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 **17**
2.1 Syntax and Operational Semantics . . . 17
2.2 Product Automaton . . . 18

3 Networks of Timed Automata – UPPAAL Style **35**
3.1 Syntax and Operational Semantics . . . 36
3.2 Equivalent State Network Automaton . . . 37

4 Implementation of UPPAAL Style Networks **62**
4.1 Pre-compiled Networks With States and Clocks as Natural Numbers . . . 66

5 Imperative Implementation of UPPAAL Style Networks **97**
5.1 Executable Successor Computation . . . 97
5.2 Check Preconditions . . . 147

6 Simple Networks of Automata With Broadcast Channels and Committed Locations **177**
6.1 Product Construction . . . 180

7 Product Bisimulation **274**

8 The Final Semantics **276**

9 Instantiating the Model Checking Locale **280**
9.1 Extracting an Efficient Implementation . . . 297

10 Deadlock Checking **302**
10.1 Notes . . . 303
10.2 Abstract Reachability Checking . . . 303
10.3 Operations . . . 306
10.4 Structural Properties . . . 327
10.5 Correctness of *dbm_subset_fed* . . . 334
10.6 Refined DBM Operations . . . 338
10.7 Transferring Properties . . . 339

10.8 Functional Refinement 347
10.9 Imperative Refinement 352
10.10Implementation for a Concrete Automaton 356
10.11Checking a Property on the Reachable Set 360
10.12Complete Deadlock Checker 363

11 Renaming Identifiers in Simple Networks **376**

12 Assembling the Model Checker and Generating Code **446**

13 Build and Test Exported Program With MLton **476**

Introduction

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

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

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

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

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

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

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

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

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

unbundle *no_library_syntax*

1 Networks of Timed Automata

1.1 Syntax and Operational Semantics

Input, output and internal transitions

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

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

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

inductive *step_n* ::
('a, 'b, 'c, 't, 's) nta ⇒ 's list ⇒ ('c, ('t::time)) cval ⇒ 'b label
⇒ 's list ⇒ ('c, 't) cval ⇒ bool
(_ ⊢_N ⟨_, _⟩ →_ ⟨_,_⟩ [61,61,61,61,61,61] 61)
where
step_n_t:
⟦∀ p ∈ {..<length N}. N!p ⊢ ⟨L!p, u⟩ →^d ⟨L!p, u ⊕ d⟩; d ≥ 0⟧
⟹ N ⊢_N ⟨L, u⟩ →_Del ⟨L, u ⊕ d⟩ |
step_n_i:
⟦
N!p ⊢ l ⟶^g,(Sil a, b),r l'; u ⊢ g; ∀ p ∈ {..<length N}. u' ⊢ inv_of (N!p) (L'!p);
L!p = l; p < length L; L' = L[p := l']; u' = [r→0]u
⟧
⟹ N ⊢_N ⟨L, u⟩ →_Act b ⟨L', u'⟩ |
step_n_s:
⟦N!p ⊢ l1 ⟶^g1,(In a, b1),r1 l1'; N!q ⊢ l2 ⟶^g2,(Out a, b2),r2 l2'; u ⊢ g1; u ⊢ g2;
∀ p ∈ {..<length N}. u' ⊢ inv_of (N!p) (L'!p);
L!p = l1; L!q = l2; p < length L; q < length L; p ≠ q;
L' = L[p := l1', q := l2']; u' = [(r1 @ r2)→0]u
⟧ ⟹ N ⊢_N ⟨L, u⟩ →_Syn b1 b2 ⟨L', u'⟩

inductive_cases[*elim*!]: *N ⊢_N ⟨L, u⟩ →_a ⟨L', u'⟩*

inductive *steps_n* ::
('a, 'b, 'c, 't, 's) nta ⇒ 's list ⇒ ('c, ('t::time)) cval
⇒ 's list ⇒ ('c, 't) cval ⇒ bool
(_ ⊢_N ⟨_, _⟩ → ⟨_,_⟩ [61,61,61] 61)*
where
refl: *N ⊢_N ⟨l, Z⟩ →* ⟨l, Z⟩ |*
step: *N ⊢_N ⟨l, Z⟩ →* ⟨l', Z'⟩ ⟹ N ⊢_N ⟨l', Z'⟩ →_ ⟨l'', Z''⟩*
⟹ N ⊢_N ⟨l, Z⟩ → ⟨l'', Z''⟩*

declare *steps_n.intros*[*intro*]

lemma *stepI2*:
 $N \vdash_N \langle l, Z\rangle \rightarrow* \langle l'', Z''\rangle$ **if** $N \vdash_N \langle l', Z'\rangle \rightarrow* \langle l'', Z''\rangle$ $N \vdash_N \langle l, Z\rangle \rightarrow_b \langle l', Z'\rangle$
 using *that*
 apply *induction*
  apply (*rule steps_n.step*)
   apply (*rule refl*)
  apply *assumption*
 apply *simp*
 by (*rule steps_n.step*; *assumption*)

lemma *step_n_t_I*:
 $[\![\forall\ p \in \{..<length\ N\}.\ u \vdash inv_of\ (N!p)\ (L!p); \forall\ p \in \{..<length\ N\}.\ u \oplus d \vdash inv_of\ (N!p)$
$(L!p)$;
  $d \geq 0$
 $]\!] \Longrightarrow N \vdash_N \langle L, u\rangle \rightarrow_{Del} \langle L, u \oplus d\rangle$
by (*auto intro*: *step_n_t*)

1.2 Product Automaton

abbreviation *state_set* :: ('*a*, '*c*, '*time*, '*s*) *transition set* ⇒ '*s set* **where**
 state_set T ≡ *fst* ‘ *T* ∪ (*snd o snd o snd o snd*) ‘ *T*

lemma *guard_concat*:
 assumes ∀ *g* ∈ *set xs*. *u* ⊢ *g*
 shows *u* ⊢ *concat xs*
 using *assms* **by** (*induction xs*) (*auto simp*: *clock_val_def*)

lemma *guard_append*:
 assumes *u* ⊢ *g1 u* ⊢ *g2*
 shows *u* ⊢ *g1* @ *g2*
 using *assms* **by** (*auto simp*: *clock_val_def*)

lemma *concat_guard*:
 assumes *u* ⊢ *concat xs g* ∈ *set xs*
 shows *u* ⊢ *g*
using *assms* **by** (*auto simp*: *list_all_iff clock_val_def*)

lemma *guard_consE*:
 assumes *u* ⊢ *ac* # *cc*
 obtains *u* ⊢$_a$ *ac u* ⊢ *cc*
 using *assms* **unfolding** *clock_val_def* **by** *auto*

lemma *guard_consI*:
 assumes *u* ⊢$_a$ *ac u* ⊢ *cc*
 shows *u* ⊢ *ac* # *cc*
 using *assms* **unfolding** *clock_val_def* **by** *auto*

lemma *collect_clock_pairs_append_cases*:
 assumes *x* ∈ *collect_clock_pairs* (*g1* @ *g2*)
 shows *x* ∈ *collect_clock_pairs g1* ∨ *x* ∈ *collect_clock_pairs g2*
  using *assms* **unfolding** *collect_clock_pairs_def* **by** *auto*

lemma *list_update_nth_split*:
 assumes *j* < *length xs*

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

locale *Product_TA_Defs* =
fixes $N :: ('a, 'b, 'c, 't, 's)\ nta$
begin

abbreviation $T \equiv map\ trans_of\ N$
abbreviation $I \equiv map\ inv_of\ N$

definition $states = \{L.\ length\ L = length\ T \wedge (\forall\ p \in \{..<length\ T\}.\ L!p \in state_set\ (T!p))\}$

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

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

definition
$product_trans \equiv product_trans_i \cup product_trans_s$

lemma *product_state_set_subs*:
assumes $q < length\ N\ l \in state_set\ product_trans$
shows $l\ !\ q \in state_set\ (T\ !\ q)$
using *assms*
unfolding *product_trans_def product_trans_i_def product_trans_s_def states_def*
apply *safe*
apply (*solves auto*)
apply (*solves auto*)
apply *simp*
apply (*subst list_update_nth_split*; *force*)
apply *simp*
apply (*subst list_update_nth_split*)
apply *simp*
apply *safe*
apply (*subst* (*asm*) (*2*) *list_update_nth_split*; *force*)
apply (*subst list_update_nth_split*)
apply *simp*
by (*subst* (*asm*) (*2*) *list_update_nth_split*; *force*) — slow: 13s

definition
$product_invariant\ L \equiv$
$concat\ (map\ (\lambda\ p.\ if\ p < length\ I\ then\ (I!p)\ (L!p)\ else\ [])\ [0..<length\ L])$

definition $product_ta :: ('a \times 'b\ label, 'c, 't, 's\ list)\ ta$ **where**
$product_ta \equiv (product_trans,\ product_invariant)$

lemma *collect_clki_product_invariant*:

```
  Timed_Automata.collect_clki product_invariant = ⋃ (Timed_Automata.collect_clki ‘ set I)
  unfolding Timed_Automata.collect_clki_def product_invariant_def
  apply (simp add: image_Union)
  apply safe
  subgoal
    unfolding collect_clock_pairs_def by fastforce
  apply clarsimp
  subgoal premises prems for a b aa ba x
  proof −
    let ?L = map (λ _. x) [0..<length N]
    let ?x = collect_clock_pairs
      (concat
      (map (λp. if p < length N then (I ! p) (?L ! p) else [])
      [0..<length ?L]))
    show ?thesis
      apply (intro exI[where x = ?x] conjI)
       apply (rule exI; rule HOL.refl)
      apply simp
      using prems unfolding collect_clock_pairs_def by (fastforce dest: mem_nth)
  qed
  done

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

lemma collect_clkt_product_trans_subs:
  Timed_Automata.collect_clkt product_trans ⊆ ⋃ (Timed_Automata.collect_clkt ‘ set T)
  unfolding
  Timed_Automata.collect_clkt_def product_trans_def product_trans_i_def product_trans_s_def
  by (fastforce dest: collect_clock_pairs_append_cases states_length)

lemma collect_clkvt_product_trans_subs:
  collect_clkvt product_trans ⊆ ⋃ (collect_clkvt ‘ set T)
  unfolding collect_clkvt_def product_trans_def product_trans_i_def product_trans_s_def
  by (fastforce dest: states_length)

lemma statesI_aux:
  fixes L
  assumes L: L ∈ states
  assumes
    p < length T
    (l, g, a, r, l′) ∈ T ! p
  shows (L[p := l]) ∈ states
  unfolding states_def apply clarsimp
  using L apply (simp add: states_def)
  apply safe
  apply (subst list_update_nth_split)
  using assms by (fastforce simp: states_def)+

lemma product_trans_s_alt_def:
  product_trans_s =
    {(L, g, (a, Syn b1 b2), r, L′) | L g a b1 b2 r L′. ∃ p q g1 g2 r1 r2 l1′ l2′.
```

```
    L ∈ states ∧ p < length L ∧ q < length L ∧ p ≠ q ∧
    g = g1 @ g2 ∧ r = r1 @ r2 ∧ L' = L[p:=l1', q:=l2']
      ∧ (L!p, g1, (In a, b1), r1, l1') ∈ T!p ∧ (L!q, g2, (Out a, b2), r2, l2') ∈ T!q
    }
  unfolding product_trans_s_def by (safe; metis)

 context
  assumes states_not_empty: states ≠ {}
  assumes trans_complete:
   ∀ p < length T. ∀ t1 ∈ T ! p. case t1 of (l1, g1, (In a, b1), r1, l1')
   ⇒ ∃ q < length T. p ≠ q ∧ (∃ l2 g2 b2 r2 l2'. (l2, g2, (Out a, b2), r2, l2') ∈ T ! q) |
   (l1, g1, (Out a, b1), r1, l1')
   ⇒ ∃ q < length T. p ≠ q ∧ (∃ l2 g2 b2 r2 l2'. (l2, g2, (In a, b2), r2, l2') ∈ T ! q) | _ ⇒
True
 begin

 lemma collect_clkt_product_trans_sups:
  Timed_Automata.collect_clkt product_trans ⊇ ⋃ (Timed_Automata.collect_clkt ‘ set T)
 proof
  fix x assume x ∈ ⋃ (Timed_Automata.collect_clkt ‘ set T)
  then obtain trans l1 g1 a' b1 r1 l1' where x:
   trans ∈ set T (l1, g1, (a', b1), r1, l1') ∈ trans x ∈ collect_clock_pairs g1
   unfolding Timed_Automata.collect_clkt_def by force
  then obtain p where p:
    p < length T T ! p = trans
   by (auto dest!: mem_nth)
  from states_not_empty obtain L where L: L ∈ states by auto
  have length T = length L by (auto simp: states_length[OF ‹L ∈ states›])
  show x ∈ Timed_Automata.collect_clkt product_trans
  proof (cases a')
   case a': (In a)
   with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
    q < length T (l2, g2, (Out a, b2), r2, l2') ∈ T ! q p ≠ q
    by atomize_elim fastforce
   have L[p := l1, q := l2] ∈ states (is ?L ∈ _)
    using L p(1) x(1,2) trans2 unfolding p(2)[symmetric] by (auto intro!: statesI_aux)
   moreover have ?L ! p = l1 ?L ! q = l2
    using ‹p ≠ q› ‹p < length T› ‹q < length T› ‹L ∈ states› by (auto dest: states_length)
   moreover note t = calculation
   have (?L, g1 @ g2, (a, Syn b1 b2), r1 @ r2, ?L[p := l1', q := l2']) ∈ product_trans_s
    unfolding product_trans_s_alt_def
    apply clarsimp
    using t p(1) x(1,2) trans2 unfolding p(2)[symmetric] a' ‹length T = length L›
    by metis
   moreover have x ∈ collect_clock_pairs (g1 @ g2)
    using x(3) by (auto simp: collect_clock_pairs_def)
   ultimately show ?thesis unfolding Timed_Automata.collect_clkt_def product_trans_def
by force
  next
   case a': (Out a)
   with x p trans_complete obtain q l2 g2 b2 r2 l2' where trans2:
    q < length T (l2, g2, (In a, b2), r2, l2') ∈ T ! q p ≠ q
    by atomize_elim fastforce
   have L[q := l2, p := l1] ∈ states (is ?L ∈ _)
```

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

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

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

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

  lemma collect_clkt_product_trans:
    Timed_Automata.collect_clkt product_trans = ⋃ (Timed_Automata.collect_clkt ‘ set T)
    using collect_clkt_product_trans_sups collect_clkt_product_trans_subs by simp

  lemma collect_clkvt_product_trans:
    collect_clkvt product_trans = ⋃ (collect_clkvt ‘ set T)
  using collect_clkvt_product_trans_sups collect_clkvt_product_trans_subs by simp

  end

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

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

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

lemma finite_product_trans_i:
  finite product_trans_i
proof −
  let ?N = ⋃ (trans_of ‘ set N)
  let ?S = {(L, p, g, (a, Act b), r, l′)|L p g a b r l′.
    L ∈ states ∧ (L ! p, g, (Sil a, b), r, l′) ∈ map trans_of N ! p ∧ p < length L}
  let ?R = {(L, p, g, (a, Act b), r, l′)|L p g a b r l′.
    L ∈ states ∧ p < length L ∧ (g, (Sil a, b), r, l′) ∈ snd ‘ ?N}
  have ?S ⊆ ?R by (fastforce simp: states_length dest: nth_mem)
  moreover have finite ?R using finite_states [[simproc add: finite_Collect]]
    apply simp
    apply (rule finite_imageI)
    apply (rule finite_SigmaI)
     apply (rule finite_subset[where B = {(p, L). p < length L ∧ L ∈ states}])
      apply force
    using states_length
     apply −
     apply (rule finite_subset[where B = {(p, L). p < length N ∧ L ∈ states}]; fastforce)
    using finite_trans
    apply (intro finite_vimageI)
    unfolding inj_on_def by auto
  ultimately have finite ?S by (rule finite_subset)
  moreover have product_trans_i = (λ (L, p, g, a, r, l′). (L, g, a, r, L[p := l′])) ‘ ?S
    unfolding product_trans_i_def by (auto 4 3 simp: image_Collect)
  ultimately show ?thesis by auto
qed

lemma finite_Collect_bounded_ex_5′ [simp]:
assumes finite {(a,b,c,d,e) . P a b c d e}
shows
  finite {x. ∃ a b c d e. P a b c d e ∧ Q x a b c d e}
  ⟷ (∀ a b c d e. P a b c d e ⟶ finite {x. Q x a b c d e})
using assms finite_Collect_bounded_ex[OF assms, where Q = λ x. λ (a, b, c, d, e). Q x a b
c d e]
by clarsimp

lemma finite_product_trans_s:
  finite product_trans_s
proof −
  let ?N = ⋃ (trans_of ‘ set N)
  have (g1, (In a, b), r1, l1′) ∈ snd ‘ ?N if
    (L ! p, g1, (In a, b), r1, l1′) ∈ map trans_of N ! p p < length L L ∈ states
```

```
    for L p g1 a b r1 l1'
    using that by (auto simp: states_length dest: nth_mem)
  let ?S = {(L, p, q, g1, g2, (a, Syn b1 b2), r1, r2, l1', l2')|L p q g1 g2 a b1 b2 r1 r2 l1' l2'.
    L ∈ states ∧ (L ! p, g1, (In a, b1), r1, l1') ∈ map trans_of N ! p ∧
      (L ! q, g2, (Out a, b2), r2, l2') ∈ map trans_of N ! q ∧ p < length L ∧ q < length L}
  define F1 where F1 ≡ {(g1, a, b, r1, l1') | g1 a b r1 l1'. (g1, (In a, b), r1, l1') ∈ snd ` ?N}
  have fin1: finite F1 unfolding F1_def using finite_trans
    using [[simproc add: finite_Collect]] by (force simp: inj_on_def intro: finite_vimageI)
  define F2 where F2 ≡ {(g1, a, b, r1, l1') | g1 a b r1 l1'. (g1, (Out a, b), r1, l1') ∈ snd `
?N}
  have fin2: finite F2 unfolding F2_def using finite_trans
    using [[simproc add: finite_Collect]] by (force simp: inj_on_def intro: finite_vimageI)
  define R where R ≡ {(L, p, q, g1, g2, (a, Syn b1 b2), r1, r2, l1', l2')|L p q g1 g2 a b1 b2
r1 r2 l1' l2'.
    L ∈ states ∧ p < length L ∧ q < length L ∧ (g1, a, b1, r1, l1') ∈ F1 ∧ (g2, a, b2, r2, l2')
∈ F2}

  have R_alt_def: R = {t. ∃ L p q g1 a b1 r1 l1' g2 b2 r2 l2'.
    L ∈ states ∧ p < length L ∧ q < length L
    ∧ (g1, a, b1, r1, l1') ∈ F1 ∧ (g2, a, b2, r2, l2') ∈ F2
    ∧ t = (L, p, q, g1, g2, (a, Syn b1 b2), r1, r2, l1', l2')
    }
    unfolding R_def by auto
  have ?S ⊆ R by (fastforce simp: R_alt_def F1_def F2_def states_length dest: nth_mem)
  moreover have finite R
    unfolding R_alt_def
    using fin1 fin2 finite_states
    using [[simproc add: finite_Collect]]
    by (auto simp: inj_on_def intro: finite_vimageI intro!: finite_imageI)
  ultimately have finite ?S by (rule finite_subset)
  moreover have
    product_trans_s
    ⊆ (λ (L, p, q, g1, g2, a, r1, r2, l1', l2').
       (L, g1 @ g2, a, r1 @ r2, L[p := l1', q := l2'])) ` ?S
    unfolding product_trans_s_def by (simp add: image_Collect) blast
  ultimately show ?thesis by (auto intro: finite_subset)
qed

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

end

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

lemma concat_map_if_aux:
  assumes (m :: nat) ≥ n
  shows concat (map (λ i. if i < n then f i else []) [0..<m])
      = concat (map (λ i. if i < n then f i else []) [0..<n])
```

```
    using assms by (induction rule: dec_induct) auto

  lemma finite_invariant_of_product[folded inv_of_product]:
    assumes ∀ A ∈ set N. finite (range (inv_of A))
    shows finite (range product_invariant)
  proof −
    let ?N = ⋃ (range ‘ inv_of ‘ set N)
    let ?X = {I. set I ⊆ ?N ∧ length I ≤ length N}
    have [] ∈ ?X by auto
    have finite ?N
      using assms
      by auto
    then have finite ?X by (rule finite_lists_length_le)
    moreover have range product_invariant ⊆ concat ‘ ?X
    proof
      fix x assume x ∈ range product_invariant
      then obtain L where L:
          x = concat
                (map (λp. if p < length (map inv_of N)
                          then (map inv_of N ! p) (L ! p) else [])
                  [0..<length L])
        unfolding product_invariant_def by auto
      show x ∈ concat ‘ ?X
      proof (cases length L ≤ length N)
        case True
        then show ?thesis using L by (fastforce dest: nth_mem)
      next
        case False
        then have x = concat
                (map (λp. if p < length (map inv_of N)
                          then (map inv_of N ! p) (L ! p) else [])
                  [0..<length N])
          using L by (auto intro: concat_map_if_aux)
        then show ?thesis by (fastforce dest: nth_mem)
      qed
    qed
    ultimately show ?thesis by − (drule finite_subset; auto)
  qed

end

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

  lemma product_invariantD:
    assumes u ⊢ product_invariant L p < length N length L = length N
    shows u ⊢ inv_of (N ! p) (L!p)
  using assms unfolding product_invariant_def by (force intro: concat_guard)

  lemma states_step:
    L′ ∈ states if N ⊢_N ⟨L, u⟩ →_a ⟨L′, u′⟩ L ∈ states
    using that unfolding states_def apply safe
       apply simp
```

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

 lemma states_steps:
   L' ∈ states if N ⊢_N ⟨L, u⟩ →* ⟨L', u'⟩ L ∈ states
   using that apply (induction N ≡ N _ _ _ _ rule: steps_n.induct)
    apply assumption
   apply simp
   by (rule states_step; assumption)

end

lemma steps_altI:
 A ⊢ ⟨l, Z⟩ →* ⟨l'', Z''⟩ if
 A ⊢ ⟨l, Z⟩ →* ⟨l', Z'⟩ A ⊢ ⟨l', Z'⟩ → ⟨l'', Z''⟩
 using that by (induction; blast)


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

 lemma
   len[simp]: length L = length N
 using states unfolding states_def by auto

 lemma product_delay_complete:
   assumes step: N ⊢_N ⟨L, u⟩ →_Del ⟨L', u'⟩
   obtains d where product_ta ⊢ ⟨L, u⟩ →^d ⟨L', u'⟩
 using step proof cases
   case prems: (step_n_t d)
```

```
  from prems have ∗:
    ∀ p∈{..<length N}. u ⊕ d ⊢ inv_of (N ! p) (L ! p)
  by (auto elim!: step_t.cases)
  from prems ∗ show ?thesis
  by (fastforce simp add: product_ta_def inv_of_def product_invariant_def[abs_def]
                intro!: guard_concat that
     )
qed

lemma product_int_complete:
  assumes step: N ⊢N ⟨L, u⟩ →Act b ⟨L′, u′⟩
  obtains a where product_ta ⊢ ⟨L, u⟩ →(a, Act b) ⟨L′, u′⟩
  using step proof cases
  case prems: (step_n_i p l g a r l′)
  from prems show ?thesis
    apply −
    apply (rule that)
    apply (rule step_a.intros[where g = g and r = r])
    by (fastforce
        simp: product_trans_def product_trans_i_def product_invariant_def product_ta_def
        trans_of_def inv_of_def
        intro: guard_concat
        )+
qed

lemma product_sync_complete:
  assumes step: N ⊢N ⟨L, u⟩ →Syn b1 b2 ⟨L′, u′⟩
  obtains a where product_ta ⊢ ⟨L, u⟩ →(a, Syn b1 b2) ⟨L′, u′⟩
  using step proof cases
  case prems: (step_n_s p l1 g1 a r1 l1′ q l2 g2 r2 l2′)
  from prems show ?thesis
    apply −
    apply (rule that)
    apply (rule step_a.intros[where g = g1 @ g2 and a = (a, Syn b1 b2) and r = r1 @ r2])
    subgoal premises prems
      apply
        (clarsimp simp add:
          product_trans_def product_trans_s_def product_invariant_def product_ta_def
          trans_of_def inv_of_def
          )
      apply (erule allE[where x = p])
      using ‹p < _› apply simp
      apply (erule allE[where x = q])
      using prems by (fastforce simp: trans_of_def)
    by (fastforce
          simp: product_invariant_def product_ta_def inv_of_def
          intro: guard_concat guard_append
       )+
qed

lemma product_complete:
  assumes step: N ⊢N ⟨L, u⟩ →b ⟨L′, u′⟩
  shows product_ta ⊢ ⟨L, u⟩ → ⟨L′, u′⟩
  using step
```

by (*cases b*) (*rule product_delay_complete product_int_complete product_sync_complete*, *simp*, *blast*)+

lemma *product_ta_cases*:
assumes *product_ta* ⊢ $L \longrightarrow^{g,a,r} L'$
shows $(L, g, a, r, L') \in$ *product_trans_i* ∨ $(L, g, a, r, L') \in$ *product_trans_s*
using *assms* **unfolding** *product_ta_def trans_of_def product_trans_def* **by** *auto*

lemma *product_delay_sound*:
assumes *step*: *product_ta* ⊢ $\langle L, u\rangle \rightarrow^{d} \langle L', u'\rangle$
shows $N \vdash_N \langle L, u\rangle \rightarrow_{Del} \langle L', u'\rangle$
using *assms* **by** *cases* (*force intro*!: *step_n_t product_invariantD len*)

lemma *product_action_sound*:
assumes *step*: *product_ta* ⊢ $\langle L, u\rangle \rightarrow_{(a, b)} \langle L', u'\rangle$
shows $N \vdash_N \langle L, u\rangle \rightarrow_b \langle L', u'\rangle$
using *assms*
apply *cases*
apply *simp*
apply (*drule product_ta_cases*)
apply (*erule disjE*)
unfolding *product_trans_i_def product_trans_s_def*
apply (*solves* ‹*auto intro*!: *product_invariantD step_n_i*›)
by (*auto intro*!: *product_invariantD step_n_s simp*: *clock_val_def*)

lemma *product_sound*:
assumes *step*: *product_ta* ⊢ $\langle L, u\rangle \rightarrow \langle L', u'\rangle$
obtains *a* **where** $N \vdash_N \langle L, u\rangle \rightarrow_a \langle L', u'\rangle$
using *step* **by** *cases* (*force dest*!: *product_action_sound product_delay_sound*)+

lemma *states_product_step*[*intro*]:
$L' \in$ *states* **if** *product_ta* ⊢ $\langle L, u\rangle \rightarrow \langle L', u'\rangle$
by (*auto intro*: *product_sound that elim*!: *states_step*)

lemma *product_steps_sound*:
$N \vdash_N \langle L, u\rangle \rightarrow* \langle L', u'\rangle$ **if** *product_ta* ⊢ $\langle L, u\rangle \rightarrow* \langle L', u'\rangle$
using *that states* **proof** (*induction A* ≡ *product_ta _ _ _ _ rule*: *steps.induct*)
case (*refl l u*)
then show *?case* **by** *blast*
next
case *prems*: (*step l u l' u' l'' u''*)
interpret *interp*: *Product_TA N l* **by** *standard* (*rule prems*)
from *prems* **have** ∗: $N \vdash_N \langle l', u'\rangle \rightarrow* \langle l'',u''\rangle$ **by** *auto*
from *prems* **show** *?case* **by** − (*erule interp.product_sound*, *rule stepI2*, *rule* ∗)
qed

lemma *product_steps_complete*:
product_ta ⊢ $\langle L, u\rangle \rightarrow* \langle L', u'\rangle$ **if** $N \vdash_N \langle L, u\rangle \rightarrow* \langle L', u'\rangle$
using *that states* **proof** (*induction N* ≡ *N _ _ _ _ rule*: *steps_n.induct*)
case (*refl l Z*)
then show *?case* **by** *blast*
next
case *prems*: (*step l Z l' Z' _ l'' Z''*)

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

lemma *product_correct*:
product_ta ⊢ ⟨*L*, *u*⟩ →* ⟨*L'*, *u'*⟩ ⟷ *N* ⊢$_N$ ⟨*L*, *u*⟩ →* ⟨*L'*, *u'*⟩
by (*metis product_steps_complete product_steps_sound*)

end

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

unbundle *no_library_syntax*

2 Networks of Timed Automata With Discrete State

2.1 Syntax and Operational Semantics

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

type_synonym
('*a*, '*c*, '*time*, '*s*, '*st*) *transition* =
'*s* * ('*st* ⇒ ('*c*, '*time*) *cconstraint*) * '*a* * ('*st* ⇒ '*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* × ('*st* ⇒ *bool*) × ('*st* ⇒ '*st option*), '*c*, '*t*, '*s*, '*st*) *sta list* × ('*s* ⇒ '*st* ⇒ *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* ⇒ '*s list* ⇒ '*st* ⇒ ('*c*, ('*t*::*time*)) *cval* ⇒ '*a label*
⇒ '*s list* ⇒ '*st* ⇒ ('*c*, '*t*) *cval* ⇒ *bool*
(_ ⊢ ⟨_, _, _⟩ →_ ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*] *61*)
where
step_sn_t:
(*N*, *I*) ⊢ ⟨*L*, *s*, *u*⟩ →$_{Del}$ ⟨*L*, *s*, *u* ⊕ *d*⟩
if ∀ *p* ∈ {..<*length N*}. *u* ⊕ *d* ⊢ *snd* (*N* ! *p*) (*L* ! *p*)
d ≥ *0 length N* = *length I* |

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

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

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

declare *steps_sn.intros*[*intro*]

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

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

2.2 Product Automaton

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

definition
$T_s\ p\ s = \{(l, g\ s, a, f\ s, l') \mid l\ g\ a\ f\ l'.\ (l, g, a, f, l') \in fst\ (fst\ A\ !\ p)\}$

definition
N_s s = map (λ p. (T_s p s, snd (fst A ! p))) [0..<length (fst A)]

abbreviation *P ≡ snd A*

definition *p ≡ length (fst A)*

abbreviation *product s ≡ Product_TA_Defs.product_ta (N_s s)*

abbreviation *T' s ≡ trans_of (product s)*
abbreviation *I' s ≡ inv_of (product s)*

definition
prod_trans_i =
{((L, s), g, a, r, (L', s')) | L s g c a r m L' s'.
(∀ q < p. (P ! q) (L ! q) s) ∧ (∀ q < p. (P ! q) (L' ! q) s')
∧ (L, g, (a, Networks.label.Act (c, m)), r, L') ∈ T' s ∧ c s ∧ Some s' = m s}

definition
prod_trans_s =
{((L, s), g, a, r, (L', s')) | L s g ci co a r mi mo L' s' so.
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') ∈ T' s
∧ Some so = mo s
∧ Some s' = mi so
}

definition
prod_trans ≡ prod_trans_i ∪ prod_trans_s

definition
prod_invariant ≡ λ (L, s). I' s L

definition *prod_ta :: ('a, 'c, 't, 's list × 'st) ta* **where**
prod_ta ≡ (prod_trans, prod_invariant)

lemma *prod_ta_cases*:
assumes *prod_ta ⊢ L ⟶^{g,a,r} L'*
shows *(L, g, a, r, L') ∈ prod_trans_i ∨ (L, g, a, r, L') ∈ prod_trans_s*
using *assms* **unfolding** *prod_ta_def trans_of_def prod_trans_def* **by** *auto*

lemma *inv_of_simp*:
inv_of prod_ta (L, s) = I' s L
unfolding *inv_of_def prod_ta_def prod_invariant_def* **by** *simp*

lemma *I'_simp*:
I' s L = I' s' L
unfolding *Product_TA_Defs.product_ta_def inv_of_def Product_TA_Defs.product_invariant_def N_s_def*
apply *simp*
apply *(rule arg_cong[***where** *f = concat])*
by *simp*

lemma *collect_clki_prod_invariant*:
Timed_Automata.collect_clki prod_invariant = Timed_Automata.collect_clki $(I'\ s)$
unfolding *prod_invariant_def Timed_Automata.collect_clki_def*
apply (*simp split*: *prod.split*)
apply *safe*
apply (*subst* (*asm*) *I'_simp*[**where** $s' = s$])
by *auto*

lemma *collect_clki_prod_invariant'*:
Timed_Automata.collect_clki prod_invariant
$\subseteq \bigcup \{$*Timed_Automata.collect_clki* (*snd* (*fst A* ! *p*)) | *p*. $p <$ *length* (*fst A*)$\}$
unfolding *collect_clki_prod_invariant*[*of s*]
unfolding *inv_of_def Product_TA_Defs.product_ta_def*
unfolding *Product_TA_Defs.product_invariant_def*
unfolding *inv_of_def N_s_def*
unfolding *Timed_Automata.collect_clki_def*
unfolding *collect_clock_pairs_def*
by *auto*

lemma *collect_clkt_prod_trans_subs*:
Timed_Automata.collect_clkt prod_trans $\subseteq$ *Timed_Automata.collect_clkt* $(\bigcup\ (T'\ `\ UNIV))$
unfolding *Timed_Automata.collect_clkt_def prod_trans_def prod_trans_i_def prod_trans_s_def*
by *fastforce*

lemma *collect_clkvt_prod_trans_subs*:
collect_clkvt prod_trans $\subseteq$ *collect_clkvt* $(\bigcup\ (T'\ `\ UNIV))$
unfolding *collect_clkvt_def prod_trans_def prod_trans_i_def prod_trans_s_def* **by** *fastforce*

lemma *T_simp*:
T ! q = *trans_of* (N ! q) **if** $q <$ *length* N
using *that* **oops**

abbreviation $N \equiv$ *fst A*

context
fixes Q
assumes *finite_state*:
$\forall\ l.\ \forall\ q < p.\ (P\ !\ q)\ l\ s \longrightarrow Q\ s$
finite $\{s.\ Q\ s\}$
and *finite_trans*: $\forall\ A \in$ *set* N. *finite* (*fst A*)
and *p_gt_0*: $p > 0$
begin

lemma *finite_state'*:
finite $\{s.\ \forall q<p.\ (P\ !\ q)\ (L\ !\ q)\ s\}$ (**is** *finite ?S*)
proof −
from *p_gt_0* **obtain** q **where** $q < p$ **by** *blast*
then have *?S* $\subseteq \{s.\ Q\ s\}$ **using** *finite_state*(*1*) **by** *auto*
moreover have *finite* ... **by** (*rule finite_state*(*2*))
ultimately show *?thesis* **by** (*rule finite_subset*)
qed

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

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

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

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

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

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

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

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

```
      Q s ∧
      (L, g, (a, Networks.label.Act (c, m)), r, L') ∈ T' s ∧
      (∀ q < p. (P ! q) (L ! q) s) ∧ (∀ q < p. (P ! q) (L' ! q) s')
      ∧ c s ∧ Some s' = m s}

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

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

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

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

end

  abbreviation states' s ≡ Product_TA_Defs.states (N_s s)
```

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

end

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

lemma A_unfold:
  A ≡ (N, P)
  by auto

lemma network_step_delay:
  assumes step: (N, P) ⊢ ⟨L, s, u⟩ →Del ⟨L', s', u'⟩ and len: length L = p
  shows N_s s ⊢N ⟨L, u⟩ →Networks.label.Del ⟨L', u'⟩
  subgoal
    using step
    apply cases
    subgoal
      apply simp
      apply (rule step_n_t)
      subgoal
        unfolding N_s_def by (auto simp: inv_of_def)
      apply assumption
      done
    done
  done

lemma network_step_silent:
  assumes step: (N, P) ⊢ ⟨L, s, u⟩ →Act a ⟨L', s', u'⟩ and len: length L = p
  obtains a where N_s s ⊢N ⟨L, u⟩ →Networks.label.Act a ⟨L', u'⟩
  subgoal premises prems
    using step
    apply cases
    subgoal
      apply (rule prems)
      apply (rule step_n_i)
      unfolding N_s_def T_s_def by (auto 4 0 simp: trans_of_def inv_of_def len p_def)
    done
  done

lemma network_step_sync:
  assumes step: (N, P) ⊢ ⟨L, s, u⟩ →Syn a ⟨L', s', u'⟩ and len: length L = p
  obtains a b where N_s s ⊢N ⟨L, u⟩ →Networks.label.Syn a b ⟨L', u'⟩
  subgoal premises prems
    using step
    apply cases
    subgoal
      subgoal premises A
        apply (rule prems)
        apply (rule step_n_s)
                defer
```

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

lemma network_step:
  assumes step: (N, P) ⊢ ⟨L, s, u⟩ →_a ⟨L', s', u'⟩ and len: length L = p
  obtains a where N_s s ⊢_N ⟨L, u⟩ →_a ⟨L', u'⟩
  subgoal
    using step
    apply (cases a; simp)
     apply (rule that, erule network_step_delay[OF _ len, simplified])
    apply (erule network_step_silent[OF _ len, simplified], erule that)
    apply (erule network_step_sync[OF _ len, simplified], erule that)
    done
  done

lemma trans_of_N_s_1:
  (fst ‘ trans_of (N_s s ! q)) = fst ‘ fst (N ! q) if q < p
  using that unfolding trans_of_def N_s_def p_def T_s_def by (auto 0 7 simp: image_iff)

lemma trans_of_N_s_2:
  ((snd o snd o snd o snd) ‘ trans_of (N_s s ! q)) = (snd o snd o snd o snd) ‘ fst (N ! q) if q <
p
  using that unfolding trans_of_def N_s_def p_def T_s_def by force

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

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

  lemma states_step:
    L' ∈ states' s if A ⊢ ⟨L, s, u⟩ →_a ⟨L', s', u'⟩ L ∈ states' s
  proof −
    interpret Product_TA_Defs' N_s s .
    from ‹L ∈ _› have L ∈ states .
    from ‹L ∈ _› have length L = p by (simp add: N_s_length states_length)
    with network_step[folded A_unfold, OF that(1)] obtain a where
      N_s s ⊢_N ⟨L, u⟩ →_a ⟨L',u'⟩
      by auto
    then show ?thesis using that(2) by (rule states_step)
```

```
  qed

  lemma states_steps:
    L' ∈ states' s' if A ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩ L ∈ states' s
    using that proof (induction A ≡ A _ _ _ _ _ _ rule: steps_sn.induct)
    case (refl L s u)
    then show ?case by assumption
  next
    case (step L s u L' s' u' L'' s'' u'')
    with states_step[of L' s' u' L'' s'' u''] states'_simp show ?case by blast
  qed

  lemma inv_step:
    ∀ p<length P. (P ! p) (L' ! p) s' if
    A ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩ ∀ p<length P. (P ! p) (L ! p) s
    using that by (cases) auto

  lemma inv_steps:
    ∀ p<length P. (P ! p) (L' ! p) s' if
    A ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩ ∀ p<length P. (P ! p) (L ! p) s
    using that by (induction A ≡ A _ _ _ _ _ _ rule: steps_sn.induct) (auto dest: inv_step)

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 by standard rule

  lemma inv_prod_simp:
    inv_of prod_ta (l, s') = Product_TA_Defs.product_invariant (N_s s') l if length l = p
   unfolding prod_ta_def prod_invariant_def Product_TA_Defs.product_ta_def N_s_def inv_of_def
    using that by (simp add: p_def)

  lemma inv_of_N_simp:
    map inv_of (N_s s') ! q = I ! q if q < p
    using that unfolding inv_of_def N_s_def p_def by simp

  lemma product_inv_prod_simp:
    inv_of prod_ta (l, s') = I' s l if length l = p
    using that
    apply (simp add: inv_prod_simp)
    apply (simp add: N_s_length inv_of_def Product_TA_Defs.product_invariant_def)
    apply (rule arg_cong[where f = concat])
    apply (clarsimp cong: map_cong)
    by (subst inv_of_N_simp; simp)

  lemma product_inv_prod[intro]:
```

```
    u ⊢ inv_of prod_ta (l, s') if u ⊢ inv_of product_ta l length l = p
    using that by (simp add: product_inv_prod_simp)

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

  lemma length_L[intro]:
    length L = p
    by (simp add: N_s_length)

  lemma prod_complete_delay:
    assumes step: A ⊢ ⟨L, s, u⟩ →Del ⟨L', s', u'⟩
    obtains d where prod_ta ⊢ ⟨(L, s), u⟩ →^d ⟨(L', s'), u'⟩
  using step proof cases
    case prems: (step_sn_t N d P)
    note [simp] = A_simp[OF prems(1)]
    from prems have N_s s ⊢N ⟨L, u⟩ →Networks.Del ⟨L', u'⟩
      unfolding N_s_def by (auto 4 3 simp: inv_of_def intro: step_n_t)
    with prems show ?thesis
      by (auto 4 4
          intro: that
          simp: product_inv_prod_simp[OF length_L]
          elim!: product_delay_complete step_t.cases)
  qed

  lemma prod_complete_silent:
    assumes step: A ⊢ ⟨L, s, u⟩ →Act a ⟨L', s', u'⟩
    obtains a where prod_ta ⊢ ⟨(L, s), u⟩ →a ⟨(L', s'), u'⟩
  using step proof cases
    case prems: (step_sn_i l g c m f l' N q r I)
    note [simp] = A_simp[OF prems(1)]
    from prems(13) have [simp]: length P = p by (simp add: p_def)
    have N_s s ⊢N ⟨L, u⟩ →Networks.label.Act (c, m) ⟨L', u'⟩
      apply (rule step_n_i)
       using prems unfolding N_s_def T_s_def by (auto 3 0 simp: trans_of_def inv_of_def
N_s_length)
    with ‹length P = p› obtain b where
      product_ta ⊢ ⟨L, u⟩ →(b, Networks.label.Act (c, m)) ⟨L', u'⟩
      by (clarsimp elim!: product_int_complete)
    with prems inv obtain g r where step:
      ((L, s), g, b, r, (L', s')) ∈ prod_trans_i
      u ⊢ g [r→0]u = u' u' ⊢ inv_of product_ta L'
       apply atomize_elim
      unfolding prod_trans_i_def by − (erule step_a.cases; auto)
    then have ((L, s), g, b, r, (L', s')) ∈ trans_of prod_ta
      by (simp add: prod_trans_def trans_of_def prod_ta_def)
    moreover have length L' = p
      using length_L prems(8) by auto
    ultimately show ?thesis
      apply −
      apply (rule that)
      apply rule
```

```
    using step(2−) by force+
qed

lemma prod_complete_sync:
  assumes step: A ⊢ ⟨L, s, u⟩ →Syn a ⟨L′, s′, u′⟩
  obtains a where prod_ta ⊢ ⟨(L, s), u⟩ →a ⟨(L′, s′), u′⟩
using step proof cases
  case prems: (step_sn_s l1 g1 ci mi f1 l1′ N q1 l2 g2 co mo f2 l2′ q2 r1 r2 I so)
  note [simp] = A_simp[OF prems(1)]
  from prems(21) have [simp]: length P = p by (simp add: p_def)

  have N_s s ⊢N ⟨L, u⟩ →Networks.label.Syn (ci, mi) (co, mo) ⟨L′, u′⟩
    apply (rule step_n_s)
            defer
            defer
            apply (rule prems; fail)
           apply (rule prems(5); fail)
    subgoal
      using prems unfolding N_s_def by (auto simp: inv_of_def)
          defer
          defer
          apply (rule prems; fail)
         apply (rule prems(12); fail)
       using prems unfolding N_s_def T_s_def by (auto 3 0 simp: trans_of_def p_def
N_s_length)
  with ‹length P = p› obtain a where
    product_ta ⊢ ⟨L, u⟩ →(a, Networks.label.Syn (ci, mi) (co, mo)) ⟨L′, u′⟩
    by (auto elim!: product_sync_complete)
  with prems inv obtain g r where step:
      ((L, s), g, a, r, (L′, s′)) ∈ prod_trans_s
      u ⊢ g [r→0]u = u′ u′ ⊢ inv_of product_ta L′
      apply atomize_elim
    unfolding prod_trans_s_def by − (erule step_a.cases; auto; blast)
  then have ((L, s), g, a, r, (L′, s′)) ∈ trans_of prod_ta
    by (simp add: prod_trans_def trans_of_def prod_ta_def)
  moreover have length L′ = p
    using length_L ‹L′ = _› by auto
  ultimately show ?thesis
    apply −
    apply (rule that)
    apply rule
    using step(2−) by force+
qed

lemma prod_complete:
  assumes step: A ⊢ ⟨L, s, u⟩ →a ⟨L′, s′, u′⟩
  shows prod_ta ⊢ ⟨(L, s), u⟩ → ⟨(L′, s′), u′⟩
  using step
 by (cases a; simp; blast elim!: prod_complete_delay prod_complete_silent prod_complete_sync)

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

lemma *prod_sound'_delay*:
assumes *step*: *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →^{d} ⟨(*L'*, *s'*), *u'*⟩
shows *A* ⊢ ⟨*L*, *s*, *u*⟩ →$_{Del}$ ⟨*L'*, *s'*, *u'*⟩ ∧ *product_ta* ⊢ ⟨*L*, *u*⟩ → ⟨*L'*, *u'*⟩
∧ (∀ *p*<*length P*. (*P* ! *p*) (*L'* ! *p*) *s'*)
using *assms* **proof** *cases*
case *prems*: *1*
then have *product_ta* ⊢ ⟨*L*, *u*⟩ →^{d} ⟨*L'*, *u'*⟩ **unfolding** *inv_of_simp* **by** *fast*
moreover from *product_delay_sound*[*OF this*] *prems*(*1*−*3*) **have** *A* ⊢ ⟨*L*, *s*, *u*⟩ →$_{Del}$ ⟨*L'*, *s'*, *u'*⟩
apply *simp*
apply (*subst A_unfold*)
apply (*rule step_sn_t*)
by (*auto simp*: *N_s_def inv_of_def step_t.simps N_s_length p_def Len intro*: ‹*0* ≤ *d*›)
ultimately show *?thesis* **using** *prems inv* **by** *fast*
qed

lemma *prod_sound'_action*:
assumes *step*: *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →$_a$ ⟨(*L'*, *s'*), *u'*⟩
obtains *a* **where** (*A* ⊢ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩ ∧ *a* ≠ *Del*) ∧ *product_ta* ⊢ ⟨*L*, *u*⟩ → ⟨*L'*, *u'*⟩
∧ (∀ *p*<*length P*. (*P* ! *p*) (*L'* ! *p*) *s'*)
using *assms*
proof *cases*
case *prems*: (*1 g r*)
from *Len* **have** [*simp*]: *length P* = *p* **by** (*simp add*: *p_def*)
from *prems*(*1*)[*THEN prod_ta_cases*] **show** *?thesis*
proof (*rule disjE*, *goal_cases*)
case *1*
then obtain *c m* **where** ∗:
Some s' = *m s* ∀ *q*<*p*. (*P* ! *q*) (*L* ! *q*) *s* ∀ *q*<*p*. (*P* ! *q*) (*L'* ! *q*) *s'*
product_ta ⊢ *L* ⟶$^{g,(a,\ Networks.label.Act\ (c,\ m)),r}$ *L' c s*
unfolding *prod_trans_i_def* **by** *auto*
with *prems* **have** *product_ta* ⊢ ⟨*L*, *u*⟩ →$_{(a,\ Networks.label.Act\ (c,\ m))}$ ⟨*L'*, *u'*⟩
unfolding *inv_of_simp* **by** (*metis I'_simp step_a.intros*)
moreover from *product_action_sound*[*OF this*] *prems*(*3*−*4*) **obtain** *a* **where**
A ⊢ ⟨*L*, *s*, *u*⟩ →$_{Act\ a}$ ⟨*L'*, *s'*, *u'*⟩
apply *safe*
apply (*simp add*: *N_s_def trans_of_def N_s_length T_s_def*)
apply (*simp only*: *ex_simps*[*symmetric*])
apply (*erule exE*, *erule exE*)
apply (*rule that*)
apply (*subst A_unfold*)
apply (*rule step_sn_i*)
apply *fast*
using ∗(*3*) **by** (*auto simp*: *N_s_def inv_of_def p_def* ‹*Some s'* = *m s*› *intro*: ‹*c s*›)
ultimately show *?thesis* **using** ∗ **by** (*auto intro*: *that*)
next
case *2*
then obtain *ci co mi mo si* **where** ∗:
Some s' = *mi si Some si* = *mo s* ∀ *q*<*p*. (*P* ! *q*) (*L* ! *q*) *s* ∀ *q*<*p*. (*P* ! *q*) (*L'* ! *q*) *s'*
product_ta ⊢ *L* ⟶$^{g,(a,\ Networks.label.Syn\ (ci,\ mi)\ (co,\ mo)),r}$ *L'*
ci s co s
unfolding *prod_trans_s_def* **by** *auto*

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

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

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

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

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

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

