumes formula_dom:
  set2_formula  $\Phi \subseteq \text{loc\_set}$ 
  locs_of_formula  $\Phi \subseteq \{0..<n\_ps\}$ 
  vars_of_formula  $\Phi \subseteq \text{var\_set}$ 
begin

```

interpretation *Simple_Network_Rename_Formula*
 $\langle \text{proof} \rangle$

lemmas *Simple_Network_Rename_intro* = *Simple_Network_Rename_Formula_axioms*

end

lemma *is_result_assert_iff*:
is_result (*assert* *b* *m*) $\longleftrightarrow$ *b*
 $\langle \text{proof} \rangle$

lemma *is_result_combine_Cons_iff*:
is_result (*combine* (*x* # *xs*)) $\longleftrightarrow$ *is_result* *x* $\wedge$ *is_result* (*combine* *xs*)
 $\langle \text{proof} \rangle$

lemma *is_result_combine_iff*:
is_result (*a* <|> *b*) $\longleftrightarrow$ *is_result* *a* $\wedge$ *is_result* *b*
 $\langle \text{proof} \rangle$

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* Φ *s0* *L0* $\longleftrightarrow$
is_result (*check_renaming* *urge* Φ *L0* *s0*)
 $\langle \text{proof} \rangle$

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 (∀ i < n_ps. let (_, _, trans, _) = (automata ! i) in ∀ (l, _, _, _, _, l') ∈ set trans.
      l < num_states i ∧ l' < num_states i)
    (STR "Number of states is correct (transitions)'),
    assert (∀ i < n_ps. let (_, _, _, inv) = (automata ! i) in ∀ (x, _) ∈ set inv. x < num_states
i)
    (STR "Number of states is correct (invariants)'),
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, _, _, f, _, _) ∈ set trans.
      ∀ (x, upd) ∈ set f. x < n_vs ∧ (∀ i ∈ vars_of_exp upd. i < n_vs))
    (STR "Variable set bounded (updates)'),
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, b, _, _, _, _, _) ∈ set trans.
      ∀ i ∈ vars_of_bexp b. i < n_vs)
    (STR "Variable set bounded (guards)'),
    assert (∀ i < n_vs. fst (bounds' ! i) = i)
    (STR "Bounds first index'),
    assert (∀ a ∈ set broadcast. a < num_actions)
    (STR "Broadcast actions bounded'),
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, _, a, _, _, _) ∈ set trans.
      pred_act (λa. a < num_actions) a)
    (STR "Actions bounded (transitions)'),
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, g, _, _, r, _) ∈ set trans.
      (∀ c ∈ set r. 0 < c ∧ c ≤ m) ∧
      (∀ (c, x) ∈ collect_clock_pairs g. 0 < c ∧ c ≤ m ∧ x ∈ ℕ))
    (STR "Clock set bounded (transitions)'),
    assert (∀ (_, _, _, inv) ∈ set automata. ∀ (l, g) ∈ set inv.
      (∀ (c, x) ∈ collect_clock_pairs g. 0 < c ∧ c ≤ m ∧ x ∈ ℕ))
    (STR "Clock set bounded (invariants)'),
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, g, a, _, _, _) ∈ set trans.
      case a of In a ⇒ a ∈ set broadcast → g = [] | _ ⇒ True)
    (STR "Broadcast receivers are unguarded'),
    assert (∀ (_, U, _, _) ∈ 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*
<proof>

context

fixes *k* :: nat list list list

and *L*₀ :: nat list

and *s*₀ :: (nat × int) list

and *formula* :: (nat, nat, nat, int) formula

begin

definition *check_precond2* **where**

check_precond2 ≡

combine [

assert (∀ i < n_ps. ∀ (l, g) ∈ set ((snd o snd o snd) (automata ! i)).

```

     $\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x)$ 
    (STR "Ceiling invariants"),
  assert ( $\forall i < n\_ps. \forall (l, \_, g, \_) \in \text{set } ((fst \circ snd \circ snd) (automata ! i)).$ 
    ( $\forall (x, m) \in \text{collect\_clock\_pairs } g. m \leq \text{int } (k ! i ! l ! x)$ ))
    (STR "Ceiling transitions"),
  assert ( $\forall i < n\_ps. \forall (l, b, g, a, upd, r, l') \in \text{set } ((fst \circ snd \circ snd) (automata ! i)).$ 
     $\forall c \in \{0..m+1\} - \text{set } r. k ! i ! l' ! c \leq k ! i ! l ! c$ )
    (STR "Ceiling resets"),
  assert (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 ( $\forall (\_, \_, \_, inv) \in \text{set } automata. \text{distinct } (\text{map } fst \text{ inv})$ )
    (STR "Unambiguous invariants"),
  assert (bounded bounds (map_of s0))
    (STR "Initial state bounded"),
  assert (length L0 = n_ps)
    (STR "Length of initial state"),
  assert ( $\forall i < n\_ps. L0 ! i \in \text{fst } ' \text{set } ((fst \circ snd \circ snd) (automata ! i))$ )
    (STR "Initial state has outgoing transitions"),
  assert (vars_of_formula 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_broadcast_bounds' automata m num_states
    k L0 s0 formula
  <proof>

```

end

definition

check_precond k L0 s0 formula $\equiv$ 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  $\longleftrightarrow$  is_result (check_precond k L0 s0 formula)
  <proof>

```

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 " "* |> ($\lambda xs.$ "[" @ *concat xs* @ "]" @ *s*)

instance
 ⟨*proof*⟩

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 " "* |> ($\lambda xs.$ "[" @ *concat xs* @ "]" @ *s*)

instance
 ⟨*proof*⟩

end

definition

pad m s = *replicate m CHR " "* @ *s*

definition *shows_rat r* $\equiv$ *case r of fract.Rat s f b* $\Rightarrow$
 (*if s then* "" *else* "-") @ *show f* @ (*if b $\neq$ 0 then* "." @ *show b* *else* "")

fun *shows_json* :: *nat* $\Rightarrow$ *JSON* $\Rightarrow$ *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* :: *JSON*) *rest* = *shows_json* 0 *x* @ *rest* **for** *p*

definition *shows_list* *jsons* *s* =
 map (*shows_json* 0) *jsons* |> intersperse ", " |> (λ*xs*. "[" @ concat *xs* @ "]" @ *s*)

instance
 ⟨*proof*⟩

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* (λ*i* *v*. *show* (*inv_renum_vars* *i*) @ "=" @ *show* *v*)

definition

$show_state\ inv_renum_states\ inv_renum_vars \equiv \lambda(L, vs).$
 $let\ L = show_locs\ inv_renum_states\ L;$ $vs = show_vars\ inv_renum_vars\ vs\ in$
 $"<"\ @\ L\ @\ ">,"<"\ @\ vs\ @\ ">"$

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 ⇒ ↑(
  (¬ has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0) →
    ¬ N broadcast automata bounds, (L0, map_of s0, λ_. 0) ⊨ formula
  ))
| Renaming_Failed ⇒ ↑(¬ Simple_Network_Rename_Formula
  broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s0 L0 automata
  formula)
| Preconds_Unsat ⇒ ↑(¬ 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)
  (map_formula renum_states renum_vars id formula))
>t
<proof>

```

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 ⇒ ↑( has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0))
| Unsat ⇒ ↑(¬ has_deadlock (N broadcast automata bounds) (L0, map_of s0, λ_. 0))

```

```

| Renaming_Failed  $\Rightarrow \uparrow(\neg \text{Simple\_Network\_Rename\_Formula}$ 
  broadcast bounds renum_acts renum_vars renum_clocks renum_states urge  $s_0$   $L_0$  automata
  (formula.EX (not sexp.true)))
| Preconds_Unsat  $\Rightarrow \uparrow(\neg \text{Simple\_Network\_Impl\_nat\_ceiling\_start\_state}$ 
  (map renum_acts broadcast)
  (map  $(\lambda(a,p). (\text{renum\_vars } a, p))$  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  $L_0$ ) (map  $(\lambda(x, v). (\text{renum\_vars } x, v))$   $s_0$ )
  (formula.EX (not sexp.true)))
>_t
<proof>