```
    by (metis Prod_TA_Defs'.states'_simp interp.states_prod_step interp.inv_prod_step)
  show ?case
    using ∗ prems by (auto simp: ‹l' = _› intro: interp.prod_sound stepI2)
qed

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

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

end

end
theory UPPAAL_Asm
  imports Main
begin

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

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

type_synonym stack = int list
type_synonym flag = bool
```

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

definition int_of :: bool ⇒ int where
  int_of x ≡ if x then 1 else 0

fun step :: instr ⇒ state ⇒ 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, ¬ f, rs) |
  step AND (pc, b # st, m, f, rs) =
    (if b = 0 ∨ b = 1
     then Some (pc + 1, st, m, b = 1 ∧ 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 ≤ 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 ≥ 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 ≥ 0 then Some (nat q, int pc # st, m, f, rs) else None) |
  step RETURN (pc, q # st, m, f, rs) =
    (if q ≥ 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 ⇒ instr option
type_synonym fuel = nat

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

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

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

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

lemmas [intro] = steps.intros

lemma exec_steps:
  assumes exec prog n s pcs = Some ((pc, st, m, f, rs), pcs')
  shows steps prog n s (pc, st, m, f, rs) ∧ prog pc = Some HALT
using assms proof (induction P ≡ prog n s pcs arbitrary: pc st m f rs rule: exec.induct)
  case 1
  then show ?case by simp
next
  case (2 n pc' st' m' f' rs' pcs')
  then obtain instr where prog pc' = Some instr by (cases prog pc') auto
  show ?case
  proof (cases instr = HALT)
    case True
    with 2.prems ‹prog pc' = _› show ?thesis by auto
  next
    case False
    with 2 ‹prog pc' = _› show ?thesis by (auto split: option.split_asm)
  qed
qed

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

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

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

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

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

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

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

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

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

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

type_synonym *'t programc* = *addr* ⇒ *'t instrc option*

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

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

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

definition *stepst prog n u start* ≡ λ (*pc*, *st*, *m*, *f*, *rs*).
stepsc prog n u start (pc, st, m, f, rs) ∧ *prog pc = Some (INSTR HALT)*

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

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

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

end
theory *UPPAAL_State_Networks*
imports *Munta_Model_Checker.State_Networks Timed_Automata.Normalized_Zone_Semantics UPPAAL_Asm_Clocks*
AutoCorres2.Subgoals
begin

3 Networks of Timed Automata – UPPAAL Style

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

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

lemma *finite_lists_boundedI*:
assumes ∀ *i* < *r*. *finite* (*S i*)
shows *finite* {*s*. *length s = r* ∧ (∀ *i*<*r*. *s* ! *i* ∈ *S i*)} (**is** *finite ?R*)
proof −
let *?S* = ⋃ {*S i* | *i*. *i* < *r*}
have *?R* ⊆ {*s*. *set s* ⊆ *?S* ∧ *length s = r*}
by (*auto dest*!: *mem_nth*)

moreover have *finite* ... **by** (*rule finite_lists_length_eq*) (*use assms* **in** *auto*)
ultimately show *?thesis* **by** (*rule finite_subset*)
qed

3.1 Syntax and Operational Semantics

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

type_synonym
('*c*, '*time*, '*s*) *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
bounded bounds s ≡
length s = *length bounds* ∧ (∀ *i* < *length s*. *fst* (*bounds* ! *i*) < *s* ! *i* ∧ *s* ! *i* < *snd* (*bounds* ! *i*))

inductive *step_u* ::
('*a*, '*t* :: *time*, '*s*) *unta* ⇒ *nat* ⇒ '*s list* ⇒ *int list* ⇒ (*nat*, '*t*) *cval* ⇒ '*a label*
⇒ '*s list* ⇒ *int list* ⇒ (*nat*, '*t*) *cval* ⇒ *bool*
(_ ⊢_ ⟨_, _, _⟩ →_ ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*,*61*] *61*)
where
step_u_t:
⟦
∀ *p* < *length N*. ∃ *pc st s*′ *rs*.
stepst P n (*u* ⊕ *d*) ((*I* ! *p*) (*L* ! *p*), [], *s*, *True*, []) (*pc*, *st*, *s*′, *True*, *rs*);
∀ *p* < *length N*. *u* ⊕ *d* ⊢ *snd* (*N* ! *p*) (*L* ! *p*);
d ≥ *0*;
bounded B s
⟧
⟹ (*P*, *N*, *I*, *B*) ⊢*n* ⟨*L*, *s*, *u*⟩ →*Del* ⟨*L*, *s*, *u* ⊕ *d*⟩ |
step_u_i:
⟦
stepst P n u (*pc_g*, [], *s*, *True*, []) (_, _, _, *True*, _);
stepst P n u (*pc_u*, [], *s*, *True*, []) (_, _, *s*′, _, *r*);
∀ *p* < *length N*. ∃ *pc st s rs*.
stepst P n u′ ((*I* ! *p*) (*L*′ ! *p*), [], *s*′, *True*, []) (*pc*, *st*, *s*, *True*, *rs*);
(*l*, *pc_g*, *Sil a*, *pc_u*, *l*′) ∈ *fst* (*N* ! *p*);
∀ *p* < *length N*. *u*′ ⊢ *snd* (*N* ! *p*) (*L*′ ! *p*);
L!*p* = *l*; *p* < *length L*; *L*′ = *L*[*p* := *l*′]; *u*′ = [*r*→*0*]*u*;

bounded B s′
]]
⟹ (*P*, *N*, *I*, *B*) ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_{Act\ a}$ ⟨*L′*, *s′*, *u′*⟩ |
step_u_s:
[[
stepst P n u (*pc_g1*, [], *s*, *True*, []) (_, _, _, *True*, _);
stepst P n u (*pc_g2*, [], *s*, *True*, []) (_, _, _, *True*, _);
stepst P n u (*pc_u2*, [], *s*, *True*, []) (_, _, *s1*, _, *r2*);
— UPPAAL semantics quirk
((∃ *pc st s′ f*. *stepst P n u* (*pc_u1*, [], *s*, *True*, []) (*pc*, *st*, *s′*, *f*, *r1*))
∨ (¬ (∃ *pc st s′ f r′*. *stepst P n u* (*pc_u1*, [], *s*, *True*, []) (*pc*, *st*, *s′*, *f*, *r′*)) ∧ *r1* = []));
stepst P n u (*pc_u1*, [], *s1*, *True*, []) (_, _, *s′*, _, _);
~~*stepst P n u (pc_u2, [], s1, True, [], _, _, s2, _, r2);*~~
∀ *p* < *length N*. ∃ *pc st s rs*.
stepst P n u′ ((*I* ! *p*) (*L′* ! *p*), [], *s′*, *True*, []) (*pc*, *st*, *s*, *True*, *rs*);
(*l1*, *pc_g1*, *In a*, *pc_u1*, *l1′*) ∈ *fst* (*N* ! *p*);
(*l2*, *pc_g2*, *Out a*, *pc_u2*, *l2′*) ∈ *fst* (*N* ! *q*);
∀ *p* < *length N*. *u′* ⊢ *snd* (*N* ! *p*) (*L′* ! *p*);
L!*p* = *l1*; *L*!*q* = *l2*; *p* < *length L*; *q* < *length L*; *p* ≠ *q*;
L′ = *L*[*p* := *l1′*, *q* := *l2′*]; *u′* = [(*r1* @ *r2*)→*0*]*u*;
bounded B s′
]] ⟹ (*P*, *N*, *I*, *B*) ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_{Syn\ a}$ ⟨*L′*, *s′*, *u′*⟩

inductive_cases[*elim*!]: *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L′*, *s′*, *u′*⟩

inductive *steps_un* ::
('*a*, '*t* :: *time*, '*s*) *unta* ⇒ *nat* ⇒ '*s list* ⇒ *int list* ⇒ (*nat*, '*t*) *cval*
⇒ '*s list* ⇒ *int list* ⇒ (*nat*, '*t*) *cval* ⇒ *bool*
(_ ⊢_ ⟨_, _, _⟩ →* ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*,*61*] *61*)
where
refl: *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →* ⟨*L*, *s*, *u*⟩ |
step: *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →* ⟨*L′*, *s′*, *u′*⟩ ⟹ *A* ⊢$_n$ ⟨*L′*, *s′*, *u′*⟩ →$_a$ ⟨*L″*, *s″*, *u″*⟩
⟹ *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →* ⟨*L″*, *s″*, *u″*⟩

declare *steps_un.intros*[*intro*]

lemma *stepI2*:
A ⊢$_n$ ⟨*L*, *s*, *u*⟩ →* ⟨*L″*, *s″*, *u″*⟩ **if**
A ⊢$_n$ ⟨*L′*, *s′*, *u′*⟩ →* ⟨*L″*, *s″*, *u″*⟩ *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L′*, *s′*, *u′*⟩
using *that*
apply *induction*
apply (*rule steps_un.step*)
apply (*rule refl*)
apply *assumption*
apply *simp*
by (*rule steps_un.step*; *assumption*)

3.2 Equivalent State Network Automaton

definition *stripp p* ≡ *map_option strip o p*
definition *stripfp p* ≡ *map_option stripf o p*
definition *striptp p* ≡ *map_option stript o p*

locale *Equiv_TA_Defs* =

```
  fixes A :: ('a, 't, 's) unta
    and n :: nat — Fuel
begin

abbreviation N ≡ fst (snd A)
abbreviation P ≡ fst A
abbreviation I ≡ fst (snd (snd A))
abbreviation B ≡ snd (snd (snd A))
abbreviation P' ≡ stripfp P
abbreviation PF ≡ stripfp P
abbreviation PT ≡ striptp P
definition p ≡ length N

definition make_f pc_u ≡ λ s.
  case (exec P' n (pc_u, [], s, True, []) []) of
    None ⇒ []
  | Some ((_, _, _, _, r), _) ⇒ r

definition make_mt pc_u ≡ λ s.
  case (exec PT n (pc_u, [], s, True, []) []) of
    None ⇒ None
  | Some ((_, _, s', _, r), _) ⇒ Some s'

definition make_mf pc_u ≡ λ s.
  case (exec PF n (pc_u, [], s, True, []) []) of
    None ⇒ None
  | Some ((_, _, s', _, r), _) ⇒ Some s'

definition make_c pc_g ≡ λ s.
  case (exec PT n (pc_g, [], s, True, []) []) of
    None ⇒ False
  | Some ((_, _, _, f, _), _) ⇒ f

definition make_g pc_g ≡ λ s.
  case (exec PT n (pc_g, [], s, True, []) []) of
    None ⇒ []
  | Some ((_, _, _, _, _), pcs) ⇒
      List.map_filter (λ pc.
        case P pc of
          Some (CEXP ac) ⇒ Some ac
        | _ ⇒ None
          )
        pcs

definition
  state_trans_t i ≡
    {(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') ∈ fst (N ! i)
    }
```

abbreviation *state_trans* ≡ *state_trans_t*

definition
state_pred i ≡ λ *l s*.
case (*exec P′ n* ((*I* ! *i*) *l*, [], *s*, *True*, []) []) *of*
None ⇒ *False*
| *Some* ((_, _, _, *f*, _), _) ⇒ *f* ∧ *bounded B s*

definition
state_inv i ≡ *snd* (*N* ! *i*)

definition
state_ta ≡ (*map* (λ *p*. (*state_trans p*, *state_inv p*)) [*0*..<*p*], *map state_pred* [*0*..<*p*])

sublocale *defs*: *Prod_TA_Defs state_ta* **.**

lemma *bounded_finite*:
finite {*s*. *bounded B s*} (**is** *finite ?S*)
proof −
have
?S ⊆ {*s*. *length s* = *length B* ∧ (∀ *i*<*length B*. *fst* (*B* ! *i*) < *s* ! *i* ∧ *s* ! *i* < *snd* (*B* ! *i*))}
unfolding *bounded_def* **by** *auto*
moreover have *finite* ... **unfolding** *bounded_def* **using** *finite_lists_boundedI* **by** *force*
ultimately show *finite ?S* **by** (*rule finite_subset*)
qed

lemma *finite_state*:
∀ *q* < *p*. ∀ *l*. *finite* {*s*. (*defs.P* ! *q*) *l s*}
proof *safe*
fix *q l* **assume** ‹*q* < *p*›
let *?S* = {*s*. (*defs.P* ! *q*) *l s*}
from ‹*q* < *p*› **have** *?S* ⊆ {*s*. *bounded B s*}
unfolding *state_ta_def state_pred_def* **by** (*auto split*: *option.splits*)
moreover have *finite* ... **by** (*rule bounded_finite*)
ultimately show *finite ?S* **by** (*rule finite_subset*)
qed

end

fun *is_instr* :: *'t instrc* ⇒ *bool* **where**
is_instr (*INSTR* _) = *True* |
is_instr _ = *False*

lemma *step_stripf*:
assumes
is_instr cmd
shows
stepc cmd u (*pc*, *st*, *s*, *f*, *rs*) = *step* (*stripf cmd*) (*pc*, *st*, *s*, *f*, *rs*)
proof (*cases cmd*)

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

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

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

lemma *stepsc_f_complete*:
assumes
stepsc P n′ u start end
⋀ *pc′ st s′ f′ rs cmd*.
stepsc P n′ u start (*pc′*, *st*, *s′*, *f′*, *rs*) ⟹ *P pc′* = *Some cmd*
⟹ *is_instr cmd*
shows
steps (*stripfp P*) *n′ start end*
using *assms* **proof** (*induction P* ≡ *P n′ u* ≡ *u x4* ≡ *start end arbitrary*: *start rule*: *stepsc.induct*)
case *1*
then show *?case* **unfolding** *stripfp_def* **by** *auto*
next
case (*2 cmd pc st m f rs s n′ s′*)
have *is_instr cmd* **if**
stepsc P n′ u s (*pc′*, *st*, *s′*, *f′*, *rs*) *P pc′* = *Some cmd* **for** *pc′ st s′ f′ rs cmd*
using *2*(*1*,*2*) *that* **by** (*auto intro*: *2*(*5*))
with *2*(*4*) **have** ∗: *steps* (*stripfp P*) *n′ s s′* **by** *auto*
show *?case*
proof (*cases cmd*)
case (*INSTR instr*)
with *2*(*1*) *step_stripf* **have**
step (*stripf cmd*) (*pc*, *st*, *m*, *f*, *rs*) = *Some s*
by (*auto split*: *option.split_asm*)
with *2*(*1*−*3*) *2*(*5*−) ∗ **show** *?thesis* **unfolding** *stripfp_def* **by** *auto*
next
case (*CEXP ac*)
with *2* **show** *?thesis* **by** *fastforce*
qed
qed

lemma *stepsc_f_sound*:
assumes
steps (*stripfp P*) *n′ start end*
⋀ *pc′ st s′ f′ rs cmd*.
stepsc P n′ u start (*pc′*, *st*, *s′*, *f′*, *rs*) ⟹ *P pc′* = *Some cmd*
⟹ *is_instr cmd*
shows
stepsc P n′ u start end
using *assms* **proof** (*induction stripfp P n′ start end*)
case (*1 n start*)
then show *?case* **by** *auto*
next
case (*2 instr pc st m f rs s n s′*)
from *2*(*2*) **obtain** *cmd* **where** *P pc* = *Some cmd* **unfolding** *stripfp_def* **by** *auto*
show *?case*
proof (*cases cmd*)
case *prems*: (*INSTR instr*)
with ‹*P pc* = _› *2*(*2*) *step_stripf*[*of cmd*, *symmetric*] *2*(*1*) **have** *step*:
stepc cmd u (*pc*, *st*, *m*, *f*, *rs*) = *Some s*
unfolding *stripfp_def* **by** *auto*
with ‹*P pc* = _› **have** *is_instr cmd* **if**
stepsc P n u s (*pc′*, *st*, *s′*, *f′*, *rs*) *P pc′* = *Some cmd* **for** *pc′ st s′ f′ rs cmd*
using *that* **unfolding** *stripfp_def* **by** (*force intro*: *2*(*5*))
with *2*(*4*) **have** *stepsc P n u s s′* **by** *auto*
with *step* ‹*P pc* = _› **show** *?thesis* **unfolding** *stripfp_def* **by** *auto*
next
case *prems*: (*CEXP ac*)
from ‹*P pc* = _› *2*(*5*) **have** *is_instr cmd* **by** *blast*
with *prems* **show** *?thesis* **by** *auto*
qed
qed

definition
time_indep P n start ≡
∀ *pc′ st s′ f′ rs cmd u*.
stepsc P n u start (*pc′*, *st*, *s′*, *f′*, *rs*) ∧ *P pc′* = *Some cmd*
⟶ *is_instr cmd*

lemma *stepsc_t_complete*:
assumes
stepsc P n′ u start end
⋀ *pc′ st s′ f′ rs ac*.
stepsc P n′ u start (*pc′*, *st*, *s′*, *f′*, *rs*) ⟹ *P pc′* = *Some* (*CEXP ac*) ⟹ *u* ⊢$_a$ *ac*
shows
steps (*striptp P*) *n′ start end*
using *assms* **proof** (*induction P* ≡ *P n′ u* ≡ *u x4* ≡ *start end arbitrary*: *start rule*: *stepsc.induct*)
case *1*
then show *?case* **unfolding** *stripfp_def* **by** *auto*
next
case (*2 cmd pc st m f rs s n′ s′*)
have *u* ⊢$_a$ *ac* **if**
stepsc P n′ u s (*pc′*, *st*, *s′*, *f′*, *rs*) *P pc′* = *Some* (*CEXP ac*) **for** *pc′ st s′ f′ rs ac*
using *2*(*1*,*2*) *that* **by** (*auto intro*: *2*(*5*))

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

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

lemma *stepsc_t_visitedc*:
assumes
stepsc P n′ u start end
⋀ *pc′ st s′ f′ rs.*
stepsc P n′ u start (*pc′*, *st*, *s′*, *f′*, *rs*) ⟹ *Q pc′*
shows ∃ *pcs. visitedc P n′ u start end pcs*
∧ (∀ *pc* ∈ *set pcs. Q pc*)
using *assms* **by** (*induction*) (*fastforce intro*!: *visitedc.intros*)+

lemma *visitedc_t_visited*:
assumes
visitedc P n′ u start end pcs
⋀ *pc′ ac. pc′* ∈ *set pcs* ⟹ *P pc′* = *Some* (*CEXP ac*) ⟹ *u* ⊢$_a$ *ac*
shows
visited (*striptp P*) *n′ start end pcs*
∧ (∀ *pc ac. pc′* ∈ *set pcs* ∧ *P pc′* = *Some* (*CEXP ac*) ⟶ *u* ⊢$_a$ *ac*)
using *assms*
proof (*induction P* ≡ *P n′ u* ≡ *u x4* ≡ *start end pcs arbitrary*: *start rule*: *visitedc.induct*)
case *1*
then show *?case* **by** (*auto intro*: *visited.intros*)
next
case (*2 cmd pc st m f rs s n′ s′ pcs*)
have *u* ⊢$_a$ *ac* **if**
pc′ ∈ *set pcs P pc′* = *Some* (*CEXP ac*) **for** *pc′ ac*
using *2*(*1*,*2*) *that* **by** (*auto intro*: *2*(*5*))
with *2*(*4*) **have** ∗:
visited (*striptp P*) *n′ s s′ pcs* ∀ *pc ac. pc′* ∈ *set pcs* ∧ *P pc′* = *Some* (*CEXP ac*) ⟶ *u* ⊢$_a$ *ac*
by *auto*
show *?case*
proof (*cases cmd*)
case (*INSTR instr*)
with *2*(*1*) *step_stript* **have**
step (*stript cmd*) (*pc*, *st*, *m*, *f*, *rs*) = *Some s*
by (*auto split*: *option.split_asm*)
with *2*(*1−3*) *2*(*5−*) ∗ **show** *?thesis* **unfolding** *striptp_def* **by** (*auto intro*: *visited.intros*)
next
case (*CEXP ac*)
with *2*(*1−3*) **have** *u* ⊢$_a$ *ac* **by** (*auto intro*: *2*(*5*))
with *2*(*1*) **have**
step (*stript cmd*) (*pc*, *st*, *m*, *f*, *rs*) = *Some s*
using ‹*cmd* = _› **by** *auto*
with *2*(*1−3*) *2*(*5−*) ∗ **show** *?thesis* **unfolding** *striptp_def* **by** (*auto intro*: *visited.intros*)
qed
qed

lemma *stepsc_t_sound*:
assumes
steps (*striptp P*) *n′ start end*
⋀ *pc′ st s′ f′ rs ac.*
stepsc P n′ u start (*pc′*, *st*, *s′*, *f′*, *rs*) ⟹ *P pc′* = *Some* (*CEXP ac*) ⟹ *u* ⊢$_a$ *ac*
shows
stepsc P n′ u start end
using *assms* **proof** (*induction striptp P n′ start end*)

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

lemma stepsc_visitedc:
  ∃ cc. visitedc P n u start end cc if stepsc P n u start end
  using that by induction (auto intro: visitedc.intros)

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

lemma steps_visited:
  ∃ cc. visited P n start end cc if steps P n start end
  using that by induction (auto intro: visited.intros)

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

context
  fixes P n u start
  assumes constraints_conj: ∀ pc' st s' f' rs ac.
    stepsc P n u start (pc', st, s', f', rs) ∧ P pc' = Some (CEXP ac) ⟶ u ⊢_a ac
begin

  lemma stepsc_t_sound':
```

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

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

lemma *stepsc_t_complete''*:
assumes
stepsc P n u start end
shows
∃ *pcs. visitedc P n u start end pcs* ∧ (∀ *pc* ∈ *set pcs.* ∀ *ac. P pc* = *Some* (*CEXP ac*) ⟶ *u* ⊢$_a$ *ac*)
using *assms constraints_conj* **by** (*auto intro*: *stepsc_t_complete stepsc_t_visitedc*)

lemma *stepsc_t_visited*:
assumes
stepsc P n u start end
shows
∃ *pcs. visited* (*striptp P*) *n start end pcs* ∧ (∀ *pc* ∈ *set pcs.* ∀ *ac. P pc* = *Some* (*CEXP ac*) ⟶ *u* ⊢$_a$ *ac*)
using *stepsc_t_complete''*[*OF assms*] *visitedc_t_visited* **by** *blast*

lemma *stepst_t_complete*:
assumes
stepst P n u start end
shows
∃ *pcs. exec* (*striptp P*) *n start* [] = *Some* (*end*, *pcs*) ∧ (∀ *pc* ∈ *set pcs.* ∀ *ac. P pc* = *Some* (*CEXP ac*) ⟶ *u* ⊢$_a$ *ac*)
using *assms* **by** (*auto dest*!: *stepsc_t_visited visited_exec' simp*: *striptp_def stepst_def*)

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

end

context

fixes P n $start$
assumes *time_indep*: $time_indep\ P\ n\ start$
begin

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

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

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

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

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

end

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

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

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

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

lemma *stepsc_mono*:

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

lemma *stepst_mono*:
assumes *stepst P n u start end* $n' \geq n$
shows *stepst P* n' *u start end*
using *assms stepsc_mono* **unfolding** *stepst_def* **by** *blast*

definition
state_indep P n ≡
$\forall$ *pc f pc' st f' rs u s1 s1' s2 s2' rs1 rs2*.
stepsc P n u (*pc*, *st*, *s1*, *f*, *rs*) (*pc'*, *st*, *s1'*, *f'*, *rs1*) ∧
stepsc P n u (*pc*, *st*, *s2*, *f*, *rs*) (*pc'*, *st*, *s2'*, *f'*, *rs2*)
⟶ *rs1* = *rs2*

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

lemma *steps_striptp_stepsc*:
assumes
⋀ *pc st m f rs ac*.
steps (*striptp P*) *n s'* (*pc*, *st*, *m*, *f*, *rs*) ⟹ *P pc* = *Some* (*CEXP ac*)
⟹ *u'* ⊢$_a$ *ac*

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

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

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

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


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

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

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

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

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

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

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

sublocale *prod*: *Prod_TA state_ta* **by** *standard* (*auto simp*: *inv'*)

lemma *inv_simp*:
snd (*defs.N* ! *q*) (*L'* ! *q*) = *snd* (*N* ! *q*) (*L'* ! *q*) **if** *q* < *p* **for** *L'*
using *that* **unfolding** *state_ta_def state_inv_def* **by** *simp*

lemma *defs_p_eq*[*simp*]:
defs.p = *p*
by (*simp add*: *defs.p_def p_def*)

lemma *ball_lessThan*[*simp*]:
(∀ *x* ∈ {..<*m*}. *Q x*) ⟷ (∀ *x* < *m*. *Q x*)
by *auto*

lemma *trans_state_taD*:
assumes (*l*, *g*, (*a*, *c*, *m*), *f*, *l'*) ∈ *fst* (*defs.N* ! *q*) *q* < *p*
shows
(*l*, *g*, (*a*, *c*, *m*), *f*, *l'*) ∈ *state_trans_t q*
using *assms* **unfolding** *state_ta_def* **by** *simp*

lemma *N_transD*:
assumes (*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *fst* (*N* ! *q*) *q* < *p*
shows (*l*, *make_g pc_g*, (*a*, *make_c pc_g*, *make_mf pc_u*), *make_f pc_u*, *l'*) ∈ *fst* (*defs.N* ! *q*)
using *assms* **unfolding** *state_ta_def state_trans_t_def* **by** *auto*

lemma *pred_time_indep'*:
∀ *L' s' u'*. ∀ *p'* < *p*. *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →* ⟨*L'*, *s'*, *u'*⟩
⟶ *time_indep P n* ((*I* ! *p'*) (*L'* ! *p'*), [], *s'*, *True*, [])
using *pred_time_indep* **oops**

lemma *P_steps_upd*:
assumes
Some s'' = *make_mf pc_u s'* (*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *fst* (*N* ! *q*) *q* < *p*
shows
∃ *pc st f*. *stepst P n u'* (*pc_u*, [], *s'*, *True*, []) (*pc*, *st*, *s''*, *f*, *make_f pc_u s'*)
proof −
from *assms upd_time_indep* **have** ∗: *time_indep P n* (*pc_u*, [], *s'*, *True*, []) **by** *auto*
from *assms*(*1*) ‹*q* < *p*› **obtain** *pc st f rs pcs* **where**
exec PF n (*pc_u*, [], *s'*, *True*, []) [] = *Some* ((*pc*, *st*, *s''*, *f*, *rs*), *pcs*)
unfolding *make_mf_def state_ta_def* **by** (*fastforce split*: *option.splits*)
with *stepst_f_equiv*[*OF* ∗] **show** *?thesis* **unfolding** *make_f_def* **by** *fastforce*
qed

lemma *P_steps_reset*:
assumes
q < *p* (*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *fst* (*N* ! *q*)
shows
(∃ *pc st s'' f*. *stepst P n u* (*pc_u*, [], *s'*, *True*, []) (*pc*, *st*, *s''*, *f*, *make_f pc_u s'*)) ∨
(∄ *pc st s'' f r'*. *stepst P n u* (*pc_u*, [], *s'*, *True*, []) (*pc*, *st*, *s''*, *f*, *r'*)) ∧

```
      make_f pc_u s' = []
proof (cases ∃ pc st s'' f r'. stepst P n u (pc_u, [], s', True, []) (pc, st, s'', f, r'))
  case True
  then obtain pc st s'' f r' where *:
    stepst P n u (pc_u, [], s', True, []) (pc, st, s'', f, r')
    by blast
  from assms upd_time_indep have time_indep P n (pc_u, [], s', True, []) by auto
  from * stepst_f_equiv[OF this, symmetric, where pcs = []] show ?thesis
    unfolding make_f_def by (force split: option.split)
next
  case False
  from assms upd_time_indep have time_indep P n (pc_u, [], s', True, []) by auto
  from False stepst_f_equiv[OF this, where pcs = []] show ?thesis
    unfolding make_f_def by (auto split: option.split)
qed

lemma steps_P_reset:
  assumes
    (∃ pc st s'' f. stepst P n u (pc_u, [], s', True, []) (pc, st, s'', f, r)) ∨
     (∄ pc st s'' f r'. stepst P n u (pc_u, [], s', True, []) (pc, st, s'', f, r')) ∧ r = []
    q < p (l, pc_g, a, pc_u, l') ∈ fst (N ! q)
  shows make_f pc_u s' = r
  using assms(1)
proof (safe, goal_cases)
  case prems: (1 pc st s'' f)
  from assms upd_time_indep have time_indep P n (pc_u, [], s', True, []) by auto
  from stepst_f_equiv[OF this, symmetric] prems ‹q < p› obtain pc st f pcs where
    exec PF n (pc_u, [], s', True, []) [] = Some ((pc, st, s'', f, r), pcs)
    unfolding make_f_def state_ta_def by (fastforce split: option.splits)
  then show ?case unfolding make_f_def by (auto split: option.split)
next
  case prems: 2
  have exec PF n (pc_u, [], s', True, []) [] = None
  proof (cases exec PF n (pc_u, [], s', True, []) [])
    case None
    then show ?thesis .
  next
    case (Some a)
    obtain pc'' st'' s'' f'' rs'' pcs'' where a = ((pc'', st'', s'', f'', rs''), pcs'')
      by (cases a) auto
    from assms upd_time_indep have time_indep P n (pc_u, [], s', True, []) by auto
    from stepst_f_equiv[OF this] ‹_ = Some a› prems ‹a = _› show ?thesis by auto metis
  qed
  then show ?case unfolding make_f_def by simp
qed


lemma steps_P_upd:
  assumes
    stepst P n u' (pc_u, [], s', True, []) (pc, st, s'', f, r)
    q < p (l, pc_g, a, pc_u, l') ∈ fst (N ! q)
  shows
    Some s'' = make_mf pc_u s' (is ?A) r = make_f pc_u s' (is ?B)
proof −
```

```
  from assms upd_time_indep have time_indep P n (pc_u, [], s', True, []) by auto
  from stepst_f_equiv[OF this, symmetric] assms(1) ‹q < p› obtain pc st f pcs where
    exec PF n (pc_u, [], s', True, []) [] = Some ((pc, st, s'', f, r), pcs)
    unfolding make_mf_def state_ta_def by (fastforce split: option.splits)
  then show ?A ?B unfolding make_f_def make_mf_def by (auto split: option.split)
qed

lemma steps_P_guard:
  assumes
    stepst P n u' (pc_g, [], s', True, []) (pc, st, s'', True, rs)
    q < p (l, pc_g, a, pc_u, l') ∈ fst (N ! q)
  shows
    make_c pc_g s' (is ?A) u' ⊢ make_g pc_g s' (is ?B)
proof −
  from stepst_t_complete[OF _ assms(1)] clock_conj assms obtain pcs where
    exec PT n (pc_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs)
    ∀ pc∈set pcs. ∀ ac. P pc = Some (CEXP ac) ⟶ u' ⊢_a ac
    by fastforce
  then show ?A ?B unfolding make_c_def make_g_def
    by (auto split: option.split instrc.split_asm simp: list_all_iff set_map_filter clock_val_def)
qed

lemma P_steps_guard:
  assumes
    make_c pc_g s' u' ⊢ make_g pc_g s'
    q < p (l, pc_g, a, pc_u, l') ∈ fst (N ! q)
  shows
    ∃ pc s'' st rs. stepst P n u' (pc_g, [], s', True, []) (pc, st, s'', True, rs)
proof −
  from assms(1) ‹q < p› obtain pc st s'' rs pcs where ∗:
    exec PT n (pc_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs)
    unfolding make_c_def state_ta_def by (fastforce split: option.splits)
  with exec_min_steps[of PT n _ [] _ pcs] have ∗∗:
    exec PT (length pcs) (pc_g, [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs @ [])
    by auto
  from ∗ assms(2) have
    u' ⊢ List.map_filter (λpc. case P pc of
        None ⇒ None
      | Some (INSTR xa) ⇒ Map.empty xa
      | Some (CEXP xa) ⇒ Some xa) pcs unfolding make_g_def
    by auto
  moreover from ∗∗ exec_steps_visited[of PT pcs _ []] have pc ∈ set pcs
    if steps PT (length pcs) (pc_g, [], s', True, []) (pc, st, m, f, rs) for pc st m f rs
    using that by fastforce
  ultimately have u' ⊢_a ac
    if steps PT (length pcs) (pc_g, [], s', True, []) (pc, st, m, f, rs) P pc = Some (CEXP ac)
    for pc st m f rs ac
    using that by (auto 4 3 split: option.splits simp: list_all_iff set_map_filter clock_val_def)
  moreover from ∗∗ have
    steps PT (length pcs) (pc_g, [], s', True, []) (pc, st, s'', True, rs) PT pc = Some HALT
    by (auto dest: exec_steps)
  ultimately have stepst P (length pcs) u' (pc_g, [], s', True, []) (pc, st, s'', True, rs)
    by (auto intro: steps_striptp_stepsc simp: stepst_def striptp_def elim: stript.elims)
  moreover from exec_len[OF ∗] have n ≥ length pcs by simp
```

ultimately show *?thesis* **by** (*blast intro*: *stepst_mono*)
qed

lemma *P_bounded*:
assumes
(*defs.P* ! *q*) (*L'* ! *q*) *s'* *q* < *p*
shows *bounded B s'*
using *assms* **unfolding** *state_pred_def state_ta_def* **by** (*auto split*: *option.splits*)

lemma *P_steps*:
assumes
(*defs.P* ! *q*) (*L'* ! *q*) *s'*
q < *p* *L'* ∈ *defs.states'* *s'*
shows
∃ *pc st s'' rs*. *stepst P n u'* ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) (*pc*, *st*, *s''*, *True*, *rs*)
proof −
from *assms pred_time_indep* **have** ∗: *time_indep P n* ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) **by** *auto*
from *assms*(*1*) ‹*q* < *p*› **obtain** *pc st s'' rs pcs* **where**
exec PF n ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) [] = *Some* ((*pc*, *st*, *s''*, *True*, *rs*), *pcs*)
unfolding *state_pred_def state_ta_def* **by** (*auto split*: *option.splits*)
with *stepst_f_equiv*[*OF* ∗] **show** *?thesis* **by** *blast*
qed

lemma *steps_P*:
assumes
stepst P n u' ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) (*pc*, *st*, *s''*, *True*, *rs*)
q < *p* *L'* ∈ *defs.states'* *s'*
bounded B s'
shows
(*defs.P* ! *q*) (*L'* ! *q*) *s'*
proof −
from *assms pred_time_indep* **have** *time_indep P n* ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) **by** *auto*
from *stepst_f_equiv*[*OF this*] *assms*(*1*) **obtain** *pcs'* **where**
exec PF n ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) [] = *Some* ((*pc*, *st*, *s''*, *True*, *rs*), *pcs'*)
by *blast*
with ‹*q* < *p*› ‹*bounded B s'*› **show** *?thesis* **unfolding** *state_pred_def state_ta_def* **by** *simp*
qed

lemma *P_iff*:
(∃ *pc st rs s''*. *stepst P n u'* ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) (*pc*, *st*, *s''*, *True*, *rs*)
∧ *bounded B s'*)
⟷ (*defs.P* ! *q*) (*L'* ! *q*) *s'* **if** *q* < *p* *L'* ∈ *defs.states'* *s'*
using *that* **by** (*metis steps_P P_steps P_bounded*)

lemma *states'_updI'*:
assumes (*L'* ! *q*, *g*, (*a*, *c*, *m*), *f*, *l'*) ∈ *fst* (*defs.N* ! *q*) *L'* ∈ *defs.states'* *s''*
shows *L'*[*q* := *l'*] ∈ *defs.states'* *s'*
using *assms*
unfolding *prod.states'_simp*[*of s' s''*]
unfolding *Product_TA_Defs.states_def*
apply *clarsimp*
subgoal for *p*
by (*cases p* = *q*; *force simp*: *prod.trans_of_N_s_2*[*simplified*] *Prod_TA_Defs.N_s_length*)
done

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

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

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

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

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

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

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

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

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

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

lemma equiv_complete':
  assumes step: A ⊢n ⟨L, s, u⟩ →a ⟨L', s', u'⟩
```

```
  shows state_ta ⊢ ⟨L, s, u⟩ →a ⟨L', s', u'⟩ ∧ L' ∈ defs.states' s'
     ∧ (∀ q < p. (defs.P ! q) (L' ! q) s')
    using step proof cases
    case (step_u_t N P d I)
    note [simp] = A_simp[OF this(1)]
    from step_u_t(2−) show ?thesis
      apply safe
      subgoal
        by (auto simp: state_ta_def p_def state_inv_def intro: step_sn_t)
      by (fastforce simp: p_def intro: steps_P intro!: states'_updI)+
  next
    case (step_u_i P pc_g uu uv uw ux pc_u uy uz va r N I l a l' p)
    note [simp] = A_simp[OF this(1)]
    from step_u_i(2−) show ?thesis
      apply −
      apply (simp add: Prod_TA_Defs.N_s_length)
      apply (subst state_ta_unfold)
      apply (frule steps_P_guard(1))
        apply assumption
        apply (simp; fail)
      apply (drule steps_P_guard(2))
        apply assumption
        apply (simp; fail)
      apply (frule steps_P_upd(1))
        apply assumption
        apply (simp; fail)
      apply (drule steps_P_upd(2))
        apply assumption
        apply (simp; fail)
      apply (drule N_transD)
       apply assumption
        apply safe
      apply (rule step_sn_i)
        apply assumption
      apply (solves auto; fail)+
      apply (simp add: state_ta_def p_def state_inv_def state_pred_def; fail)
      apply (solves auto; fail)+
      apply (simp add: Prod_TA_Defs.N_s_length; fail)
      apply (solves auto; fail)+
      by (fastforce simp: p_def intro: steps_P intro!: states'_updI)+
  next
    case (step_u_s P pc_g1 vb vc vd ve pc_g2 vf vg vh vi pc_u2 vj vk s1 vl r2 pc_u1 r1 vm vn
vo vp N
          I l1 a l1' p' l2 l2' q
        )
    note [simp] = A_simp[OF this(1)]
    from ‹q < length L› have q < p by (simp add: Prod_TA_Defs.N_s_length)
    from step_u_s(2−) show ?thesis
      apply −
      apply (simp add: Prod_TA_Defs.N_s_length)
      apply (subst state_ta_unfold)
      apply (frule steps_P_guard(1))
        apply assumption
        apply (simp; fail)
```

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

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

lemma equiv_complete'':
  assumes step: A ⊢n ⟨L, s, u⟩ →a ⟨L', s', u'⟩ p > 0
    shows (∀ q<p. ∃ pc st s'' rs pcs.
           exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
           Some ((pc, st, s'', True, rs), pcs)) (is ?A)
         bounded B s' (is ?B)
proof −
  from assms equiv_complete' have ∗: ∀ q<p. (defs.P ! q) (L' ! q) s' by simp
  then show ?A unfolding state_ta_def state_pred_def by (fastforce split: option.splits)
```

```
  from ‹p > 0› * have (defs.P ! 0) (L' ! 0) s' by auto
  with ‹p > 0› show ?B unfolding state_ta_def state_pred_def by (auto split: option.splits)
 qed

 lemma equiv_steps_sound':
  assumes step: state_ta ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩
  shows A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩ ∧ L' ∈ defs.states' s' ∧
     (∀ q<p. ∃ pc st s'' rs pcs.
        exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
        Some ((pc, st, s'', True, rs), pcs)) ∧ bounded B s'
  using step states inv
 proof (induction A ≡ state_ta L ≡ L s ≡ s u ≡ u L' s' u' arbitrary: rule: steps_sn.induct)
  case (refl)
  with bounded show ?case by blast
 next
  case prems: (step L' s' u' a L'' s'' u'')
  from prems have *:
   A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩ L' ∈ defs.states' s'
   (∀ q<p. ∃ pc st s'' rs pcs.
        exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
        Some ((pc, st, s'', True, rs), pcs))
   bounded B s'
   by auto
  interpret interp: Equiv_TA A n L' s'
    using pred_time_indep upd_time_indep clock_conj * by unfold_locales (auto simp: Len intro!: *)
  from prems(3) have
   A ⊢n ⟨L', s', u'⟩ →a ⟨L'', s'', u''⟩ L'' ∈ defs.states' s''
   ∀ q<p. ∃ pc st s''' rs pcs.
        exec PF n ((I ! q) (L'' ! q), [], s'', True, []) [] =
        Some ((pc, st, s''', True, rs), pcs)
   bounded B s''
   by (force dest!: interp.equiv_sound')+
  with * interp.states show ?case
   by − (assumption | rule conjI steps_un.intros)+
 qed

lemma equiv_steps_complete':
  state_ta ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩ ∧ L' ∈ defs.states' s' ∧
     (∀ q<p. ∃ pc st s'' rs pcs.
        exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
        Some ((pc, st, s'', True, rs), pcs)) ∧ bounded B s'
  if A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩ p > 0
   using that states inv proof (induction A ≡ A n ≡ n L ≡ L s ≡ s u ≡ u _ _ _ rule: steps_un.induct)
  case refl
  with bounded show ?case by blast
 next
  case prems: (step L' s' u' a L'' s'' u'')
  from prems have *:
   state_ta ⊢ ⟨L, s, u⟩ →* ⟨L', s', u'⟩ L' ∈ defs.states' s'
   (∀ q<p. ∃ pc st s'' rs pcs.
        exec PF n ((I ! q) (L' ! q), [], s', True, []) [] =
        Some ((pc, st, s'', True, rs), pcs))
```

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

lemmas *equiv_steps_sound* = *equiv_steps_sound′*[*THEN conjunct1*]
lemmas *equiv_steps_complete* = *equiv_steps_complete′*[*THEN conjunct1*]

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

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

end

end
theory *UPPAAL_State_Networks_Impl*
imports *Munta_Base.Normalized_Zone_Semantics_Impl UPPAAL_State_Networks*
begin

4 Implementation of UPPAAL Style Networks

no_notation *OR* (**infix** *or 60*)

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

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

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

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

lemma *steps_reset*:
 ∀ *c* ∈ *set r'*. ∃ *x pc*. *Some* (*STOREC c x*) = *P pc*
 if *steps P n* (*pc*, *st*, *s*, *f*, *r*) (*pc'*, *st'*, *s'*, *f'*, *r'*) ∀ *c* ∈ *set r*. ∃ *x pc*. *Some* (*STOREC c x*) = *P pc*
 using *that*
 by (*induction P* ≡ *P n* (*pc*, *st*, *s*, *f*, *r* :: *nat list*) (*pc'*, *st'*, *s'*, *f'*, *r'*) *arbitrary*: *pc st s f r rule*: *steps.induct*)
  (*auto dest*!: *step_resets''*[**where** *P* = *P*])

lemma *exec_reset*:
 ∀ *c* ∈ *set r'*. ∃ *x pc*. *Some* (*STOREC c x*) = *P pc*
 if *Some* ((*pc'*, *st'*, *s'*, *f'*, *r'*), *pcs'*) = *exec P n* (*pc*, *st*, *s*, *f*, []) *pcs*
 using *exec_steps*[*OF that*[*symmetric*]] *steps_reset* **by** *force*

lemma *exec_pointers*:
 ∀ *pc* ∈ *set pcs'*. ∃ *pc instr*. *Some instr* = *P pc*
 if *Some* ((*pc'*, *st'*, *s'*, *f'*, *r'*), *pcs'*) = *exec P n* (*pc*, *st*, *s*, *f*, *r*) *pcs*
  ∀ *pc* ∈ *set pcs*. ∃ *pc instr*. *Some instr* = *P pc*
 using *that*
 apply (*induction rule*: *exec.induct*)
 by (*auto split*: *option.splits if_splits*) *metis*+

lemma *exec_pointers'*:
 ∀ *pc* ∈ *set pcs'*. ∃ *pc instr*. *Some instr* = *P pc*
 if *Some* ((*pc'*, *st'*, *s'*, *f'*, *r'*), *pcs'*) = *exec P n* (*pc*, *st*, *s*, *f*, *r*) []
 using *that exec_pointers* **by** *fastforce*

context *Prod_TA_Defs*
begin

lemma *finite_range_I'*:
 assumes ∀ *A*∈{*0*..<*p*}. *finite* (*range* (*snd* (*N* ! *A*)))
  shows *finite* (*range* (*I'* *s*))
  using *assms* **unfolding** *inv_of_def Product_TA_Defs.product_ta_def N_s_def*
  by (*auto simp*: *inv_of_def p_def intro*!: *Product_TA_Defs.finite_invariant_of_product*)

lemma *range_prod_invariant*:
 range prod_invariant = *range* (*I'* *s*)
 unfolding *prod_invariant_def* **using** *I'_simp* **by** *auto*

lemma *finite_rangeI*:
 assumes ∀ *A*∈{*0*..<*p*}. *finite* (*range* (*snd* (*N* ! *A*)))
 shows *finite* (*range prod_invariant*)
 using *assms* **by** (*metis finite_range_I' range_prod_invariant*)

end

context *Equiv_TA_Defs*
begin

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

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

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

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

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

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

unfolding *make_f_def*
apply (*clarsimp split*: *option.split_asm*)
by (*auto dest*: *exec_reset simp*: *P_Storec_iff*)

lemma *product_trans_resets*:
collect_clkvt ($\bigcup$ *s. defs.T' s*) $\subseteq$ {*c.* $\exists$ *x pc. Some* (*INSTR* (*STOREC c x*)) = *P pc*}
unfolding *trans_of_def*
unfolding *Product_TA_Defs.product_ta_def*
apply *simp*
unfolding *Product_TA_Defs.product_trans_def*
unfolding *collect_clkvt_def*
apply *safe*
unfolding *Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def*
apply *clarsimp_all*
unfolding *defs.N_s_def*
unfolding *trans_of_def*
unfolding *defs.T_s_def*
unfolding *state_ta_def*
unfolding *state_trans_t_def*
unfolding *make_f_def*
apply (*clarsimp_all split*: *option.split_asm*)
by (*auto dest*: *exec_reset simp*: *P_Storec_iff*)

lemma *product_trans_guards*:
Timed_Automata.collect_clkt ($\bigcup$ *s. defs.T' s*)
$\subseteq$ {*constraint_pair ac* | *ac.* $\exists$ *pc. Some* (*CEXP ac*) = *P pc*}
unfolding *trans_of_def*
unfolding *Product_TA_Defs.product_ta_def*
apply *simp*
unfolding *Product_TA_Defs.product_trans_def*
unfolding *Timed_Automata.collect_clkt_def collect_clock_pairs_def*
apply *safe*
unfolding *Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def*
apply *clarsimp_all*
unfolding *defs.N_s_def*
unfolding *trans_of_def*
unfolding *defs.T_s_def*
unfolding *state_ta_def*
unfolding *state_trans_t_def*
unfolding *make_g_def*
apply (*clarsimp_all split*: *option.split_asm*)
subgoal premises *prems*
using *prems*(*1*,*3*−)
unfolding *set_map_filter*
by (*smt* (*verit*, *best*) *instrc.simps*(*5*,*6*) *is_instr.cases*
mem_Collect_eq option.exhaust option.inject
option.simps(*3*,*4*,*5*))+
subgoal premises *prems*
using *prems*(*1*,*2*,*4*−)
apply *safe*
unfolding *set_map_filter*
apply (*drule exec_pointers'*)
by (*smt* (*verit*, *best*) *instrc.simps*(*5*,*6*) *is_instr.cases*
mem_Collect_eq option.exhaust option.inject

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

end

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

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

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

abbreviation repeat x n ≡ map (λ _. x) [0..<n]

abbreviation conv_prog P pc ≡ map_option (map_instrc real_of_int) (P pc)
abbreviation conv_A' ≡ λ (T, I). (T, conv_cc o I)

fun hd_of_formula :: formula ⇒ nat list ⇒ int list ⇒ 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 φ _) = 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* .

abbreviation *EA* ≡ *equiv.state_ta*

abbreviation *A* ≡ *equiv.defs.prod_ta*

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

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

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

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

end

lemma *snd_comp*[*simp*]:

snd o (λ *i*. (*f i*, *g i*)) = *g*
by *auto*

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*: *clk_set′* = {*1*..*m*}
and *p_gt_0*: *p* > *0*
and *m_gt_0*: *m* > *0*
and *processes_have_trans*: ∀ *i* < *p*. *trans* ! *i* ≠ [] — Necessary for refinement
and *start_has_trans*: ∀ *q* < *p*. *trans* ! *q* ! *0* ≠ [] — Necessary for refinement

assumes *resets_zero*: ∀ *x c*. *Some* (*INSTR* (*STOREC c x*)) ∈ *set prog* ⟶ *x* = *0*

begin

lemma *consts_nats′*:
∀ *I* ∈ *set inv*. ∀ *cc* ∈ *set I*. ∀ (*c*, *d*) ∈ *collect_clock_pairs cc*. *d* ∈ ℕ

∀ *ac*. *Some* (*CEXP ac*) ∈ *set prog* ⟶ (*snd* (*constraint_pair ac*) ∈ ℕ)
using *consts_nats* **unfolding** *clkp_set′_def* **by** *force*+

lemma *clk_pairs_N_inv*:
⋃ (*collect_clock_pairs* ‘ *range* (*snd x*)) ⊆ ⋃ (*collect_clock_pairs* ‘ *set* (*concat inv*))
if *x* ∈ *set equiv.defs.N* **for** *x*
using *that process_length*(*1*)
unfolding *equiv.state_ta_def equiv.state_inv_def equiv.p_def*
unfolding *N_def I_def*
by *clarsimp* (*auto split*: *if_split_asm dest*: *nth_mem*)+

lemma *clkp_set_simp_1*:
⋃ (*collect_clock_pairs* ‘ *set* (*concat inv*)) ⊇ *Timed_Automata.collect_clki* (*snd A*)
unfolding *equiv.defs.prod_ta_def inv_of_def*
apply (*rule subset_trans*)
apply *simp*
apply (*rule equiv.defs.collect_clki_prod_invariant′*)
unfolding *Timed_Automata.collect_clki_def* **using** *clk_pairs_N_inv nth_mem* **by** *blast*

lemma *clk_set_simp_2*:
{*c*. ∃ *x*. *Some* (*INSTR* (*STOREC c x*)) ∈ *set prog*} ⊇ *collect_clkvt* (*trans_of A*)
unfolding *equiv.defs.prod_ta_def trans_of_def*
apply (*rule subset_trans*)
apply *simp*
apply (*rule equiv.defs.collect_clkvt_prod_trans_subs*)
apply (*rule subset_trans*)
apply (*rule equiv.product_trans_resets*)
unfolding *N_def PROG_def* **by** (*auto dest*!: *nth_mem*) *metis*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma P_p[simp]:
```

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

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

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

end

end
theory Program_Analysis
  imports
    UPPAAL_State_Networks_Impl
    Munta_Base.More_Methods
begin

fun steps_approx :: nat ⇒ 't instrc option list ⇒ addr ⇒ addr set where
  steps_approx 0 prog pc = (if pc < length prog then {pc} else {}) |
  steps_approx (Suc n) prog pc =
    (
      if pc ≥ length prog
      then {}
      else
        case prog ! pc of
          None ⇒ {pc}
        | Some cmd ⇒
          let succs =
            (
              case cmd of
                CEXP ac ⇒ {pc + 1}
              | INSTR instr ⇒ (
                  case instr of
                    CALL   ⇒ {..<length prog}
                  | RETURN ⇒ {..<length prog}
                  | JMPZ pc' ⇒ {pc + 1, pc'}
                  | HALT ⇒ {}
                  |    _ ⇒ {pc + 1})
            )
          in {pc} ∪ ⋃ (steps_approx n prog ‘ succs)
    )


lemma bounded_less_simp[simp]:
  ∀ q∈{..<p::nat}. P q ≡ ∀ q < p. P q
  by (rule eq_reflection) auto

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

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


private definition [simp]:
  P' ≡ map_option strip o (conv_prog (λ i. if i < length prog then prog ! i else None))

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

lemmas steps_out_of_range = steps_out_of_range'[unfolded striptp_def P'_def]

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

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

lemmas steps_steps_approx = steps_steps_approx'[unfolded P'_def]

end


context
 fixes prog :: int instrc option list
begin

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

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

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

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

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

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

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

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

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

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

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

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

end

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

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

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

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

lemma visitedc_reset:
 ∃ x. ∃ pc ∈ set pcs. Some (INSTR (STOREC c x)) = P pc
 if visitedc P n u (pc, st, s, f, r) (pc', st', s', f', r') pcs c ∈ set r' − set r
 using that
 apply (
    induction P ≡ P n u (pc, st, s, f, r :: nat list) (pc', st', s', f', r') pcs
    arbitrary: pc st s f r rule: visitedc.induct
    )
 apply blast
by (erule stepc.elims; force split: option.split_asm if_split_asm elim!: step.elims)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

unbundle lattice_syntax

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

  by auto

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

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

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

lemmas resets_start' = resets_start[OF find_resets_start]

context UPPAAL_Reachability_Problem_precompiled_defs
begin

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

end

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

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

lemma guaranteed_execution:
  assumes
    ∀ pc ∈ {pc..<pc_t}.
      prog ! pc ≠ None
      ∧ prog ! pc ∉ Some ` INSTR `
        {STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,
instr.LE, instr.EQ}
      ∧ (∀ c d. prog ! pc = Some (INSTR (STOREC c d)) ⟶ d = 0)

    ∀ pc ∈ {pc..<pc_t}. ∀ pc'. prog ! pc = Some (INSTR (JMPZ pc')) ⟶ pc' > pc ∧ pc' ≤ pc_t
    prog ! pc_t = Some (INSTR HALT) pc_t ≥ pc n > pc_t − pc pc_t < length prog
  shows ∃ st' s' f' r'. steps
      (map_option stripf o (λpc. if pc < size prog then prog ! pc else None))
      n (pc, st, s, f, r) (pc_t, st', s', f', r')
  using assms
proof (induction pc_t − pc arbitrary: pc st s f r n rule: less_induct)
  case less
  let ?prog = (map_option stripf ∘ (λpc. if pc < length prog then prog ! pc else None))
  from ‹pc_t − pc < n› obtain n' where [simp]: n = Suc n' by (cases n) auto
```

```
  from less.prems(3−) have [simp]: pc < length prog pc_t < length prog by auto
  show ?case
  proof (cases pc_t = pc)
    case True
    then show ?thesis by force
  next
    case False
    with less.prems(1,4) have valid_instr:
      prog ! pc ∉ Some ‘ INSTR ‘
       {STORE, HALT, POP, CALL, RETURN, instr.AND, instr.NOT, instr.ADD, instr.LT,
instr.LE, instr.EQ}
      prog ! pc ≠ None
      by auto
    from ‹pc ≤ _› ‹pc_t ≠ _› less.prems(2) have jumps:
      pc' > pc ∧ pc' ≤ pc_t if prog ! pc = Some (INSTR (JMPZ pc')) for pc'
      using that by fastforce
    from ‹pc ≤ _› ‹pc_t ≠ _› less.prems(1) have stores:
      d = 0 if prog ! pc = Some (INSTR (STOREC c d)) for c d
      using that by fastforce
    show ?thesis
    proof (cases prog ! pc)
      case None
      from ‹pc_t ≠ _› less.prems(1,4) have prog ! pc ≠ None
        by simp
      with None show ?thesis by auto
    next
      case (Some a)
      then show ?thesis
      proof (cases a)
        case (INSTR instr)
        have ∗:
          ∃ st' s' f' r'. steps ?prog n' (pc', st, s, f, r) (pc_t, st', s', f', r')
          if pc < pc' pc' ≤ pc_t for pc' st s f r
          apply (rule less.hyps[of pc'])
          subgoal
            using that by simp
          subgoal
            using that less.prems(1) by force
          subgoal
            apply clarsimp
            subgoal premises prems for pc_s pc'
            proof −
              from prems that have pc_s ∈ {pc..<pc_t} by simp
              with prems(1) less.prems(2) show ?thesis by blast
            qed
            done
          using ‹prog ! pc_t = _› ‹pc_t − pc < n› that by auto
        from ‹pc_t ≠ pc› ‹pc ≤ _› have Suc pc ≤ pc_t by simp
        then obtain pc' st' s' f' r' where
          step instr (pc, st, s, f, r) = Some (pc', st', s', f', r') pc < pc' pc' ≤ pc_t
          apply atomize_elim
          apply (cases instr)
          using valid_instr ‹a = _› ‹_ = Some a› by (auto simp: stores jumps)
        with ‹a = _› ‹_ = Some a› show ?thesis by (force dest!: ∗)
```

```
      next
        case (CEXP x2)
        have ∗:
          ∃ st' s' f' r'. steps ?prog n' (Suc pc, st, s, f, r) (pc_t, st', s', f', r')
          if Suc pc ≤ pc_t for st s f r
          apply (rule less.hyps[of Suc pc])
          subgoal
            using ‹pc ≤ _› ‹pc_t ≠ pc› by simp
          subgoal
            using less.prems(1) by force
          subgoal
            apply clarsimp
            subgoal premises prems for pc_s pc'
            proof −
              from prems have pc_s ∈ {pc..<pc_t} by simp
              with prems(1) less.prems(2) show ?thesis by blast
            qed
            done
          using ‹prog ! pc_t = _› ‹pc_t − pc < n› that by auto
        from ‹pc_t ≠ pc› ‹pc ≤ _› have Suc pc ≤ pc_t by simp
        with ‹a = _› ‹_ = Some a› show ?thesis by (force dest!: ∗)
      qed
    qed
  qed
qed

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

  by auto

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

lemma find_next_halt_finds_halt:
  prog ! pc' = Some (INSTR HALT) ∧ pc ≤ pc' ∧ pc' < length prog
  if find_next_halt prog pc = Some pc'
using that proof (induction prog pc rule: find_next_halt.induct)
  case prems: (1 prog pc)
  from prems(20) show ?case
    by (
        simp,
        simp
        split: if_split_asm option.split_asm instrc.split_asm instr.split_asm
        del: find_next_halt.simps;
        fastforce dest: prems(1−19) simp del: find_next_halt.simps)
```

qed

definition
guaranteed_execution_cond prog pc_s n ≡
case find_next_halt prog pc_s of
None ⇒ *False* |
Some pc_t ⇒
(
∀ *pc* ∈ {*pc_s*..<*pc_t*}.
prog ! *pc* ≠ *None*
∧ *prog* ! *pc* ∉ *Some* ‘ *INSTR* ‘
{*STORE*, *HALT*, *POP*, *CALL*, *RETURN*, *instr.AND*, *instr.NOT*, *instr.ADD*, *instr.LT*, *instr.LE*, *instr.EQ*}
∧ (∀ *c d*. *prog* ! *pc* = *Some* (*INSTR* (*STOREC c d*)) ⟶ *d* = *0*)
) ∧
(∀ *pc* ∈ {*pc_s*..<*pc_t*}. ∀ *pc′*. *prog* ! *pc* = *Some* (*INSTR* (*JMPZ pc′*)) ⟶ *pc′* > *pc* ∧ *pc′* ≤ *pc_t*)
∧ *n* > *pc_t* − *pc_s*

lemma *guaranteed_execution_cond_alt_def*[*code*]:
guaranteed_execution_cond prog pc_s n ≡
case find_next_halt prog pc_s of
None ⇒ *False* |
Some pc_t ⇒
(
∀ *pc* ∈ {*pc_s*..<*pc_t*}.
prog ! *pc* ≠ *None*
∧ *prog* ! *pc* ∉ *Some* ‘ *INSTR* ‘
{*STORE*, *HALT*, *POP*, *CALL*, *RETURN*, *instr.AND*, *instr.NOT*, *instr.ADD*, *instr.LT*, *instr.LE*, *instr.EQ*}
∧ (*case prog* ! *pc of Some* (*INSTR* (*STOREC c d*)) ⇒ *d* = *0* | _ ⇒ *True*)
) ∧
(∀ *pc* ∈ {*pc_s*..<*pc_t*}.
case prog ! *pc of Some* (*INSTR* (*JMPZ pc′*)) ⇒ *pc′* > *pc* ∧ *pc′* ≤ *pc_t* | _ ⇒ *True*)
∧ *n* > *pc_t* − *pc_s*

proof (*rule eq_reflection*, *goal_cases*)
case *1*
have ∗:
(∀ *c d*. *prog* ! *pc* = *Some* (*INSTR* (*STOREC c d*)) ⟶ *d* = *0*) ⟷
(*case prog* ! *pc of Some* (*INSTR* (*STOREC c d*)) ⇒ *d* = *0* | _ ⇒ *True*) **for** *pc*
by (*auto split*: *option.split instrc.split instr.split*)
have ∗∗:
(∀ *pc′*. *prog* ! *pc* = *Some* (*INSTR* (*JMPZ pc′*)) ⟶ *pc′* > *pc* ∧ *pc′* ≤ *pc_t*) ⟷
(*case prog* ! *pc of Some* (*INSTR* (*JMPZ pc′*)) ⇒ *pc′* > *pc* ∧ *pc′* ≤ *pc_t* | _ ⇒ *True*) **for** *pc pc_t*
by (*auto split*: *option.split instrc.split instr.split*)
show *?case* **unfolding** *guaranteed_execution_cond_def* ∗ ∗∗ **..**
qed

lemma *guaranteed_execution′*:
∃ *pc_t st′ s′ f′ r′ pcs′*. *exec*
(*map_option stripf o* (λ*pc*. *if pc* < *size prog then prog* ! *pc else None*))

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


context UPPAAL_Reachability_Problem_precompiled_defs
begin

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

  lemma clkp_set'_alt_def:
    clkp_set' =
      ⋃ (collect_clock_pairs ‘ set (concat inv)) ∪ (constraint_pair ‘ collect_cexp)
    unfolding clkp_set'_def collect_cexp_def by auto

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

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

  lemma clk_set'_alt_def: clk_set' = (fst ‘ clkp_set' ∪ fst ‘ collect_store)
    unfolding clk_set'_def collect_store_def by auto

end


fun conj_instr :: 't instrc ⇒ addr ⇒ 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 ⇒ addr ⇒ addr ⇒ 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' prog (pc + 4) pc'

inductive_cases stepscE: stepsc prog n u (pc, st, m, f, rs) (pc', st', m', f', rs')

function check_conj_block' :: 't instrc option list ⇒ addr ⇒ addr option where
  check_conj_block' prog pc = (
    if pc ≥ length prog then None
    else if prog ! pc = Some (INSTR HALT) then Some pc
    else if
      prog ! pc = Some (INSTR COPY) ∧ (case prog ! (pc + 1) of Some (CEXP ac) ⇒ True |
_ ⇒ False)
      ∧ prog ! (pc + 2) = Some (INSTR instr.AND)
    then check_conj_block' prog (pc + 3)
    else if
      prog ! pc = Some (INSTR COPY) ∧ (case prog ! (pc + 2) of Some (CEXP ac) ⇒ True |
_ ⇒ False)
      ∧ prog ! (pc + 3) = Some (INSTR instr.AND)
      ∧ (case prog ! (pc + 1) of Some (INSTR (JMPZ pc')) ⇒ True | _ ⇒ False)
    then
      (case (prog ! (pc + 1), check_conj_block' prog (pc + 4)) of (Some (INSTR (JMPZ pc')),
Some pc'')
       ⇒ if pc' = pc'' then Some pc' else None | _ ⇒ None)
    else None
    )

  by pat_completeness auto

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

lemma is_conj_block'_len_prog:
  pc' < length prog if is_conj_block' prog pc pc'
```

```
  using that by induction auto

lemma check_conj_block':
  check_conj_block' prog pc = Some pc' ⟹ is_conj_block' prog pc pc'
  apply (induction prog pc rule: check_conj_block'.induct)
    apply (erule check_conj_block'.elims)
  by (auto
        split: if_split_asm option.split_asm instrc.split_asm instr.split_asm
        simp del: check_conj_block'.simps
        intro: is_conj_block'.intros is_conj_block'_len_prog)

lemma stepsc_reverseE':
  assumes stepsc prog (Suc n) u s s'' s'' ≠ s
  obtains pc' st' m' f' rs' cmd where
    stepc cmd u (pc', st', m', f', rs') = Some s''
    prog pc' = Some cmd
    stepsc prog n u s (pc', st', m', f', rs')
  apply atomize_elim
  using assms
proof (induction prog Suc n u s s'' arbitrary: n rule: stepsc.induct)
  case (1 prog n u start)
  then show ?case by simp
next
  case prems: (2 cmd u pc st m f rs s prog n s')
  then show ?case
  proof (cases s' = s)
    case True
    with prems show ?thesis
      apply simp
      apply solve_ex_triv+
      by (auto elim: stepsc.cases)
  next
    case False
    with prems(3) have n > 0 by (auto elim!: stepsc.cases)
    then obtain n' where n = Suc n' by (cases n) auto
    from prems(4)[OF this False] obtain cmd pc' st' m' f' rs' where
      stepc cmd u (pc', st', m', f', rs') = Some s' prog pc' = Some cmd
      stepsc prog n' u s (pc', st', m', f', rs')
      by atomize_elim
    with prems show ?thesis unfolding ‹n = _› by blast
  qed
qed

lemma stepsc_reverseE:
  assumes stepsc prog n u s s'' s'' ≠ s
  obtains n' pc' st' m' f' rs' cmd where
    n = Suc n'
    stepc cmd u (pc', st', m', f', rs') = Some s''
    prog pc' = Some cmd
    stepsc prog n' u s (pc', st', m', f', rs')
proof −
  from assms have n > 0 by (auto elim!: stepsc.cases)
  then obtain n' where n = Suc n' by (cases n) auto
  with assms show ?thesis by (auto elim!: stepsc_reverseE' intro!: that)
```

qed

lemma
pc = *pc′* − *1* **if**
stepc (*CEXP cc*) *u* (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc′*, *st′*, *m′*, *f′*, *rs′*)
using *that* **by** *auto*

lemma
pc = *pc′* − *1* **if**
stepc (*INSTR instr*) *u* (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc′*, *st′*, *m′*, *f′*, *rs′*)
¬ (∃ *x*. *instr* = *JMPZ pc′*) *instr* ≠ *RETURN instr* ≠ *CALL*
using *that* **by** (*auto split*: *option.split_asm if_split_asm elim*!: *step.elims*)

lemma *stepc_pc_no_jump*:
pc = *pc′* − *1* **if**
stepc cmd u (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc′*, *st′*, *m′*, *f′*, *rs′*)
cmd ≠ *INSTR* (*JMPZ pc′*) *cmd* ≠ *INSTR RETURN cmd* ≠ *INSTR CALL*
using *that* **by** (*cases cmd*) (*auto split*: *option.split_asm if_split_asm elim*!: *step.elims*)

inductive *stepsn* :: *′t programc* ⇒ *nat* ⇒ (*nat*, *′t* :: *time*) *cval* ⇒ *state* ⇒ *state* ⇒ *bool*
for *prog* **where**
stepsn prog 0 u start start |
stepsn prog (*Suc n*) *u* (*pc*, *st*, *m*, *f*, *rs*) *s′* **if**
stepc cmd u (*pc*, *st*, *m*, *f*, *rs*) = *Some s*
prog pc = *Some cmd*
stepsn prog n u s s′

declare *stepsn.intros*[*intro*]

lemma *stepsc_stepsn*:
assumes *stepsc P n u s s′*
obtains *n′* **where** *stepsn P n′ u s s′ n′* < *n*
using *assms* **by** *induction auto*

lemma *stepsn_stepsc*:
assumes *stepsn P n′ u s s′ n′* < *n*
shows *stepsc P n u s s′*
using *assms*
proof (*induction arbitrary*: *n*)
case (*1 u start*)
then obtain *n′* **where** *n* = *Suc n′* **by** (*cases n*) *auto*
then show *?case* **by** *auto*
next
case (*2 cmd u pc st m f rs s n s′ n′*)
from ‹*_* < *n′*› **have** *n* < *n′* − *1* **by** *simp*
from *2*(*4*)[*OF this*] *2*(*1*,*2*,*3*,*5*) **show** *?case*
proof −
have (∃*fa n fb p*. *P* = *fa* ∧ *n′* = *Suc n* ∧ *u* = *fb* ∧ (*pc*, *st*, *m*, *f*, *rs*) = *p* ∧ *s′* = *p*) ∨ (∃ *i fa n is isa b ns p fb na pa*. *P* = *fb* ∧ *n′* = *Suc na* ∧ *u* = *fa* ∧ (*pc*, *st*, *m*, *f*, *rs*) = (*n*, *is*, *isa*, *b*, *ns*) ∧ *s′* = *pa* ∧ *stepc i fa* (*n*, *is*, *isa*, *b*, *ns*) = *Some p* ∧ *fb n* = *Some i* ∧ *stepsc fb na fa p pa*)
using *2.prems Suc_pred′* ‹*P pc* = *Some cmd*› ‹*stepc cmd u* (*pc*, *st*, *m*, *f*, *rs*) = *Some s*› ‹*stepsc P* (*n′* − *1*) *u s s′*› *gr_implies_not_zero* **by** *blast*
then show *?thesis*
by *blast*

```
    qed
  qed

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

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

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

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

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

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

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

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

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

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

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

  shows f ∧ pc_t = pc'
  using assms
 unfolding stepst_def by (auto split: if_split_asm elim!: is_conj_block'_is_conj stepsc_stepsn)

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

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

```
    apply (erule stepsc.cases)
    by (auto split: if_split_asm option.split_asm elim!: UPPAAL_Asm.step.elims)
  from is_conj_block'_is_conj'[OF assms(1) this] show ?thesis by simp
qed

definition
  is_conj_block P pc pc' ≡
   (∃ ac. P ! pc = Some (CEXP ac)) ∧ is_conj_block' P (pc + 1) pc'
   ∨ (∃ 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'
  unfolding is_conj_block_def
  by (rule eq_reflection) (auto split: option.split_asm instrc.split_asm)

lemma is_conj_block_is_conj:
  assumes is_conj_block P pc pc' P ! pc = Some (CEXP ac)
    and
      stepst
        (λ i. if i < length P then P ! i else None) n u
        (pc, st, s, f, rs)
        (pc_t, st_t, s_t, True, rs_t)
  shows (u ⊢_a ac) ∧ pc_t = pc'
  using assms
  unfolding is_conj_block_def
  apply safe
  by ((drule is_conj_block'_is_conj2 is_conj_block'_is_conj3; simp), simp)+

lemma is_conj_block'_decomp:
  is_conj_block P pc' pc'' if
  is_conj_block' P pc pc'' P ! pc' = Some (CEXP ac) pc ≤ pc' pc' ≤ pc''
  using that
proof (induction arbitrary: pc')
  case (1 pc prog)
  then show ?case by (simp add: is_conj_block_def)
next
  case prems: (2 pc'' prog pc ac)
  with ‹pc ≤ _› consider pc' = pc | pc' = Suc pc | pc' = Suc (Suc pc) | pc + 3 ≤ pc'
    by force
  then show ?case using prems by cases (auto simp add: numeral_3_eq_3 is_conj_block_def)
next
  case prems: (3 pc'' prog pc ac)
  with ‹pc ≤ _› consider
    pc' = pc | pc' = Suc pc | pc' = Suc (Suc pc) | pc' = Suc (Suc (Suc pc)) | pc + 4 ≤ pc'
    by force
  then show ?case using prems by cases (auto simp add: numeral_3_eq_3 numeral_4_eq_4
is_conj_block_def)+
qed
```

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

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

abbreviation *conv_P* ≡ *map* (*map_option* (*map_instrc real_of_int*))

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

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

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

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

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

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

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

lemma *is_conj*:
fixes *u* :: *nat* ⇒ *real*
defines *S* ≡ *steps_approx n P pc_s*
defines *pc_c* ≡ *Min* {*pc*. ∃ *ac*. *pc* ∈ *S* ∧ *P* ! *pc* = *Some* (*CEXP ac*)}
assumes *is_conj_block P pc_c* (*Max S*)
and *stepst* (*conv_prog prog*) *n u* (*pc_s*, *st*, *s*, *f*, *rs*) (*pc_t*, *st_t*, *s_t*, *True*, *rs_t*)
and *stepsc* (*conv_prog prog*) *n u* (*pc_s*, *st*, *s*, *f*, *rs*) (*pc'*, *st'*, *s'*, *f'*, *rs'*)
and *P* ! *pc'* = *Some* (*CEXP ac*) *pc'* < *length P*
shows (*u* ⊢$_a$ *conv_ac ac*) ∧ *pc_t* = *Max S*
proof −
from *stepst_stepc_extend*[*OF assms*(*4*,*5*)] **have** ∗:
stepst (*conv_prog prog*) *n u* (*pc'*, *st'*, *s'*, *f'*, *rs'*) (*pc_t*, *st_t*, *s_t*, *True*, *rs_t*) **.**
from ‹*stepsc* _ _ _ _ _ _› ‹*P* ! *pc'* = _› ‹*pc'* < _› **have** *pc_c* ≤ *pc' pc'* ≤ *Max S*
unfolding *pc_c_def S_def* **by** (*auto intro*: *stepsc_steps_approx Min_le Max_ge*)
from *is_conj_block_decomp*[*OF assms*(*3*) ‹*P* ! *pc'* = _› *this*] **have**
is_conj_block P pc' (*Max S*) **.**
then have *is_conj_block* (*conv_P P*) *pc'* (*Max S*) **by** *auto*
from *is_conj_block_is_conj*[*OF this* _ *stepst_conv_P*[*OF* ∗]] ‹*P* ! *pc'* = _› ‹*pc'* < _›
show *?thesis*
by *auto*
qed

lemma *is_conj'*:

fixes $u :: nat \Rightarrow real$
defines $S \equiv steps_approx\ n\ P\ pc_s$
assumes $\{pc.\ \exists\ ac.\ pc \in S \wedge P\ !\ pc = Some\ (CEXP\ ac)\} = \{\}$
and $stepsc\ (conv_prog\ prog)\ n\ u\ (pc_s,\ st,\ s,\ f,\ rs)\ (pc',\ st',\ s',\ f',\ rs')$
and $P\ !\ pc' = Some\ (CEXP\ ac)\ pc' < length\ P$
shows *False*
using $stepsc_steps_approx[OF\ assms(3,5)]\ assms(4)\ assms(2)$ **unfolding** *S_def* **by** *auto*

definition
$check_conj_block\ pc\ pc' \equiv$
$(case\ P\ !\ pc\ of\ Some\ (CEXP\ ac) \Rightarrow True \mid _ \Rightarrow False) \wedge check_conj_block'\ P\ (pc + 1) = Some\ pc'$
$\vee\ (case\ P\ !\ pc\ of\ Some\ (CEXP\ ac) \Rightarrow True \mid _ \Rightarrow False) \wedge P\ !\ (pc + 1) = Some\ (INSTR\ instr.AND)$
$\wedge\ check_conj_block'\ P\ (pc + 2) = Some\ pc'$

lemma *check_conj_block*:
$check_conj_block\ pc\ pc' \Longrightarrow is_conj_block\ P\ pc\ pc'$
unfolding *is_conj_block_alt_def check_conj_block_def*
by (*auto dest*!: *check_conj_block' simp del*: *check_conj_block'.simps*)

definition
conjunction_check $\equiv$
$let\ S = steps_approx\ n\ P\ pc_s;\ S' = \{pc.\ \exists\ ac.\ pc \in S \wedge P\ !\ pc = Some\ (CEXP\ ac)\}\ in$
$S' = \{\} \vee check_conj_block\ (Min\ S')\ (Max\ S)$

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

proof −
let *?S* = $steps_approx\ n\ P\ pc_s$
have
$\{pc.\ pc \in ?S \wedge (case\ P\ !\ pc\ of\ Some\ (CEXP\ ac) \Rightarrow True \mid _ \Rightarrow False)\}$
$= \{pc.\ \exists\ ac.\ pc \in ?S \wedge P\ !\ pc = Some\ (CEXP\ ac)\}$
by *safe* (*auto split*: *option.splits instrc.splits*)
show *?thesis* **unfolding** *conjunction_check_def Let_def* ‹*_* = *_*› **..**
qed

lemma *conjunction_check*:
fixes $u :: nat \Rightarrow real$
assumes *conjunction_check*
and $stepst\ (conv_prog\ prog)\ n\ u\ (pc_s,\ st,\ s,\ f,\ rs)\ (pc_t,\ st_t,\ s_t,\ True,\ rs_t)$
and $stepsc\ (conv_prog\ prog)\ n\ u\ (pc_s,\ st,\ s,\ f,\ rs)\ (pc',\ st',\ s',\ f',\ rs')$
and $P\ !\ pc' = Some\ (CEXP\ ac)\ pc' < length\ P$
shows $u \vdash_a conv_ac\ ac$
using *assms*

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

end

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

unbundle *no_library_syntax*

5 Imperative Implementation of UPPAAL Style Networks

5.1 Executable Successor Computation

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

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

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

definition
trans_i_map =
map (*map* (*List.map_filter*
(λ (*g*, *a*, *m*, *l'*). *case a of Sil a* ⇒ *Some* (*g*, *a*, *m*, *l'*) | _ ⇒ *None*)
)) *trans*

definition

```
  trans_in_map ≡
    map (map (map
      (λ (g, a, m, l′). case a of In a ⇒ (g, a, m, l′)) o filter (λ (g, a, m, l′).
        case a of In a ⇒ True | _ ⇒ False))
        ) trans

definition
  trans_out_map ≡
    map (map (map
      (λ (g, a, m, l′). case a of Out a ⇒ (g, a, m, l′)) o filter (λ (g, a, m, l′).
        case a of Out a ⇒ True | _ ⇒ False))
        ) trans

abbreviation
  nested_list_to_iarray xs ≡ IArray (map IArray xs)


definition
  actions_by_state i ≡ fold (λ t acc. acc[fst (snd t) := (i, t) # (acc ! fst (snd t))])

definition
  all_actions_by_state t L ≡
    fold (λ i. actions_by_state i (t !! i !! (L ! i))) [0..<p] (repeat [] na)

definition PROG′ pc ≡ (if pc < length prog then (IArray prog) !! pc else None)
abbreviation PF ≡ stripfp PROG′
abbreviation PT ≡ striptp PROG′
definition runf pc s ≡ exec PF max_steps (pc, [], s, True, []) []
definition runt pc s ≡ exec PT max_steps (pc, [], s, True, []) []

lemma PROG′_PROG [simp]:
  PROG′ = PROG
  unfolding PROG′_def PROG_def by (rule ext) simp

definition bounded′ s ≡
  (∀ i<length s. fst (IArray bounds !! i) < s ! i ∧ s ! i < snd (IArray bounds !! i))

definition
  check_pred L s ≡
    list_all
      (λ q.
        case runf (pred ! q ! (L ! q)) s of
          Some ((_, _, _, f, _), _) ⇒ f ∧ bounded′ s
        | None ⇒ False
      )
      [0..<p]


definition
  make_cconstr pcs = List.map_filter
    (λ pc.
      case PROG pc of
        Some (CEXP ac) ⇒ Some ac
      | _ ⇒ None
```

```
  )
  pcs

definition
  check_g pc s ≡
    case runt pc s of
      Some ((_, _, _, True, _), pcs) ⇒ Some (make_cconstr pcs)
    | _ ⇒ None


 definition
  trans_i_from ≡ λ (L, s) i.
    List.map_filter (λ (g, a, m, l').
      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 ≡ 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 ≡ *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*)
using *assms* **unfolding** *trans_fun_def* **by** *auto*

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*)
using *assms* **unfolding** *trans_fun_def* **by** *auto*

end

context *Prod_TA_Defs*
begin

lemma *prod_trans_i_alt_def*:
prod_trans_i =
{((*L*, *s*), *g*, *a*, *r*, (*L′*, *s′*)) | *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*}
unfolding
prod_trans_i_def trans_of_def Product_TA_Defs.product_ta_def
Product_TA_Defs.product_trans_def
Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def
by (*safe*; *simp*; *metis*)

lemma *prod_trans_s_alt_def*:
prod_trans_s =
{((*L*, *s*), *g*, *a*, *r*, (*L′*, *s′*)) | *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*
}
unfolding
prod_trans_s_def trans_of_def Product_TA_Defs.product_ta_def
Product_TA_Defs.product_trans_def

```
    Product_TA_Defs.product_trans_i_def
    Product_TA_Defs.product_trans_s_def
  by (safe; simp; metis)

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

  lemma T_s_unfold_1:
    fst ‘ equiv.defs.T_s q s = fst ‘ fst (equiv.N ! q) if q < p
    using ‹q < p›
    unfolding equiv.defs.T_s_def
    unfolding equiv.state_ta_def
    unfolding equiv.state_trans_t_def
    by force

  lemma T_s_unfold_2:
    (snd o snd o snd o snd) ‘ equiv.defs.T_s q s = (snd o snd o snd o snd) ‘ fst (equiv.N ! q)
    if q < p
    using ‹q < p›
    unfolding equiv.defs.T_s_def
    unfolding equiv.state_ta_def
    unfolding equiv.state_trans_t_def
    by force

end

lemma (in Equiv_TA_Defs) p_p:
  defs.p = p
  by simp

context
  Prod_TA_Defs
begin

  lemma states'_alt_def:
    states' s =
    {L. length L = p ∧
        (∀ q < p. (L ! q) ∈ fst ‘ (fst (fst A ! q)) ∪ (snd o snd o snd o snd) ‘ (fst (fst A ! q)))}
    unfolding trans_of_def Product_TA_Defs.product_ta_def N_s_def
    unfolding Product_TA_Defs.product_trans_def
    unfolding Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def
    apply simp
    apply safe
    unfolding T_s_def
     apply (fastforce simp: Product_TA_Defs.states_def trans_of_def p_def)
    apply (force simp: Product_TA_Defs.states_def trans_of_def p_def)
    by (fastforce simp: Product_TA_Defs.states_def trans_of_def p_def image_iff)

end


context UPPAAL_Reachability_Problem_precompiled
```

begin

lemma *PF_unfold*:
equiv.PF = stripfp (conv_prog PROG)
using [[*show_abbrevs=false*]]
unfolding *N_def stripfp_def*
apply (*rule ext*)
subgoal for *x*
apply (*cases PROG x*)
apply (*solves simp*)
subgoal for *a*
by (*cases a*) *auto*
done
done

lemma *PT_unfold*:
equiv.PT = striptp (conv_prog PROG)
using [[*show_abbrevs=false*]]
unfolding *N_def striptp_def*
apply (*rule ext*)
subgoal for *x*
apply (*cases PROG x*)
apply (*solves simp*)
subgoal for *a*
by (*cases a*) *auto*
done
done

lemma *states'I*:
l ∈ *equiv.defs.states' s* **if** *A* ⊢ (*l*, *s*) $\longrightarrow^{g,a,r}$ (*l'*, *s'*)
using *equiv.defs.prod_ta_cases*[*OF that*]
unfolding *equiv.defs.prod_trans_i_alt_def equiv.defs.prod_trans_s_alt_def*
unfolding *Product_TA_Defs.product_trans_def*
unfolding *Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def*
by *fastforce*

lemma *A_lengthD*:
length l = p **if** *A* ⊢ (*l*, *s*) $\longrightarrow^{g,a,r}$ (*l'*, *s'*)
using *that* **by** (*auto dest*: *states'I*)

lemma *N_s_state_trans*:
assumes *equiv.defs.N_s s* ! *q* ⊢ *l* ! *q* $\longrightarrow^{g,(a,\ c,\ m'),r}$ *l'* *q* < *p*
obtains *f' g'* **where**
(*l* ! *q*, *g'*, (*a*, *c*, *m'*), *f'*, *l'*) ∈ *equiv.state_trans q g* = *g' s r* = *f' s*
using *assms*
unfolding *equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def*
unfolding *equiv.state_ta_def* **by** *auto*

lemma *make_f_collect_store*:
assumes (*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *fst* (*equiv.N* ! *q*) *c* ∈ *set* (*equiv.make_f pc_u s*) *q* < *p*
shows *c* ∈ *fst* ‘ *collect_store' pc_u*
proof −
from *assms*(*1*) ‹*q* < *p*› **have** (*pc_g*, *a*, *pc_u*, *l'*) ∈ *set* (*trans* ! *q* ! *l*)
unfolding *N_def T_def* **by** (*auto dest*!: *nth_mem*)

```
  from assms obtain pc x2 x3 x4 pcs r where exec:
    exec equiv.PF max_steps (pc_u, [], s, True, []) [] = Some ((pc, x2, x3, x4, r), pcs)
    unfolding equiv.make_f_def by (auto split: option.split_asm) metis
  with assms have c ∈ set r unfolding equiv.make_f_def by auto
  with exec_reset'[OF exec] obtain pc' d where Some (STOREC c d) = equiv.PF pc' pc' ∈ set
pcs
    by force
  with exec obtain y2 y3 y4 y5 where steps:
    steps equiv.PF max_steps (pc_u, [], s, True, []) (pc', y2, y3, y4, y5)
    by (auto intro: exec_steps')
  from ‹_ = equiv.PF pc'› have pc' < length prog
    unfolding N_def PROG_def stripfp_def by (simp split: if_split_asm)
  from steps have pc' ∈ steps_approx max_steps prog pc_u
    unfolding PF_unfold
    unfolding stripfp_def
   by (auto simp: PROG_def intro: steps_steps_approx[of stripf, OF _ _ _ ‹pc' < length prog›])
  with ‹_ = equiv.PF pc'› ‹_ ∈ set (trans ! q ! l)› show ?thesis
    unfolding collect_store'_def stripfp_def N_def PROG_def
    apply (clarsimp split: if_split_asm)
    apply (cases prog ! pc')
     apply (simp; fail)
    subgoal for x
      by (cases x; force)
    done
qed

lemma resets_approx:
  set r ⊆
  ⋃ {fst ‘ collect_store' r | i g a r. (g, a, r, (l' ! i)) ∈ set (trans ! i ! (l ! i))}
  if A ⊢ (l, s) ⟶^{g,a,r} (l', s')
proof −
  from that have [simp]: length l = p by (auto dest: A_lengthD)
  show ?thesis using that
    apply clarsimp
    apply (drule equiv.defs.prod_ta_cases)
    apply safe
    subgoal for x
      unfolding equiv.defs.prod_trans_i_alt_def
      apply simp
      unfolding Product_TA_Defs.product_trans_def
      apply safe
      unfolding Product_TA_Defs.product_trans_i_def
      apply clarsimp
      apply (erule N_s_state_trans, assumption)
      unfolding equiv.state_trans_t_def
      apply clarsimp
      subgoal for q l'' pc_g pc_u
        apply (frule make_f_collect_store, assumption+)
        unfolding N_def T_def
        apply (clarsimp dest!: nth_mem)
        subgoal premises prems for b j
        proof −
          from prems(6,8,11−) have
            (pc_g, Sil a, pc_u, l[q := l''] ! q) ∈ set (trans ! q ! (l ! q))
```