```

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)
<proof>

```

lemma int_Nat_range_iff:

```

(n :: int)  $\in \mathbb{N} \longleftrightarrow n \geq 0$  for n
<proof>

```

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 precondition_mc in SML module_name Test

export_code precondition_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_cconstraint_def
  Simple_Network_Rename_Defs.map_cconstraint_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:
  
$$L \in \text{states} \iff \text{length } L = n\_ps \wedge (\forall i. i < n\_ps \longrightarrow$$


$$(\exists (l, b, g, a, r, u, l') \in \text{fst} (\text{snd} (\text{snd} (\text{fst} (\text{snd } A) ! i))). L ! i = l \vee L ! i = l')$$

  <proof>

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  $\equiv$ 

$$\bigcup g \in \text{set} (\text{filter } (\lambda(a, b). a = l) (\text{snd} (\text{snd} (\text{snd} (\text{automata} ! i))))). \text{collect\_clock\_pairs } (\text{snd } g)$$


definition clkp_set'' i l  $\equiv$ 

$$\text{clkp\_inv } i \ l \cup (\bigcup (l', b, g, \_) \in \text{set} (\text{fst} (\text{snd} (\text{snd} (\text{automata} ! i)))).$$


$$\text{if } l' = l \text{ then collect\_clock\_pairs } g \text{ else } \{\}$$


```

definition

$collect_resets\ i\ l = (\bigcup (l', b, g, a, f, r, _) \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))).$
if $l' = l$ then set r else $\{\}$

context

fixes $q\ c :: nat$

begin

definition $n \equiv num_states\ q$

definition $V \equiv \lambda\ v.\ v \leq n$

definition

$bound_g\ l \equiv$
 $Max\ (\{0\} \cup \bigcup ((\lambda\ (x, d). \text{if } x = c \text{ then } \{d\} \text{ else } \{\})\ 'clkp_set''\ q\ l))$

definition

$bound_inv\ l \equiv$
 $Max\ (\{0\} \cup \bigcup ((\lambda\ (x, d). \text{if } x = c \text{ then } \{d\} \text{ else } \{\})\ 'clkp_inv\ q\ l))$

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 ≡
let
  w = calc_shortest_scc_paths G n
in
  map (λ x. case x of None ⇒ 0 | Some x ⇒ 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 $mem_assoc\ x = list_ex\ (\lambda(y, _). x = y)$

definition $mk_renaming\ str\ xs \equiv$

```
do {
  mapping  $\leftarrow fold\_error$ 
    ( $\lambda x\ m.$ 
      if  $mem\_assoc\ x\ m$  then  $Error\ [STR\ "Duplicate\ name:\ " + str\ x]$  else  $(x, length\ m) \# m |>$ 
    )  $xs\ []$ ;
  Result (let
     $m = map\_of\ mapping$ ;
     $f = (\lambda x.$ 
      case  $m\ x$  of
        None  $\Rightarrow let\ \_ = println\ (STR\ "Key\ error:\ " + str\ x)$  in undefined
      | Some  $v \Rightarrow v$ );
     $m = map\_of\ (map\ prod.swap\ mapping)$ ;
     $f\_inv = (\lambda x.$ 
      case  $m\ x$  of
        None  $\Rightarrow let\ \_ = println\ (STR\ "Key\ error:\ " + String.implode\ (show\ x))$  in undefined
      | Some  $v \Rightarrow v$ );
    in  $(f, f\_inv)$ 
  )
}
```

definition

```
extend_domain  $m\ d\ n \equiv$ 
  let
    ( $i, xs$ ) = fold
      ( $\lambda x\ (i, xs).$  if  $x \in set\ d$  then  $(i + 1, (x, i + 1) \# xs)$  else  $(i, xs)$ )  $d\ (n, [])$ ;
     $m' = map\_of\ xs$ 
  in
    ( $\lambda x.$  if  $x \in set\ d$  then the  $(m'\ x)$  else  $m\ x$ )
```