```
        by simp
      with prems(6,8,11−) show ?thesis by blast
    qed
    done
  done
 subgoal for x
  unfolding equiv.defs.prod_trans_s_alt_def
  apply simp
  unfolding Product_TA_Defs.product_trans_def
  apply safe
  unfolding Product_TA_Defs.product_trans_s_def
  apply clarsimp
  apply (erule N_s_state_trans, assumption)
  apply (erule N_s_state_trans, assumption)
  unfolding equiv.state_trans_t_def
  apply clarsimp
  apply (erule disjE)
  subgoal for s1 p' q l'' l'aa pc_g pc_ga pc_u pc_ua
   apply (frule make_f_collect_store, assumption+)
   unfolding N_def T_def
   apply (clarsimp dest!: nth_mem)
   subgoal premises prems for b j j'
   proof −
    from prems(9−) have
      (pc_g, In a, pc_u, l[p' := l'', q := l'aa] ! p') ∈ set (trans ! p' ! (l ! p'))
      by simp
    with prems(9−) show ?thesis by blast
   qed
   done
  subgoal for s1 q p' l'aa l'' pc_ga pc_g pc_ua pc_u
   apply (frule make_f_collect_store, assumption+)
   unfolding N_def T_def
   apply (clarsimp dest!: nth_mem)
   subgoal premises prems for b j j'
   proof −
    from prems(9−) have
      (pc_g, Out a, pc_u, l[q := l'aa, p' := l''] ! p') ∈ set (trans ! p' ! (l ! p'))
      by simp
    with prems(9−) show ?thesis by blast
   qed
   done
  done
 done
qed

lemma make_g_clkp_set'':
 fixes x
 assumes
  (l, pc_g, a, pc_u, l') ∈ fst (equiv.N ! q) x ∈ collect_clock_pairs (equiv.make_g pc_g s)
  q < p
 shows x ∈ clkp_set'' q l
proof −
 from assms(1) ‹q < p› have (pc_g, a, pc_u, l') ∈ set (trans ! q ! l)
  unfolding N_def T_def by (auto dest!: nth_mem)
```

```
  from assms obtain pc x2 x3 x4 r pcs where exec:
    exec equiv.PT max_steps (pc_g, [], s, True, []) [] = Some ((pc, x2, x3, x4, r), pcs)
    unfolding equiv.make_g_def by (auto split: option.split_asm)
  with assms have x ∈ collect_clock_pairs (List.map_filter (λ pc.
        case equiv.P pc of
          Some (CEXP ac) ⇒ Some ac
        | _ ⇒ None
          )
        pcs)
    unfolding equiv.make_g_def by auto
  then obtain pc' ac where
    equiv.P pc' = Some (CEXP ac) x = constraint_pair ac pc' ∈ set pcs
    unfolding equiv.make_g_def collect_clock_pairs_def set_map_filter
    by (clarsimp split: option.split_asm; auto split: instrc.split_asm)
  with exec obtain y2 y3 y4 y5 where steps:
    steps equiv.PT max_steps (pc_g, [], s, True, []) (pc', y2, y3, y4, y5)
    by (auto intro: exec_steps')
  from ‹equiv.P pc' = _› have pc' < length prog
    unfolding N_def PROG_def by (simp split: if_split_asm)
  from steps have pc' ∈ steps_approx max_steps prog pc_g
    unfolding PT_unfold
    unfolding striptp_def
   by (auto simp: PROG_def intro: steps_steps_approx[of stript, OF _ _ _ ‹pc' < length prog›])
  with ‹equiv.P pc' = _› ‹_ ∈ set (trans ! q ! l)› ‹x = _› show ?thesis
    unfolding clkp_set''_def collect_cexp'_def N_def PROG_def by (force split: if_split_asm)
qed

lemma guard_approx:
  collect_clock_pairs g ⊆
  ⋃ {clkp_set'' i (l ! i) | i g a r.
     (g, a, r, (l' ! i)) ∈ set (trans ! i ! (l ! i)) ∧ l ∈ equiv.defs.states' s ∧ i < p
    }
  if A ⊢ (l, s) ⟶^{g,a,r} (l', s')
proof −
  from that have [simp]: length l = p by (auto dest: A_lengthD)
  show ?thesis using that
    apply clarsimp
    apply (drule equiv.defs.prod_ta_cases)
    apply safe
    subgoal for x b
      unfolding equiv.defs.prod_trans_i_alt_def
      apply simp
      unfolding Product_TA_Defs.product_trans_def
      apply safe
      unfolding Product_TA_Defs.product_trans_i_def
      apply clarsimp
      apply (erule N_s_state_trans, assumption)
      unfolding equiv.state_trans_t_def
      apply clarsimp
      subgoal for q l'' pc_g pc_u
        apply (frule make_g_clkp_set'', assumption+)
        unfolding N_def T_def
        apply (clarsimp dest!: nth_mem)
        subgoal premises prems for j
```

```
      proof −
        from prems(6,8,11−) have
          (pc_g, Sil a, pc_u, l[q := l''] ! q) ∈ set (trans ! q ! (l ! q))
          by simp
        with prems(6,8,11−) prems show ?thesis by blast
      qed
      done
    done
  subgoal for x b
    unfolding equiv.defs.prod_trans_s_alt_def
    apply simp
    unfolding Product_TA_Defs.product_trans_def
    apply safe
    unfolding Product_TA_Defs.product_trans_s_def
    apply clarsimp
    apply (erule N_s_state_trans, assumption)
    apply (erule N_s_state_trans, assumption)
    unfolding equiv.state_trans_t_def
    apply clarsimp
    apply (drule collect_clock_pairs_append_cases)
    apply (erule disjE)
    subgoal for s1 p' q l'' l'aa pc_g pc_ga pc_u pc_ua
      unfolding equiv.make_c_def
        apply (clarsimp split: option.split_asm)
      apply (frule make_g_clkp_set'', assumption+)
      unfolding N_def T_def
      apply (clarsimp dest!: nth_mem)
      subgoal premises prems for j j'
      proof −
        from prems(9−) have
          (pc_g, In a, pc_u, l[p' := l'', q := l'aa] ! p') ∈ set (trans ! p' ! (l ! p'))
          by simp
        with prems(9−) show ?thesis by blast
      qed
      done
    subgoal for s1 q p' l'aa l'' pc_ga pc_g pc_ua pc_u
      apply (frule make_g_clkp_set'', assumption+)
      unfolding N_def T_def
      apply (clarsimp dest!: nth_mem)
      subgoal premises prems for j j'
      proof −
        from prems(9−) have
          (pc_g, Out a, pc_u, l[q := l'aa, p' := l''] ! p') ∈ set (trans ! p' ! (l ! p'))
          by simp
        with prems(9−) show ?thesis by blast
      qed
      done
    done
  done
qed

end
```

abbreviation *conv B* ≡ (*conv_prog* (*fst B*), (*map conv_A′* (*fst* (*snd B*))), *snd* (*snd B*))

context *UPPAAL_Reachability_Problem_precompiled*
begin

sublocale *defs′*:
Equiv_TA_Defs conv N max_steps **.**

lemma *equiv_states′_alt_def*:
equiv.defs.states′ s =
{*L*. *length L* = *p* ∧
(∀ *q* < *p*. *L* ! *q* ∈ *fst* ‘ *fst* (*equiv.N* ! *q*)
∨ *L* ! *q* ∈ (*snd o snd o snd o snd*) ‘ *fst* (*equiv.N* ! *q*))}
unfolding *Product_TA_Defs.states_def*
unfolding *equiv.defs.N_s_def trans_of_def*
using *T_s_unfold_1 T_s_unfold_2* **by** *simp*

lemma *init_states*:
init ∈ *equiv.defs.states′* s_0
using *processes_have_trans start_has_trans*
unfolding *equiv_states′_alt_def*
unfolding *init_def N_def T_def* **by** *force*

lemma *p_p*[*simp*]:
defs′.p = *p*
unfolding *defs′.p_def* **by** *simp*

lemma *T_s_unfold_1′*:
fst ‘ *defs′.defs.T_s q s* = *fst* ‘ *fst* (*defs′.N* ! *q*) **if** *q* < *p*
using ‹*q* < *p*›
unfolding *defs′.defs.T_s_def*
unfolding *defs′.state_ta_def*
unfolding *defs′.state_trans_t_def p_p*
by *force*

lemma *T_s_unfold_2′*:
(*snd o snd o snd o snd*) ‘ *defs′.defs.T_s q s* = (*snd o snd o snd o snd*) ‘ *fst* (*defs′.N* ! *q*)
if *q* < *p*
using ‹*q* < *p*›
unfolding *defs′.defs.T_s_def*
unfolding *defs′.state_ta_def*
unfolding *defs′.state_trans_t_def p_p*
by *force*

lemma *product_states′_alt_def*:
defs′.defs.states′ s =
{*L*. *length L* = *p* ∧
(∀ *q* < *p*. *L* ! *q* ∈ *fst* ‘ *fst* (*defs′.N* ! *q*)
∨ *L* ! *q* ∈ (*snd o snd o snd o snd*) ‘ *fst* (*defs′.N* ! *q*))}
unfolding *Product_TA_Defs.states_def*
unfolding *defs′.defs.N_s_def trans_of_def*
using *T_s_unfold_1′ T_s_unfold_2′*
by *force*

```
lemma states'_conv[simp]:
  defs'.defs.states' s = equiv.defs.states' s
  unfolding product_states'_alt_def equiv_states'_alt_def
  unfolding N_def T_def by simp

lemma init_states_defs'[intro]:
  init ∈ defs'.defs.states' s₀
  using init_states by simp

lemma
  defs'.I = equiv.I
  by simp

lemma PF_PF[simp]:
  defs'.PF = equiv.PF
  apply simp
  unfolding stripfp_def
  apply (rule ext)
  apply clarsimp
  subgoal for x
    apply (cases equiv.P x)
     apply simp
    subgoal for a
      by (cases a) auto
    done
  done

lemma PF_PROG[simp]:
  equiv.PF = stripfp PROG
  unfolding N_def by simp

lemma I_simp[simp]:
  (equiv.I ! q) l = pred ! q ! l if q < p
  unfolding N_def P_def using ‹q < p› process_length(3) by simp

lemma
  defs'.P = conv_prog PROG
  by (simp add: N_def)

lemma states_len[intro]:
  assumes
    q < p L ∈ equiv.defs.states' s
  shows
    L ! q < length (trans ! q)
  using assms unfolding Product_TA_Defs.states_def
  apply simp
  unfolding trans_of_def equiv.defs.N_s_def
  apply (simp add: T_s_unfold_1[simplified] T_s_unfold_2[simplified])
  unfolding N_def
  apply simp
  unfolding T_def
    using state_set
  unfolding process_length(2)[symmetric]
  apply safe
```

```
    apply (erule allE)
    apply (erule impE)
     apply assumption
    apply safe
    by (clarsimp dest!: nth_mem; force)+

end


locale UPPAAL_Reachability_Problem_precompiled_ceiling =
  UPPAAL_Reachability_Problem_precompiled +
  fixes k :: 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:
    ∀ i < p. ∀ l < length (trans ! i). ∀ (g, a, r, l') ∈ set (trans ! i ! l).
      guaranteed_execution_cond prog r max_steps

begin

definition k_fun l c ≡ if c > 0 ∧ c ≤ m then Max {k ! i ! (fst l ! i) ! c | i . i < p} else 0


lemma p_p':
  equiv.p = p
  by simp

lemma clkp_set_clk_set_subs:
  fst ‘ clkp_set A (l, s) ⊆ clk_set A
  unfolding TA_clkp_set_unfold by auto

lemma k_ceiling_1:
  ∀ l. ∀(x,m) ∈ clkp_set A l. m ≤ k_fun l x

  apply safe
  subgoal premises prems for l s x d
  proof −
    from ‹(x, d) ∈ _› have 0 < x x ≤ m
      using clkp_set_clk_set_subs[of l s] clk_set by force+
    from prems show ?thesis
      unfolding clkp_set_def
      apply safe
      subgoal
        unfolding collect_clki_def
```

```
      unfolding inv_of_def
      unfolding equiv.defs.prod_ta_def
      unfolding equiv.defs.prod_invariant_def
      unfolding inv_of_def
        Product_TA_Defs.product_ta_def
        Product_TA_Defs.product_invariant_def
        equiv.defs.N_s_def
      unfolding length_N
      unfolding equiv.state_ta_def
      unfolding p_p'
      unfolding equiv.state_inv_def
      unfolding N_def
      unfolding collect_clock_pairs_def
      apply (clarsimp cong: if_cong simp: I_def)
      subgoal premises prems for l' i
      proof −
        have nat d ≤ k ! i ! (l ! i) ! x
          using prems lengths k_ceiling(2)
          unfolding collect_clock_pairs_def
          by (auto 4 4)
        also from ‹_ < p› have ... ≤ Max {k ! i ! (l ! i) ! x |i. i < p}
          by (auto intro: Max_ge)
        finally show ?thesis
          unfolding k_fun_def using ‹0 < x› ‹x ≤ m› by auto
      qed
      done
    subgoal
      unfolding collect_clkt_def
      apply clarsimp
      subgoal premises prems for g a r l' s'
      proof −
        from guard_approx[OF prems(2)] prems(1) obtain i g a r where ∗:
          (x, d) ∈ clkp_set'' i (l ! i) (g, a, r, l' ! i) ∈ set (trans ! i ! (l ! i))
          l ∈ equiv.defs.states' s i < p
          by auto
        from ‹i < p› ‹l ∈ _› have l ! i < length (trans ! i)
          by auto
        with k_ceiling(1) ∗ have nat d ≤ k ! i ! (l ! i) ! x
          by force
        also from ‹_ < p› have ... ≤ Max {k ! i ! (l ! i) ! x |i. i < p}
          by (auto intro: Max_ge)
        finally show ?thesis
          unfolding k_fun_def using ‹0 < x› ‹x ≤ m› by auto
      qed
      done
    done
  qed
done

lemma k_ceiling_2:
   ∀ l g a r l' c. A ⊢ l ⟶^{g,a,r} l' ∧ c ∉ set r ⟶ k_fun l' c ≤ k_fun l c
  unfolding trans_of_def equiv.defs.prod_ta_def equiv.defs.prod_trans_def
  apply clarsimp
  apply safe
```

```
subgoal premises prems for l s g a r l' s' c
proof −
 from prems obtain p' l'' pc_g pc_u where ∗:
  p' < p l' = l[p' := l'']
  r = equiv.make_f pc_u s Some s' = equiv.make_mf pc_u s
  (l ! p', pc_g, Sil a, pc_u, l'') ∈ fst (equiv.N ! p')
  l ∈ equiv.defs.states' s
  apply atomize_elim
  unfolding equiv.defs.prod_trans_i_def
 unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def trans_of_def
  apply clarsimp
  apply safe
  unfolding Product_TA_Defs.product_trans_i_def
  unfolding trans_of_def
   apply clarsimp
  unfolding equiv.defs.N_s_def
  unfolding equiv.defs.T_s_def
  unfolding Equiv_TA_Defs.state_ta_def
  unfolding equiv.state_trans_t_def
  unfolding Product_TA_Defs.product_trans_s_def
  by auto
 from ‹l ∈ _› have [simp]: length l = p
  by simp
 from ‹r = _› have fst ‘ collect_store'' pc_u ⊆ set r
  supply find_resets_start.simps[simp del]
  unfolding collect_store''_def equiv.make_f_def
  apply (clarsimp split: option.split_asm)
  subgoal
   using ‹Some s' = _› unfolding equiv.make_mf_def
   by (auto split: option.split_asm)
  subgoal premises prems for pc' g st f pcs c d pc_t pc''
  proof −
   from prems have
    steps (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max_steps
     (pc_u, [], s, True, []) (pc', g, st, f, r)
    prog ! pc' = Some (INSTR HALT)
    unfolding PF_unfold stripfp_def N_def PROG_def
    by (auto dest!: exec_steps split: if_split_asm elim!: stripf.elims)
   with prems show ?thesis
    by (force intro: sym dest!: resets_start')
  qed
  done
 with ‹c ∉ _› have c ∉ fst ‘ collect_store'' pc_u by blast
 show ?thesis
 proof (cases c > m)
  case True
  then show ?thesis
   unfolding k_fun_def by auto
 next
  case False
  with ‹c ∉ fst ‘ _ _› have c ∈ {0..<m+1} − fst ‘ collect_store'' pc_u
   by auto
  from ∗ have (l ! p') < length (trans ! p')
   unfolding N_def T_def by auto
```

```
    from ∗ have (pc_g, Sil a, pc_u, l'') ∈ set (trans ! p' ! (l ! p'))
      (l ! p') < length (trans ! p')
      unfolding N_def T_def by auto
    with k_resets ‹c ∈ _› ‹p' < _› have k ! p' ! l'' ! c ≤ k ! p' ! (l ! p') ! c
      unfolding k_fun_def by force
    with ‹l' = _› show ?thesis
      unfolding k_fun_def
      apply clarsimp
      apply (rule Max.boundedI)
       apply force
      using p_gt_0 apply force
      apply clarsimp
      subgoal for i
       apply (cases i = p')
        apply simp
        apply (rule le_trans)
       by (auto intro: Max_ge)
      done
  qed
qed
subgoal premises prems for l s g a r l' s' c
proof −
  from prems obtain p1 l1 pc_g1 pc_u1 p2 l2 pc_g2 pc_u2 s'' where ∗:
    p1 < p p2 < p l' = l[p1 := l1, p2 := l2]
    r = equiv.make_f pc_u1 s @ equiv.make_f pc_u2 s
    Some s' = equiv.make_mf pc_u1 s'' Some s'' = equiv.make_mf pc_u2 s
    (l ! p1, pc_g1, In a, pc_u1, l1) ∈ fst (equiv.N ! p1)
    (l ! p2, pc_g2, Out a, pc_u2, l2) ∈ fst (equiv.N ! p2)
    l ∈ equiv.defs.states' s
    apply atomize_elim
    unfolding equiv.defs.prod_trans_s_def
   unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def trans_of_def
    apply clarsimp
    apply safe
    subgoal
      unfolding Product_TA_Defs.product_trans_i_def
      by auto
    unfolding Product_TA_Defs.product_trans_s_def
    unfolding trans_of_def
    apply clarsimp
    unfolding equiv.defs.N_s_def
    unfolding equiv.defs.T_s_def
    unfolding Equiv_TA_Defs.state_ta_def
    unfolding equiv.state_trans_t_def
    apply clarsimp
    by blast
  from ‹l ∈ _› have [simp]: length l = p
    by simp
  from ∗ have ∗∗:
    (pc_g1, In a, pc_u1, l1) ∈ set (trans ! p1 ! (l ! p1)) (l ! p1) < length (trans ! p1)
    (pc_g2, Out a, pc_u2, l2) ∈ set (trans ! p2 ! (l ! p2)) (l ! p2) < length (trans ! p2)
    unfolding N_def T_def by auto
  with ‹p1 < p› guaranteed_resets have guaranteed_execution:
    guaranteed_execution_cond prog pc_u1 max_steps
```

```
  by blast
from ‹r = _› have fst ‘ collect_store'' pc_u1 ⊆ set r
  supply find_resets_start.simps[simp del]
  unfolding collect_store''_def
    equiv.make_f_def
  apply (clarsimp split: option.split_asm)
  subgoal
    using ‹Some s'' = _› unfolding equiv.make_mf_def
    by (auto split: option.split_asm)
  subgoal
    using ‹Some s'' = _› unfolding equiv.make_mf_def
    by (auto split: option.split_asm)
  subgoal
    using ‹Some s' = _› unfolding equiv.make_mf_def
    using guaranteed_execution'[OF guaranteed_execution, of [] s True [] []]
    unfolding stripfp_def PROG_def by auto
  subgoal premises prems for _ _ _ _ r2 _ pc' g st f r1 pcs c d pc_t pc''
  proof −
    from prems have
      steps (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max_steps
        (pc_u1, [], s, True, []) (pc', g, st, f, r1)
      prog ! pc' = Some (INSTR HALT) r = r1 @ r2
      unfolding PF_unfold stripfp_def N_def PROG_def
      by (auto dest!: exec_steps split: if_split_asm elim!: stripf.elims)
    with prems show ?thesis
      by (force intro: sym dest!: resets_start')
  qed
  done
moreover from ‹r = _› have fst ‘ collect_store'' pc_u2 ⊆ set r
  supply find_resets_start.simps[simp del]
  unfolding collect_store''_def
    equiv.make_f_def
  apply (clarsimp split: option.split_asm)
  subgoal
    using ‹Some s'' = _› unfolding equiv.make_mf_def
    by (auto split: option.split_asm)
  subgoal
    using ‹Some s'' = _› unfolding equiv.make_mf_def
    by (auto split: option.split_asm)
  subgoal
    using ‹Some s' = _› unfolding equiv.make_mf_def
    using guaranteed_execution'[OF guaranteed_execution, of [] s True [] []]
    unfolding stripfp_def PROG_def by auto
  subgoal premises prems for pc' g st f r1 pcs _ _ _ _ r2 _ c d pc_t pc''
  proof −
    from prems have
      steps (map_option stripf o (λpc. if pc < size prog then prog ! pc else None)) max_steps
        (pc_u2, [], s, True, []) (pc', g, st, f, r1)
      prog ! pc' = Some (INSTR HALT) r = r2 @ r1
      unfolding PF_unfold stripfp_def N_def PROG_def
      by (auto dest!: exec_steps split: if_split_asm elim!: stripf.elims)
    with prems show ?thesis
      by (force intro: sym dest!: resets_start')
  qed
```

```
    done
  ultimately have c_not_elem: c ∉ fst ‘ collect_store'' pc_u1 c ∉ fst ‘ collect_store'' pc_u2
    using ‹c ∉ _› by auto
  show ?thesis
  proof (cases c > m)
    case True
    then show ?thesis
      unfolding k_fun_def by auto
  next
    case False
    with c_not_elem have
      c ∈ {0..<m+1} − fst ‘ collect_store'' pc_u1
      c ∈ {0..<m+1} − fst ‘ collect_store'' pc_u2
      by auto
    with ∗∗ k_resets ‹p1 < _› ‹p2 < _› have
      k ! p1 ! l1 ! c ≤ k ! p1 ! (l ! p1) ! c k ! p2 ! l2 ! c ≤ k ! p2 ! (l ! p2) ! c
      by (auto split: prod.split_asm)
    with ‹l' = _› show ?thesis
      unfolding k_fun_def
      apply clarsimp
      apply (rule Max.boundedI)
        apply force
      using p_gt_0 apply force
      apply clarsimp
      subgoal for i
        apply (cases i = p2)
        subgoal
          apply simp
          apply (rule le_trans)
          by (auto intro: Max_ge)
        apply (cases i = p1)
         apply simp
         apply (rule le_trans)
        by (auto intro: Max_ge)
      done
  qed
 qed
 done

lemma
 shows k_ceiling':
   ∀ l. ∀(x,m) ∈ clkp_set A l. m ≤ k_fun l x
   ∀ l g a r l' c. A ⊢ l ⟶^{g,a,r} l' ∧ c ∉ set r ⟶ k_fun l' c ≤ k_fun l c
 and k_bound':
   ∀ l. ∀ i > m. k_fun l i = 0
 and k_0':
   ∀ l. k_fun l 0 = 0
 using k_ceiling_1 k_ceiling_2 unfolding k_fun_def by auto

sublocale Reachability_Problem A (init, s0) m k_fun PR_CONST (λ (l, s). F l s)
 by (standard; rule k_ceiling' k_bound' k_0')

end
```

locale *UPPAAL_Reachability_Problem_precompiled_start_state* =
UPPAAL_Reachability_Problem_precompiled _ _ _ _ *pred*
for *pred* :: *nat list list* +
fixes s_0 :: *int list*
assumes *start_pred*:
$\forall\ q < p.\ \exists\ pc\ st\ s'\ rs\ pcs.$
exec (*stripfp PROG*) *max_steps* ((*pred* ! *q* ! (*init* ! *q*)), [], s_0, *True*, []) []
$=$ *Some* ((*pc*, *st*, s', *True*, *rs*), *pcs*)
and *bounded*: *bounded bounds* s_0
and *pred_time_indep*: $\forall\ x \in$ *set pred*. $\forall\ pc \in$ *set x*. *time_indep_check prog pc max_steps*
and *upd_time_indep*:
$\forall\ T \in$ *set trans*. $\forall\ xs \in$ *set T*. $\forall\ (_, _, pc_u, _) \in$ *set xs*.
time_indep_check prog pc_u max_steps
and *clock_conj*:
$\forall\ T \in$ *set trans*. $\forall\ xs \in$ *set T*. $\forall\ (pc_g, _, _, _) \in$ *set xs*.
conjunction_check prog pc_g max_steps
begin

lemma *B0*[*intro*]:
bounded defs'.B s_0
using *bounded* **unfolding** *bounded_def N_def* **by** *simp*

lemma *equiv_P_simp*:
equiv.P = *PROG*
unfolding *N_def* **by** *simp*

lemma *time_indep_P*[*intro*]:
time_indep (*conv_prog equiv.P*) *max_steps* (*pred* ! *q* ! (*L* ! *q*), [], *s*, *True*, [])
if *q* < *p* *L* ∈ *equiv.defs.states'* *s*
proof −
from *that lengths process_length* **have** *q* < *length pred L* ! *q* < *length* (*pred* ! *q*) **by** *auto*
then have *pred* ! *q* ∈ *set pred pred* ! *q* ! (*L* ! *q*) ∈ *set* (*pred* ! *q*) **by** *auto*
with *pred_time_indep time_indep_overapprox* **show** *?thesis*
by (*auto simp*: *PROG_def equiv_P_simp*)
qed

lemma *time_indep_PROG*[*intro*]:
time_indep (*conv_prog PROG*) *max_steps* (*pc_u*, [], *s*, *True*, [])
if *q* < *p* (*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *T q*
proof −
from *that lengths process_length* **have**
q < *length trans l* < *length* (*trans* ! *q*) (*pc_g*, *a*, *pc_u*, *l'*) ∈ *set* (*trans* ! *q* ! *l*)
unfolding *T_def* **by** *auto*
moreover then have *trans* ! *q* ∈ *set trans trans* ! *q* ! *l* ∈ *set* (*trans* ! *q*) **by** *auto*
ultimately show *?thesis* **using** *upd_time_indep time_indep_overapprox*
unfolding *PROG_def* **by** *blast*
qed

lemma *facts*[*intro*]:
u ⊢$_a$ *ac* **if**
q < *defs'.p*
(*l*, *pc_g*, *a*, *pc_u*, *l'*) ∈ *fst* (*defs'.N* ! *q*)
stepst defs'.P max_steps u (*pc_g*, [], *s*, *True*, []) (*pc_t*, *st_t*, *s_t*, *True*, *rs_t*)

```
    stepsc defs'.P max_steps u (pc_g, [], s, True, []) (pc', st, s', f', rs)
    defs'.P pc' = Some (CEXP ac)
  proof −
    let ?P = conv_P prog
    from that(5) obtain ac' where
      ac = conv_ac ac' prog ! pc' = Some (CEXP ac') pc' < length prog
      apply (clarsimp split: option.split_asm if_split_asm simp add: PROG_def N_def)
      subgoal for z
        by (cases z) auto
      done
    with that have u ⊢_a conv_ac ac'
      apply −
      apply (rule conjunction_check)
      using clock_conj apply simp_all
      unfolding N_def apply simp_all
     using lengths process_length(2) by (force dest!: nth_mem simp: PROG_def N_def T_def)+
    with ‹ac = _› show ?thesis by simp
  qed

  sublocale product': 
    Equiv_TA conv N max_steps init s_0
    apply standard
         apply standard
        apply (simp; blast)
        subgoal
          apply clarsimp
          apply (force simp: N_def)
          done
       apply blast
      apply (simp; fail)
    unfolding PF_PF using start_pred apply simp
    by (rule B0)

  lemma fst_S[simp]:
    fst ‘ (λ(l, g, a, r, l'). (l, map conv_ac g, a, r, l')) ‘ S = fst ‘ S
    by force

  lemma snd_S[simp]:
    (snd ∘ snd ∘ snd ∘ snd) ‘ (λ(l, g, a, r, l'). (l, map conv_ac g, a, r, l')) ‘ S
    = (snd ∘ snd ∘ snd ∘ snd) ‘ S
    by force

end

locale UPPAAL_Reachability_Problem_precompiled' =
  UPPAAL_Reachability_Problem_precompiled_start_state +
  UPPAAL_Reachability_Problem_precompiled_defs' +
  UPPAAL_Reachability_Problem_precompiled_ceiling +
  assumes action_set:
    ∀ T ∈ set trans. ∀ xs ∈ set T. ∀ (_, a, _) ∈ set xs. pred_act (λ a. a < na) a
begin


  sublocale Reachability_Problem_Impl_Defs _ _ A (init, s_0) m k_fun PR_CONST (λ (l, s).
```

F l s) .

definition
states' = {(*L*, *s*). *L* ∈ *equiv.defs.states' s* ∧ *check_pred L s* ∧ *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*) ∈ *set* (*IArray* (*map IArray trans_in_map*) !! *q* !! *l*)
using *assms process_length*(*2*) **unfolding** *trans_in_map_def*
by (*force dest*: *nth_mem intro*!: *image_eqI*[**where** *x* = (*g1*, *In a*, *r1*)])

lemma *in_trans_out_mapI*:
assumes
q < *p l* < *length* (*trans* ! *q*) *i* < *length* (*trans* ! *q* ! *l*)
(*g1*, *Out a*, *r1*) = *trans* ! *q* ! *l* ! *i*
shows (*g1*, *a*, *r1*) ∈ *set* (*IArray* (*map IArray trans_out_map*) !! *q* !! *l*)
using *assms process_length*(*2*) **unfolding** *trans_out_map_def*
by (*force dest*: *nth_mem intro*!: *image_eqI*[**where** *x* = (*g1*, *Out a*, *r1*)])

lemma *in_trans_in_mapD*:
assumes
(*g1*, *a*, *r1*) ∈ *set* (*IArray* (*map IArray trans_in_map*) !! *q* !! *l*)
q < *p l* < *length* (*trans* ! *q*)
obtains *i* **where**
i < *length* (*trans* ! *q* ! *l*) ∧ *trans* ! *q* ! *l* ! *i* = (*g1*, *In a*, *r1*)
using *assms process_length*(*2*) **unfolding** *trans_in_map_def*
by (*fastforce dest*: *mem_nth split*: *act.split_asm*)

lemma *in_trans_out_mapD*:
assumes
(*g1*, *a*, *r1*) ∈ *set* (*IArray* (*map IArray trans_out_map*) !! *q* !! *l*)
q < *p l* < *length* (*trans* ! *q*)
obtains *i* **where**
i < *length* (*trans* ! *q* ! *l*) ∧ *trans* ! *q* ! *l* ! *i* = (*g1*, *Out a*, *r1*)
using *assms process_length*(*2*) **unfolding** *trans_out_map_def*
by (*fastforce dest*: *mem_nth split*: *act.split_asm*)

lemma *in_actions_by_stateI*:
assumes
(*g1*, *a*, *r1*) ∈ *set xs a* < *length acc*
shows
(*q*, *g1*, *a*, *r1*) ∈ *set* (*actions_by_state q xs acc* ! *a*)
∧ *a* < *length* (*actions_by_state q xs acc*)
using *assms* **unfolding** *actions_by_state_def*
apply (*induction xs arbitrary*: *acc*)
apply (*simp*; *fail*)
apply *simp*
apply (*erule disjE*)
apply (*rule fold_acc_preserv*
[**where** *P* = λ *acc*. (*q*, *g1*, *a*, *r1*) ∈ *set* (*acc* ! *a*) ∧ *a* < *length acc*]
)
apply (*subst list_update_nth_split*; *auto*)

by *auto*

lemma *in_actions_by_state_preserv*:
assumes
$(q, g1, a, r1) \in set\ (acc\ !\ a)\ a < length\ acc$
shows
$(q, g1, a, r1) \in set\ (actions_by_state\ y\ xs\ acc\ !\ a)$
$\wedge\ a < length\ (actions_by_state\ y\ xs\ acc)$
using *assms* **unfolding** *actions_by_state_def*
apply −
apply (*rule fold_acc_preserv*
[**where** $P = \lambda\ acc.\ (q, g1, a, r1) \in set\ (acc\ !\ a) \wedge a < length\ acc$]
)
apply (*subst list_update_nth_split*; *auto*)
by *auto*

lemma *length_actions_by_state_preserv*[*simp*]:
shows *length* (*actions_by_state y xs acc*) = *length acc*
unfolding *actions_by_state_def* **by** (*auto intro*: *fold_acc_preserv simp*: *list_update_nth_split*)

lemma *in_all_actions_by_stateI*:
assumes
$a < na\ q < p\ (g1, a, r1) \in set\ (M\ !!\ q\ !!\ (L\ !\ q))$
shows
$(q, g1, a, r1) \in set\ (all_actions_by_state\ M\ L\ !\ a)$
unfolding *all_actions_by_state_def*
apply (*rule fold_acc_ev_preserv*
[**where** $P = \lambda\ acc.\ (q, g1, a, r1) \in set\ (acc\ !\ a)$ **and** $Q = \lambda\ acc.\ a < length\ acc$,
THEN conjunct1]
)
apply (*rule in_actions_by_state_preserv*[*THEN conjunct1*])
using *assms* **by** (*auto intro*: *in_actions_by_stateI*[*THEN conjunct1*])

lemma *actions_by_state_inj*:
assumes $j < length\ acc$
shows $\forall\ (q, a) \in set\ (actions_by_state\ i\ xs\ acc\ !\ j).\ (q, a) \notin set\ (acc\ !\ j) \longrightarrow i = q$
unfolding *actions_by_state_def*
apply (*rule fold_acc_preserv*
[**where** $P =$
$\lambda\ acc'.\ (\forall\ (q, a) \in set\ (acc'\ !\ j).\ (q, a) \notin set\ (acc\ !\ j) \longrightarrow i = q) \wedge j < length\ acc'$,
THEN conjunct1])
subgoal for *x acc*
by (*cases fst* (*snd x*) = *j*; *simp*)
using *assms* **by** *auto*

lemma *actions_by_state_inj'*:
assumes $j < length\ acc\ (q, a) \notin set\ (acc\ !\ j)\ (q, a) \in set\ (actions_by_state\ i\ xs\ acc\ !\ j)$
shows $i = q$
using *actions_by_state_inj*[*OF assms*(*1*)] *assms*(*2*−) **by** *fast*

lemma *in_actions_by_stateD*:
assumes
$(q, g, a, t) \in set\ (actions_by_state\ i\ xs\ acc\ !\ j)\ (q, g, a, t) \notin set\ (acc\ !\ j)$
$j < length\ acc$

shows
(*g*, *a*, *t*) ∈ *set xs* ∧ *j* = *a*
using *assms* **unfolding** *actions_by_state_def*
apply −
apply (*drule fold_evD*
[**where** *y* = (*g*, *a*, *t*) **and** *Q* = λ *acc'*. *length acc'* = *length acc*
and *R* = λ (_, *a'*, *t*). *a'* = *j*]
)
apply *assumption*
apply (*subst* (*asm*) *list_update_nth_split*[*of j*]; *force*)
apply *simp*+
apply (*subst* (*asm*) *list_update_nth_split*[*of j*]; *force*)
by *auto*

lemma *in_all_actions_by_stateD*:
assumes
(*q*, *g1*, *a*, *r1*) ∈ *set* (*all_actions_by_state M L* ! *a'*) *a'* < *na*
shows
(*g1*, *a*, *r1*) ∈ *set* (*M* !! *q* !! (*L* ! *q*)) ∧ *q* < *p* ∧ *a'* = *a*
using *assms*
unfolding *all_actions_by_state_def*
apply −
apply (*drule fold_evD''*[**where** *y* = *q* **and** *Q* = λ *acc*. *length acc* = *na*])
apply (*simp*; *fail*)
apply (*drule actions_by_state_inj'*[*rotated*])
apply (*simp*; *fail*)+
apply *safe*
apply (*drule in_actions_by_stateD*)
apply *assumption*
apply (*rule fold_acc_preserv*)
apply (*simp*; *fail*)+
subgoal premises *prems*
proof −
from *prems*(*2*) **have** *q* ∈ *set* [*0*..<*p*] **by** *auto*
then show *?thesis* **by** *simp*
qed
by (*auto intro*: *fold_acc_preserv dest*!: *in_actions_by_stateD*)

lemma *length_all_actions_by_state_preserv*:
length (*all_actions_by_state M L*) = *na*
unfolding *all_actions_by_state_def* **by** (*auto intro*: *fold_acc_preserv*)

lemma *less_naI*:
assumes
q < *p*
(*g1*, *a*, *r1*) = *trans* ! *q* ! *l* ! *j*
l < *length* (*trans* ! *q*)
j < *length* (*trans* ! *q* ! *l*)
shows *pred_act* (λ*a*. *a* < *na*) *a*
using *action_set assms process_length*(*2*) **by** (*force dest*!: *nth_mem*)

lemma *in_actions_trans_in_mapI*:
assumes
pa < *p*

```
    (g1, In a, r1) = trans ! pa ! (L ! pa) ! j
    L ! pa < length (trans ! pa)
    j < length (trans ! pa ! (L ! pa))
  shows (pa, g1, a, r1) ∈ set (all_actions_by_state (IArray (map IArray trans_in_map)) L !
a)
  apply (rule in_all_actions_by_stateI)
  using assms action_set process_length(2) apply (fastforce dest!: nth_mem)
  using assms by (fastforce intro: in_trans_in_mapI)+

lemma in_actions_trans_out_mapI:
  assumes
    pa < p
    (g1, Out a, r1) = trans ! pa ! (L ! pa) ! j
    L ! pa < length (trans ! pa)
    j < length (trans ! pa ! (L ! pa))
  shows (pa, g1, a, r1) ∈ set (all_actions_by_state (IArray (map IArray trans_out_map)) L
! a)
  apply (rule in_all_actions_by_stateI)
  using assms action_set process_length(2) apply (fastforce dest!: nth_mem)
  using assms by (fastforce intro: in_trans_out_mapI)+

lemma in_pairs_by_actionD2:
  assumes
    (g, a, r, L', s') ∈ set (pairs_by_action (L, s) xs ys)
    ∀ (q, g, a'', m, l) ∈ set xs. a'' = a'
    ∀ (q, g, a'', m, l) ∈ set ys. a'' = a'
  shows check_pred L' s'
  using assms(1) unfolding pairs_by_action_def using assms(2,3)
  by (clarsimp split: option.split_asm simp: set_map_filter) (clarsimp split: if_split_asm)

lemma in_pairs_by_actionD1:
  assumes
    (g, a, r, L', s') ∈ set (pairs_by_action (L, s) xs ys)
    ∀ (q, g, a'', m, l) ∈ set xs. a'' = a'
    ∀ (q, g, a'', m, l) ∈ set ys. a'' = a'
  obtains
    pa q pc_g1 pc_g2 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
    x1 x2 x3 x4 x5 y1 y2 y3 y4
  where
    pa ≠ q
    (pa, pc_g1, a, pc_u1, l1') ∈ set ys
    (q, pc_g2, a, pc_u2, l2') ∈ set xs
    L' = L[pa := l1', q := l2']
    runf pc_u1 s1 = Some ((x1, x2, s', x3, x4), x5)
    runf pc_u2 s = Some ((y1, y2, s1, y3, r2), y4)
    check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
    r1 = make_reset pc_u1 s
    g = g1 @ g2 r = r1 @ r2
proof −
  obtain
    pa q pc_g1 pc_g2 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
    x1 x2 x3 x4 x5 y1 y2 y3 y4
    where
    (q, pc_g1, a, pc_u1, l1') ∈ set ys (pa, pc_g2, a, pc_u2, l2') ∈ set xs q ≠ pa
```

check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
runf pc_u1 s1 = Some ((x1, x2, s', x3, x4), x5)
runf pc_u2 s = Some ((y1, y2, s1, y3, r2), y4)
r1 = make_reset pc_u1 s
Some (g1 @ g2, a, r1 @ r2, L[q := l1', pa := l2'], s') = Some (g, a, r, L', s')
proof −
from *assms(1)* **show** *?thesis*
unfolding *pairs_by_action_def* **using** *assms(2,3)*
apply *clarsimp*
unfolding *set_map_filter*
apply *clarsimp*
apply (*clarsimp split*: *option.split_asm if_split_asm*)
by (*force intro*!: *that*)
qed
then show *?thesis* **by** (*fast intro*: *that*)
qed

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

using *in_pairs_by_actionD1*[*OF assms*] *in_pairs_by_actionD2*[*OF assms*] **by** *metis*

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*)
using *assms* **unfolding** *trans_fun_def* **by** *auto*

lemma *states'_states'*[*intro*]:
$L \in$ *equiv.defs.states' s* **if** $(L, s) \in$ *states'*
using *that* **unfolding** *states'_def* **by** *auto*

lemma *bounded'_bounded*:
bounded' s $\longleftrightarrow$ *bounded bounds s* **if** *length s = length bounds*
using *that* **unfolding** *bounded'_def bounded_def* **by** *simp*

lemma *bounded_bounded'*:
bounded bounds s $\Longrightarrow$ *bounded' s*
unfolding *bounded'_def bounded_def* **by** *simp*

lemma *P_unfold*:
($\forall$ *q*<*p*. (*equiv.defs.P* ! *q*) (*L* ! *q*) *s*) $\longleftrightarrow$ (*check_pred L s*) **if** *length s* = *length bounds*
unfolding *equiv.state_ta_def equiv.state_pred_def check_pred_def* **using** *process_length*(*3*)
that
apply *simp*
unfolding *list_all_iff*
unfolding *N_def*
unfolding *runf_def P_def*
by (*fastforce split*: *option.splits simp*: *bounded'_bounded*)

lemma *P_unfold_1*:
($\forall$ *q*<*p*. (*equiv.defs.P* ! *q*) (*L* ! *q*) *s*) $\Longrightarrow$ (*check_pred L s*)
unfolding *equiv.state_ta_def equiv.state_pred_def check_pred_def* **using** *process_length*(*3*)
apply *simp*
unfolding *list_all_iff*
unfolding *N_def*
unfolding *runf_def P_def*
by (*fastforce split*: *option.split_asm simp*: *bounded_bounded'*)

lemma *equiv_PT*[*simp*]:
equiv.PT = *PT*
unfolding *striptp_def N_def* **by** *simp*

lemmas [*simp*] = *equiv_P_simp*

lemma *transD*:
assumes
(*pc_g*, *a*, *pc_u*, *l'*) = *trans* ! *q* ! (*L* ! *q*) ! *j*
L ! *q* < *length* (*trans* ! *q*) *j* < *length* (*trans* ! *q* ! (*L* ! *q*))
q < *p*
shows (*L* ! *q*, *pc_g*, *a*, *pc_u*, *l'*) $\in$ *fst* (*equiv.N* ! *q*)
using *assms* **unfolding** *N_def T_def* **by** *simp solve_ex_triv*

lemma *trans_ND*:
assumes
(*L* ! *q*, *pc_g*, *a*, *pc_u*, *l'*) $\in$ *fst* (*equiv.N* ! *q*)
q < *p*
shows
equiv.defs.N_s s ! *q* $\vdash$ *L* ! *q* $\longrightarrow^{equiv.make_g\ pc_g\ s,(a,\ equiv.make_c\ pc_g,\ equiv.make_mf\ pc_u),equiv.make_f\ pc_u\ s}$ *l'*
using *assms*
unfolding *equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def*
unfolding *equiv.state_ta_def equiv.state_trans_t_def*
by *clarsimp solve_ex_triv+*

lemma *make_f_unfold*:
equiv.make_f pc s = *make_reset pc s*
unfolding *make_reset_def equiv.make_f_def runf_def* **by** *simp*

lemma *make_g_simp*:

```
  assumes check_g pc_g s = Some g1
  shows g1 = equiv.make_g pc_g s
  using assms unfolding check_g_def equiv.make_g_def runt_def
  by (clarsimp split: option.splits bool.splits simp: make_cconstr_def)

lemma make_c_simp:
  assumes check_g pc_g s = Some g1
  shows equiv.make_c pc_g s
  using assms unfolding check_g_def equiv.make_c_def runt_def
  by (clarsimp split: option.splits bool.splits simp: make_cconstr_def)

lemma make_reset_simp:
  assumes runf pc_u s = Some ((y1, y2, s1, y3, r2), y4)
  shows make_reset pc_u s = r2
  using assms unfolding runf_def make_reset_def by (auto split: option.splits)

lemma make_mf_simp:
  assumes runf pc_u s = Some ((y1, y2, s1, y3, r2), y4)
  shows equiv.make_mf pc_u s = Some s1
  using assms unfolding runf_def equiv.make_mf_def by (auto split: option.splits)

lemma trans_fun_trans_of':
  (trans_fun, trans_of A) ∈ transition_rel states'
  unfolding transition_rel_def T_def
  apply simp
  unfolding trans_of_def
  apply safe
  subgoal for L s g a r L' s'
    unfolding equiv.defs.prod_ta_def equiv.defs.prod_trans_def
    apply simp
    apply safe
    subgoal
      apply (rule trans_i_fun_trans_fun)
      unfolding equiv.defs.prod_trans_i_alt_def
      apply safe
        unfolding trans_fun_def trans_i_from_def trans_i_fun_def
        unfolding Product_TA_Defs.product_trans_i_def
        apply clarsimp
        subgoal premises prems for c m p' l'
        proof −
          from prems have L ! p' < length (trans ! p') by auto
          from prems obtain pc_g pc_u where
            (L ! p', pc_g, Sil a, pc_u, l') ∈ T p'
            g = equiv.make_g pc_g s r = equiv.make_f pc_u s
            c = equiv.make_c pc_g m = equiv.make_mf pc_u
            unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
            unfolding equiv.state_ta_def equiv.state_trans_t_def
            apply clarsimp
            unfolding N_def T_def by clarsimp
          from this(1) have (pc_g, Sil a, pc_u, l') ∈ set (trans ! p' ! (L ! p'))
            unfolding T_def by auto
          moreover have check_g pc_g s = Some g
            using ‹g = _› ‹c = _› ‹c s›
              unfolding check_g_def equiv.make_g_def equiv.make_c_def
```

```
        by (auto split: option.splits simp: runt_def make_cconstr_def)
      moreover obtain x1 x2 x3 pcs where runf pc_u s = Some ((x1, x2, s', x3, r), pcs)
        using ‹r = _› ‹m = _› prems(5)
        unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
        by (auto split: option.splits)
      moreover have check_pred (L[p' := l']) s'
        using prems(3) by (auto intro: P_unfold_1)
      ultimately show ?thesis using process_length(2) ‹p' < _› ‹L ! p' < _›
        by (force simp: set_map_filter trans_i_map_def)
    qed
    done
  subgoal
    apply (rule trans_s_fun_trans_fun)
      unfolding equiv.defs.prod_trans_s_alt_def
      apply safe
    unfolding trans_fun_def trans_s_fun_def
    unfolding Product_TA_Defs.product_trans_s_def
    apply clarsimp
    subgoal premises prems for ci co mi mo s1 q1 q2 g1 g2 r1 r2 l1' l2'
    proof −
      from prems have L ! q1 < length (trans ! q1) L ! q2 < length (trans ! q2) by auto
      from prems obtain pc_g1 pc_u1 where
        (L ! q1, pc_g1, In a, pc_u1, l1') ∈ T q1
        g1 = equiv.make_g pc_g1 s r1 = equiv.make_f pc_u1 s
        ci = equiv.make_c pc_g1 mi = equiv.make_mf pc_u1
        unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
        unfolding equiv.state_ta_def equiv.state_trans_t_def
        apply clarsimp
        unfolding N_def T_def by clarsimp
      from prems obtain pc_g2 pc_u2 where
        (L ! q2, pc_g2, Out a, pc_u2, l2') ∈ T q2
        g2 = equiv.make_g pc_g2 s r2 = equiv.make_f pc_u2 s
        co = equiv.make_c pc_g2 mo = equiv.make_mf pc_u2
        unfolding equiv.defs.N_s_def trans_of_def equiv.defs.T_s_def
        unfolding equiv.state_ta_def equiv.state_trans_t_def
        apply clarsimp
        unfolding N_def T_def by clarsimp
      from ‹_ ∈ T q1› have (pc_g1, In a, pc_u1, l1') ∈ set (trans ! q1 ! (L ! q1))
        unfolding T_def by auto
      from ‹_ ∈ T q2› have (pc_g2, Out a, pc_u2, l2') ∈ set (trans ! q2 ! (L ! q2))
        unfolding T_def by auto
      moreover have check_g pc_g1 s = Some g1 check_g pc_g2 s = Some g2
        using ‹g1 = _› ‹ci = _› ‹ci s› ‹g2 = _› ‹co = _› ‹co s›
          unfolding check_g_def equiv.make_g_def equiv.make_c_def
          by (auto split: option.splits simp: runt_def make_cconstr_def)
     moreover obtain x1 x2 x3 pcs where runf pc_u2 s = Some ((x1, x2, s1, x3, r2), pcs)
        using ‹r2 = _› ‹mo = _› ‹Some s1 = _›
        unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
        by (auto split: option.splits)
      moreover obtain x1 x2 x3 x4 pcs where runf pc_u1 s1 = Some ((x1, x2, s', x3, x4),
pcs)
        using ‹r1 = _› ‹mi = _› ‹Some s' = _›
        unfolding equiv.make_f_def equiv.make_mf_def runf_def trans_of_def
        by (auto split: option.splits)
```

```
      moreover have r1 = make_reset pc_u1 s
       using ‹r1 = _› unfolding make_reset_def equiv.make_f_def runf_def by auto
      moreover have check_pred (L[q1 := l1', q2 := l2']) s'
       using prems(5) by (auto intro: P_unfold_1)
      moreover have a < na
        using action_set ‹_ ∈ set (trans ! q1 ! (L ! q1))› ‹q1 < _› ‹L ! q1 < _›
            process_length(2)
        by (fastforce dest!: nth_mem)
       moreover have
        (q1, pc_g1, a, pc_u1, l1')
        ∈ set (all_actions_by_state (nested_list_to_iarray trans_in_map) L ! a)
       using ‹L ! q1 < _› ‹_ ∈ set (trans ! q1 ! (L ! q1))› ‹q1 < p›
       by (force intro: in_actions_trans_in_mapI dest: mem_nth)
      moreover have
       (q2, pc_g2, a, pc_u2, l2')
       ∈ set (all_actions_by_state (nested_list_to_iarray trans_out_map) L ! a)
       using ‹L ! q2 < _› ‹_ ∈ set (trans ! q2 ! (L ! q2))› ‹q2 < p›
       by (force intro: in_actions_trans_out_mapI dest: mem_nth)
      ultimately show ?thesis
       using process_length(2) ‹q1 < _› ‹L ! q1 < _› ‹q2 < _› ‹L ! q2 < _› ‹_ ≠ _›
       unfolding pairs_by_action_def
        apply −
        apply (rule bexI[where x = a])
        by auto (force simp: set_map_filter)
    qed
    done
  done
subgoal for L s g a r L' s'
apply (drule in_trans_funD)
apply (erule disjE)
unfolding equiv.defs.prod_ta_def equiv.defs.prod_trans_def
 apply simp
 apply (rule disjI2)
subgoal
  unfolding equiv.defs.prod_trans_s_alt_def
  apply safe
  unfolding trans_s_fun_def
  apply clarsimp
   subgoal for x
  apply (erule in_pairs_by_actionD[where a' = x])
  apply (solves ‹auto dest: in_all_actions_by_stateD›)
      apply (solves ‹auto dest: in_all_actions_by_stateD›)
       apply (drule in_all_actions_by_stateD, assumption)
    apply (drule in_all_actions_by_stateD, assumption)
     apply safe
    apply (erule in_trans_in_mapD)
       apply (simp; fail)
      apply blast
       apply (erule in_trans_out_mapD)
     apply blast
      apply blast
     apply (simp only: ex_simps[symmetric])
     unfolding states'_def
     apply (clarsimp)
```

```
      apply (subst P_unfold, assumption)
      apply (subst P_unfold)
      subgoal
        unfolding runf_def by (auto dest!: exec_state_length)
      apply simp
      apply (drule transD[rotated 2], solve_triv+, blast)
      apply (drule transD[rotated 2], solve_triv+, blast)
      apply (drule_tac s = s in trans_ND, assumption)
      apply (drule_tac s = s in trans_ND, assumption)
      unfolding Product_TA_Defs.product_trans_s_def
      apply clarsimp
      unfolding trans_of_def
      subgoal
        apply (simp only: ex_simps[symmetric])
        apply defer_ex
        apply defer_ex
        apply solve_ex_triv
        apply solve_ex_triv
        apply solve_ex_triv
        unfolding make_f_unfold
    by (auto simp add: make_c_simp make_mf_simp make_reset_simp make_g_simp[symmetric])
      done
    done
subgoal
  apply simp
  apply (rule disjI1)
  using process_length(2)
  unfolding equiv.defs.prod_trans_i_alt_def
  apply simp
  unfolding P_unfold
  unfolding trans_i_fun_def trans_i_from_def states'_def
  apply simp
  apply (erule bexE)

  unfolding set_map_filter
  apply simp
  subgoal premises prems for q
  proof −
    from prems have len: q < length trans L ! q < length (trans ! q) by auto
    from prems(4) obtain pc_g pc_u l' x1 x2 x3 x4 where
      (pc_g, a, pc_u, l') ∈ set (IArray.list_of (map IArray trans_i_map ! q) ! (L ! q))
      check_g pc_g s = Some g
      r = make_reset pc_u s
      runf pc_u s = Some ((x1, x2, s', x3, r), x4)
      check_pred (L[q := l']) s'
      L' = L[q := l']
      apply atomize_elim
        unfolding make_reset_def
        by (force split: option.splits if_split_asm)
    moreover then have
        (L, g, (a, Networks.label.Act (equiv.make_c pc_g, equiv.make_mf pc_u)), r, L')
        ∈ Product_TA_Defs.product_trans_i (equiv.defs.N_s s)
    unfolding Product_TA_Defs.product_trans_i_def
    apply clarsimp
```

```
        apply solve_ex_triv
        apply safe
        using prems apply simp
        unfolding trans_i_map_def
        using len ‹q < p› apply (clarsimp simp: set_map_filter)
         apply (clarsimp split: act.split_asm)
        apply (frule make_c_simp)
        apply (drule mem_nth)
        apply safe
        apply (drule transD[rotated], solve_triv+)
        apply (drule trans_ND)
         apply solve_triv
          apply (subst make_g_simp)
        using ‹q < p› prems(1) by (auto simp add: make_f_unfold)
      ultimately show ?thesis
        apply (subst P_unfold)
        subgoal
          using prems(1) by fast
        apply (subst P_unfold)
        subgoal
          using ‹runf _ _ = _› prems(1)
          unfolding runf_def by (auto dest!: exec_state_length)
        using prems(1) by (force simp: make_mf_simp dest: make_c_simp)
    qed
    done
  done
done


  lemma transition_rel_mono:
    (a, b) ∈ transition_rel B if (a, b) ∈ transition_rel C B ⊆ C
    using that unfolding transition_rel_def b_rel_def fun_rel_def by auto

end

context
  Equiv_TA_Defs
begin

  lemma state_set_subs:
    state_set (trans_of (defs.product s''))
  ⊆ {L. length L = defs.p ∧ (∀ q<defs.p. L ! q ∈ State_Networks.state_set (fst (defs.N ! q)))}
    unfolding defs.states'_alt_def[symmetric]
    unfolding defs.N_s_def
    unfolding state_set_def Product_TA_Defs.states_def
    unfolding trans_of_def
    unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
    unfolding Product_TA_Defs.product_trans_i_def Product_TA_Defs.product_trans_s_def
    unfolding defs.T_s_def
      unfolding Product_TA_Defs.states_def trans_of_def
      apply simp
      apply safe
            apply (simp; fail)
      using [[goals_limit = 1]]
```

```
          apply force
         apply force
        apply (force simp: image_iff)
       apply force
      subgoal for x _ _ _ _ _ _ _ L p g _ _ _ r l' q
       apply (cases p = q)
       apply (force simp: image_iff)
       apply (force simp: image_iff)
       done
     apply force
    subgoal for _ _ _ _ _ _ _ _ _ p q g1 g2 _ _ _ _ _ _ r1 r2 l1' l2' qa
     apply (cases qa = q)
     apply force
     apply (cases qa = p)
     by (force simp: image_iff)+
    done

end

context UPPAAL_Reachability_Problem_precompiled_defs
begin

  lemma N_s_state_indep:
    assumes (L ! q, g, a, r, l') ∈ map trans_of (equiv.defs.N_s s) ! q q < p
    obtains g r where (L ! q, g, a, r, l') ∈ map trans_of (equiv.defs.N_s s') ! q
    using assms unfolding trans_of_def equiv.defs.N_s_def equiv.defs.T_s_def by force

  lemma fst_product_state_indep:
    fst ‘ fst (equiv.defs.product s) = fst ‘ fst (equiv.defs.product s')
    unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
    unfolding Product_TA_Defs.product_trans_s_def Product_TA_Defs.product_trans_i_def
    apply simp
    unfolding equiv.defs.states'_alt_def
    apply clarsimp
    apply (rule equalityI)
    subgoal
     apply (rule subsetI)
     apply clarsimp
     apply (erule disjE)
      apply (erule conjE exE)+
      apply (erule N_s_state_indep)
       apply (simp; fail)
      apply (rule img_fst)
      apply (rule Set.UnI1)
      apply (subst mem_Collect_eq)
      apply solve_ex_triv+
     apply (erule conjE exE)+
     apply (erule N_s_state_indep, (simp; fail))
     apply (erule N_s_state_indep, (simp; fail))
     apply (rule img_fst)
     apply (rule Set.UnI2)
     apply (subst mem_Collect_eq)
     apply (rule exI)+
     apply (rule conjI)
```

```
    defer
   by solve_ex_triv+
  subgoal
   apply (rule subsetI)
   apply clarsimp
   apply (erule disjE)
    apply (erule conjE exE)+
    apply (erule N_s_state_indep)
     apply (simp; fail)
    apply (rule img_fst)
    apply (rule Set.UnI1)
    apply (subst mem_Collect_eq)
    apply solve_ex_triv+
   apply (erule conjE exE)+
   apply (erule N_s_state_indep, (simp; fail))
   apply (erule N_s_state_indep, (simp; fail))
   apply (rule img_fst)
   apply (rule Set.UnI2)
   apply (subst mem_Collect_eq)
   apply (rule exI)+
   apply (rule conjI)
    defer
   by solve_ex_triv+
  done

lemma last_product_state_indep:
  (snd o snd o snd o snd) ‘ fst (equiv.defs.product s)
 = (snd o snd o snd o snd) ‘ fst (equiv.defs.product s')
  unfolding Product_TA_Defs.product_ta_def Product_TA_Defs.product_trans_def
  unfolding Product_TA_Defs.product_trans_s_def Product_TA_Defs.product_trans_i_def
  apply simp
  unfolding equiv.defs.states'_alt_def
  apply clarsimp
  apply (rule equalityI)
  subgoal
   apply (rule subsetI)
   apply clarsimp
   apply (erule disjE)
    apply (erule conjE exE)+
    apply (erule N_s_state_indep)
     apply (simp; fail)
    apply (rule image_eqI)
     prefer 2
     apply (rule Set.UnI1)
     apply (subst mem_Collect_eq)
     apply solve_ex_triv+
   apply (erule conjE exE)+
   apply (erule N_s_state_indep, (simp; fail))
   apply (erule N_s_state_indep, (simp; fail))
   apply (rule image_eqI)
    defer
    apply (rule Set.UnI2)
    apply (subst mem_Collect_eq)
    apply (rule exI)+
```

```
    apply (rule conjI)
     defer
     apply solve_ex_triv+
    defer
    apply (rule HOL.refl)
   by simp
  subgoal
   apply (rule subsetI)
   apply clarsimp
   apply (erule disjE)
   apply (erule conjE exE)+
   apply (erule N_s_state_indep)
   apply (simp; fail)
   apply (rule image_eqI)
   prefer 2
   apply (rule Set.UnI1)
   apply (subst mem_Collect_eq)
   apply solve_ex_triv+
   apply (erule conjE exE)+
   apply (erule N_s_state_indep, (simp; fail))
   apply (erule N_s_state_indep, (simp; fail))
   apply (rule image_eqI)
   defer
   apply (rule Set.UnI2)
   apply (subst mem_Collect_eq)
   apply (rule exI)+
   apply (rule conjI)
   defer
   apply solve_ex_triv+
   defer
   apply (rule HOL.refl)
   by simp
  done

lemma state_set_T':
  state_set (equiv.defs.T' s'') ⊇ fst ` state_set (trans_of A)
  unfolding trans_of_def
  unfolding state_set_def
  unfolding Prod_TA_Defs.prod_ta_def equiv.defs.prod_trans_def
  apply simp
  unfolding equiv.defs.prod_trans_i_def equiv.defs.prod_trans_s_def
  unfolding trans_of_def
  apply safe
    apply (subst fst_product_state_indep; force)
   apply (subst fst_product_state_indep; force)
  apply (subst (asm) last_product_state_indep[simplified]; force)
  by (subst (asm) last_product_state_indep[simplified]; force)

lemma state_set_T'2:
  length L = equiv.defs.p
  ∀ q<equiv.defs.p. L ! q ∈ State_Networks.state_set (fst (equiv.defs.N ! q))
  if (L, s) ∈ state_set (trans_of A)
  using subset_trans [OF state_set_T' equiv.state_set_subs] that by blast+
```

lemma *state_set_states'*:
L ∈ *equiv.defs.states' s* **if** (*L*, *s*) ∈ *state_set* (*trans_of A*)
using *state_set_T'2*[*OF that*] **unfolding** *equiv.defs.states'_alt_def* **by** *simp*

lemma *state_set_pred*:
∀ *q*<*p*. (*equiv.defs.P* ! *q*) (*L* ! *q*) *s* **if** (*L*, *s*) ∈ *state_set* (*trans_of A*)
using *that*
unfolding *Normalized_Zone_Semantics_Impl_Refine.state_set_def*
unfolding *trans_of_def Prod_TA_Defs.prod_ta_def Prod_TA_Defs.prod_trans_def*
unfolding *Prod_TA_Defs.prod_trans_i_def Prod_TA_Defs.prod_trans_s_def*
by *force*

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *bounded_bounded''*:
bounded bounds s ⟹ *length s* = *length bounds*
unfolding *bounded'_def bounded_def* **by** *simp*

lemma *P_bounded*:
(∀ *q*<*p*. (*equiv.defs.P* ! *q*) (*L* ! *q*) *s*) ⟹ *bounded bounds s*
unfolding *equiv.state_ta_def equiv.state_pred_def check_pred_def* **using** *process_length*(*3*) *p_gt_0*
apply *simp*
unfolding *list_all_iff*
unfolding *N_def*
unfolding *runf_def P_def*
apply (*drule spec*[*of _ 0*])
by (*simp split*: *option.split*; *auto split*: *option.split_asm*)

lemma *P_state_length*:
(∀ *q*<*p*. (*equiv.defs.P* ! *q*) (*L* ! *q*) *s*) ⟹ *length s* = *length bounds*
by (*intro P_bounded bounded_bounded''*)

lemma *state_set_state_length*:
length s = *length bounds* **if** (*L*, *s*) ∈ *state_set* (*trans_of A*)
using *that* **unfolding** *state_set_def*
apply (*safe dest*!: *equiv.defs.prod_ta_cases*)
unfolding *equiv.defs.prod_trans_i_alt_def equiv.defs.prod_trans_s_alt_def*
by *safe* (*auto dest*: *P_state_length*)

lemma *state_set_states*:
state_set (*trans_of A*) ⊆ *states'*
using *state_set_states' state_set_pred* **unfolding** *states'_def*
by (*auto intro*: *P_unfold_1 state_set_state_length*)

lemma *p_p_2*[*simp*]:
defs'.defs.p = *p*
unfolding *defs'.p_p p_p* **..**

lemma *len_product'_N*[*simp*]:

length defs'.defs.N = p
unfolding *defs'.defs.p_def*[*symmetric*] **by** (*rule p_p_2*)

lemma *len_equiv_N*:
length equiv.defs.N = p
unfolding *equiv.defs.p_def*[*symmetric*] **by** *simp*

lemma
defs'.p = p
unfolding *defs'.p_def* **by** *simp*

lemma *equiv_p_p*: *equiv.p = p*
by *simp*

lemma *init_states*:
init ∈ *equiv.defs.states'* s_0
using *processes_have_trans start_has_trans*
unfolding *equiv_states'_alt_def*
unfolding *init_def N_def T_def* **by** *force*

lemma *start_pred'*:
check_pred init s_0
using *start_pred bounded* **unfolding** *check_pred_def runf_def list_all_iff*
by (*fastforce split*: *option.split intro*: *bounded_bounded'*)

lemma *start_states'*:
(*init*, s_0) ∈ *states'*
using *start_pred' init_states bounded* **unfolding** *states'_def bounded_def* **by** *auto*

lemma *trans_fun_trans_of*[*intro*, *simp*]:
(*trans_fun*, *trans_of A*) ∈ *transition_rel states*
using *trans_fun_trans_of' state_set_states start_states'* **unfolding** *transition_rel_def* **by** *blast*

definition
inv_fun ≡ λ (*L*, _). *concat* (*map* (λ *i*. *IArray* (*map IArray inv*) !! *i* !! (*L* ! *i*)) [*0*..<*p*])

lemma *states_states'*:
states ⊆ *states'*
using *state_set_states start_states'* **by** *auto*

lemma *states'_length*[*simp*]:
length L = p **if** (*L*, *s*) ∈ *states'*
using *that* **unfolding** *states'_def* **by** *auto*

lemma *inv_simp*:
I q (*L* ! *q*) = *inv* ! *q* ! (*L* ! *q*) **if** *q* < *p* (*L*, *s*) ∈ *states'*
unfolding *I_def* **using** *that states'_states'*[*OF that*(*2*)] *lengths* **by** (*auto dest*!: *states_len*)

lemma *inv_fun_inv_of'*:
(*inv_fun*, *inv_of A*) ∈ *inv_rel Id states'*
unfolding *inv_rel_def*
apply (*clarsimp simp*: *equiv.defs.inv_of_simp Product_TA_Defs.inv_of_product*)

```
    using process_length(1)
    unfolding inv_fun_def Product_TA_Defs.product_invariant_def init_def
    unfolding equiv.defs.N_s_def
    apply simp
    apply (rule arg_cong[where f = concat])
    unfolding inv_of_def Equiv_TA_Defs.state_ta_def apply simp
    unfolding equiv.state_inv_def N_def Equiv_TA_Defs.state_inv_def
    by (auto simp: inv_simp)

  lemma inv_rel_mono:
    (a, b) ∈ inv_rel Id B if (a, b) ∈ inv_rel Id C B ⊆ C
    using that unfolding inv_rel_def b_rel_def fun_rel_def by auto

  lemma inv_fun_inv_of[intro, simp]:
    (inv_fun, inv_of A) ∈ inv_rel Id states
    using inv_fun_inv_of' states_states' by (rule inv_rel_mono)

  definition final_fun ≡ λ (L, s). hd_of_formula formula L s

  lemma final_fun_final':
    (final_fun, (λ (l, s). F l s)) ∈ inv_rel Id states'
    unfolding F_def final_fun_def inv_rel_def in_set_member[symmetric] list_ex_iff
     by (force dest!: states'_states')

  lemma final_fun_final[intro, simp]:
    (final_fun, (λ (l, s). F l s)) ∈ inv_rel Id states
    using final_fun_final' states_states' by (rule inv_rel_mono)

  lemma fst_clkp_setD:
    assumes (c, d) ∈ clkp_set A l
    shows c > 0 c ≤ m d ∈ range int
    using assms clock_set consts_nats clkp_set'_subs
    unfolding Nats_def clk_set'_def TA_clkp_set_unfold by force+

  lemma init_has_trans:
    (init, s0) ∈ fst ` (trans_of A) ⟷ trans_fun (init, s0) ≠ []
    apply standard
    using trans_fun_trans_of unfolding transition_rel_def apply force
    using trans_fun_trans_of' start_states' unfolding transition_rel_def by fast

end

context UPPAAL_Reachability_Problem_precompiled'
begin

  abbreviation k_i ≡ IArray (map (IArray o (map (IArray o map int))) k)

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

  lemma k_impl_alt_def:
    k_impl =
    (λ (l, _). IArray (map (λ c. Max ((λ i. k_i !! i !! (l ! i) !! c) ` {0..<p})) [0..<m+1]))
  proof −
```

```
  have {i. i < p} = {0..<p}
    by auto
  then show ?thesis unfolding k_impl_def setcompr_eq_image by auto
qed

lemma k_length_alt:
  ∀ i < p. ∀ j < length (k ! i). length (k ! i ! j) = m + 1
  using k_length(1,3) by (auto dest: nth_mem)

lemma Max_int_commute:
  int (Max S) = Max (int ‘ S) if finite S S ≠ {}
  apply (rule mono_Max_commute)
    apply standard
  using that by auto

lemma k_impl_k'[intro]:
  k_impl (l, s) = IArray (k' (l, s)) if (l, s) ∈ states'
proof −
  have l_len[simp]: l ! i < length (trans ! i) if i < p for i
    using ‹i < p› ‹(l, s) ∈ _› by auto
  have ∗: k_i !! i !! (l ! i) !! c = k ! i ! (l ! i) ! c
    if c ≤ m i < p for c i
  proof −
    from k_length_alt that k_length(1,2) have length (k ! i ! (l ! i)) = m + 1
      by auto
    with that k_length process_length(2) processes_have_trans start_has_trans show ?thesis
      unfolding init_def by auto
  qed
  show ?thesis
    unfolding k_impl_def k'_def k_fun_def

    apply clarsimp
    apply safe
    subgoal
      apply (subst Max_int_commute)
      subgoal
        by auto
      subgoal
        using p_gt_0 by auto
      apply (rule arg_cong[where f = Max])
      apply safe
      using ∗ apply (solves auto)
      by (solves ‹auto simp add: ∗[symmetric]›)

    subgoal
      apply (rule Max_eqI)
        apply (solves auto)
      using k_length_alt k_length processes_have_trans k_0 p_gt_0 unfolding init_def
       apply (solves auto)

      using k_length_alt k_length processes_have_trans k_0 p_gt_0 unfolding init_def
      apply clarsimp
      apply (rule exI[where x = 0])
      by simp
```

```
    subgoal
      apply (subst Max_int_commute)
      subgoal
        by auto
      subgoal
        using p_gt_0 by auto
      apply (rule arg_cong[where f = Max])
      apply safe
      using ∗ apply (solves auto)
      by (solves ‹auto simp add: ∗[symmetric]›)
    done
qed

lemma k_impl_k'2[intro]:
  k_impl (l, s) = IArray (k' (l, s)) if
  (l, s) ∈ Normalized_Zone_Semantics_Impl_Refine.state_set (trans_of A)
  using that states_states' by auto

lemma k_impl_k'_0[intro]:
  k_impl (init, s₀) = IArray (k' (init, s₀))
  using states_states' by auto

sublocale impl:
  Reachability_Problem_Impl
  where trans_fun = trans_fun
  and trans_impl = trans_fun
  and inv_fun = inv_fun
  and F_fun = final_fun
  and ceiling = k_impl
  and A = A
  and l₀ = (init, s₀)
  and l₀i = (init, s₀)
  and F = PR_CONST ((λ (l, s). F l s))
  and n = m
  and k = k_fun
  and loc_rel = Id
  and show_clock = show
  and show_state = show
  and states' = states
  unfolding PR_CONST_def
  apply standard
        apply (fastforce simp: inv_rel_def b_rel_def)
  subgoal
    by auto (metis IdI list_rel_id_simp relAPP_def)
      by (fastforce simp: inv_rel_def b_rel_def)+

lemma F_reachable_correct':
  impl.op.F_reachable
  ⟷ (∃ L' s' u u'.
     conv_A A ⊢' ⟨(init, s₀), u⟩ →∗ ⟨(L', s'), u'⟩
     ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
   ) if formula = formula.EX φ
  using that E_op''.E_from_op_reachability_check[of F_rel PR_CONST (λ(x, y). F x y),
```

```
    unfolded F_rel_def, OF HOL.refl]
   reachability_check
 unfolding impl.E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
 unfolding F_rel_def unfolding F_def by force

lemma PT_PT:
 defs'.PT = equiv.PT
 apply simp
 unfolding striptp_def
 apply (rule ext)
 apply clarsimp
 subgoal for x
  apply (cases PROG x)
   apply (simp; fail)
  subgoal for a
   by (cases a) auto
  done
 done

lemma P_P[simp]:
 defs'.defs.P = equiv.defs.P
 unfolding Equiv_TA_Defs.state_ta_def
 unfolding Equiv_TA_Defs.p_def
 unfolding Equiv_TA_Defs.state_pred_def
 using PF_PF by (auto split: option.split)

lemma map_map_filter:
 map f (List.map_filter g xs) = List.map_filter (map_option f o g) xs
 by (induction xs; simp add: List.map_filter_simps split: option.split)

lemma make_g_conv:
 defs'.make_g = conv_cc oo equiv.make_g
 unfolding Equiv_TA_Defs.make_g_def PT_PT PF_PF apply simp
 apply (intro ext)
 apply (clarsimp split: option.splits simp: map_map_filter)
 apply (rule arg_cong2[where f = List.map_filter])
 by (auto split: option.split instrc.split)

lemma make_c_conv:
 defs'.make_c = equiv.make_c
 unfolding Equiv_TA_Defs.make_c_def PT_PT by simp

lemma make_f_conv:
 defs'.make_f = equiv.make_f
 unfolding Equiv_TA_Defs.make_f_def PF_PF by simp

lemma make_mf_conv:
 defs'.make_mf = equiv.make_mf
 unfolding Equiv_TA_Defs.make_mf_def PF_PF by simp

lemmas make_convs = make_g_conv make_c_conv make_f_conv make_mf_conv

lemma state_trans_conv:
 Equiv_TA_Defs.state_trans_t (conv N) max_steps q
```

```
  = (λ (a, b, c). (a, λ x. conv_cc (b x), c)) ‘ equiv.state_trans q if ‹q < p›
  unfolding Equiv_TA_Defs.state_trans_t_def image_Collect
  using ‹q < _› make_convs by (force split: prod.splits)+

lemma map_conv_t:
  map trans_of (defs'.defs.N_s s) ! q = conv_t ‘ (map trans_of (equiv.defs.N_s s) ! q)
  if ‹q < p›
  using ‹q < p›
  apply (subst nth_map)
  unfolding defs'.defs.N_s_length p_p_2
   apply assumption
  apply (subst nth_map)
  unfolding equiv.defs.N_s_length
   apply simp
  unfolding trans_of_def Prod_TA_Defs.N_s_def
  unfolding len_equiv_N len_product'_N
  apply simp
  unfolding Prod_TA_Defs.T_s_def
  unfolding image_Collect
  unfolding Equiv_TA_Defs.state_ta_def Equiv_TA_Defs.p_def
  apply simp
  using state_trans_conv[of q]
  apply simp
  apply safe
   apply force
  apply solve_ex_triv+
  unfolding image_iff by force

lemma product_trans_t_conv:
  Product_TA_Defs.product_trans_s (defs'.defs.N_s s)
   = conv_t ‘ Product_TA_Defs.product_trans_s (equiv.defs.N_s s)
  unfolding Product_TA_Defs.product_trans_s_def
  apply (simp only: states'_conv)
    apply safe
   apply (simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
    unfolding image_Collect
    apply (simp split: prod.split)
     apply safe
    subgoal
      by defer_ex solve_ex_triv+
    subgoal
      by solve_ex_triv+ (force simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
    done

lemma product_trans_t_conv':
  Product_TA_Defs.product_trans_i (defs'.defs.N_s s)
   = conv_t ‘ Product_TA_Defs.product_trans_i (equiv.defs.N_s s)
  unfolding Product_TA_Defs.product_trans_i_def
  apply (simp only: states'_conv)
    apply safe
   apply (simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
    unfolding image_Collect
    apply (simp split: prod.split)
     apply safe
```

```
    subgoal
      by defer_ex solve_ex_triv+
    subgoal
      by solve_ex_triv+ (force simp only: equiv.states'_len_simp equiv_p_p map_conv_t)
    done

lemma prod_trans_s_conv:
  defs'.defs.prod_trans_s = conv_t ‘ equiv.defs.prod_trans_s
  unfolding defs'.defs.prod_trans_s_alt_def
  unfolding equiv.defs.prod_trans_s_alt_def
  unfolding product_trans_t_conv
  unfolding p_p_2 P_P
  apply simp
  apply safe
  unfolding p_p P_P
   apply (simp add: image_Collect)
   apply solve_ex_triv
  subgoal
    apply defer_ex
    apply defer_ex
    by solve_ex_triv+
  subgoal
    apply defer_ex
    apply defer_ex
    by solve_ex_triv+
  done

lemma prod_trans_i_conv:
  defs'.defs.prod_trans_i = conv_t ‘ equiv.defs.prod_trans_i
  unfolding defs'.defs.prod_trans_i_alt_def
  unfolding equiv.defs.prod_trans_i_alt_def
  unfolding product_trans_t_conv'
  unfolding p_p_2 P_P
  apply simp
  apply safe
  unfolding p_p P_P
   apply (simp add: image_Collect)
   apply solve_ex_triv
  subgoal
    apply defer_ex
    apply defer_ex
    by solve_ex_triv+
  subgoal
    apply defer_ex
    apply defer_ex
    by solve_ex_triv+
  done

lemma prod_trans_conv:
  defs'.defs.prod_trans = conv_t ‘ equiv.defs.prod_trans
  unfolding defs'.defs.prod_trans_def
  unfolding equiv.defs.prod_trans_def
  unfolding prod_trans_s_conv
  unfolding prod_trans_i_conv image_Un ..
```

lemma *prod_invariant_conv*:
defs'.defs.prod_invariant = (*map conv_ac* ∘∘ *Prod_TA_Defs.prod_invariant*) *EA*
apply (*rule ext*)
apply *safe*
unfolding *defs'.defs.prod_invariant_def equiv.defs.prod_invariant_def*
unfolding *Product_TA_Defs.product_ta_def inv_of_def*
apply *simp*
unfolding *Product_TA_Defs.product_invariant_def List.map_concat*
apply (*simp add*: *Prod_TA_Defs.N_s_length*)
unfolding *Equiv_TA_Defs.p_p Equiv_TA_Defs.p_def* **apply** *simp*
apply (*rule cong*[**where** *f* = *concat*])
apply (*rule HOL.refl*)
unfolding *Prod_TA_Defs.N_s_def inv_of_def Equiv_TA_Defs.state_ta_def*
unfolding *Equiv_TA_Defs.p_def* **unfolding** *Equiv_TA_Defs.state_inv_def*
by (*simp split*: *prod.split*)

lemma *prod_conv*: *defs'.defs.prod_ta* = *conv_A A*
unfolding *defs'.defs.prod_ta_def*
unfolding *equiv.defs.prod_ta_def*
unfolding *conv_A_def*
by (*simp add*: *prod_invariant_conv*[*symmetric*] *prod_trans_conv*[*symmetric*])

lemma *F_reachable_correct*:
impl.op.F_reachable
⟷ (∃ *L' s' u u'*.
conv N ⊢$_m$ *ax_steps* ⟨*init*, s_0, *u*⟩ →∗ ⟨*L'*, *s'*, *u'*⟩
∧ (∀ *c* ∈ {*1..m*}. *u c* = *0*) ∧ *check_bexp φ L' s'*
) **if** *formula* = *formula.EX φ start_inv_check*
unfolding *F_reachable_correct'*[*OF that*(*1*)]
apply (*subst product'.prod_correct*[*symmetric*])
using *prod_conv p_p p_gt_0* **apply** *simp*
using *prod_conv p_p p_gt_0* **apply** *simp*
using *F_reachable_equiv*[*OF that*(*2*)]
by (*simp add*: *F_def*, *simp add*: *that*(*1*))

definition
reachability_checker_old ≡
worklist_algo2_impl
impl.subsumes_impl impl.a_0*_impl impl.F_impl impl.succs_impl impl.emptiness_check_impl*

definition
reachability_checker' ≡
pw_impl
(*return o fst*) *impl.state_copy_impl impl.tracei impl.subsumes_impl impl.*a_0*_impl impl.F_impl*
impl.succs_impl impl.emptiness_check_impl

theorem *reachability_check'*:
(*uncurry0 reachability_checker'*,
uncurry0 (
Refine_Basic.RETURN (∃ *L' s' u u'*.
conv_A A ⊢' ⟨(*init*, s_0), *u*⟩ →∗ ⟨(*L'*, *s'*), *u'*⟩
∧ (∀ *c* ∈ {*1..m*}. *u c* = *0*) ∧ *check_bexp φ L' s'*
)

```
   )
  )
  ∈ unit_assn^k →_a bool_assn if formula = formula.EX φ
  using impl.pw_impl_hnr_F_reachable
  unfolding reachability_checker'_def F_reachable_correct'[OF that] .

corollary reachability_checker'_hoare:
  <emp> reachability_checker'
  <λ r. ↑(r = (∃ L' s' u u'.
      conv_A A ⊢' ⟨(init, s_0), u⟩ →* ⟨(L', s'), u'⟩
      ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
    ))
  >_t if formula = formula.EX φ
 apply (rule cons_post_rule)
 using reachability_check'[OF that, to_hnr] apply (simp add: hn_refine_def)
 by (sep_auto simp: pure_def)

definition reachability_checker where
  reachability_checker ≡ do
    {
      init_sat ← impl.start_inv_check_impl;
      if init_sat then do
        { x ← 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 ⊢_max_steps ⟨init, s_0, u⟩ →* ⟨L', s', u'⟩
            ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp φ L' s'
            )
          then REACHABLE
          else UNREACHABLE
        else INIT_INV_ERR
    )
   ))
  ∈ unit_assn^k →_a id_assn if formula = formula.EX φ
  apply (simp only: F_reachable_correct[OF that, symmetric] cong: if_cong)
  supply
    impl.pw_impl_hnr_F_reachable
    [unfolded reachability_checker'_def[symmetric], to_hnr, unfolded hn_refine_def,
     rule_format, sep_heap_rules]
  supply
   impl.start_inv_check_impl.refine[to_hnr, unfolded hn_refine_def, rule_format, sep_heap_rules]
```

```
  unfolding reachability_checker_def
  by sepref_to_hoare (sep_auto simp: pure_def)

corollary reachability_checker_hoare:
  <emp> reachability_checker
  <λ r. ↑(r =
     (
       if start_inv_check
       then
         if
           (
           ∃ L' s' u u'.
             conv N ⊢_m ax_steps ⟨init, s_0, 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
 apply (rule cons_post_rule)
 using reachability_check[OF that, to_hnr] apply (simp add: hn_refine_def)
 by (sep_auto simp: pure_def)

lemma list_all_concat:
  list_all Q (concat xxs) ⟷ (∀ xs ∈ set xxs. list_all Q xs)
  unfolding list_all_iff by auto

lemma inv_of_init_unfold:
  u ⊢ inv_of (conv_A A) (init, s_0) ⟷ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0))
proof −
  have ∗: inv_of (conv_A A) (init, s_0) = conv_cc (equiv.defs.I' s_0 init)
    using equiv.defs.inv_of_simp[of init s_0]
    unfolding inv_of_def conv_A_def by (auto split: prod.split)
  have u ⊢ inv_of (conv_A A) (init, s_0) ⟷ (∀ i < p. u ⊢ conv_cc (I i 0))
    unfolding ∗ Product_TA_Defs.inv_of_product Product_TA_Defs.product_invariant_def
    apply (simp only: product'.prod.length_L p_p_2 cong: list.map_cong_simp)
    unfolding equiv.defs.N_s_def length_N
    apply (simp cong: list.map_cong_simp)
    unfolding inv_of_def
    apply (simp cong: list.map_cong_simp)
    unfolding init_def
    apply (simp cong: list.map_cong_simp)
    unfolding Equiv_TA_Defs.state_ta_def
    apply (simp cong: list.map_cong_simp)
    unfolding equiv.state_inv_def
    unfolding N_def
    by (force simp: map_concat list_all_concat clock_val_def cong: list.map_cong_simp)
  also have (∀ i < p. u ⊢ conv_cc (I i 0)) ⟷ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0))
    unfolding I_def using lengths processes_have_trans by fastforce
  finally show ?thesis .
qed
```

```
corollary reachability_checker_hoare':
  <emp> reachability_checker
  <λ r. ↑(r =
     (
      if (∀ u. (∀ c∈{1..m}. u c = 0) ⟶ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0)))
      then
        if
          (
          ∃ L' s' u u'.
            conv N ⊢_m ax_steps ⟨init, s_0, 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
  using reachability_checker_hoare[OF that] unfolding start_inv_check_correct inv_of_init_unfold
.