lemma $is_result_make_err$:

```
 $is\_result\ (make\_err\ m\ e) \longleftrightarrow False$ 
<proof>
```

lemma $is_result_assert_msg_iff$:

```
 $is\_result\ (assert\_msg\ b\ m\ r) \longleftrightarrow is\_result\ r \wedge b$ 
<proof>
```

context $Simple_Network_Impl$

begin

definition $action_set$ **where**

```
 $action\_set \equiv$ 
  ( $\bigcup (\_, \_, trans, \_) \in set\ automata. \bigcup (\_, \_, \_, a, \_, \_, \_) \in set\ trans. set\_act\ a$ )
   $\cup set\ broadcast$ 
```

definition loc_set' **where**

```
 $loc\_set'\ p = (\bigcup (l, \_, \_, \_, \_, \_, l') \in set\ (fst\ (snd\ (snd\ (automata\ !\ p))))). \{l, l'\}$  for  $p$ 
```

end

definition

concat_str = *String.implode* o *concat* o *map String.explode*

Unsafe Glue Code definition *list_of_set'* :: 'a set $\Rightarrow$ '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'* $\hookrightarrow$ (*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* $\equiv$ λ *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' = (λi . *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 = (λi . *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* (λi . *loc_set' i* |> *remdups* |> *length*) [*0..<n_ps*];
num_states = (λi . *num_states_list ! i*);
mk_renaming = *mk_renaming* (λx . *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 = (λc . *if* *c* = 0 *then STR "0"* *else inv_renum_clocks (c - 1)*);
renum_states_list' $\leftarrow$ *combine_map* (λi . *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 = (λi . *renum_states_list ! i*);
let inv_renum_states = (λi . *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* $\equiv \lambda dc\ ids_to_names\ (broadcast, automata, bounds)\ L_0\ s_0\ formula.$

```

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  $\equiv \lambda dc\ ids\_to\_names\ (broadcast, automata, bounds)\ L_0\ s_0\ formula.$  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 $\Rightarrow$ *map_of as |> Result*
 | *_* $\Rightarrow$ *Error [STR "json_to_map: expected object"]*
for json

definition *get* **where**
get m x $\equiv$ *case m x of*
 None $\Rightarrow$ *Error [STR "(Get) key not found: " + String.implode (show x)]*
 | *Some a* $\Rightarrow$ *Result a*

definition
get_default def m x $\equiv$ *case m x of None* $\Rightarrow$ *def* | *Some a* $\Rightarrow$ *a*

definition *default* **where**
default def x $\equiv$ *case x of Result s* $\Rightarrow$ *s* | *Error e* $\Rightarrow$ *def*

definition *of_array* **where**
of_array json $\equiv$ *case json of*
 JSON.Array s $\Rightarrow$ *Result s*
 | *_* $\Rightarrow$ *Error [STR "of_array: expected sequence"]*
for json

definition *of_string* **where**
of_string json $\equiv$ *case json of*
 JSON.String s $\Rightarrow$ *Result (String.implode s)*
 | *_* $\Rightarrow$ *Error [STR "of_array: expected sequence"]*
for json

definition *of_nat* **where**
of_nat json $\equiv$ *case json of*
 JSON.Nat n $\Rightarrow$ *Result n*
 | *_* $\Rightarrow$ *Error [STR "of_nat: expected natural number"]*
for json

definition *of_int* **where**
of_int json $\equiv$ *case json of*
 JSON.Int n $\Rightarrow$ *Result n*
 | *_* $\Rightarrow$ *Error [STR "of_int: expected integral number"]*
for json

definition [*consuming*]:
lx_underscore = *exactly " _" with ($\lambda_.$ CHR " _")*

definition [*consuming*]:
lx_hyphen = *exactly " -" with ($\lambda_.$ 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* ≡ *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)]
⟨proof⟩
```

lemma *parse parse_bounds* (STR "'") = Result []
 ⟨proof⟩

definition [*consuming*]:

```
scan_var = ta_var_ident
```

abbreviation *seq_ignore_left_ws* (**infixr** *-- 60)