Post-processing    schematic_goal succs_impl_alt_def:
    impl.succs_impl ≡ ?impl
    unfolding impl.succs_impl_def
    unfolding k_impl_alt_def
    apply (abstract_let
        λ (l, _ :: int list). IArray (map (λ c. MAX i∈{0..<p}. k_i !! i !! (l ! i) !! c) [0..<m+1])
        k_i
      )
    apply (abstract_let inv_fun :: nat list × int list ⇒ (nat, int) acconstraint list inv_fun)
    apply (abstract_let trans_fun trans_fun)
   unfolding inv_fun_def[abs_def] trans_fun_def[abs_def] trans_s_fun_def trans_i_fun_def
trans_i_from_def
    apply (abstract_let IArray (map IArray inv) inv)
    apply (abstract_let IArray (map IArray trans_out_map) trans_out_map)
    apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
    apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
    by (rule Pure.reflexive)

  lemma reachability_checker'_alt_def':
    reachability_checker' ≡
      let
        key = return ∘ fst;
        sub = impl.subsumes_impl;
        copy = impl.state_copy_impl;
        start = impl.a_0_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
    unfolding reachability_checker'_def by simp
```

schematic_goal *reachability_checker_alt_def*:
reachability_checker ≡ *?impl*
unfolding *reachability_checker_def*
unfolding *reachability_checker'_alt_def' impl.succs_impl_def*
unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)
apply (*abstract_let inv_fun* :: *nat list* × *int list* ⇒ (*nat*, *int*) *acconstraint list*)
apply (*abstract_let trans_fun trans_fun*)
unfolding *impl.init_dbm_impl_def impl.a₀_impl_def*
unfolding *impl.F_impl_def*
unfolding *final_fun_def*[*abs_def*]
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

end

lemmas [*code*] = *UPPAAL_Reachability_Problem_precompiled'.k_impl_def*

Some post refinements **code_thms** *fw_upd'*
code_thms *fw_impl'*
code_thms *fw_impl*

abbreviation *plus_int* :: *int* ⇒ *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*)) |
dbm_add_int (*Le a*) (*Lt b*) = (*Lt* (*plus_int a b*)) |
dbm_add_int (*Lt a*) (*Le b*) = (*Lt* (*plus_int a b*)) |
dbm_add_int (*Lt a*) (*Lt b*) = (*Lt* (*plus_int a b*))

lemma *dbm_add_int*:
dbm_add = *dbm_add_int*
apply (*rule ext*)+
subgoal for *x y*
by (*cases x*; *cases y*) *auto*
done

definition
fw_upd'_int m k i j =
Refine_Basic.RETURN
(*op_mtx_set m* (*i*, *j*)
(*min* (*op_mtx_get m* (*i*, *j*)) (*dbm_add_int* (*op_mtx_get m* (*i*, *k*)) (*op_mtx_get m* (*k*, *j*)))))

definition

```
  fw_upd_impl_int n ≡ λai bib bia bi. do {
                        xa ← mtx_get (Suc n) ai (bia, bib);
                        xb ← mtx_get (Suc n) ai (bib, bi);
                        x ← mtx_get (Suc n) ai (bia, bi);
                        let e = (dbm_add_int xa xb);
                        if e < x then mtx_set (Suc n) ai (bia, bi) e else Heap_Monad.return ai
                      }

lemma fw_upd_impl_int_eq:
  fw_upd_impl_int = fw_upd_impl
  unfolding fw_upd_impl_int_def fw_upd_impl_def
  unfolding dbm_add_int add
  unfolding Let_def ..

definition
  fw_impl_int n ≡
    imp_for' 0 (n + 1)
     (λxb. imp_for' 0 (n + 1)
           (λxd. imp_for' 0 (n + 1) (λxf σ''''. fw_upd_impl_int n σ'''' xb xd xf)))

lemma fw_impl'_int:
  fw_impl = fw_impl_int
  unfolding fw_impl_def fw_impl_int_def
  unfolding fw_upd_impl_int_eq ..

context UPPAAL_Reachability_Problem_precompiled_defs'
begin

  definition run_impl program pc s ≡ exec program max_steps (pc, [], s, True, []) []

  lemma runf_impl:
    runf = run_impl PF
    unfolding runf_def run_impl_def ..

  lemma runt_impl:
    runt = run_impl PT
    unfolding runt_def run_impl_def ..

  definition
    make_cconstr_impl program pcs =
    List.map_filter
     (λpc. case program pc of None ⇒ None | Some (INSTR x) ⇒ Map.empty x
           | Some (CEXP ac) ⇒ Some ac)
     pcs

  lemma make_cconstr_impl:
    make_cconstr = make_cconstr_impl PROG
    unfolding make_cconstr_def make_cconstr_impl_def ..

  definition
    check_g_impl programf program pc s ≡
    case run_impl programf pc s of None ⇒ None
    | Some ((x, xa, xb, True, xc), pcs) ⇒ Some (make_cconstr_impl program pcs)
    | Some ((x, xa, xb, False, xc), pcs) ⇒ None
```

lemma *check_g_impl*:
check_g = check_g_impl PT PROG'
unfolding *check_g_impl_def check_g_def runt_impl PROG'_PROG make_cconstr_impl* **..**

definition
make_reset_impl program m1 s ≡
case run_impl program m1 s of
Some ((_, _, _, _, r1), _) ⇒ r1
| None ⇒ []

lemma *make_reset_impl*:
make_reset = make_reset_impl PF
unfolding *make_reset_def make_reset_impl_def runf_impl* **..**

definition
check_pred_impl program bnds L s ≡
list_all
(λq. case run_impl program (pred ! q ! (L ! q)) s of None ⇒ False
| Some ((x, xa, xb, f, xc), xd) ⇒
f ∧ (∀ i<length s. fst (bnds !! i) < s ! i ∧ s ! i < snd (bnds !! i)))
[0..<p]

lemma *check_pred_impl*:
check_pred = check_pred_impl PF (IArray bounds)
unfolding *check_pred_def check_pred_impl_def runf_impl bounded'_def* **..**

definition
pairs_by_action_impl pf pt porig bnds ≡ 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_impl pt porig g1 s, check_g_impl pt porig g2 s) of
(Some cc1, Some cc2) ⇒
(case run_impl pf m2 s of
Some ((_, _, s1, _, r2), _) ⇒
(case run_impl pf m1 s1 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)
unfolding *pairs_by_action_def pairs_by_action_impl_def*
unfolding *check_g_impl make_reset_impl check_pred_impl runf_impl* **..**

definition
all_actions_by_state_impl upt_p empty_ran i L ≡
fold (λ*ia*. *actions_by_state ia* (*i* !! *ia* !! (*L* ! *ia*))) *upt_p empty_ran*

lemma *all_actions_by_state_impl*:
all_actions_by_state = *all_actions_by_state_impl* [*0*..<*p*] (*repeat* [] *na*)
unfolding *all_actions_by_state_def all_actions_by_state_impl_def* **..**

definition
trans_i_from_impl programf programt program bnds trans_i_array ≡
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*))
unfolding *trans_i_from_def trans_i_from_impl_def*
unfolding *check_g_impl runf_impl check_pred_impl* **..**

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *PF_alt_def*:
PF = (λ *pc*. *if pc* < *length prog then* (*IArray* (*map* (*map_option stripf*) *prog*)) !! *pc else None*)
unfolding *stripfp_def PROG'_def* **by** *auto*

lemma *PT_alt_def*:
PT = (λ *pc*. *if pc* < *length prog then* (*IArray* (*map* (*map_option stript*) *prog*)) !! *pc else None*)
unfolding *striptp_def PROG'_def* **by** *auto*

schematic_goal *reachability_checker_alt_def_refined*:
reachability_checker ≡ *?impl*
unfolding *reachability_checker_alt_def*
unfolding *fw_impl'_int*
unfolding *inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def*
unfolding *trans_i_from_impl*
unfolding *runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl*
apply (*abstract_let IArray* (*map IArray inv*) *inv*)
apply (*abstract_let IArray* (*map IArray trans_out_map*) *trans_out_map*)
apply (*abstract_let IArray* (*map IArray trans_in_map*) *trans_in_map*)
apply (*abstract_let IArray* (*map IArray trans_i_map*) *trans_i_map*)
apply (*abstract_let IArray bounds bounds*)
apply (*abstract_let PF PF*)

apply (*abstract_let PT PT*)
unfolding *PF_alt_def PT_alt_def*
apply (*abstract_let PROG' PROG'*)
unfolding *PROG'_def*
apply (*abstract_let length prog len_prof*)
apply (*abstract_let IArray* (*map* (*map_option stripf*) *prog*) *prog_f*)
apply (*abstract_let IArray* (*map* (*map_option stript*) *prog*) *prog_t*)
apply (*abstract_let IArray prog prog*)
unfolding *all_actions_by_state_impl*
apply (*abstract_let* [*0..<p*])
apply (*abstract_let* [*0..<na*])
apply (*abstract_let* {*0..<p*})
apply (*abstract_let* [*0..<m+1*])
by (*rule Pure.reflexive*)

end

5.2 Check Preconditions

context *UPPAAL_Reachability_Problem_precompiled_defs*
begin

abbreviation
check_nat_subs ≡ ∀ (_, *d*) ∈ *clkp_set'*. *d* ≥ *0*

lemma *check_nat_subs*:
check_nat_subs ⟷ *snd* ‘ *clkp_set'* ⊆ ℕ
unfolding *Nats_def* **apply** *safe*
subgoal for _ _ *b* **using** *rangeI*[*of int nat b*] **by** *auto*
by *auto*

definition
check_resets ≡ ∀ *x c*. *Some* (*INSTR* (*STOREC c x*)) ∈ *set prog* ⟶ *x* = *0*

lemma *check_resets_alt_def*:
check_resets =
(∀ (*c*, *x*) ∈ *collect_store*. *x* = *0*)
unfolding *check_resets_def collect_store_def* **by** *auto*

definition
check_pre ≡
length inv = *p* ∧ *length trans* = *p* ∧ *length pred* = *p*
∧ (∀ *i* < *p*. *length* (*pred* ! *i*) = *length* (*trans* ! *i*) ∧ *length* (*inv* ! *i*) = *length* (*trans* ! *i*))
∧ (∀ *T* ∈ *set trans*. ∀ *xs* ∈ *set T*. ∀ (_, _, _, *l*) ∈ *set xs*. *l* < *length T*)
∧ *p* > *0* ∧ *m* > *0*
∧ (∀ *i* < *p*. *trans* ! *i* ≠ []) ∧ (∀ *q* < *p*. *trans* ! *q* ! *0* ≠ [])
∧ *check_nat_subs* ∧ *clk_set'* = {*1..m*}
∧ *check_resets*

lemma *finite_clkp_set'*[*intro*, *simp*]:
finite clkp_set'
unfolding *clkp_set'_def*
using [[*simproc add*: *finite_Collect*]]

by (*auto intro*!: *finite_vimageI simp*: *inj_on_def*)

lemma *check_pre*:
UPPAAL_Reachability_Problem_precompiled p m inv pred trans prog ⟷ *check_pre*
unfolding
UPPAAL_Reachability_Problem_precompiled_def
check_pre_def check_nat_subs check_resets_def
by *auto*

end

context *UPPAAL_Reachability_Problem_precompiled_defs*
begin

context
fixes *k* :: *nat list list list*
begin

definition
check_ceiling ≡
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
⟷ *check_pre* ∧ *check_ceiling*
unfolding *UPPAAL_Reachability_Problem_precompiled_ceiling_def check_pre check_ceiling_def*
by *auto*

end

end

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp_alt_def
UPPAAL_Reachability_Problem_precompiled_defs.collect_store_alt_def
UPPAAL_Reachability_Problem_precompiled_defs.check_resets_alt_def

export_code *UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp* **in** *SML* **module_name** *Test*

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs.check_pre
UPPAAL_Reachability_Problem_precompiled_defs.check_axioms
UPPAAL_Reachability_Problem_precompiled_defs.clkp_set'_alt_def
UPPAAL_Reachability_Problem_precompiled_defs.clk_set'_alt_def
UPPAAL_Reachability_Problem_precompiled_defs.check_pre_def
UPPAAL_Reachability_Problem_precompiled_defs.check_ceiling_def
UPPAAL_Reachability_Problem_precompiled_defs.init_def

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs'.trans_out_map_def
UPPAAL_Reachability_Problem_precompiled_defs'.trans_in_map_def

UPPAAL_Reachability_Problem_precompiled_defs'.trans_i_map_def
UPPAAL_Reachability_Problem_precompiled_defs'.all_actions_by_state_def
UPPAAL_Reachability_Problem_precompiled_defs'.actions_by_state_def
UPPAAL_Reachability_Problem_precompiled_defs'.pairs_by_action_def

code_pred *clock_val_a* **.**

concrete_definition *reachability_checker_impl*
uses *UPPAAL_Reachability_Problem_precompiled'.reachability_checker_alt_def_refined*

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs'.make_cconstr_def
UPPAAL_Reachability_Problem_precompiled_defs'.make_reset_def
UPPAAL_Reachability_Problem_precompiled_defs'.check_pred_def
UPPAAL_Reachability_Problem_precompiled_defs'.check_g_def
UPPAAL_Reachability_Problem_precompiled_defs'.runf_def
UPPAAL_Reachability_Problem_precompiled_defs'.runt_def
UPPAAL_Reachability_Problem_precompiled_defs.PROG_def

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs.P_def

lemma *exec_code*[*code*]:
exec prog n (*pc*, *st*, *m*, *f*, *rs*) *pcs* =
(*case n of 0* ⇒ *None*
| *Suc n* ⇒
(*case prog pc of None* ⇒ *None*
| *Some instr* ⇒
if instr = *HALT*
then Some ((*pc*, *st*, *m*, *f*, *rs*), *pc* # *pcs*)
else
(*case UPPAAL_Asm.step instr* (*pc*, *st*, *m*, *f*, *rs*) *of*
None ⇒ *None* | *Some s* ⇒ *exec prog n s* (*pc* # *pcs*))))
by (*cases n*) *auto*

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

definition
init_pred_check ≡ λ *p prog max_steps pred* s_0.
(∀ *q* < *p*.
case (*exec*
(*stripfp* (*UPPAAL_Reachability_Problem_precompiled_defs.PROG prog*))
max_steps
((*pred* ! *q* ! (*UPPAAL_Reachability_Problem_precompiled_defs.init p* ! *q*)), [], s_0, *True*,
[])
[])
of Some ((*pc*, *st*, *s'*, *True*, *rs*), *pcs*) ⇒ *True* | _ ⇒ *False*)

definition
time_indep_check1 ≡ λ *pred prog max_steps.*

(∀ x∈set pred. ∀ pc∈set x. time_indep_check prog pc max_steps)

definition
time_indep_check2 ≡ λ trans prog max_steps.
(∀ T∈set trans. ∀ xs∈set T. ∀ (_, _, pc_u, _)∈set xs. time_indep_check prog pc_u max_steps)

definition
conjunction_check2 ≡ λ trans prog max_steps.
(∀ T∈set trans. ∀ xs∈set T. ∀ (pc_g, _, _, _)∈set xs. conjunction_check prog pc_g max_steps)

lemma start_pred[code]:
UPPAAL_Reachability_Problem_precompiled_start_state_axioms = (λ p max_steps trans prog bounds pred s_0.
init_pred_check p prog max_steps pred s_0
∧ bounded bounds s_0
∧ time_indep_check1 pred prog max_steps
∧ time_indep_check2 trans prog max_steps
∧ conjunction_check2 trans prog max_steps
)
unfolding UPPAAL_Reachability_Problem_precompiled_start_state_axioms_def
unfolding init_pred_check_def bounded_def time_indep_check1_def time_indep_check2_def conjunction_check2_def
apply (rule ext)+
apply safe
apply (fastforce split: option.split_asm bool.split_asm)
subgoal premises prems
using prems(1,7) **by** (fastforce split: option.split_asm bool.split_asm)
done

export_code UPPAAL_Reachability_Problem_precompiled_start_state_axioms

context UPPAAL_Reachability_Problem_precompiled_defs
begin

lemma collect_store''_alt_def:
collect_store'' pc ≡
case find_resets_start prog pc of
None ⇒ {} |
Some pc' ⇒
⋃ (
(λ cmd. case cmd of Some (INSTR (STOREC c x)) ⇒ {(c, x)} | _ ⇒ {}) ‘
((!) prog) ‘ {pc .. pc'}
)
unfolding collect_store''_def
apply (rule eq_reflection)
apply safe
subgoal
apply (simp del: find_resets_start.simps split: option.split_asm)
by (auto intro: sym intro!: bexI split: option.split)
subgoal
apply (clarsimp simp del: find_resets_start.simps split: option.split_asm)

by (*auto intro*: *sym split*: *instrc.split_asm instr.split_asm*)
done

lemma *collect_cexp'_alt_def*:
collect_cexp' pc ≡
⋃ ((λ *cmd. case cmd of Some* (*CEXP ac*) ⇒ {*ac*} | _ ⇒ {}) ‘
((!) *prog*) ‘ *steps_approx max_steps prog pc*
)
unfolding *collect_cexp'_def*
by (*auto 4 3 intro*!: *eq_reflection bexI intro*: *sym split*: *option.splits instrc.split_asm*)

end

lemmas [*code*] =
UPPAAL_Reachability_Problem_precompiled_defs'.PROG'_def
UPPAAL_Reachability_Problem_precompiled_start_state_def
UPPAAL_Reachability_Problem_precompiled_ceiling_axioms_def
UPPAAL_Reachability_Problem_precompiled_defs.N_def
UPPAAL_Reachability_Problem_precompiled_defs.collect_store''_alt_def
UPPAAL_Reachability_Problem_precompiled_defs.clkp_set''_def
UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp'_alt_def
UPPAAL_Reachability_Problem_precompiled_defs'.pairs_by_action_impl_def
UPPAAL_Reachability_Problem_precompiled_defs'.make_reset_impl_def
UPPAAL_Reachability_Problem_precompiled_defs'.check_g_impl_def
UPPAAL_Reachability_Problem_precompiled_defs'.run_impl_def
UPPAAL_Reachability_Problem_precompiled_defs'.make_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 s_0 *na* ≡
if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s_0 *na k*
then
reachability_checker_impl p m max_steps I T prog bounds P s_0 *na k final*
≫= (λ *x. return* (*Some x*))
else return None

abbreviation *N* ≡ *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* s_0

```
na),
    uncurry0 (
      Refine_Basic.RETURN (
       if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s₀ na
k
       then Some (
        if (∀ u. (∀ c∈{1..m}. u c = 0) ⟶ (∀ i < p. u ⊢ conv_cc (I ! i ! 0)))
          then
            if
              (
              ∃ L' s' u u'.
                conv (N p I P T prog bounds) ⊢_m ax_steps
                ⟨repeat 0 p, s₀, u⟩ →* ⟨L', s', u'⟩
              ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp formula L' s'
              )
            then REACHABLE
            else UNREACHABLE
          else INIT_INV_ERR
          )
       else None
      )
    )
   )
    ∈ unit_assn^k →_a id_assn
proof −
  define A where A ≡ conv (N p I P T prog bounds)
  define start_inv where
    start_inv ≡ (∀ u. (∀ c∈{1..m}. u c = 0) ⟶ (∀ i < p. u ⊢ conv_cc (I ! i ! 0)))
  define reach where
    reach ≡
      ∃ L' s' u u'.
        A ⊢_m ax_steps
        ⟨repeat 0 p, s₀, u⟩ →* ⟨L', s', u'⟩
      ∧ (∀ c ∈ {1..m}. u c = 0) ∧ check_bexp formula L' s'
  note [sep_heap_rules] =
    UPPAAL_Reachability_Problem_precompiled'.reachability_checker_hoare'
    [ OF _ HOL.refl[of formula.EX formula],
      unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
      of p m max_steps I T prog bounds P s₀ na k,
      unfolded A_def[symmetric] start_inv_def[symmetric] reach_def[symmetric]
    ]
  show ?thesis
    unfolding A_def[symmetric] start_inv_def[symmetric] reach_def[symmetric]
    unfolding check_and_verify_def
    by sepref_to_hoare (sep_auto simp: reachability_checker_impl.refine[symmetric])
qed

export_code open
  check_and_verify init_pred_check time_indep_check1 time_indep_check1 conjunction_check2
  checking SML_imp

end
theory UPPAAL_Model_Checking
  imports
```

UPPAAL_State_Networks_Impl_Refine
Munta_Base.TA_More
Munta_Base.Abstract_Term
begin

hide_const *models*

inductive *step_u'* ::
(*'a*, *'t* :: *time*, *'s*) *unta* ⇒ *nat* ⇒ *'s list* ⇒ *int list* ⇒ (*nat*, *'t*) *cval*
⇒ *'s list* ⇒ *int list* ⇒ (*nat*, *'t*) *cval* ⇒ *bool*
(_ ⊢– ⟨_, _, _⟩ → ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*,*61*] *61*) **where**
A ⊢*n* ⟨*L*, *s*, *u*⟩ → ⟨*L''*, *s''*, *u''*⟩ **if**
A ⊢*n* ⟨*L*, *s*, *u*⟩ →*Del* ⟨*L'*, *s'*, *u'*⟩ *a* ≠ *Del* *A* ⊢*n* ⟨*L'*, *s'*, *u'*⟩ →*a* ⟨*L''*, *s''*, *u''*⟩

inductive *steps_un'* ::
(*'a*, *'t* :: *time*, *'s*) *unta* ⇒ *nat* ⇒ *'s list* ⇒ *int list* ⇒ (*nat*, *'t*) *cval*
⇒ *'s list* ⇒ *int list* ⇒ (*nat*, *'t*) *cval* ⇒ *bool*
(_ ⊢– ⟨_, _, _⟩ →* ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*,*61*] *61*)
where
refl: *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →* ⟨*L*, *s*, *u*⟩ |
step: *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →* ⟨*L'*, *s'*, *u'*⟩ ⟹ *A* ⊢*n* ⟨*L'*, *s'*, *u'*⟩ → ⟨*L''*, *s''*, *u''*⟩
⟹ *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →* ⟨*L''*, *s''*, *u''*⟩

declare *steps_un'.intros*[*intro*]

lemma *stepI2*:
A ⊢*n* ⟨*L*, *s*, *u*⟩ →* ⟨*L''*, *s''*, *u''*⟩ **if**
A ⊢*n* ⟨*L'*, *s'*, *u'*⟩ →* ⟨*L''*, *s''*, *u''*⟩ *A* ⊢*n* ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩
using *that* **by** *induction auto*

context *Equiv_TA*
begin

lemma *prod_correct'_action*:
(∃ *a*. *defs.prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →*a* ⟨(*L'*, *s'*), *u'*⟩) =
(∃ *a*. *state_ta* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩ ∧ *a* ≠ *Del*)
apply *standard*
subgoal
by (*blast elim*: *prod.prod_sound'_action*)
apply *clarify*
subgoal for *a*
apply (*cases a*; *simp*; *erule prod.prod_complete_silent prod.prod_complete_sync*, *fast*)
done
done

lemma *prod_correct'_delay*:
(∃ *d*. *defs.prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →*d* ⟨(*L'*, *s'*), *u'*⟩) =
state_ta ⊢ ⟨*L*, *s*, *u*⟩ →*Del* ⟨*L'*, *s'*, *u'*⟩
by (*blast dest*: *prod.prod_sound'_delay elim*: *prod.prod_complete_delay*)

lemma *equiv_correct*:
state_ta ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩ = *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩
by (*blast intro*!: *equiv_sound equiv_complete*)

lemma *prod_correct_action*:
(∃ *a*. *defs.prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →$_a$ ⟨(*L'*, *s'*), *u'*⟩) =
(∃ *a*. *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩ ∧ *a* ≠ *State_Networks.label.Del*)
unfolding *prod_correct'_action equiv_correct* **..**

lemma *prod_correct_delay*:
(∃ *d*. *defs.prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →^{d} ⟨(*L'*, *s'*), *u'*⟩) =
A ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_{Del}$ ⟨*L'*, *s'*, *u'*⟩
unfolding *prod_correct'_delay equiv_correct* **..**

lemma *prod_correct*:
defs.prod_ta ⊢ ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩ =
(∃ *a*. *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩)
apply *standard*
subgoal
using *prod_correct_action*[*of u L' s' u'*] *prod_correct_delay*[*of u L' s' u'*]
Timed_Automata.step.cases **by** *metis*
subgoal
apply *clarify*
subgoal for *a*
apply (*cases a*)
using *prod_correct_action*[*of u L' s' u'*] *prod_correct_delay*[*of u L' s' u'*]
Timed_Automata.step.intros **apply** *metis+*
done
done
done

context
assumes *0* < *p*
begin

lemmas *equiv_complete''* = *equiv_complete''*[*OF* _ ‹*0* < *p*›]

definition
all_prop L' s' ≡
(∀ *q*<*p*. ∃ *pc st s'' rs pcs*.
exec PF n ((*I* ! *q*) (*L'* ! *q*), [], *s'*, *True*, []) [] =
Some ((*pc*, *st*, *s''*, *True*, *rs*), *pcs*)
) ∧ *bounded B s'* ∧ *L'* ∈ *defs.states' s'* ~~(∀ q<p. ∃ pc st s'' rs pcs. exec PF n ((I ! q) (L'' ! q), [], s', True, []) [] = Some ((pc, st, s'', True, rs), pcs))~~

lemma *step_u_inv*:
all_prop L' s' **if** *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩
using *equiv_complete''*[*OF that*] *equiv_complete'*[*OF that*] **unfolding** *all_prop_def* **by** *auto*

lemma *step_inv*:
all_prop L' s' **if** *state_ta* ⊢ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩
using *step_u_inv*[*OF equiv_sound*[*OF that*]] **.**

lemma *Equiv_TA_I*:
Equiv_TA A n L' s' **if** ∗[*unfolded all_prop_def*]: *all_prop L' s'*
using ∗ **by** − (*standard*, *auto intro*!: *pred_time_indep upd_time_indep clock_conj Len*)

lemma *step_u'_inv*:
  all_prop L'' s'' ∧ *defs.prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L''*, *s''*), *u''*⟩
  if *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L''*, *s''*, *u''*⟩
using *that* **proof** *cases*
  case *prems*: (*1 L' s' u'*)
  from *step_u_inv*[*OF prems*(*1*)] **have** ∗[*unfolded all_prop_def*]: *all_prop L' s'* **.**
  interpret *equiv*: *Equiv_TA A n L' s'*
    using *Equiv_TA_I*[*OF step_u_inv*[*OF prems*(*1*)]] **.**
  from *equiv.step_u_inv*[*OF* ‹*0* < *p*› *prems*(*3*)] **show** *?thesis*
    using *prems prod_correct_delay*[*of u L' s' u'*] *equiv.prod_correct_action*[*of u' L'' s'' u''*]
      Timed_Automata.step'.intros
    by *metis*
qed

lemma *step'_inv*:
  all_prop L'' s'' ∧ *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L''*, *s''*, *u''*⟩
  if *defs.prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L''*, *s''*), *u''*⟩
using *that* **proof** *cases*
  case *prems*: (*step' d l' u' a*)
  obtain *L' s'* **where** *l'* = (*L'*, *s'*)
    by *force*
  from *step_inv prod_correct'_delay prems*(*1*) **have** ∗:
    all_prop L' s'
    unfolding ‹*l'* = _› **by** *fast*
  interpret *equiv*: *Equiv_TA A n L' s'*
    by (*rule Equiv_TA_I*[*OF* ∗])
  from *equiv.step_inv*[*OF* ‹*0* < *p*›] *equiv.prod_correct'_action prems*(*2*)[*unfolded* ‹*l'* = _›] **have**
    all_prop L'' s''
    by *metis*
  then show *?thesis*
    using *prems prod_correct_delay*[*of u L' s' u'*] *equiv.prod_correct_action*[*of u' L'' s'' u''*]
      step_u'.intros
    unfolding ‹*l'* = _› **by** *metis*
qed

lemma *prod_correct_step'*:
  defs.prod_ta ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩ =
  A ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩
  using *step'_inv step_u'_inv* **by** *blast*

lemma *all_prop_start*:
  all_prop L s
  using *Equiv_TA_axioms* **unfolding** *Equiv_TA_def all_prop_def* **by** *auto*

lemma *steps_u'_inv*:
  all_prop L'' s'' ∧ *defs.prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L''*, *s''*), *u''*⟩
  if *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ →∗ ⟨*L''*, *s''*, *u''*⟩
  using *that*
proof (*induction A* ≡ *A n* ≡ *n L* ≡ *L s* ≡ *s u L'' s'' u''*)
  case (*refl u*)
  show *?case* **using** *all_prop_start* **by** *auto*
next
  case (*step u L' s' u' L'' s'' u''*)
  then interpret *equiv*: *Equiv_TA A n L' s'*

```
    by (blast intro: Equiv_TA_I)
  from equiv.step_u'_inv[OF ‹0 < p› step.hyps(3)] step.hyps(1−2) show ?case
    by (blast intro: steps'_altI)
qed

lemma steps'_inv:
  all_prop L'' s'' ∧ A ⊢n ⟨L, s, u⟩ →* ⟨L'', s'', u''⟩
  if defs.prod_ta ⊢' ⟨(L, s), u⟩ →* ⟨(L'', s''), u''⟩
  using that all_prop_start
proof (induction defs.prod_ta (L, s) u (L'', s'') u'' arbitrary: L s)
  case (refl' u)
  then show ?case using all_prop_start by auto
next
  case (step' u l' u' u'' L s)
  obtain L' s' where l' = (L', s') by force
  from step' interpret equiv: Equiv_TA A n L s
    by (blast intro: Equiv_TA_I)
  from equiv.step'_inv[OF ‹0 < p› step'(1)[unfolded ‹l' = _›]] step'(3)[OF ‹l' = _›] show ?case
    by (auto intro: stepI2)
qed

lemma steps_un'_complete:
  defs.prod_ta ⊢' ⟨(L, s), u⟩ →* ⟨(L'', s''), u''⟩
  if A ⊢n ⟨L, s, u⟩ →* ⟨L'', s'', u''⟩
  using steps_u'_inv[OF that] ..

lemma steps'_sound:
  A ⊢n ⟨L, s, u⟩ →* ⟨L'', s'', u''⟩
  if defs.prod_ta ⊢' ⟨(L, s), u⟩ →* ⟨(L'', s''), u''⟩
  using steps'_inv[OF that] ..

lemma prod_reachable_correct:
  defs.prod_ta ⊢' ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩ ⟷ A ⊢n ⟨L, s, u⟩ →* ⟨L', s', u'⟩
  using steps'_sound steps_un'_complete by fast

lemma Bisimulation_Invariant_I:
  Bisimulation_Invariant
  (λ (L, s, u) (L', s', u'). defs.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
  (λ (L, s, u) (L', s', u'). A ⊢n ⟨L, s, u⟩ → ⟨L', s', u'⟩)
  (=)
  (λ (L, s, u). all_prop L s)
  (λ (L, s, u). all_prop L s)
  apply (standard; clarsimp)
  apply (simp add: Equiv_TA.prod_correct_step' Equiv_TA_I ‹0 < p›)+
  using Equiv_TA.step_u'_inv Equiv_TA_I ‹0 < p› apply blast+
  done

interpretation Bisimulation_Invariant
  λ (L, s, u) (L', s', u'). defs.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩
  λ (L, s, u) (L', s', u'). A ⊢n ⟨L, s, u⟩ → ⟨L', s', u'⟩
  (=)
  λ (L, s, u). all_prop L s
  λ (L, s, u). all_prop L s
  by (rule Bisimulation_Invariant_I)
```

end

end

definition *models* (_,_ ⊨_ _ [*61*,*61*] *61*) **where**
  A,a_0 ⊨$_n$ Φ ≡ (*case* Φ *of*
    formula.EX φ ⇒
      Graph_Defs.Ex_ev
        (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩)
        (λ (*L*, *s*, _). *check_bexp* φ *L* *s*)
  | *formula.EG* φ ⇒
      Graph_Defs.Ex_alw
        (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩)
        (λ (*L*, *s*, _). *check_bexp* φ *L* *s*)
  | *formula.AX* φ ⇒
      Graph_Defs.Alw_ev
        (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩)
        (λ (*L*, *s*, _). *check_bexp* φ *L* *s*)
  | *formula.AG* φ ⇒
      Graph_Defs.Alw_alw
        (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩)
        (λ (*L*, *s*, _). *check_bexp* φ *L* *s*)
  | *formula.Leadsto* φ ψ ⇒
      Graph_Defs.leadsto
        (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩)
        (λ (*L*, *s*, _). *check_bexp* φ *L* *s*)
        (λ (*L*, *s*, _). *check_bexp* ψ *L* *s*)
) a_0

definition
  has_deadlock *A* *n* a_0 ≡
    Graph_Defs.deadlock (λ (*L*, *s*, *u*) (*L*′, *s*′, *u*′). *A* ⊢^{n} ⟨*L*, *s*, *u*⟩ → ⟨*L*′, *s*′, *u*′⟩) a_0

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

context *Reachability_Problem*
begin

lemma *reaches_steps*′:
  reaches (*l*, *u*) (*l*′, *u*′) ⟷ *conv_A* *A* ⊢′ ⟨*l*, *u*⟩ →* ⟨*l*′, *u*′⟩
  apply *standard*
  subgoal premises *prems*
    using *prems* **by** (*induction* (*l*, *u*) (*l*′, *u*′) *arbitrary*: *l*′ *u*′) (*auto intro*: *steps*′*_altI*)
  subgoal premises *prems*
    using *prems* **by** *induction* (*auto intro*: *converse_rtranclp_into_rtranclp*)
  done

lemma *clocks_I*:
  (∀ *c*. *c* ∈ *clk_set* (*conv_A* *A*) ⟶ *u* *c* = *u*′ *c*) **if** ∀ *c* ∈ {*1*..*n*}. *u* *c* = *u*′ *c*
  unfolding *clk_set_conv_A* **using** *clocks_n* **using** *that* **by** *auto*

lemma *init_dbm_reaches_iff*:

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

proof −
interpret *ta_bisim*: *Bisimulation_Invariant*
$(\lambda(l,\ u)\ (l',\ u').\ conv_A\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda(l,\ u)\ (l',\ u').\ conv_A\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda(l,\ u)\ (l',\ u').\ l' = l \wedge (\forall\ c.\ c \in clk_set\ (conv_A\ A) \longrightarrow u\ c = u'\ c))$
$(\lambda_.\ True)\ (\lambda_.\ True)$
by (*rule ta_bisimulation*[*of conv_A A*])
show *?thesis*
apply *safe*
apply *force*
subgoal for *u1 u' u2*
unfolding *init_dbm_semantics reaches_steps'*[*symmetric*]
apply (*drule ta_bisim.A_B.simulation_reaches*[*of _ _* $(l_0,\ u2)$])
subgoal
using *clocks_I*[*of u1 u2*] **by** *fastforce*
by *auto*
subgoal for *u*
by *blast*
done
qed

theorem *reachable_decides_emptiness_new*:
$(\exists\ D'.\ E^{**}\ a_0\ (l',\ D') \wedge [curry\ (conv_M\ D')]_{v,n} \neq \{\})$
$\longleftrightarrow [curry\ (init_dbm :: real\ DBM')]_{v,n} \neq \{\} \wedge$
$(\forall\ u \in [curry\ init_dbm]_{v,n}.\ \exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u\rangle \rightarrow* \langle l',\ u'\rangle)$
unfolding *reachable_decides_emptiness init_dbm_reaches_iff* **..**

lemma *reachable_decides_emptiness'_new*:
$(\exists\ D'.\ E^{**}\ a_0\ (l',\ D') \wedge [curry\ (conv_M\ D')]_{v,n} \neq \{\})$
$\longleftrightarrow (\forall\ u.\ (\forall\ c \in \{1..n\}.\ u\ c = 0) \longrightarrow (\exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u\rangle \rightarrow* \langle l',\ u'\rangle))$
unfolding *reachable_decides_emptiness_new*
using *init_dbm_semantics' init_dbm_semantics'' init_dbm_non_empty* **by** *blast*

lemma *reachability_check_new_aux*:
$(\exists\ D'.\ E^{**}\ a_0\ (l',\ D') \wedge [curry\ (conv_M\ D')]_{v,n} \neq \{\} \wedge F\ l')$
$\longleftrightarrow (\forall\ u.\ (\forall\ c \in \{1..n\}.\ u\ c = 0) \longrightarrow (\exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u\rangle \rightarrow* \langle l',\ u'\rangle \wedge F\ l'))$
using *reachable_decides_emptiness'_new*[*of l'*] **by** *fast*

theorem *reachability_check_new*:
$(\exists\ D'.\ E^{**}\ a_0\ (l',\ D') \wedge F_rel\ (l',\ D'))$
$\longleftrightarrow (\forall\ u.\ (\forall\ c \in \{1..n\}.\ u\ c = 0) \longrightarrow (\exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u\rangle \rightarrow* \langle l',\ u'\rangle \wedge F\ l'))$
using *reachability_check_new_aux*[*of l'*] *check_diag_empty_spec reachable_empty_check_diag*
unfolding *F_rel_def* **by** *auto*

lemma *init_state_in_state_set*:
$l_0 \in state_set\ (trans_of\ A)$ **if** $\neg\ deadlock\ (l_0,\ u_0)$
proof −
obtain *l u* **where** $conv_A\ A \vdash' \langle l_0,\ u_0\rangle \rightarrow \langle l,\ u\rangle$
using ‹$\neg\ deadlock$ _› **unfolding** *deadlock_def deadlocked_def* **by** *force*
then have $l_0 \in state_set\ (trans_of\ (conv_A\ A))$

unfolding *state_set_def*
by *cases* (*auto elim*!: *step_a.cases step_t.cases*)
then show *?thesis*
unfolding *state_set_def* **unfolding** *trans_of_def conv_A_def* **by** (*cases A*) *force*
qed

lemma *init_state_in_state_set'*:
$l_0 \in$ *state_set* (*trans_of A*) **if** ($\forall\, u_0.\ (\forall\, c{\in}\{1..n\}.\ u_0\ c = 0) \longrightarrow \neg$ *deadlock* (l_0, u_0))
using *init_state_in_state_set that* **by** *auto*

end

context *Reachability_Problem_Impl*
begin

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

lemma *leadsto_spec_refine*:
leadsto_spec_alt Q
$\leq$ *SPEC* (λ *r*. $\neg$ *r* $\longleftrightarrow$
($\nexists\, x.\ (\lambda a\ b.\ E_op''.E_from_op\ a\ b \wedge \neg\ check_diag\ n\ (snd\ b))^{**}\ (l_0,\ init_dbm)\ x \wedge$
F (*fst x*) $\wedge$ *Q* (*fst x*) $\wedge$
($\exists\, a.\ (\lambda a\ b.\ E_op''.E_from_op\ a\ b \wedge \neg\ check_diag\ n\ (snd\ b) \wedge Q\ (fst\ b))^{**}\ x\ a \wedge$
$(\lambda a\ b.\ E_op''.E_from_op\ a\ b \wedge \neg\ check_diag\ n\ (snd\ b) \wedge Q\ (fst\ b))^{++}\ a\ a$)
))
proof −
have $*$:
($\lambda x\ y.$ (*case y of* (l', M') $\Rightarrow$ *E_op''.E_from_op x* (l', M') $\wedge \neg$ *check_diag n M'*) $\wedge$
$\neg$ (*case y of* (l, M) $\Rightarrow$ *check_diag n M*))
= ($\lambda a\ b.$ *E_op''.E_from_op a b* $\wedge \neg$ *check_diag n* (*snd b*))
by (*intro ext*) *auto*
have $**$:
($\lambda x\ y.$ (*case y of* (l', M') $\Rightarrow$ *E_op''.E_from_op x* (l', M') $\wedge \neg$ *check_diag n M'*) $\wedge$
(*case y of* (l, M) $\Rightarrow$ *Q l*) $\wedge \neg$ (*case y of* (l, M) $\Rightarrow$ *check_diag n M*))
= ($\lambda a\ b.$ *E_op''.E_from_op a b* $\wedge \neg$ *check_diag n* (*snd b*) $\wedge$ *Q* (*fst b*))
by (*intro ext*) *auto*
have $***$: $\neg$ *check_diag n b*
if ($\lambda a\ b.$ *E_op''.E_from_op a b* $\wedge \neg$ *check_diag n* (*snd b*))** a_0 (*a*, *b*) **for** *a b*
using *that* **by** *cases* (*auto simp*: a_0*_def*)
show *?thesis*
unfolding *leadsto_spec_alt_def*[*OF Q_fun*]
unfolding *PR_CONST_def* a_0*_def*[*symmetric*] **by** (*auto dest*: $***$ *simp*: $*$ $**$)
qed

lemma *leadsto_impl_hnr'*:
(*uncurry0*
(*leadsto_impl state_copy_impl*
(*succs_P_impl' Q_fun*) a_0*_impl subsumes_impl* (*return* $\circ$ *fst*)
succs_impl' emptiness_check_impl F_impl (*Q_impl Q_fun*) *tracei*),
uncurry0
(*SPEC*

```
    (λr. (∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ ¬ deadlock (l₀, u₀)) ⟶
        (¬ r) =
        (∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶
            leadsto (λ(l, u). F l) (λ(l, u). ¬ Q l) (l₀, u₀)))))
  ∈ unit_assn^k →ₐ bool_assn
proof −
  define p1 where p1 ≡
    (♯ x. (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b))** (l₀, init_dbm) x ∧
      F (fst x) ∧ Q (fst x) ∧
      (∃ a. (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))** x a ∧
        (λa b. E_op''.E_from_op a b ∧ ¬ check_diag n (snd b) ∧ Q (fst b))++ a a))
  define p2 where p2 ≡ (∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ ¬ deadlock (l₀, u₀))
  define p3 where
    p3 ≡ (∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ leadsto (λ(l, u). F l) (λ(l, u). ¬ Q l) (l₀, u₀))
  show ?thesis
  unfolding state_set_eq[symmetric]
  using Reachability_Problem_Impl_Op.leadsto_impl_hnr[OF Reachability_Problem_Impl_Op_axioms,
    OF Q_fun precond_a₀,
     FCOMP leadsto_spec_refine[THEN Id_SPEC_refine, THEN nres_relI], to_hnr, unfolded
hn_refine_def,
    of show_state show_clock
  ]
  using init_state_in_state_set'
  using leadsto_mc[of F Q]
  unfolding p1_def[symmetric] p2_def[symmetric] p3_def[symmetric]
  apply (simp del: state_set_eq)
  apply sepref_to_hoare
  apply sep_auto
  apply (erule cons_post_rule)
  apply sep_auto
  done
qed

end

end

context UPPAAL_Reachability_Problem_precompiled'
begin

lemma F_reachable_correct'_new:
  impl.op.F_reachable
  ⟷ (∃ L' s'. ∀ u. (∀ c ∈ {1..m}. u c = 0) ⟶ (∃ u'.
      conv_A A ⊢' ⟨(init, s₀), u⟩ →* ⟨(L', s'), u'⟩
      ∧ check_bexp φ L' s')
    ) if formula = formula.EX φ
  using
    that E_op''.E_from_op_reachability_check[of F_rel PR_CONST (λ(x, y). F x y)]
    reachability_check_new
  unfolding impl.E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
  unfolding F_rel_def unfolding F_def by force

lemma F_reachable_correct'_new':
  impl.op.F_reachable
```

```
  ⟷ (∃ L' s'. ∀ u. (∀ c ∈ {1..m}. u c = 0) ⟶ (∃ u'.
      conv_A A ⊢' ⟨(init, s₀), u⟩ →* ⟨(L', s'), u'⟩
      ∧ ¬ check_bexp φ L' s')
    ) if formula = formula.AG φ
  using
    that E_op''.E_from_op_reachability_check[of F_rel PR_CONST (λ(x, y). F x y)]
    reachability_check_new
  unfolding impl.E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def
  unfolding F_rel_def unfolding F_def by force

lemma F_reachable_correct_new:
  impl.op.F_reachable
  ⟷ (∃ L' s'. ∀ u. (∀ c ∈ {1..m}. u c = 0) ⟶ (∃ u'.
      conv N ⊢^m ax_steps ⟨init, s₀, u⟩ →* ⟨L', s', u'⟩
      ∧ check_bexp φ L' s')
    ) if formula = formula.EX φ
    unfolding F_reachable_correct'_new[OF that]
    apply (subst product'.prod_reachable_correct[symmetric])
    using prod_conv p_p p_gt_0 by simp+

lemma F_reachable_correct_new':
  impl.op.F_reachable
  ⟷ (∃ L' s'. ∀ u. (∀ c ∈ {1..m}. u c = 0) ⟶ (∃ u'.
      conv N ⊢^m ax_steps ⟨init, s₀, u⟩ →* ⟨L', s', u'⟩
      ∧ ¬ check_bexp φ L' s')
    ) if formula = formula.AG φ
    unfolding F_reachable_correct'_new'[OF that]
    apply (subst product'.prod_reachable_correct[symmetric])
    using prod_conv p_p p_gt_0 by simp+

definition
  Alw_ev_checker = dfs_map_impl'
    (impl.succs_P_impl' final_fun) impl.a₀_impl impl.subsumes_impl (return ∘ fst)
    impl.state_copy_impl

definition
  leadsto_checker ψ = do {
    r ← leadsto_impl
    impl.state_copy_impl (impl.succs_P_impl' (λ (L, s). ¬ check_bexp ψ L s))

    impl.a₀_impl impl.subsumes_impl (return ∘ fst)
    impl.succs_impl' impl.emptiness_check_impl impl.F_impl
    (impl.Q_impl (λ (L, s). ¬ check_bexp ψ L s))
    impl.tracei;
    return (¬ r)
  }

definition
  model_checker = (
  case formula of
    formula.EX _ ⇒ reachability_checker' |
    formula.AG _ ⇒ do {
      r ← reachability_checker';
      return (¬ r)
```

```
    } |
    formula.AX _ ⇒ do {
      r ← if PR_CONST (λ(x, y). F x y) (init, s₀)
      then Alw_ev_checker
      else return False;
      return (¬ r)
    } |
    formula.EG _ ⇒
      if PR_CONST (λ(x, y). F x y) (init, s₀)
      then Alw_ev_checker
      else return False |
    formula.Leadsto _ ψ ⇒ leadsto_checker ψ
  )


lemma p'_gt_0:
  0 < defs'.p
  unfolding p_p by (rule p_gt_0)

interpretation Bisim_A: Bisimulation_Invariant
  (λ(L, s, u) (L', s', u').
      defs'.defs.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
  (λ(L, s, u) (L', s', u').
      conv N ⊢^m ax_steps ⟨L, s, u⟩ → ⟨L', s', u'⟩)
  (=) (λ(L, s, u). product'.all_prop L s)
  (λ(L, s, u). product'.all_prop L s)
  by (rule product'.Bisimulation_Invariant_I) (rule p'_gt_0)

interpretation Bisim_B: Bisimulation_Invariant
  (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(L, s, u) (L', s', u').
      defs'.defs.prod_ta ⊢' ⟨(L, s), u⟩ → ⟨(L', s'), u'⟩)
   (λ (l, u') (L, s, u). (l, u') = ((L, s), u)) λ _. True
   (λ(L, s, u). product'.all_prop L s)
  unfolding prod_conv[symmetric]
  apply standard
  apply (solves auto)+
  by (rule Bisim_A.A_invariant)

interpretation Bisimulation_Invariant
   (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
   (λ(L, s, u) (L', s', u').
      conv N ⊢^m ax_steps ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λ (l, u') (L, s, u). (l, u') = ((L, s), u)) λ _. True
   (λ(L, s, u). product'.all_prop L s)
proof −
  interpret bisim: Bisimulation_Invariant
    (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
   (λ(L, s, u) (L', s', u').
      conv N ⊢^m ax_steps ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λa c. ∃ b. (case b of (L, s, u) ⇒ product'.all_prop L s) ∧
            (case a of (l, u') ⇒ λ(L, s, u). (l, u') = ((L, s), u)) b ∧ b = c)
   λ _. True
   (λ(L, s, u). product'.all_prop L s)
```

```
    using Bisimulation_Invariant_composition[OF
        Bisim_B.Bisimulation_Invariant_axioms Bisim_A.Bisimulation_Invariant_axioms
        ]
    .
  show Bisimulation_Invariant
    (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
    (λ(L, s, u) (L', s', u').
      conv N ⊢m ax_steps ⟨L, s, u⟩ → ⟨L', s', u'⟩)
    (λ(l, u') (L, s, u). (l, u') = ((L, s), u)) (λ_. True) (λ(L, s, u). product'.all_prop L s)
    apply standard
    subgoal for a b a'
     apply (drule bisim.A_B_step[of a b a'])
       apply auto
     done
    subgoal for a b a'
     apply (drule bisim.B_A_step[of b a' a])
       apply auto
     done
    apply simp
   apply (drule bisim.B_invariant)
    apply auto
   done
qed

lemma models_correct:
  conv N,(init, s0, u0) ⊨m ax_steps Φ = (case Φ of
    formula.EX φ ⇒
       (λ ((L, s), u). ∃ L' s' u'. conv_A A ⊢' ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩ ∧ check_bexp φ L' s')
  | formula.EG φ ⇒
      Not o Graph_Defs.Alw_ev
       (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
       (λ ((L, s), _). ¬ check_bexp φ L s)
  | formula.AX φ ⇒
      Graph_Defs.Alw_ev
       (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
       (λ ((L, s), _). check_bexp φ L s)
  | formula.AG φ ⇒
      Not o (λ ((L, s), u).
       ∃ L' s' u'. conv_A A ⊢' ⟨(L, s), u⟩ →* ⟨(L', s'), u'⟩ ∧ ¬ check_bexp φ L' s'
      )
  | formula.Leadsto φ ψ ⇒
      Graph_Defs.leadsto
       (λ (l, u) (l', u'). conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
       (λ ((L, s), _). check_bexp φ L s)
       (λ ((L, s), _). check_bexp ψ L s)
  ) ((init, s0), u0) if ¬ deadlock ((init, s0), u0)
proof −
  have ∗: ((Not ∘∘ case_prod) (λ(L, s) _. check_bexp φ L s)) =
    (λ((L, s), _). ¬ check_bexp φ L s) for φ by auto

  show ?thesis
    apply (subst models_def)
    apply (cases Φ)
    subgoal for φ
```