where *p* *-- *q* ≡ *token p* *-- *q* **for** *p q*

abbreviation *seq_ignore_right_ws* (**infixr** --* 60)

where *p* --* *q* ≡ *token p* --* *q* **for** *p q*

abbreviation *seq_ws* (**infixr** --- 60)

where *seq_ws p q* ≡ *token p* -- *q* **for** *p q*

definition *scan_acconstraint* **where** [*unfolded Let_def, consuming*]:

```
scan_acconstraint ≡
  let scan =
    (λs c. scan_var --- exactly s *-- token lx_int with (λ(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 ≡ exactly lparen *-- inner --* token (exactly rparen)
```

definition [*consuming*]: *scan_loc* ≡

```
(scan_var --- (exactly "." *-- scan_var))
with (λ (p, s). loc (String.implode p) (String.implode s))
```

definition $[consuming]$: $scan_bexp_elem \equiv scan_acconstraint \parallel scan_loc$

abbreviation $scan_parens' \equiv scan_parens \text{ "" "" }$

definition $[consuming]$: $scan_infix_pair\ a\ b\ s \equiv a \text{ --- exactly } s \text{ **--- token } b$

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 $scan_infix_pair\ A\ B\ s\ l = scan_infix_pair\ A'\ B'\ s'\ l'$

$\langle proof \rangle$

lemma $[fundef_cong]$:

assumes $\bigwedge l2. ll_fuel\ l2 < ll_fuel\ l \implies A\ l2 = A'\ l2\ l = l'$

shows $scan_parens'\ A\ l = scan_parens'\ A'\ l'$

$\langle proof \rangle$

lemma $token_cong[fundef_cong]$:

assumes $\bigwedge l2. ll_fuel\ l2 \leq ll_fuel\ l \implies A\ l2 = A'\ l2\ l = l'$

shows $token\ A\ l = token\ A'\ l'$

$\langle proof \rangle$

lemma $is_cparser_scan_parens'[parser_rules]$:

$is_cparser\ (scan_parens'\ a)$

$\langle proof \rangle$

fun $aexp$ and $mexp$ and $scan_exp$ and $scan_7$ and $scan_6$ and $scan_0$ **where** — slow: 90s

$aexp ::=$

$token\ lx_int\ with\ exp.const \parallel token\ scan_var\ with\ exp.var\ o\ String.implode \parallel$

$scan_parens'\ (scan_exp \text{ --- exactly } "?" \text{ **--- scan_7 --- exactly } ":" \text{ **--- scan_exp})$

$with\ (\lambda\ (e1, b, e2). exp.if_then_else\ b\ e1\ e2) \parallel$

$tk_lparen \text{ **--- scan_exp ---** } tk_rparen$

$| mexp ::= chainL1\ aexp\ (multiplicative_op\ with\ (\lambda f\ a\ b. exp.binop\ f\ a\ b))$

$| scan_exp ::= chainL1\ mexp\ (additive_op\ with\ (\lambda f\ a\ b. exp.binop\ f\ a\ b))$

$| scan_7 ::=$

$scan_infix_pair\ scan_6\ scan_7\ " \rightarrow " \text{ with } uncurry\ bexp.implies \parallel$

$scan_infix_pair\ scan_6\ scan_7\ " || " \text{ with } uncurry\ bexp.or \parallel$

$scan_6 \mid$

$scan_6 ::=$

$scan_infix_pair\ scan_0\ scan_6\ "&\&" \text{ with } uncurry\ bexp.and \parallel$

$scan_0 \mid$

$scan_0 ::=$

$(exactly\ "\sim" \parallel exactly\ "\^") \text{ **--- scan_parens'\ scan_7 with } bexp.not \parallel$

$token\ (exactly\ "true") \text{ with } (\lambda _. bexp.true) \parallel$

$scan_infix_pair\ aexp\ aexp\ "<=" \text{ with } uncurry\ bexp.le \parallel$

$scan_infix_pair\ aexp\ aexp\ "<" \text{ with } uncurry\ bexp.lt \parallel$

$scan_infix_pair\ aexp\ aexp\ "==" \text{ with } uncurry\ bexp.eq \parallel$

$scan_infix_pair\ aexp\ aexp\ ">" \text{ with } uncurry\ bexp.gt \parallel$

$scan_infix_pair\ aexp\ aexp\ ">=" \text{ with } uncurry\ bexp.ge \parallel$

$scan_parens'\ scan_7$

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'
  ⟨proof⟩

lemma [parser_rules]:
  is_cparser scan_6'
  ⟨proof⟩

end
end

abbreviation scan_bexp  $\equiv$  scan_7' scan_bexp_elem sexp.imply sexp.or sexp.and sexp.not

lemma [parser_rules]:
  is_cparser scan_bexp
  ⟨proof⟩

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)))
  ⟨proof⟩

definition [consuming]: scan_prefix p head = exactly head **-- p for p

definition [consuming]: scan_formula  $\equiv$ 
  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*
 ⟨proof⟩

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 ≡ *default b a*

definition
parse_action s ≡ *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* ∈ set clocks then (eq *a b* # *cs, es*) else (*cs, eq a b* # *es*)) |
chop_sexp clocks (le *a b*) (*cs, es*) =
 (if *a* ∈ set clocks then (le *a b* # *cs, es*) else (*cs, le a b* # *es*)) |
chop_sexp clocks (lt *a b*) (*cs, es*) =
 (if *a* ∈ set clocks then (lt *a b* # *cs, es*) else (*cs, lt a b* # *es*)) |
chop_sexp clocks (ge *a b*) (*cs, es*) =
 (if *a* ∈ set clocks then (ge *a b* # *cs, es*) else (*cs, ge a b* # *es*)) |
chop_sexp clocks (gt *a b*) (*cs, es*) =
 (if *a* ∈ 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* ⇒ *_* **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_berp* :: (*String.literal, String.literal, String.literal, int*) *sexp* ⇒ *_* **where**
sexp_to_berp (lt *a* (*b* :: *int*)) = *berp.lt* (*exp.var a*) (*exp.const b*) |> *Result* |
sexp_to_berp (le *a b*) = *berp.le* (*exp.var a*) (*exp.const b*) |> *Result* |
sexp_to_berp (eq *a b*) = *berp.eq* (*exp.var a*) (*exp.const b*) |> *Result* |
sexp_to_berp (ge *a b*) = *berp.ge* (*exp.var a*) (*exp.const b*) |> *Result* |
sexp_to_berp (gt *a b*) = *berp.gt* (*exp.var a*) (*exp.const b*) |> *Result* |
sexp_to_berp (and *a b*) =
 do {*a* ← *sexp_to_berp a*; *b* ← *sexp_to_berp b*; *berp.and a b* |> *Result*} |
sexp_to_berp (*sexp.or a b*) =
 do {*a* ← *sexp_to_berp a*; *b* ← *sexp_to_berp b*; *berp.or a b* |> *Result*} |
sexp_to_berp (*imply a b*) =
 do {*a* ← *sexp_to_berp a*; *b* ← *sexp_to_berp b*; *berp.imply a b* |> *Result*} |
sexp_to_berp x = *Error [STR "Illegal construct in binary operation"]*

definition [*consuming*]:

```
scan_update ≡
scan_var --- (exactly "" || exactly "!=") **--- scan_exp
with (λ(s, x). (String.implode s, x))
```

abbreviation *scan_updates* ≡ *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 ≡
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 ← sexp_to_bexp e;
    assert (set_bexp b ⊆ set vars) (String.implode ("Unknown variable in bexp: " @ show b));
    Result (g, b)
  } for inv
```

definition *compile_invariant'* **where**

```
compile_invariant' clocks vars inv ≡
if inv = STR "" then
  Result ([], bexp.true)
else do {
  inv ← 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 ≡ do {
  n ← of_object n;
  ID ← get n "id" >> of_nat;
  name ← get n "name" >> of_string;
  inv ← get n "invariant" >> of_string;
  (inv, inv_vars) ←
    compile_invariant' clocks vars inv |> err_msg (STR "Failed to parse invariant!");
  assert (case inv_vars of bexp.true ⇒ True | _ ⇒ False)
    (STR "State invariants on nodes are not supported");
  Result ((name, ID), inv)
}
```

definition *convert_edge* **where**