```
    apply simp

    apply (subst Ex_ev_iff[
          of (λ((L, s), _). check_bexp φ L s) _ ((init, s₀), u₀), symmetric, simplified
          ])
     apply (drule equiv'_D[simplified], force)
    using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
    apply (subst Ex_ev[OF that])
    unfolding reaches_steps'[symmetric]
    apply auto
    done
  subgoal for φ
    apply simp
    apply (subst Ex_alw_iff[
          of (λ((L, s), _). check_bexp φ L s) _ ((init, s₀), u₀), symmetric, simplified
          ])
     apply (drule equiv'_D[simplified]; force)
    using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
    unfolding Graph_Defs.Ex_alw_iff * ..
  subgoal for φ
    apply simp
    apply (subst Alw_ev_iff[
          of (λ((L, s), _). check_bexp φ L s) _ ((init, s₀), u₀), symmetric, simplified
          ])
     apply (drule equiv'_D[simplified]; force)
    using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
    unfolding Graph_Defs.Ex_alw_iff * ..
  subgoal for φ
    apply simp
    unfolding Graph_Defs.Alw_alw_iff
    apply (subst Ex_ev_iff[
          of (λ((L, s), _). ¬check_bexp φ L s) _ ((init, s₀), u₀), symmetric, simplified
          ])
     apply (drule equiv'_D[simplified], subst *[symmetric], force)
    using product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
    apply (subst Ex_ev[OF that])
    unfolding reaches_steps'[symmetric]
    apply auto
    done
  subgoal for φ ψ
    apply simp
    apply (subst Leadsto_iff[
          of (λ((L, s), _). check_bexp φ L s) _
            (λ((L, s), _). check_bexp ψ L s) _ ((init, s₀), u₀), symmetric, simplified
          ])
      apply (drule equiv'_D[simplified]; force)
     apply (drule equiv'_D[simplified]; force)
    using  product'.all_prop_start[OF p'_gt_0] apply (simp add: A_B.equiv'_def[simplified];
fail)
    ..
```

```
    done
qed

lemma final_fun_final':
  ((λ (L, s). P L s), (λ (L, s). P L s)) ∈ inv_rel Id states' for P
  unfolding F_def final_fun_def inv_rel_def in_set_member[symmetric] list_ex_iff
  by (force dest!: states'_states')

lemma final_fun_final[intro, simp]:
  ((λ (L, s). P L s), (λ (L, s). P L s)) ∈ inv_rel Id states for P
  using final_fun_final' states_states' by (rule inv_rel_mono)

abbreviation u_0 ≡ (λ _. 0 :: real)

lemma deadlock_start_iff:
  Bisim_A.B.deadlock (init, s_0, λ_. 0) ⟷ deadlock ((init, s_0), u_0)
  using product'.all_prop_start[OF p'_gt_0]
  by − (rule deadlock_iff[of _ (init, s_0, u_0), symmetric]; simp)

theorem model_check':
  (uncurry0 model_checker,
    uncurry0 (
      SPEC (λ r.
        ¬ Graph_Defs.deadlock
          (λ (L, s, u) (L', s', u'). conv N ⊢^m ax_steps ⟨L, s, u⟩ → ⟨L', s', u'⟩) (init, s_0, u_0) ⟶
        r = (conv N,(init, s_0, u_0) ⊨_m ax_steps formula)
      )
    )
  )
  ∈ unit_assn^k →_a bool_assn
proof −
  have ∗: (λ(l, u). ¬ (case l of (L, s) ⇒ (Not ∘∘∘ check_bexp) φ L s))
    = (λ((L, s), _). check_bexp φ L s) for φ
    by auto
  have ∗∗:
    (λ(l, u). ¬ (case l of (L, s) ⇒ check_bexp φ L s)) = (λ((L, s), _). ¬ check_bexp φ L s)
    for φ by auto
  have ∗∗∗:
    (λ(l, u). case l of (L, s) ⇒ φ L s) = (λ((L, s), _). φ L s) for φ
    by auto
  have ∗∗∗∗: (λ(L, y). (Not ∘∘∘ check_bexp) ψ L y) = (λ(L, y). ¬check_bexp ψ L y) for ψ
    by auto
  have ∗∗∗∗∗:
    return True = (return False ⤜ return o Not)
    by auto

  interpret ta_bisim: Bisimulation_Invariant
    (λ(l, u) (l', u').
      conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
    (λ(l, u) (l', u').
      conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
    (λ(l, u) (l', u').
      l' = l ∧
      (∀ c. c ∈ clk_set (conv_A A) ⟶
```

```
        u c = u' c))
   (λ_. True) (λ_. True)
   by (rule ta_bisimulation[of conv_A A])

  have bisim2:
   (∃ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ∧
              ¬ Alw_ev (λ(l, u). φ l) ((init, s₀), u₀))
   ⟷ (¬ Alw_ev (λ(l, u). φ l) ((init, s₀), u₀))
    for φ
   apply safe
   subgoal for u
    apply (subst (asm) ta_bisim.Alw_ev_iff[of _ (λ(l, u). φ l) _ ((init, s₀), λ_. 0)])
    using clocks_I[of u λ_. 0] unfolding ta_bisim.A_B.equiv'_def
     apply auto
    done
   by force

  have bisim1:
   (∃ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ∧ ¬ Alw_ev (λ((L, s), _). ¬ check_bexp φ L s) ((init, s₀),
u₀)) =
    (¬ Alw_ev (λ((L, s), _). ¬ check_bexp φ L s) ((init, s₀), u₀)) for φ
   using bisim2[of λ (L, s). ¬ check_bexp φ L s] unfolding *** .

  have bisim3:
   (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶
    leadsto (λ((L, s), _). check_bexp φ L s) (λ((L, s), _). check_bexp ψ L s) ((init, s₀), u₀)) =
    leadsto (λ((L, s), _). check_bexp φ L s) (λ((L, s), _). check_bexp ψ L s) ((init, s₀), u₀)
    for φ ψ
   apply safe
    apply force
   subgoal for u
    apply (subst (asm) ta_bisim.Leadsto_iff[of
          _ (λ((L, s), _). check_bexp φ L s) _ (λ((L, s), _). check_bexp ψ L s)
          _ ((init, s₀), u)
          ])
    using clocks_I[of u λ_. 0] unfolding ta_bisim.A_B.equiv'_def by auto
   done

  have bisim4:
   (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ ¬ deadlock ((init, s₀), u₀))
   ⟷ ¬ deadlock ((init, s₀), u₀)

   apply safe
    apply fast
   subgoal for u
    apply (subst (asm) ta_bisim.deadlock_iff[of _ ((init, s₀), u)])
    using clocks_I[of u λ_. 0] unfolding ta_bisim.A_B.equiv'_def by auto
   done

  have bisim5:
   (∀ u. (∀ c∈{1..m}. u c = 0) ⟶ (∃ u'. conv_A A ⊢' ⟨(init, s₀), u⟩ →* ⟨(L', s'), u'⟩ ∧ φ L' s'))
  ⟷ (∃ u'. conv_A A ⊢' ⟨(init, s₀), u₀⟩ →* ⟨(L', s'), u'⟩ ∧ φ L' s')
   for φ L' s'
   unfolding reaches_steps'[symmetric]
```

```
  apply safe
   apply fast
  subgoal for u' u
    apply (drule ta_bisim.bisim.A_B_reaches[of _ _ ((init, s₀), u)])
    subgoal
      using clocks_I[of u λ_. 0] unfolding ta_bisim.equiv'_def by auto
    unfolding ta_bisim.equiv'_def by auto
  done

define protect where
  protect = ((λ(l, u) (l', u').
                          conv_A A ⊢' ⟨l, u⟩ → ⟨l', u'⟩))

show ?thesis
  unfolding deadlock_start_iff
  using models_correct
  apply simp
  unfolding protect_def[symmetric]
  apply sepref_to_hoare
  apply sep_auto
 unfolding model_checker_def reachability_checker'_def Alw_ev_checker_def leadsto_checker_def
  apply (cases formula; simp)

    — EX
  subgoal premises prems for φ
    using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
    apply (subst (asm) (2) F_reachable_correct'_new)
     apply (rule prems; fail)
    apply (subst (asm) bisim5)
    apply sep_auto
    unfolding final_fun_def F_def prems
    apply (erule cons_post_rule)
    apply (sep_auto simp: pure_def)
    done

      — EG
  subgoal premises prems for φ
    using impl.Alw_ev_impl_hnr[
        to_hnr, unfolded hn_refine_def
        ]
    unfolding final_fun_def F_def prems(2)
      UPPAAL_Reachability_Problem_precompiled'.final_fun_def[
        OF UPPAAL_Reachability_Problem_precompiled'_axioms
        ]
      UPPAAL_Reachability_Problem_precompiled_defs.F_def
    apply sep_auto
    unfolding **
    subgoal
      apply (subst (asm) bisim1)
      apply (erule cons_post_rule)
      using init_state_in_state_set[of u₀]
      apply (sep_auto simp: pure_def protect_def ***)
      done
    subgoal
```

```
    apply (subst (asm) bisim1)
    apply simp
    apply (erule cons_post_rule)
    using init_state_in_state_set[of u0]
    apply (sep_auto simp: pure_def protect_def ∗∗∗)
    done
  done

    — AX
subgoal premises prems for φ
  using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def]
  unfolding final_fun_def F_def
  unfolding UPPAAL_Reachability_Problem_precompiled_defs.F_def
  apply (subst
      UPPAAL_Reachability_Problem_precompiled'.final_fun_def[
       OF UPPAAL_Reachability_Problem_precompiled'_axioms
       ])
  apply (safe; clarsimp simp: prems(2))
  unfolding bisim2
  unfolding ∗ ∗∗∗
  subgoal
    using init_state_in_state_set[of u0]
    by (sep_auto simp: pure_def protect_def)
  subgoal
    unfolding protect_def ∗∗∗∗∗
    apply (erule bind_rule)
    using init_state_in_state_set[of u0]
    apply (sep_auto simp: pure_def)
    done
  done

    — AG
subgoal premises prems for φ
  using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
  apply (subst (asm) (2) F_reachable_correct'_new')
   apply (rule prems; fail)
  apply (subst (asm) bisim5)
  unfolding final_fun_def F_def prems
  apply (sep_auto simp: pure_def)
  done

    — Leadsto
subgoal premises prems for φ ψ
  using impl.leadsto_impl_hnr'[
      OF final_fun_final, of Not oo check_bexp ψ,
      to_hnr, unfolded hn_refine_def
      ]
  unfolding ∗ F_def
  apply (simp add: prems(2))
  unfolding ∗∗∗ ∗∗∗∗ bisim3 bisim4
  apply (erule bind_rule)
  apply (sep_auto simp: pure_def protect_def)
  done
done
```

qed

theorem *model_check'_hoare*:
<*emp*>
model_checker
<λ*r*. ↑ ((¬ *Bisim_A.B.deadlock* (*init*, s_0, λ_. *0*)) ⟶ *r* = (
conv N,(*init*, s_0, u_0) ⊨$_m$ *ax_steps formula*
))>$_t$
using *model_check'*[*to_hnr*, *unfolded hn_refine_def*]
by (*sep_auto simp*: *pure_def elim*!: *cons_post_rule*)

lemma *Alw_ev_checker_alt_def'*:
Alw_ev_checker ≡
do {
x ← *let*
key = *return* ∘ *fst*;
sub = *impl.subsumes_impl*;
copy = *impl.state_copy_impl*;
start = *impl.*a_0*_impl*;
succs = *impl.succs_P_impl' final_fun*
in dfs_map_impl' succs start sub key copy;
_ ← *return* ();
return x
}
unfolding *Alw_ev_checker_def* **by** *simp*

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.*a_0*_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.emptiness_check_impl*;
trace = *impl.tracei*
in
leadsto_impl copy succs start sub key succs' empty final final' trace;
return (¬ *r*)
}
unfolding *leadsto_checker_def* **by** *simp*

schematic_goal *succs_P_impl_alt_def*:
impl.succs_P_impl (λ(*L*, *s*). *P L s*) ≡ *?impl* **for** *P*
unfolding *impl.succs_P_impl_def*[*OF final_fun_final*]
unfolding *k_impl_alt_def*
apply (*abstract_let trans_fun trans_fun*)
unfolding *inv_fun_def*[*abs_def*] *trans_fun_def*[*abs_def*] *trans_s_fun_def trans_i_fun_def trans_i_from_def*
apply (*abstract_let IArray* (*map IArray inv*) *inv_a*)

```
  apply (abstract_let IArray (map IArray trans_out_map) trans_out_map)
  apply (abstract_let IArray (map IArray trans_in_map) trans_in_map)
  apply (abstract_let IArray (map IArray trans_i_map) trans_i_map)
  by (rule Pure.reflexive)


lemmas succs_P'_impl_alt_def =
  impl.succs_P_impl'_def[OF final_fun_final, unfolded succs_P_impl_alt_def]

lemmas succs_impl'_alt_def =
  impl.succs_impl'_def[unfolded succs_impl_alt_def]

lemma reachability_checker'_alt_def':
  reachability_checker' ≡
    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.emptiness_check_impl;
        let tracei = impl.tracei;
        pw_impl key copy tracei sub start final succs empty
      };
      _ ← return ();
      return x
    }
  unfolding reachability_checker'_def by simp

schematic_goal reachability_checker'_alt_def:
  reachability_checker' ≡ ?impl
  unfolding reachability_checker'_alt_def' impl.succs_impl_def
  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
  unfolding k_impl_alt_def
  apply (abstract_let k_i k_i)

  apply (abstract_let trans_fun trans_fun)
  unfolding impl.init_dbm_impl_def impl.a₀_impl_def
  unfolding impl.F_impl_def
  unfolding final_fun_def[abs_def]
  unfolding impl.subsumes_impl_def
  unfolding impl.emptiness_check_impl_def
  unfolding impl.state_copy_impl_def
  by (rule Pure.reflexive)

schematic_goal Alw_ev_checker_alt_def:
  Alw_ev_checker ≡ ?impl
  unfolding Alw_ev_checker_alt_def' final_fun_def
    impl.succs_P_impl_def[OF final_fun_final] impl.succs_P_impl'_def[OF final_fun_final]
  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
```

unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)

apply (*abstract_let trans_fun trans_fun*)
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *impl.F_impl_def*
unfolding *final_fun_def*[*abs_def*]
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

schematic_goal *leadsto_checker_alt_def*:
leadsto_checker ≡ *?impl*
unfolding *leadsto_checker_alt_def'*
unfolding *impl.F_impl_def impl.Q_impl_def*[*OF final_fun_final*]
unfolding *impl.succs_P_impl'_def*[*OF final_fun_final*]
unfolding *impl.succs_impl'_def impl.succs_impl_def impl.succs_P_impl_def*[*OF final_fun_final*]
unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)

apply (*abstract_let trans_fun trans_fun*)
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *final_fun_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

schematic_goal *reachability_checker'_alt_def_refined*:
reachability_checker' ≡ *?impl*
unfolding *reachability_checker'_alt_def*
unfolding *fw_impl'_int*
unfolding *inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def*
unfolding *trans_i_from_impl*
unfolding *runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl*
apply (*abstract_let IArray* (*map IArray inv*) *inv_ia*)
apply (*abstract_let IArray* (*map IArray trans_out_map*) *trans_out_map*)
apply (*abstract_let IArray* (*map IArray trans_in_map*) *trans_in_map*)
apply (*abstract_let IArray* (*map IArray trans_i_map*) *trans_i_map*)
apply (*abstract_let IArray bounds bounds_ia*)
apply (*abstract_let PF PF*)
apply (*abstract_let PT PT*)
unfolding *PF_alt_def PT_alt_def*
apply (*abstract_let PROG' PROG'*)
unfolding *PROG'_def*
apply (*abstract_let length prog len_prog*)
apply (*abstract_let IArray* (*map* (*map_option stripf*) *prog*) *progf_ia*)
apply (*abstract_let IArray* (*map* (*map_option stript*) *prog*) *progt_ia*)
apply (*abstract_let IArray prog prog_ia*)
unfolding *all_actions_by_state_impl*
apply (*abstract_let* [*0..<p*])

apply (*abstract_let* [*0..<na*])
apply (*abstract_let* {*0..<p*})
apply (*abstract_let* [*0..<m+1*])
by (*rule Pure.reflexive*)

schematic_goal *Alw_ev_checker_alt_def_refined*:
Alw_ev_checker ≡ *?impl*
unfolding *Alw_ev_checker_alt_def*
unfolding *fw_impl'_int*
unfolding *inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def*
unfolding *trans_i_from_impl*
unfolding *runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl*
apply (*abstract_let IArray* (*map IArray inv*) *inv_ia*)
apply (*abstract_let IArray* (*map IArray trans_out_map*) *trans_out_map*)
apply (*abstract_let IArray* (*map IArray trans_in_map*) *trans_in_map*)
apply (*abstract_let IArray* (*map IArray trans_i_map*) *trans_i_map*)
apply (*abstract_let IArray bounds bounds*)
apply (*abstract_let PF PF*)
apply (*abstract_let PT PT*)
unfolding *PF_alt_def PT_alt_def*
apply (*abstract_let PROG' PROG'*)
unfolding *PROG'_def*
apply (*abstract_let length prog len_prog*)
apply (*abstract_let IArray* (*map* (*map_option stripf*) *prog*) *progf_ia*)
apply (*abstract_let IArray* (*map* (*map_option stript*) *prog*) *progt_ia*)
apply (*abstract_let IArray prog prog*)
unfolding *all_actions_by_state_impl*
apply (*abstract_let* [*0..<p*])
apply (*abstract_let* [*0..<na*])
apply (*abstract_let* {*0..<p*})
apply (*abstract_let* [*0..<m+1*])
by (*rule Pure.reflexive*)

schematic_goal *leadsto_checker_alt_def_refined*:
leadsto_checker ≡ *?impl*
unfolding *leadsto_checker_alt_def*
unfolding *fw_impl'_int*
unfolding *inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def*
unfolding *trans_i_from_impl*
unfolding *runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl*
apply (*abstract_let IArray* (*map IArray inv*) *inv_ia*)
apply (*abstract_let IArray* (*map IArray trans_out_map*) *trans_out_map*)
apply (*abstract_let IArray* (*map IArray trans_in_map*) *trans_in_map*)
apply (*abstract_let IArray* (*map IArray trans_i_map*) *trans_i_map*)
apply (*abstract_let IArray bounds bounds_ia*)
apply (*abstract_let PF PF*)
apply (*abstract_let PT PT*)
unfolding *PF_alt_def PT_alt_def*
apply (*abstract_let PROG' PROG'*)
unfolding *PROG'_def*
apply (*abstract_let length prog len_prog*)
apply (*abstract_let IArray* (*map* (*map_option stripf*) *prog*) *progf_ia*)
apply (*abstract_let IArray* (*map* (*map_option stript*) *prog*) *progt_ia*)
apply (*abstract_let IArray prog prog_ia*)

unfolding *all_actions_by_state_impl*
apply (*abstract_let* [*0..<p*])
apply (*abstract_let* [*0..<na*])
apply (*abstract_let* {*0..<p*})
apply (*abstract_let* [*0..<m+1*])
by (*rule Pure.reflexive*)

end

concrete_definition *reachability_checker'*
uses *UPPAAL_Reachability_Problem_precompiled'.reachability_checker'_alt_def_refined*

concrete_definition *Alw_ev_checker*
uses *UPPAAL_Reachability_Problem_precompiled'.Alw_ev_checker_alt_def_refined*

concrete_definition *leadsto_checker*
uses *UPPAAL_Reachability_Problem_precompiled'.leadsto_checker_alt_def_refined*

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemmas *model_checker_def_refined = model_checker_def*[*unfolded*
reachability_checker'.refine[*OF UPPAAL_Reachability_Problem_precompiled'_axioms*]
Alw_ev_checker.refine[*OF UPPAAL_Reachability_Problem_precompiled'_axioms*]
leadsto_checker.refine[*OF UPPAAL_Reachability_Problem_precompiled'_axioms*]
]

end

concrete_definition *model_checker* **uses**
UPPAAL_Reachability_Problem_precompiled'.model_checker_def_refined

definition [*code*]:
precond_mc p m k max_steps I T prog final bounds P s_0 *na* ≡
if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s_0 *na k*
then
model_checker p m max_steps I T prog bounds P s_0 *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* s_0 *na*
<λ *Some r* ⇒ ↑(
UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s_0 *na k*
∧
(¬ *has_deadlock* (*conv* (*N p I P T prog bounds*)) *max_steps* (*repeat 0 p*, s_0, λ_ . *0*) ⟶
r = *conv* (*N p I P T prog bounds*),(*repeat 0 p*, s_0, λ_ . *0*) ⊨$_m$*ax_steps formula*
))
| *None* ⇒ ↑(¬ *UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds*
P s_0 *na k*)
>$_t$
proof −
define *A* **where** *A* ≡ *conv* (*N p I P T prog bounds*)
define *check* **where** *check* ≡ *A*,(*repeat 0 p*, s_0, λ_ . *0*) ⊨$_m$*ax_steps formula*

```
  note [sep_heap_rules] =
    UPPAAL_Reachability_Problem_precompiled'.model_check'_hoare[
      of p m max_steps I T prog bounds P s₀ na k formula,
      unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
      folded A_def check_def has_deadlock_def
      ]
  show ?thesis
    unfolding UPPAAL_Reachability_Problem_precompiled_defs.init_def
    unfolding A_def[symmetric] check_def[symmetric]
    unfolding precond_mc_def by (sep_auto simp: model_checker.refine[symmetric])
qed

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) ⊨_m ax_steps formula
      ))
    else r = None
    )>_t
proof −
  define A where A ≡ conv (N p I P T prog bounds)
  define check where check ≡ A,(repeat 0 p, s₀, λ_ . 0) ⊨_m ax_steps formula
  note [sep_heap_rules] =
    UPPAAL_Reachability_Problem_precompiled'.model_check'_hoare[
      of p m max_steps I T prog bounds P s₀ na k formula,
      unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
      folded A_def check_def has_deadlock_def
      ]
  show ?thesis
    unfolding UPPAAL_Reachability_Problem_precompiled_defs.init_def
    unfolding A_def[symmetric] check_def[symmetric]
    unfolding precond_mc_def by (sep_auto simp: model_checker.refine[symmetric])
qed

prepare_code_thms dfs_map_impl'_def leadsto_impl_def

lemmas [code] =
  reachability_checker'_def
  Alw_ev_checker_def
  leadsto_checker_def
  model_checker_def[unfolded UPPAAL_Reachability_Problem_precompiled_defs.F_def PR_CONST_def]

export_code
  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 $'b \Rightarrow 'b \Rightarrow 'b$ $('a, 'b)$ *exp* $('a, 'b)$ *exp* | *unop* $'b \Rightarrow 'b$ $('a, 'b)$ *exp*

inductive *check_bexp* :: $('a \rightharpoonup 'b) \Rightarrow ('a, 'b :: linorder)$ *bexp* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **and** *is_val*
where
check_bexp s true True |
check_bexp s (*not e*) $(\neg\ b)$ **if** *check_bexp s e b* |
check_bexp s (*and e1 e2*) $(a \wedge b)$ **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*or e1 e2*) $(a \vee b)$ **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*imply e1 e2*) $(a \longrightarrow b)$ **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*eq a b*) $(x = v)$ **if** *is_val s a v is_val s b x* |
check_bexp s (*le a b*) $(v \leq x)$ **if** *is_val s a v is_val s b x* |
check_bexp s (*lt a b*) $(v < x)$ **if** *is_val s a v is_val s b x* |
check_bexp s (*ge a b*) $(v \geq x)$ **if** *is_val s a v is_val s b x* |
check_bexp s (*gt a b*) $(v > x)$ **if** *is_val s a v is_val s b x* |
is_val s (*const c*) *c* |
is_val s (*var x*) *v* **if** *s x* = *Some v* |
is_val s (*if_then_else b e1 e2*) (*if bv then v1 else v2*)
if *is_val s e1 v1 is_val s e2 v2 check_bexp s b bv* |
is_val s (*binop f e1 e2*) (*f v1 v2*) **if** *is_val s e1 v1 is_val s e2 v2* |
is_val s (*unop f e*) (*f v*) **if** *is_val s e v*

inductive_simps *check_bexp_simps*:
check_bexp s bexp.true bv
check_bexp s (*bexp.not b*) *bv*
check_bexp s (*bexp.and b1 b2*) *bv*
check_bexp s (*bexp.or b1 b2*) *bv*
check_bexp s (*bexp.imply b1 b2*) *bv*
check_bexp s (*le i x*) *bv*
check_bexp s (*lt i x*) *bv*
check_bexp s (*ge i x*) *bv*
check_bexp s (*eq i x*) *bv*
check_bexp s (*gt i x*) *bv*

inductive_simps *is_val_simps*:
is_val s (*const c*) *d*
is_val s (*var x*) *v*
is_val s (*if_then_else b e1 e2*) *v*
is_val s (*binop f e1 e2*) *v*
is_val s (*unop f e*) *v*

inductive_cases *check_bexp_elims*:
check_bexp s bexp.true bv
check_bexp s (bexp.not b) bv
check_bexp s (bexp.and b1 b2) bv
check_bexp s (bexp.or b1 b2) bv
check_bexp s (bexp.imply b1 b2) bv
check_bexp s (le i x) bv
check_bexp s (lt i x) bv
check_bexp s (ge i x) bv
check_bexp s (eq i x) bv
check_bexp s (gt i x) bv

inductive_cases *is_val_elims*:
is_val s (const c) d
is_val s (var x) v
is_val s (if_then_else b e1 e2) v
is_val s (binop f e1 e2) v
is_val s (unop f e) v

type_synonym
*('a, 'b) upd = ('a * ('a, 'b) exp) list*

definition *is_upd* **where**
is_upd s upd s' ≡ ∃ x e v. upd = (x, e) ∧ is_val s e v ∧ 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''*
using *assms* **by** *induction* (*auto intro*: *is_upds.intros*)

lemma *is_upds_appendE*:
assumes *is_upds s (upds1 @ upds2) s''*
obtains *s'* **where** *is_upds s upds1 s' is_upds s' upds2 s''*
using *assms* **by** (*induction upds1 arbitrary*: *s*) (*auto intro*: *is_upds.intros elim*!: *is_upds_ConsE*)

lemma *is_upds_NilD*:
s' = s **if** *is_upds s [] s'*
using *that* **by** (*rule is_upds_NilE*)

lemma *is_upds_all_NilD*:

assumes *is_upds s* (*concat upds*) *s'* (∀ *xs*∈*set upds*. *xs* = []) **shows** *s'* = *s* **using** *assms* **by** (*simp del*: *Nil_eq_concat_conv flip*: *concat_eq_Nil_conv*) (*rule is_upds_NilD*)

lemma *is_upd_const_simp*: *is_upd s* (*x*, *const c*) *s'* ⟷ *s'* = *s*(*x* := *Some c*) **unfolding** *is_upd_def* **by** (*simp add*: *is_val_simps*)

end
theory *Simple_Network_Language*
imports *Main*
Munta_Model_Checker.State_Networks
Munta_Model_Checker.UPPAAL_State_Networks_Impl
FinFun.FinFun
Simple_Expressions
Munta_Tagging
begin

6 Simple Networks of Automata With Broadcast Channels and Committed Locations

no_notation *top_assn* (*true*)

abbreviation *concat_map* **where** *concat_map f xs* ≡ *concat* (*map f xs*)

type_synonym ('*c*, '*t*, '*s*) *invassn* = '*s* ⇒ ('*c*, '*t*) *cconstraint*

type_synonym ('*a*, '*s*, '*c*, '*t*, '*x*, '*v*) *transition* = '*s* × ('*x*, '*v*) *bexp* × ('*c*, '*t*) *cconstraint* × '*a* × ('*x*, '*v*) *upd* × '*c list* × '*s*

type_synonym ('*a*, '*s*, '*c*, '*t*, '*x*, '*v*) *sta* = '*s set* × '*s set* × ('*a*, '*s*, '*c*, '*t*, '*x*, '*v*) *transition set* × ('*c*, '*t*, '*s*) *invassn*

type_synonym ('*a*, '*s*, '*c*, '*t*, '*x*, '*v*) *nta* = '*a set* × ('*a act*, '*s*, '*c*, '*t*, '*x*, '*v*) *sta list* × ('*x* ⇀ '*v* ∗ '*v*)

context begin

qualified definition *conv_t* **where** *conv_t* ≡ λ (*l*,*b*,*g*,*a*,*f*,*r*,*l'*). (*l*,*b*,*conv_cc g*,*a*,*f*,*r*,*l'*)

qualified definition *conv_A* **where** *conv_A* ≡ λ (*C*, *U*, *T*, *I*). (*C*, *U*, *conv_t* ‘ *T*, *conv_cc o I*)

definition *conv* **where** *conv* ≡ λ(*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 ≡ dom s = dom bounds ∧
(∀ x ∈ dom s. fst (the (bounds x)) ≤ the (s x) ∧ the (s x) ≤ snd (the (bounds x)))

definition *committed* :: *('a, 's, 'c, 't, 'x, 'v) sta ⇒ 's set* **where**
committed A ≡ fst A

definition *urgent* :: *('a, 's, 'c, 't, 'x, 'v) sta ⇒ 's set* **where**
urgent A ≡ fst (snd A)

definition *trans* :: *('a, 's, 'c, 't, 'x, 'v) sta ⇒ ('a, 's, 'c, 't, 'x, 'v) transition set*
where
trans A ≡ fst (snd (snd A))

definition *inv* :: *('a, 's, 'c, 't, 'x, 'v) sta ⇒ ('c, 't, 's) invassn* **where**
inv A ≡ snd (snd (snd A))

no_notation *step_sn* (*_ ⊢ ⟨_, _, _⟩ →_ ⟨_, _, _⟩ [61,61,61,61,61] 61*)
no_notation *steps_sn* (*_ ⊢ ⟨_, _, _⟩ →∗ ⟨_, _, _⟩ [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 ⇒ 's list ⇒ ('x ⇀ 'v) ⇒ ('c, 't) cval
⇒ 'a label ⇒ 's list ⇒ ('x ⇀ 'v) ⇒ ('c, 't) cval ⇒ bool
(_ ⊢ ⟨_, _, _⟩ →_ ⟨_, _, _⟩ [61,61,61,61,61] 61)
where
step_t:
⟦
''target invariant'' | ∀ p < length N. u ⊕ d ⊢ inv (N ! p) (L ! p);
''nonnegative'' | d ≥ 0;
''urgent'' | (∃ p < length N. L ! p ∈ urgent (N ! p)) ⟶ d = 0;
''bounded'' | bounded B s
⟧
⟹ (broadcast, N, B) ⊢ ⟨L, s, u⟩ →Del ⟨L, s, u ⊕ d⟩ |
step_int:
⟦
TRANS ''silent'' | (l, b, g, Sil a, f, r, l′) ∈ trans (N ! p);
''committed'' | l ∈ committed (N ! p) ∨ (∀ p < length N. L ! p ∉ committed (N ! p));
''bexp'' | check_bexp s b True;
''guard'' | u ⊢ g;
''target invariant'' | ∀ p < length N. u′ ⊢ inv (N ! p) (L′ ! p);
''loc'' | L!p = l;
''range'' | p < length L;
''new loc'' | L′ = L[p := l′];
''new valuation'' | u′ = [r→0]u;
''is_upd'' | is_upds s f s′;
''bounded'' | bounded B s′
⟧
⟹ (broadcast, N, B) ⊢ ⟨L, s, u⟩ →Internal a ⟨L′, s′, u′⟩ |
step_bin:

```
  ⟦
    ''not broadcast'' | (a ∉ broadcast);
    TRANS ''in''  | (l1, b1, g1, In a,  f1, r1, l1') ∈ trans (N ! p);
    TRANS ''out'' | (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N ! q);
    ''committed'' |
      l1 ∈ committed (N ! p) ∨ l2 ∈ committed (N ! q) ∨ (∀ p < length N. L ! p ∉ committed
(N ! p));
    ''bexp'' | check_bexp s b1 True; ''bexp'' | check_bexp s b2 True;
    ''guard'' | u ⊢ g1; ''guard'' | u ⊢ g2;
    ''target invariant'' | ∀ p < length N. u' ⊢ inv (N ! p) (L' ! p);
    RECV ''loc'' | L!p = l1; SEND ''loc'' | L!q = l2;
    RECV ''range'' | p < length L; SEND ''range'' | q < length L;
    ''different'' | p ≠ q;
    ''new loc''       | L' = L[p := l1', q := l2'];
    ''new valuation'' | u' = [r1@r2→0]u;
    ''upd'' | is_upds s f1 s'; ''upd'' | is_upds s' f2 s'';
    ''bounded'' | bounded B s''
  ⟧
  ⟹ (broadcast, N, B) ⊢ ⟨L, s, u⟩ →Bin a ⟨L', s'', u'⟩ |
 step_broad:
  ⟦
    ''broadcast'' | a ∈ broadcast;
    TRANS ''out'' | (l, b, g, Out a, f, r, l') ∈ trans (N ! p);
    TRANS ''in''  | (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N ! p));
    ''committed'' | (l ∈ committed (N ! p)) ∨ (∃ p ∈ set ps. L ! p ∈ committed (N ! p))
    ∨ (∀ p < length N. L ! p ∉ committed (N ! p)));
    ''bexp''    | check_bexp s b True;
    ''bexp''    | ∀ p ∈ set ps. check_bexp s (bs p) True;
    ''guard''   | u ⊢ g;
    ''guard''   | ∀ p ∈ set ps. u ⊢ gs p;
    ''maximal'' | ∀ q < length N. q ∉ set ps ∧ p ≠ q
      ⟶ (∀ b g f r l'. (L!q, b, g, In a, f, r, l') ∈ trans (N ! q)
      ⟶ ¬ check_bexp s b True ∨ ¬ u ⊢ g);
    ''target invariant'' | ∀ p < length N. u' ⊢ inv (N ! p) (L' ! p);
    SEND ''loc''       | L!p = l;
    SEND ''range''     | p < length L;
    SEL ''range''      | set ps ⊆ {0..<length N};
    SEL ''not sender'' | p ∉ set ps;
    SEL ''distinct''   | distinct ps;
    SEL ''sorted''     | sorted ps;
    ''new loc'' | L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'];
    ''new valuation'' | u' = [r@concat (map rs ps)→0]u;
    ''upd''     | is_upds s f s';
    ''upds''    | is_upds s' (concat_map fs ps) s'';
    ''bounded'' | bounded B s''
  ⟧
  ⟹ (broadcast, N, B) ⊢ ⟨L, s, u⟩ →Broad a ⟨L', s'', u'⟩
lemmas [intro?] = step_u.intros[unfolded TAG_def]
```

Comments:

- Should there be an error transition + state if states run of bounds or updates are undefined? Then one could run a reachability check for the error state.
- Should the state be total?

- Note that intermediate states are allowed to run out of bounds

definition *steps_u* ::
  ('*a*, '*s*, '*c*, '*t* :: *time*, '*x*, '*v* :: *linorder*) *nta* ⇒ '*s list* ⇒ ('*x* ⇀ '*v*) ⇒ ('*c*, '*t*) *cval*
  ⇒ '*s list* ⇒ ('*x* ⇀ '*v*) ⇒ ('*c*, '*t*) *cval* ⇒ *bool*
(_ ⊢ ⟨_, _, _⟩ →* ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*] *61*)
where
  A ⊢ ⟨*L*, *s*, *u*⟩ →* ⟨*L'*, *s'*, *u'*⟩ ≡
    (λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). ∃ *a*. *A* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩)** (*L*, *s*, *u*) (*L'*, *s'*, *u'*)

Misc **lemma** *clock_val_concat_iff*:
  ‹*u* ⊢ *concat gs* ⟷ (∀ *g* ∈ *set gs*. *u* ⊢ *g*)›
  by (*auto intro*: *guard_concat concat_guard*)

lemma *clock_val_append_iff*:
  ‹*u* ⊢ *g1* @ *g2* ⟷ *u* ⊢ *g1* ∧ *u* ⊢ *g2*›
proof −
  have *u* ⊢ *g1* @ *g2* ⟷ *u* ⊢ *concat* [*g1*, *g2*]
    by *simp*
  also have ... ⟷ *u* ⊢ *g1* ∧ *u* ⊢ *g2*
    unfolding *clock_val_concat_iff* **by** *simp*
  finally show *?thesis* **.**
qed

6.1 Product Construction

locale *Prod_TA_Defs* =
  fixes *A* :: ('*a*, '*s*, '*c*, '*t*, '*x*, '*v* :: *linorder*) *nta*
begin

definition
  broadcast = *fst A*

definition
  N i = *fst* (*snd A*) ! *i*

definition
  bounds = *snd* (*snd A*)

definition — Number of processes
  n_ps = *length* (*fst* (*snd A*))

definition *states* :: '*s list set* **where**
  states ≡ {*L*. *length L* = *n_ps* ∧
    (∀ *i*. *i* < *n_ps* ⟶ *L* ! *i* ∈ (⋃ (*l*, *e*, *g*, *a*, *r*, *u*, *l'*) ∈ (*trans* (*N i*)). {*l*, *l'*}))}

definition
  prod_inv ≡ λ(*L*, *s*). *if L* ∈ *states then concat* (*map* (λ*i*. *inv* (*N i*) (*L* ! *i*)) [*0*..<*n_ps*]) *else* []

definition
  trans_int =
    {((*L*, *s*), *g*, *Internal a*, *r*, (*L'*, *s'*)) | *L s l b g f p a r l' L' s'*.
      (*l*, *b*, *g*, *Sil a*, *f*, *r*, *l'*) ∈ *trans* (*N p*) ∧
      (*l* ∈ *committed* (*N p*) ∨ (∀ *p* < *n_ps*. *L* ! *p* ∉ *committed* (*N p*))) ∧