```

convert_edge clocks vars e ≡ do {
  e ← of_object e;
  source ← get e "source" ≫ of_nat;
  target ← get e "target" ≫ 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) |
rename_locs_sexp f (gt a b) = Result (gt a b)

```

fun rename_locs_formula **where**

```

rename_locs_formula f (formula.EX  $\varphi$ ) = rename_locs_sexp f  $\varphi$   $\gg$  Result o formula.EX |
rename_locs_formula f (EG  $\varphi$ ) = rename_locs_sexp f  $\varphi$   $\gg$  Result o EG |
rename_locs_formula f (AX  $\varphi$ ) = rename_locs_sexp f  $\varphi$   $\gg$  Result o AX |
rename_locs_formula f (AG  $\varphi$ ) = rename_locs_sexp f  $\varphi$   $\gg$  Result o AG |
rename_locs_formula f (Leadsto  $\varphi$   $\psi$ ) =
do { $\varphi$  ← rename_locs_sexp f  $\varphi$ ;  $\psi$  ← rename_locs_sexp f  $\psi$ ; Leadsto  $\varphi$   $\psi$  |> Result}

```

definition convert :: JSON $\Rightarrow$

```

((nat  $\Rightarrow$  nat  $\Rightarrow$  String.literal)  $\times$  (String.literal  $\Rightarrow$  nat)  $\times$  String.literal list  $\times$ 
 (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  $\times$ 
 (String.literal  $\times$  int  $\times$  int) list  $\times$ 
 (nat, nat, String.literal, int) formula  $\times$  nat list  $\times$  (String.literal  $\times$  int) list
) Error_List_Monad.result where

```

```

convert json  $\equiv$  do {
  all ← of_object json;
  automata ← get all "automata";
  automata ← of_array automata;
  let broadcast = default [] (do {x ← get all "broadcast"; of_array x});
  broadcast ← combine_map of_string broadcast;
  let _ = trace_level 3
    ( $\lambda$ _. return (STR "Broadcast channels " + String.implode (show broadcast)));
  let bounds = default (STR "") (do {
    x ← get all "vars"; of_string x
  });
  bounds ← parse parse_bounds bounds |> err_msg (STR "Failed to parse bounds");
  clocks ← get all "clocks";
  clocks ← of_string clocks;
  clocks ← parse (parse_list ( $\lambda$ x ws  $\rightarrow$  ta_var_ident with String.implode)) clocks
    |> err_msg (STR "Failed to parse clocks");
  formula ← get all "formula";
  formula ← of_string formula;
  formula ← parse scan_formula formula |> err_msg (STR "Failed to parse formula");
  automata ← combine_map of_object automata;
  process_names ← combine_map ( $\lambda$ a. get a "name"  $\gg$  of_string) automata;
  assert (distinct process_names) (STR "Process names are ambiguous");
  assert (locs_of_formula formula  $\subseteq$  set process_names) (STR "Unknown process name in
formula");
  let process_names_to_index = List_Index.index process_names;
  init_locs ← combine_map
    ( $\lambda$ a. do {x ← get a "initial"; x ← of_nat x; x |> Result})
    automata;
  let formula = formula.map_formula process_names_to_index id id id formula;
  let vars = map fst bounds;

```

```

let init_vars = map (λx. (x, 0::int)) vars;
names_automata ← combine_map (convert_automaton clocks vars) automata;
let automata = map (snd o snd) names_automata;
let names     = map fst names_automata;
let ids_to_names = map (fst o snd) names_automata;
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⟩

Now we can run MUNTA on a few examples:

⟨ML⟩

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⟩

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
  ⟨Unsynchronized.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>
  <csmma_05.muntax>
  <csmma_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/muntax##
       — 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 116245 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, 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, 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, Formal proof development.