$L!p = l \wedge p < length\ L \wedge L' = L[p := l'] \wedge is_upds\ s\ f\ s' \wedge check_bexp\ s\ b\ True \wedge$
$L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s'$
}

definition
trans_bin =
{((*L*, *s*), *g1* @ *g2*, *Bin a*, *r1* @ *r2*, (*L'*, *s''*)) |
L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
$a \notin broadcast \wedge$
$(l1, b1, g1, In\ a, f1, r1, l1') \in trans\ (N\ p) \wedge$
$(l2, b2, g2, Out\ a, f2, r2, l2') \in trans\ (N\ q) \wedge$
$(l1 \in committed\ (N\ p) \vee l2 \in committed\ (N\ q) \vee (\forall p < n_ps.\ L\ !\ p \notin committed\ (N\ p))) \wedge$
$L!p = l1 \wedge L!q = l2 \wedge p < length\ L \wedge q < length\ L \wedge p \neq q \wedge$
$check_bexp\ s\ b1\ True \wedge check_bexp\ s\ b2\ True \wedge$
$L' = L[p := l1', q := l2'] \wedge is_upds\ s\ f1\ s' \wedge is_upds\ s'\ f2\ s'' \wedge$
$L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s''$
}

definition
trans_broad =
{((*L*, *s*), *g* @ *concat* (*map gs ps*), *Broad a*, *r* @ *concat* (*map rs ps*), (*L'*, *s''*)) |
L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
$a \in broadcast\ \wedge$
$(l, b, g, Out\ a, f, r, l') \in trans\ (N\ p) \wedge$
$(\forall p \in set\ ps.\ (L\ !\ p, bs\ p, gs\ p, In\ a, fs\ p, rs\ p, ls'\ p) \in trans\ (N\ p)) \wedge$
$(l \in committed\ (N\ p)) \vee (\exists p \in set\ ps.\ L\ !\ p \in committed\ (N\ p))$
$\vee (\forall p < n_ps.\ L\ !\ p \notin committed\ (N\ p))) \wedge$
$(\forall q < n_ps.\ q \notin set\ ps \wedge p \neq q \longrightarrow$
$\neg (\exists b\ g\ f\ r\ l'.\ (L!q, b, g, In\ a, f, r, l') \in trans\ (N\ q) \wedge check_bexp\ s\ b\ True)) \wedge$
$L!p = l \wedge$
$p < length\ L \wedge set\ ps \subseteq \{0..<n_ps\} \wedge p \notin set\ ps \wedge distinct\ ps \wedge sorted\ ps \wedge$
$check_bexp\ s\ b\ True \wedge (\forall p \in set\ ps.\ check_bexp\ s\ (bs\ p)\ True) \wedge$
$L' = (fold\ (\lambda p\ L\ .\ L[p := ls'\ p])\ ps\ L)[p := l'] \wedge$
$is_upds\ s\ f\ s' \wedge is_upds\ s'\ (concat_map\ fs\ ps)\ s'' \wedge$
$L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s''$
}

definition
trans_prod = *trans_int* ∪ *trans_bin* ∪ *trans_broad*

definition
prod_ta = (*trans_prod*, *prod_inv* :: ('*s list* × ('*x* ⇒ '*v option*) ⇒ _))

lemma *inv_of_prod*[*simp*]:
inv_of prod_ta = *prod_inv*
unfolding *prod_ta_def inv_of_def* **by** *simp*

lemma *trans_of_prod*[*simp*]:
trans_of prod_ta = *trans_prod*
unfolding *prod_ta_def trans_of_def* **by** *simp*

lemma *A_split*:
‹*A* = (*broadcast*, *map N* [*0..<n_ps*], *bounds*)›
unfolding *broadcast_def N_def bounds_def n_ps_def* **by** (*cases A*) (*simp add*: *List.map_nth*)

lemma *map_N_n_ps_simp*:
map N [*0*..<*n_ps*] ! *p* = *N p*
unfolding *N_def n_ps_def List.map_nth* **..**

lemma *N_split_simp*[*simp*]:
assumes *A* = (*broadcast'*, *N'*, *B*)
shows *N'* ! *i* = *N i*
unfolding *N_def* **unfolding** *assms* **by** *simp*

lemma *state_preservation_updI*:
assumes *l'* ∈ (⋃ (*l*, *b*, *g*, *a*, *r*, *u*, *l'*) ∈ *trans* (*N p*). {*l*, *l'*}) *L* ∈ *states*
shows *L*[*p* := *l'*] ∈ *states*
using *assms* **unfolding** *states_def* **by** (*fastforce simp*: *nth_list_update'*)

lemma *state_preservation_fold_updI*:
assumes ∀ *p* ∈ *set ps*. *ls' p* ∈ (⋃ (*l*, *b*, *g*, *a*, *r*, *u*, *l'*) ∈ *trans* (*N p*). {*l*, *l'*}) *L* ∈ *states*
shows *fold* (λ*p L*. *L*[*p* := *ls' p*]) *ps L* ∈ *states*
using *assms* **by** (*induction ps arbitrary*: *L*) (*auto intro*: *state_preservation_updI*)

lemma *state_set_states*:
Simulation_Graphs_TA.state_set prod_ta ⊆ {(*l*, *s*). *l* ∈ *states*}
unfolding *prod_ta_def state_set_def*
unfolding *trans_of_def trans_prod_def*
unfolding *trans_int_def trans_bin_def trans_broad_def*
by *auto* (*auto intro*: *state_preservation_updI state_preservation_fold_updI*)

lemma *trans_prod_bounded_inv*:
‹*bounded bounds s'*› **if** ‹((*L*, *s*), *g*, *a*, *r*, (*L'*, *s'*)) ∈ *trans_prod*›
using *that* **unfolding** *bounds_def trans_prod_def trans_int_def trans_bin_def trans_broad_def*
by (*auto simp*: *bounds_def*)

lemma *trans_prod_states_inv*:
‹*L'* ∈ *states*› **if** ‹((*L*, *s*), *g*, *a*, *r*, (*L'*, *s'*)) ∈ *trans_prod*› ‹*L* ∈ *states*›
using *that*
unfolding *bounds_def trans_prod_def trans_int_def trans_bin_def trans_broad_def*
apply *clarsimp*
apply *safe*
apply (*force intro*!: *state_preservation_updI*)
apply (*force intro*!: *state_preservation_updI*)
apply (*force intro*!: *state_preservation_updI*)
apply (*force intro*!: *state_preservation_updI*)
apply (*force intro*!: *state_preservation_updI*)
apply (*fastforce intro*!: *state_preservation_updI state_preservation_fold_updI*)
subgoal
apply (*rule state_preservation_updI*)
apply *force*
apply (*force intro*!: *state_preservation_fold_updI*)
done
apply (*fastforce intro*!: *state_preservation_updI state_preservation_fold_updI*)
done

end

locale *Prod_TA_sem* =
  Prod_TA_Defs A **for** *A* :: ('*a*, '*s*, '*c*, '*t* :: *time*, '*x*, '*v* :: *linorder*) *nta*
begin

lemma *prod_invI*[*intro*]:
  ‹*u* ⊢ *prod_inv* (*L*, *s*)› **if** ‹∀ *p*<*n_ps*. *u* ⊢ *Simple_Network_Language.inv* (*N p*) (*L* ! *p*)›
  using *that* **unfolding** *prod_inv_def* **by** (*auto intro*!: *guard_concat*)

lemma *prod_invD*[*dest*]:
  ‹∀ *p*<*n_ps*. *u* ⊢ *Simple_Network_Language.inv* (*N p*) (*L* ! *p*)› **if** ‹*u* ⊢ *prod_inv* (*L*, *s*)› ‹*L* ∈ *states*›
  using *that* **unfolding** *prod_inv_def* **by** (*auto elim*: *concat_guard*)

lemma *delay_sound*:
  assumes *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →^*d* ⟨(*L'*, *s'*), *u'*⟩ *L* ∈ *states bounded bounds s*
        ∀ *N* ∈ *set* (*fst* (*snd A*)). *urgent N* = {}
      shows *A* ⊢ ⟨*L*, *s*, *u*⟩ →*Del* ⟨*L'*, *s'*, *u'*⟩
proof −
  from *assms*(*4*) **have** *l* ∉ *urgent* (*N p*) **if** *p* < *n_ps* **for** *p l*
    using *that* **unfolding** *N_def n_ps_def* **by** *auto*
  then show *?thesis*
    by (*subst A_split*) (*use assms* **in** ‹*cases*, *auto intro*!: *step_t simp*: *TAG_def*›)
qed

lemma *action_sound*:
  A ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩ **if** *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →*a* ⟨(*L'*, *s'*), *u'*⟩
  using *that*
proof *cases*
  case *prems*: (*1 g r*)
  note [*simp add*] = *map_N_n_ps_simp clock_val_append_iff clock_val_concat_iff*
  from ‹*prod_ta* ⊢ (*L*, *s*) ⟶^*g*,*a*,*r* (*L'*, *s'*)›[*THEN state_setI2*] **have** *L'* ∈ *states*
    using *state_set_states that* **by** *fast*
  from ‹*prod_ta* ⊢ (*L*, *s*) ⟶^*g*,*a*,*r* (*L'*, *s'*)› **show** *?thesis*
    unfolding *trans_of_prod trans_prod_def*
  proof *safe*
    assume ((*L*, *s*), *g*, *a*, *r*, *L'*, *s'*) ∈ *trans_int*
    then show *A* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩
      unfolding *trans_int_def*
      apply *clarsimp*
      using *prems* ‹*L'* ∈ _›
      by (*subst A_split*) (*intro step_int*; *simp add*: *TAG_def*; *elim prod_invD*; *assumption*)
  next
    assume ((*L*, *s*), *g*, *a*, *r*, *L'*, *s'*) ∈ *trans_bin*
    then show *A* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩
      unfolding *trans_bin_def*
      using *prems* ‹*L'* ∈ _›
      apply *clarsimp*
      apply (*subst A_split*, *standard*)
                apply (*assumption* | *simp*; *elim prod_invD*; *assumption*)+
      done
  next
    assume ((*L*, *s*), *g*, *a*, *r*, *L'*, *s'*) ∈ *trans_broad*
    then show *A* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩

```
    using prems ‹L' ∈ _›
    unfolding trans_broad_def inv_of_prod
    apply clarsimp
    apply (subst A_split)
    apply standard
    apply (simp; elim prod_invD; assumption)+
    apply fastforce+
    done
  qed
qed

lemma bounded_inv:
  ‹bounded bounds s'› if ‹A ⊢ ⟨L, s, u⟩ →_a ⟨L', s', u'⟩›
  using that unfolding bounds_def by cases (simp add: TAG_def)+

lemma states_inv:
  ‹L' ∈ states› if ‹A ⊢ ⟨L, s, u⟩ →_a ⟨L', s', u'⟩› ‹L ∈ states›
  using that unfolding bounds_def
proof cases
  case (step_t N d B broadcast)
  with ‹L ∈ states› show ?thesis
    by simp
next
  case prems[unfolded TAG_def]: (step_int l b g a f r l' N' p B broadcast)
  from ‹A = _› prems(3) have l' ∈ (⋃ (l, b, g, a, r, u, l') ∈ trans (N p). {l, l'})
    by force
  with ‹L ∈ states› show ?thesis
    unfolding ‹L' = _› by (intro state_preservation_updI)
next
  case prems[unfolded TAG_def]:
    (step_bin broadcast a l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s' B)
  from ‹A = _› prems(4, 5) have
    l1' ∈ (⋃ (l, b, g, a, r, u, l') ∈ trans (N p). {l, l'})
    l2' ∈ (⋃ (l, b, g, a, r, u, l') ∈ trans (N q). {l, l'})
    by force+
  with ‹L ∈ states› show ?thesis
    unfolding ‹L' = _› by (intro state_preservation_updI)
next
  case prems[unfolded TAG_def]: (step_broad a broadcast l b g f r l' N' p ps bs gs fs rs ls' s' B)
  from ‹A = _› prems(4, 5) have
    l' ∈ (⋃ (l, b, g, a, r, u, l') ∈ trans (N p). {l, l'})
    ∀ q ∈ set ps. ls' q ∈ (⋃ (l, b, g, a, r, u, l') ∈ trans (N q). {l, l'})
    by force+
  with ‹L ∈ states› show ?thesis
    unfolding ‹L' = _› by (intro state_preservation_updI state_preservation_fold_updI)
qed

end


locale Prod_TA =
  Prod_TA_sem A for A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder) nta +
  assumes broadcast_receivers_unguarded:
    ∀ p < n_ps. ∀ l b g a f r l'. (l, b, g, In a, f, r, l') ∈ trans (N p) ∧ a ∈ broadcast ⟶ g = []
```

begin

lemma *action_complete*:
prod_ta ⊢ ⟨(*L*, *s*), *u*⟩ →$_a$ ⟨(*L'*, *s'*), *u'*⟩
if *A* ⊢ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩ *a* ≠ *Del* *L* ∈ *states bounded bounds s*
using *that*(*1*) **proof** *cases*
case (*step_t N d B broadcast*)
then show *?thesis*
using *that*(*2*) **by** *auto*
next
case *prems*[*unfolded TAG_def*]: (*step_int l b g a' f r l' N' p B broadcast'*)
have [*simp*]:
B = *bounds broadcast'* = *broadcast length N'* = *n_ps*
unfolding *bounds_def broadcast_def n_ps_def* **unfolding** *prems*(*1*) **by** *simp*+
have [*simp*]:
N' ! *i* = *N i* **for** *i*
unfolding *N_def* **unfolding** *prems*(*1*) **by** *simp*
have *prod_ta* ⊢ (*L*, *s*) ⟶$^{g,Internal\ a',r}$ (*L'*, *s'*)
proof −
from *prems* ‹*L* ∈ *states*› ‹*bounded* _ *s*› **have** ((*L*, *s*),*g*,*Internal a'*,*r*,(*L'*, *s'*)) ∈ *trans_int*
unfolding *trans_int_def*
by *simp* (*rule exI conjI HOL.refl* | *assumption* | (*simp*; *fail*))+
then show *?thesis*
unfolding *prod_ta_def trans_of_def trans_prod_def* **by** *simp*
qed
moreover have *u* ⊢ *g*
by (*rule prems*)
moreover have *u'* ⊢ *inv_of prod_ta* (*L'*, *s'*)
using *prems*(*7*) **by** *auto*
moreover have *u'* = [*r*→*0*]*u*
by (*rule prems*)
ultimately show *?thesis*
unfolding ‹*a* = _› **..**
next
case *prems*[*unfolded TAG_def*]:
(*step_bin a' broadcast' l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s'' B*)
have [*simp*]:
B = *bounds broadcast'* = *broadcast length N'* = *n_ps*
unfolding *bounds_def broadcast_def n_ps_def* **unfolding** *prems*(*1*) **by** *simp*+
have [*simp*]:
N' ! *i* = *N i* **for** *i*
unfolding *N_def* **unfolding** *prems*(*1*) **by** *simp*
have *prod_ta* ⊢ (*L*, *s*) ⟶$^{g1\ @\ g2,Bin\ a',r1\ @\ r2}$ (*L'*, *s'*)
proof −
from *prems* ‹*L* ∈ *states*› ‹*bounded bounds s*› **have**
((*L*, *s*),*g1* @ *g2*,*Bin a'*,*r1* @ *r2*,(*L'*, *s'*)) ∈ *trans_bin*
unfolding *trans_bin_def*
using [[*simproc add*: *ex_reorder4*]]
by *simp* (*rule exI conjI HOL.refl* | *assumption* | *fast*)+
then show *?thesis*
unfolding *prod_ta_def trans_of_def trans_prod_def* **by** *simp*
qed
moreover have *u* ⊢ *g1* @ *g2*

```
    using ‹u ⊢ g1› ‹u ⊢ g2› by (rule guard_append)
  moreover have u' ⊢ inv_of prod_ta (L', s')
    using prems by auto
  moreover have u' = [r1@r2→0]u
    by (rule prems)
  ultimately show ?thesis
    unfolding ‹a = _› ..
next
  case prems[unfolded TAG_def]: (step_broad a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s''
B)
  have [simp]:
    B = bounds broadcast' = broadcast length N' = n_ps
    unfolding bounds_def broadcast_def n_ps_def unfolding prems(1) by simp+
  have [simp]:
    N' ! i = N i for i
    unfolding N_def unfolding prems(1) by simp
  let ?r = r @ concat (map rs ps) and ?g = g @ concat (map gs ps)
  have prod_ta ⊢ (L, s) ⟶^?g,Broad a',?r (L', s')
  proof −
    have ∗: ¬ u ⊢ g ⟷ False if
      p < n_ps (l, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
      a' ∈ broadcast
      for l b g a' f r l' p
    proof −
      from that broadcast_receivers_unguarded have ‹g = []›
        by blast
      then show ?thesis
        by auto
    qed
   from prems ‹L ∈ states› ‹bounded bounds s› have ((L, s),?g,Broad a',?r,(L', s')) ∈ trans_broad
      unfolding trans_broad_def
      by clarsimp
        (intro exI conjI HOL.refl; (rule HOL.refl | assumption | fastforce simp: ∗))
   then show ?thesis
      unfolding prod_ta_def trans_of_def trans_prod_def by simp
  qed
  moreover have u ⊢ ?g
    using prems by (auto intro!: guard_append guard_concat)
  moreover have u' ⊢ inv_of prod_ta (L', s')
    using prems by auto
  moreover have u' = [?r→0]u
    by (rule prems)
  ultimately show ?thesis
    unfolding ‹a = _› ..
qed

lemma delay_complete:
  assumes A ⊢ ⟨L, s, u⟩ →Del ⟨L', s', u'⟩
  obtains d where prod_ta ⊢ ⟨(L, s), u⟩ →^d ⟨(L', s'), u'⟩
  using assms
proof cases
  case prems: (step_t N' d B broadcast)
  have [simp]:
```

length $N' = n_ps$
unfolding *n_ps_def* **unfolding** *prems*(*1*) **by** *simp*+
have [*simp*]:
N' ! $i = N\ i$ **for** i
unfolding *N_def* **unfolding** *prems*(*1*) **by** *simp*
from *prems* **show** *?thesis*
by (*intro that*[*of d*]; *unfold* ‹$u' = _$› ‹$L' = L$› ‹$s' = s$› *TAG_def*)
(*rule step_t.intros*; (*unfold inv_of_prod*; *rule prod_invI*)*?*; *simp*)
qed

lemma *step_iff*:
($\exists$ *a*. $A \vdash \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$) $\longleftrightarrow$ *prod_ta* $\vdash \langle(L, s), u\rangle \rightarrow \langle(L', s'), u'\rangle$
if *bounded bounds s* $L \in$ *states* $\forall N \in$ *set* (*fst* (*snd A*)). *urgent* $N = \{\}$
using *that*(*1*,*2*)
apply *safe*
subgoal for *a*
by (*cases a*; *blast dest*: *action_complete elim*: *delay_complete*)
by (*auto intro*: *action_sound delay_sound*[*OF* _ _ _ *that*(*3*)])

end

end
theory *TA_Impl_Misc2*
imports
Munta_Base.TA_Impl_Misc
HOL−Library.Sublist
List−Index.List_Index
Automatic_Refinement.Misc
begin

lemma *mem_nth*:
$x \in$ *set xs* $\Longrightarrow \exists\ i <$ *length xs*. *xs* ! $i = x$
by (*metis index_less_size_conv nth_index*)

lemma *union_subsetI*:
$A \cup B \subseteq C \cup D$ **if** $A \subseteq C\ B \subseteq D$
using *that* **by** *blast*

lemma *map_eq_imageD*:
f ‘ *set xs* = *set ys* **if** *map f xs* = *ys*
using *that* **by** *auto*

lemma *if_contract*:
(*if a then x else if b then x else y*) = (*if a* $\vee$ *b then x else y*) **for** *a x b y*
by (*rule SMT.z3_rule*)

More intros named_theorems *more_intros*
named_theorems *more_elims*
lemmas [*more_intros*] =
image_eqI[*rotated*] *CollectI subsetI*

lemmas [*more_elims*] =
CollectE

Finiteness **lemma** *finite_prodI*:
finite {(*a*,*b*). *P a* ∧ *Q b*} **if** *finite* {*a*. *P a*} *finite* {*a*. *Q a*}
using *that* **by** *simp*

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*}
using *that* **by** *simp*

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*}
using *that* **by** *simp*

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*}
using *that* **by** *simp*

named_theorems *finite_intros*
named_theorems *more_finite_intros*

lemmas [*finite_intros*] =
finite_UnI finite_Union finite_imageI
finite_lists_length_eq finite_lists_length_le
finite_subset_distinct distinct_finite_set

lemmas [*more_finite_intros*] =
finite_prodI finite_prodI3 finite_prodI4 finite_prodI5

Lists **lemma** *fold_evD2*:
assumes
P y (*fold f xs acc*) ¬ *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 −
from *fold_evD'*[*OF assms*(*2*,*1*)] **obtain** *x ys zs* **where** ∗:
xs = *ys* @ *x* # *zs* ¬ *P y* (*fold f ys acc*) *P y* (*f x* (*fold f ys acc*))
by *auto*
moreover from *assms*(*4*−) **have** *Q* (*fold f ys acc*) **by** (*auto intro*: *fold_acc_preserv*)
moreover from ‹*xs* = _› **have** *x* ∈ *set xs*
by *auto*
ultimately show *?thesis* **using** *assms*(*3*,*6*) **by** *auto*
qed

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*
by (*induction xs*; *simp add*: *Let_def List.map_filter_simps*)

```
lemma distinct_map_filterI:
  distinct (List.map_filter f xs)
  if ∀ x ∈ set xs. ∀ y ∈ set xs. ∀ a. f x = Some a ∧ f y = Some a ⟶ x = y distinct xs
  using that by (induction xs) (auto simp: map_filter_simps set_map_filter split: option.split)

lemma filter_eqI:
  assumes
    subseq ys xs ∀ x ∈ set ys. P x
    ∀ zs. subseq zs xs ∧ length zs > length ys ⟶ (∃ x ∈ set zs. ¬ P x)
  shows filter P xs = ys
  using assms
proof (induction xs arbitrary: ys rule: list.induct)
  case Nil
  then show ?case
    by − (cases ys; simp)
next
  case (Cons x xs)
  show ?case
  proof (cases P x)
    case True
    show ?thesis
    proof (cases ys)
      case Nil
      have subseq [x] (x # xs)
        by auto
      with Cons.prems Nil ‹P x› show ?thesis
        by fastforce
    next
      case (Cons y ys')
      have x = y
      proof (rule ccontr)
        assume x ≠ y
        with ‹subseq ys (x # xs)› ‹ys = _› have subseq (x # ys) (x # xs)
          by simp
        with Cons.prems(2−) ‹P x› show False
          by fastforce
      qed
      have ∃ x∈set zs. ¬ P x if subseq zs xs and length ys' < length zs for zs
      proof −
        from ‹subseq zs xs› have subseq (x # zs) (x # xs)
          by simp
        with ‹length ys' < length zs› Cons.prems(3) ‹ys = _› have ∃ x∈set (x # zs). ¬ P x
          by (intro Cons.prems(3)[rule_format]; simp)
        with ‹P x› show ?thesis
          by auto
      qed
      with Cons.prems ‹P x› ‹ys = _› ‹x = y› show ?thesis
        by (auto intro!: Cons.IH)
    qed
  next
    case False
    with Cons.prems show ?thesis
      by (cases ys) (auto split: if_split_asm intro!: Cons.IH)
  qed
```

qed

lemma *filter_greatest_subseqD*:
  $\exists\ x \in set\ zs.\ \neg\ P\ x$ **if** *subseq zs xs length zs > length* (*filter P xs*)
  using *that* **by** (*metis filter_id_conv not_subseq_length subseq_filter*)

lemma *filter_eq_iff_greatest_subseq*:
  filter P xs = ys $\longleftrightarrow$
  subseq ys xs $\wedge$ $(\forall\, x \in set\ ys.\ P\ x)$ $\wedge$
  $(\forall\, zs.\ subseq\ zs\ xs \wedge length\ zs > length\ ys \longrightarrow (\exists\ x \in set\ zs.\ \neg\ P\ x))$
  using *filter_greatest_subseqD filter_eqI* **by** *auto*

lemma *subseq_subsetD*:
  set xs $\subseteq$ *set ys* **if** *subseq xs ys*
  using *that*
  by (*intro subsetI*) (*unfold subseq_singleton_left*[*symmetric*], *erule subseq_order.trans*)

lemma *subseq_distinct*:
  distinct xs **if** *distinct ys subseq xs ys*
  using *subseqs_distinctD that* **by** *simp*

lemma *finite_subseqs*:
  finite {*xs. subseq xs ys*} (**is** *finite ?S*)
proof −
  have *?S* $\subseteq$ {*xs. set xs* $\subseteq$ *set ys* $\wedge$ *length xs* $\leq$ *length ys*}
    using *not_subseq_length* **by** (*force dest*: *subseq_subsetD*)
  also have *finite* ...
    by (*auto intro*: *finite_lists_length_le*)
  finally (*finite_subset*) **show** *?thesis* **.**
qed

lemma *filter_distinct_eqI*:
  assumes
    subseq ys xs $\forall\, x \in set\ ys.\ P\ x\ \forall\, x \in set\ xs.\ x \notin set\ ys \longrightarrow \neg\ P\ x\ distinct\ xs$
  shows *filter P xs = ys*
proof (*intro filter_eqI*, *safe*)
  fix *zs* **assume** *prems*: *subseq zs xs length ys < length zs*
  obtain *x* **where** *x* $\in$ *set zs x* $\notin$ *set ys*
  proof (*atomize_elim*, *rule ccontr*)
    assume $\nexists x.\ x \in set\ zs \wedge x \notin set\ ys$
    then have *set zs* $\subseteq$ *set ys*
      by *auto*
    moreover from *prems assms* **have** *distinct zs distinct ys*
      by (*blast intro*: *subseq_distinct*)+
    ultimately show *False*
      using ‹*length ys < length zs*›
      by (*auto dest*: *card_mono*[*rotated*] *simp*: *distinct_card*[*symmetric*])
  qed
  with *prems assms* **show** $\exists x \in set\ zs.\ \neg\ P\ x$
    by (*auto 4 3 dest*: *subseq_subsetD*)
qed (*use assms* **in** *blast*)+

lemma *subseq_sorted_wrt*:
  sorted_wrt R xs **if** *sorted_wrt R ys subseq xs ys*

using *that*
by (*induction xs arbitrary*: *ys*)
(*auto 0 4 dest*: *subseq_subsetD list_emb_ConsD subseq_Cons′ simp*: *sorted_wrt_append*)

lemma *subseq_sorted*:
sorted xs **if** *sorted ys subseq xs ys*
using *that* **by** (*rule subseq_sorted_wrt*)

lemma *sorted_distinct_subset_subseqI*:
assumes *sorted xs distinct xs sorted ys set xs* ⊆ *set ys*
shows *subseq xs ys*
using *assms*
proof (*induction ys arbitrary*: *xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons y ys xs*)
from *Cons.prems* **show** *?case*
by (*cases xs*; *simp*) (*safe*; *rule Cons.IH*; *auto 4 4*)
qed

lemma *sorted_distinct_subseq_iff*:
assumes *sorted ys distinct ys*
shows *subseq xs ys* ⟷ (*sorted xs* ∧ *distinct xs* ∧ *set xs* ⊆ *set ys*)
using *assms*
by *safe*
(*erule*
subseq_subsetD[*THEN subsetD*] *sorted_distinct_subset_subseqI subseq_distinct subseq_sorted*;
assumption
)+

lemma *subseq_mapE*:
assumes *subseq xs* (*map f ys*)
obtains *xs′* **where** *subseq xs′ ys map f xs′* = *xs*
using *assms*
by (*induct x1* ≡ *xs x2* ≡ *map f ys arbitrary*: *xs ys rule*: *list_emb.induct*)
(*auto*, *metis map_consI*(*1*) *subseq_Cons2*)

lemma *list_all2_map_fst_aux*:
assumes *list_all2* (λ*x y*. *x* ∈ *Pair y* ‘ (*zs y*)) *xs ys*
shows *list_all2* (=) (*map fst xs*) *ys*
using *assms* **by** (*smt fstI imageE list.rel_mono_strong list_all2_map1*)

lemma *list_all2_fst_aux*:
map fst xs = *ys* **if** *list_all2* (λ*x y*. *fst x* = *y*) *xs ys*
using *that* **by** (*induction*) *auto*

Stronger version of ⟦*inj ?f*; *map_of ?t ?k* = *Some ?x*⟧ ⟹ *map_of* (*map* (λ(*k*, *y*). (*?f k*, *y*)) *?t*) (*?f ?k*) = *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* (λ(*k*, *y*). (*f k*, *g y*)) *t*) (*f k*) = *Some* (*g x*)

using *assms*
apply (*induction t*)
apply (*solves simp*)
apply (*clarsimp split*: *if_split_asm*)
apply (*metis DiffI imageI img_fst map_of_SomeD singletonD*)
done

theorem *map_of_mapk_SomeI*:
assumes *inj f*
and *map_of t k* = *Some x*
shows *map_of* (*map* (λ(*k*, *y*). (*f k*, *g y*)) *t*) (*f k*) = *Some* (*g x*)
using *assms* **by** − (*rule map_of_mapk_SomeI'*, *erule inj_on_subset*, *auto*)

lemma *list_all2_map_eq_iff*:
list_all2 (λ*x y*. *f x* = *g y*) *xs ys* ⟷ *map f xs* = *map g ys*
proof
assume *list_all2* (λ*x y*. *f x* = *g y*) *xs ys*
then show *map f xs* = *map g ys*
by *induction auto*
next
assume *map f xs* = *map g ys*
then have *length xs* = *length ys*
by (*rule map_eq_imp_length_eq*)
then show *list_all2* (λ*x y*. *f x* = *g y*) *xs ys*
using ‹*map f xs* = _› **by** (*induction rule*: *list_induct2*; *simp*)
qed

end
theory *TA_More2*
imports *Munta_Base.TA_More*
begin

lemma *collect_clock_pairs_concat*:
collect_clock_pairs (*concat xxs*) = (⋃ *x* ∈ *set xxs*. *collect_clock_pairs x*)
unfolding *collect_clock_pairs_def* **by** *auto*

end
theory *TA_Equivalences*
imports
Timed_Automata.Timed_Automata
Munta_Base.Normalized_Zone_Semantics_Impl_Refine
begin

locale *TA_Equiv* =
fixes *A B* :: ('*a*, '*c*, '*t* :: *time*, '*s*) *ta*
fixes *S* :: '*s set*
assumes *state_set_trans_of*: *state_set* (*trans_of A*) ⊆ *S*
assumes *invs*: ∀ *l* ∈ *S*. *inv_of A l* = *inv_of B l*
assumes *trans_eq*: *trans_of A* = *trans_of B*
begin

lemma *delay1*:
A ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l'*,*u'*⟩ **if** *l* ∈ *S B* ⊢ ⟨*l*, *u*⟩ →^{d} ⟨*l'*,*u'*⟩
using *that invs* **by** (*auto elim*!: *step_t.cases*)

lemma *action1*:
  $A \vdash \langle l, u\rangle \rightarrow_a \langle l',u'\rangle$ **if** $B \vdash \langle l, u\rangle \rightarrow_a \langle l',u'\rangle$
proof −
  from *that* **have** $l \in S\ l' \in S$
    using *state_set_trans_of* **by** (*auto elim*!: *step_a.cases simp*: *trans_eq state_set_def*)
  with *that invs* **show** *?thesis*
    by (*auto elim*!: *step_a.cases simp*: *trans_eq intro*: *step_a.intros*)
qed

lemma *delay2*:
  $B \vdash \langle l, u\rangle \rightarrow^d \langle l',u'\rangle$ **if** $l \in S\ A \vdash \langle l, u\rangle \rightarrow^d \langle l',u'\rangle$
  using *that invs* **by** (*auto elim*!: *step_t.cases*)

lemma *action2*:
  $B \vdash \langle l, u\rangle \rightarrow_a \langle l',u'\rangle$ **if** $A \vdash \langle l, u\rangle \rightarrow_a \langle l',u'\rangle$
proof −
  from *that* **have** $l \in S\ l' \in S$
   using *state_set_trans_of* **by** (*auto elim*!: *step_a.cases simp*: *trans_eq*[*symmetric*] *state_set_def*)
  with *that invs* **show** *?thesis*
    by (*auto elim*!: *step_a.cases simp*: *trans_eq* [*symmetric*] *intro*: *step_a.intros*)
qed

lemma *step1*:
  $A \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$ **if** $l \in S\ B \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$
  using *that*(*2*,*1*) **by** *cases* (*auto dest*: *action1 delay1*)

lemma *step2*:
  $B \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$ **if** $l \in S\ A \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$
  using *that*(*2*,*1*) **by** *cases* (*auto dest*: *action2 delay2*)

lemma *S_inv*:
  $l' \in S$ **if** $A \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle\ l \in S$
  using *that state_set_trans_of*
  by (*auto elim*!: *step.cases step_a.cases step_t.cases simp*: *state_set_def*)

interpretation *interp*: *Bisimulation_Invariant*
  $\lambda\ (l, u)\ (l', u').\ A \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$
  $\lambda\ (l, u)\ (l', u').\ B \vdash \langle l, u\rangle \rightarrow \langle l',u'\rangle$
  $\lambda\ lu\ lu'.\ lu' = lu\ \lambda\ (l, u).\ l \in S\ \lambda\ (l, u).\ l \in S$
  by *standard* (*auto dest*: *step1 step2 intro*: *S_inv*)

end

end
theory *Simple_Network_Language_Impl*
  imports
    Simple_Network_Language
    Munta_Base.Normalized_Zone_Semantics_Impl_Refine
    HOL−Library.IArray HOL−Library.AList
    Munta_Base.More_Methods
    Munta_Base.Bijective_Embedding

TA_Impl_Misc2
TA_More2
TA_Equivalences
HOL−Eisbach.Eisbach_Tools
Munta_Base.TA_Syntax_Bundles
begin

unbundle *no_library_syntax*

Default maps **definition** *default_map_of* :: *'b* ⇒ (*'a* × *'b*) *list* ⇒ *'a* ⇒ *'b* **where**
default_map_of a xs ≡ *FinFun.map_default a* (*map_of xs*)

lemma *default_map_of_alt_def*:
default_map_of a xs x = (*if x* ∈ *dom* (*map_of xs*) *then the* (*map_of xs x*) *else a*)
unfolding *default_map_of_def map_default_def* **by** (*auto split*: *option.split_asm*)

lemma *range_default_map_of*:
range (*default_map_of x xs*) ⊆ *snd* ‘ *set xs* ∪ {*x*}
unfolding *default_map_of_def map_default_def*
by (*auto split*: *option.split_asm*) (*meson img_snd map_of_SomeD*)

lemma *finite_range_default_map_of*:
finite (*range* (*default_map_of x m*))
proof −
have *range* (*default_map_of x m*) ⊆ *the* ‘ *range* (*map_of m*) ∪ {*x*}
unfolding *default_map_of_def FinFun.map_default_def*
by (*auto split*: *option.splits*) (*metis image_eqI option.sel rangeI*)
also have *finite* . . .
by (*blast intro*: *finite_range_map_of*)
finally show *?thesis* **.**
qed

lemma *map_index'_inj*:
L = *L'*
if *length L* = *length L' map_index' n f L* = *map_index' n g L' set L* ⊆ *S set L'* ⊆ *T*
∀ *i* < *length L* + *n*. ∀ *x* ∈ *S*. ∀ *y* ∈ *T*. *f i x* = *g i y* ⟶ *x* = *y*
using *that*
proof (*induction length L arbitrary*: *L L' n*)
case *0*
then show *?case*
by *simp*
next
case (*Suc x*)
from *Suc.prems* **obtain** *a b L1 L1'* **where** ∗:
length L1 = *x length L1'* = *x L* = *a* # *L1 L'* = *b* # *L1'*
by (*smt Suc.hyps*(*2*) *length_Suc_conv*)
show *?case*
unfolding ‹*L* = _› ‹*L'* = _›
apply (*clarsimp*, *rule conjI*)
subgoal
by (*smt* ∗(*3*,*4*) *Suc.hyps*(*2*) *Suc.prems Suc_less_eq add_Suc_shift less_add_Suc2 list.set_intros*(*1*)
list_tail_coinc map_index'.simps(*2*) *set_mp*)
subgoal

```
    apply (rule Suc.hyps)
    using Suc.prems ∗ by auto
  done
qed

lemma map_index_inj:
  L = L′
  if map_index f L = map_index g L′ set L ⊆ S set L′ ⊆ T
    ∀ i < length L. ∀ x ∈ S. ∀ y ∈ T. f i x = g i y ⟶ x = y
  using that by − (rule map_index′_inj, auto dest: map_index_eq_imp_length_eq)

lemma map_index_inj1:
  L = L′
  if map_index f L = map_index g L′
    ∀ i < length L. f i (L ! i) = g i (L′ ! i) ⟶ L ! i = L′ ! i
proof (intros add: nth_equalityI)
  from that(1) show ‹length L = length L′›
    by (rule map_index_eq_imp_length_eq)
  fix i :: ‹nat›
  assume ‹i < length L›
  with that have map_index f L ! i = map_index g L′ ! i
    by auto
  with ‹i < length L› ‹length L = length L′› have f i (L ! i) = g i (L′ ! i)
    by simp
  with ‹i < length L› that(2) show ‹L ! i = L′ ! i›
    by simp
qed

lemma map_index_update:
  map_index f (xs[i := a]) = (map_index f xs)[i := f i a]
  by (rule nth_equalityI) (auto simp: nth_list_update′)

lemma map_trans_broad_aux1:
  map_index map_loc (fold (λp L. L[p := ls′ p]) ps L) =
  fold (λp L. L[p := map_loc p (ls′ p)]) ps (map_index map_loc L)
  by (induction ps arbitrary: L) (auto simp: map_index_update)

lemma InD2:
  fixes map_action
  assumes inj map_action In (map_action a) = map_act map_action a′
  shows a′ = In a
  using assms(2) by (cases a′)  (auto simp: injD[OF assms(1)])

lemma OutD2:
  fixes map_action
  assumes inj map_action Out (map_action a) = map_act map_action a′
  shows a′ = Out a
  using assms(2) by (cases a′)  (auto simp: injD[OF assms(1)])

lemma (in Prod_TA_Defs) trans_broad_alt_def:
  trans_broad =
    {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L′, s′′)) |
    L s L′ s′ s′′ a p l b g f r l′ bs gs fs rs ls′ ps.
      a ∈ broadcast  ∧
```

```
    (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
    (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
    (l ∈ committed (N p) ∨ (∃ p ∈ set ps. L ! p ∈ committed (N p))
    ∨ (∀ p < n_ps. L ! p ∉ committed (N p))) ∧
    (∀ q < n_ps. q ∉ set ps ∧ p ≠ q ⟶
      ¬ (∃ b g f r l'. (L!q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
    L!p = l ∧
    p < length L ∧ set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
    check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
    L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'] ∧
    is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
    L ∈ states ∧ bounded bounds s ∧ bounded bounds s'' ∧
    (∀ p. p∉set ps ⟶ bs p = bexp.true) ∧ (∀ p. p∉set ps ⟶ gs p = []) ∧
    (∀ p. p∉set ps ⟶ fs p = []) ∧ (∀ p. p∉set ps ⟶ rs p = [])
  }
 unfolding trans_broad_def
proof ((intro Collect_eqI iffI; elims add: more_elims), goal_cases)
 case prems: (1 x L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps)
 let ?f = λgs p. if p ∈ set ps then gs p else []
 let ?bs = λp. if p ∈ set ps then bs p else bexp.true
 let ?gs = ?f gs let ?fs = ?f fs let ?rs = ?f rs
 have [simp]: map gs ps = map ?gs ps map rs ps = map ?rs ps map fs ps = map ?fs ps
   by (simp cong: map_cong)+
 with prems show ?case
   by (inst_existentials L s L' s' s'' a p l b g f r l' ?bs ?gs ?fs ?rs ls' ps; (assumption | simp))
next
 case (2 x L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps)
 then show ?case
   by blast
qed


definition
 conv_automaton ≡ λ(committed, urgent, trans, inv).
   (committed,
    urgent,
    map (λ(l, b, g, a, f, r, l'). (l, b, conv_cc g, a, f, r, l')) trans,
    map (λ(s, cc). (s, conv_cc cc)) inv)

definition
 automaton_of ≡
   λ(committed, urgent, trans, inv). (set committed, set urgent, set trans, default_map_of [] inv)

locale Simple_Network_Impl_Defs =
 fixes automata ::
   ('s list × 's list × ('a act, 's, 'c, 't, 'x, int) transition list
     × ('s × ('c, 't) cconstraint) list) list
   and broadcast :: 'a list
   and bounds' :: ('x × (int × int)) list
begin

definition — Number of state variables
 n_vs = length bounds'
```

definition
B x ≡ *if x* ∈ *dom* (*map_of bounds′*) *then the* (*map_of bounds′ x*) *else* (*0*, *0*)

sublocale *Prod_TA_Defs*
(*set broadcast*, *map automaton_of automata*, *map_of bounds′*) **.**

lemma *L_len*[*intro*, *dest*]:
length L = *n_ps* **if** *L* ∈ *states*
using *that* **unfolding** *states_def* **by** *simp*

lemma *N_eq*:
‹*N i* = *automaton_of* (*automata* ! *i*)› **if** ‹*i* < *n_ps*›
using *that* **unfolding** *N_def n_ps_def fst_conv snd_conv* **by** (*intro nth_map*; *simp*)

end

locale *Simple_Network_Impl* =
fixes *automata* ::
(′*s list* × ′*s list* × (′*a act*, ′*s*, ′*c*, *int*, ′*x*, *int*) *transition list*
× (′*s* × (′*c*, *int*) *cconstraint*) *list*) *list*
and *broadcast* :: ′*a list*
and *bounds′* :: (′*x* × (*int* × *int*)) *list*
begin

sublocale *Simple_Network_Impl_Defs automata broadcast bounds′* **.**

end

Mapping through the product construction **lemma** *f_the_inv_f*:
f (*the_inv f x*) = *x* **if** *inj f x* ∈ *range f*
using *that* **by** (*auto simp*: *the_inv_f_f*)

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′*) **.**

lemma (**in** −) *conv_ac_inj*:
ac = *ac′* **if** *conv_ac ac* = *conv_ac ac′*
using *that* **by** (*cases ac*; *cases ac′*; *auto*)

lemma (**in** −) *conv_cc_inj*:
cc = *cc′* **if** *conv_cc cc* = *conv_cc cc′*
using *that* **by** (*subst* (*asm*) *inj_map_eq_map*) (*auto simp add*: *conv_ac_inj intro*: *injI*)

context
begin

lemma *conv_alt_def*:
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'*)
unfolding *conv_def* **by** *simp*

private 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*)))
apply (*rule ext*)
apply *clarsimp*
unfolding *Simple_Network_Language.conv_A_def automaton_of_def*
apply (*clarsimp split*: *prod.split*)
unfolding *default_map_of_def*
unfolding *FinFun.map_default_def*
apply (*rule ext*)
subgoal for *xs a*
by (*induction xs*) *auto*
done

lemma *conv_n_ps_eq*:
conv.n_ps = *n_ps*
by (*simp add*: *Prod_TA_Defs.n_ps_def*)

lemma *conv_N_eq*:
conv.N i = *Simple_Network_Language.conv_A* (*N i*) **if** *i* < *n_ps*
using *that* **unfolding** *n_ps_def Prod_TA_Defs.N_def fst_conv snd_conv* **by** (*subst nth_map* | *simp*)+

private lemma *5*:
inv (*conv.N i*) = *conv_cc o* (*inv* (*N i*)) **if** *i* < *n_ps*
unfolding *conv_N_eq*[*OF that*] **unfolding** *Simple_Network_Language.conv_A_def*
by (*simp split*: *prod.split add*: *inv_def*)

lemma *trans_conv_N_eq*:
trans (*conv.N i*) = *Simple_Network_Language.conv_t* ‘ (*trans* (*N i*)) **if** *i* < *n_ps*
unfolding *conv_N_eq*[*OF that*] **unfolding** *Simple_Network_Language.conv_A_def*
by (*simp split*: *prod.split add*: *trans_def*)

private lemma *71*:
(*l*, *b*, *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*
using *that* **by** (*force simp add*: *trans_conv_N_eq Simple_Network_Language.conv_t_def*)

private 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*
by (*auto simp*: *trans_conv_N_eq*[*OF that*] *Simple_Network_Language.conv_t_def*
dest: *conv_cc_inj intro*: *image_eqI*[*rotated*])

private lemma *73*:
 $\exists g'.\ g = conv_cc\ g' \wedge (l,\ b,\ g',\ a,\ r,\ u,\ l') \in Simple_Network_Language.trans\ (N\ i)$
 if $(l,\ b,\ g,\ a,\ r,\ u,\ l') \in Simple_Network_Language.trans\ (conv.N\ i)\ i < n_ps$
 using *that* **by** (*force simp*: *trans_conv_N_eq Simple_Network_Language.conv_t_def*)

lemma *conv_bounds*[*simp*]:
 conv.bounds = *bounds*
 unfolding *conv.bounds_def bounds_def* **by** *simp*

lemma *conv_states*[*simp*]:
 conv.states = *states*
 unfolding *conv.states_def states_def conv_n_ps_eq*
 by (*auto simp add*: *trans_conv_N_eq Simple_Network_Language.conv_t_def*) (*fastforce*, *force*)

private lemma *9*:
 committed (*conv.N p*) = *committed* (*N p*) **if** ‹*p* < *n_ps*›
 unfolding *conv_N_eq*[*OF that*] **unfolding** *Simple_Network_Language.conv_A_def*
 by (*simp split*: *prod.split add*: *committed_def*)

private lemma *10*:
 conv.broadcast = *set broadcast*
 unfolding *conv.broadcast_def* **by** *simp*

lemma *conv_trans_int*:
 conv.trans_int = $(\lambda(l,\ g,\ a,\ r,\ l').\ (l,\ map\ conv_ac\ g,\ a,\ r,\ l'))$ ‘ *trans_int*
 unfolding *conv.trans_int_def trans_int_def*
 supply [*simp*] = *L_len*
 apply *standard*
 subgoal
  by (*clarsimp simp*: *Simple_Network_Language.conv_t_def conv_n_ps_eq trans_conv_N_eq 9*)
    (*intros add*: *more_intros*, *solve_triv*+)
 subgoal
  by (*rule subsetI*,
      clarsimp simp: *Simple_Network_Language.conv_t_def conv_n_ps_eq trans_conv_N_eq 9*[*symmetric*])
    ((*drule* (*1*) *71*)+, *intros add*: *more_intros*, *solve_triv*+)
 done

lemma *conv_trans_bin*:
 conv.trans_bin = $(\lambda(l,\ g,\ a,\ r,\ l').\ (l,\ map\ conv_ac\ g,\ a,\ r,\ l'))$ ‘ *trans_bin*
 unfolding *conv.trans_bin_def trans_bin_def 10 broadcast_def*
 supply [*simp*] = *L_len*
 apply *standard*
 subgoal
  by (*clarsimp simp add*: *Simple_Network_Language.conv_t_def conv_n_ps_eq trans_conv_N_eq 9*)
    (*intros add*: *more_intros*, *solve_triv*+)
 subgoal
  by (*rule subsetI*,
      clarsimp simp: *Simple_Network_Language.conv_t_def conv_n_ps_eq trans_conv_N_eq 9*[*symmetric*])
    ((*drule* (*1*) *71*)+, *intros add*: *more_intros*, *solve_triv*+)
 done

lemma *n_ps_rangeD*:
p < *n_ps* **if** *p* ∈ *set ps set ps* ⊆ {*0*..<*n_ps*}
using *that* **by** *auto*

lemma *maximalD*:
(∀ *g f r l'*.
(*L* ! *q*, *b*, *g*, *In a'*, *f*, *r*, *l'*)
∉ (λ(*l*, *b*, *g*, *a*, *f*, *r*, *l'*).
(*l*, *b*, *map conv_ac g*, *a*, *f*, *r*, *l'*)) ‘ *trans* (*N q*)) ∨ ¬ *check_bexp s b True* **if**
∀ *q*<*n_ps*. *q* ∉ *set ps* ∧ *p* ≠ *q* ⟶ (∀ *b*. (∀ *g f r l'*.
(*L* ! *q*, *b*, *g*, *In a'*, *f*, *r*, *l'*) ∉ *trans* (*N q*)) ∨ ¬ *check_bexp s b True*)
q<*n_ps q* ∉ *set ps p* ≠ *q*
for *b ps p q L a' s* **using** *that* **by** *fastforce*

lemma *conv_trans_broad*:
conv.trans_broad = (λ(*l*, *g*, *a*, *r*, *l'*). (*l*, *map conv_ac g*, *a*, *r*, *l'*)) ‘ *trans_broad*
unfolding *conv.trans_broad_alt_def trans_broad_alt_def*
apply *standard*
supply [*simp*] = *L_len*
subgoal
proof −
have ∗∗: ∃ *gs'*. *gs* = *conv_cc o gs'* ∧
(∀ *p*∈*set ps*.(*L* ! *p*, *bs p*, *gs' p*, *In a*, *fs p*, *rs p*, *ls' p*) ∈ *trans* (*N p*))
if *assms*:
∀ *p*∈*set ps*. (*L* ! *p*, *bs p*, *gs p*, *In a*, *fs p*, *rs p*, *ls' p*) ∈ *trans* (*conv.N p*)
set ps ⊆ {*0*..<*n_ps*} ∀ *p*. *p* ∉ *set ps* ⟶ *gs p* = []
for *L ps bs gs a fs rs ls'*
proof −
have ∗: *gs p* = *conv_cc* (*Hilbert_Choice.inv conv_cc* (*gs p*)) **if** *p* ∈ *set ps* **for** *p*
using *that assms* **by** (*auto 4 3 simp*: *f_inv_into_f dest*!: *73*)
show *?thesis*
apply (*inst_existentials Hilbert_Choice.inv conv_cc o gs*)
subgoal
apply (*rule ext*)
subgoal for *p*
apply (*cases p* ∈ *set ps*)
subgoal
by (*simp*, *erule* ∗)
subgoal
using *that*(*3*) **by** (*auto intro*: *injI inv_f_eq conv_ac_inj*)
done
done
subgoal
using *that* ∗ **by** (*force dest*!: *conv_cc_inj 73*)
done
qed
have ∗: *set ps* ⊆ {*0*..<*n_ps*} ⟷ (∀ *p* ∈ *set ps*. *p* < *n_ps*) **for** *ps*
by *auto*
have *maximalI*:
∀ *q*<*n_ps*. *q* ∉ *set ps* ∧ *p* ≠ *q* ⟶ (∀ *b*. (∀ *g f r l'*.
(*L* ! *q*, *b*, *g*, *In a'*, *f*, *r*, *l'*) ∉ *trans* (*N q*)) ∨ ¬ *check_bexp s b True*) **if**
∀ *q*<*n_ps*. *q* ∉ *set ps* ∧ *p* ≠ *q* ⟶ (∀ *b*. (∀ *g f r l'*.
(*L* ! *q*, *b*, *g*, *In a'*, *f*, *r*, *l'*) ∉ *trans* (*conv.N q*)) ∨ ¬ *check_bexp s b True*)

```
    for ps p L a' s
      using that by (blast dest!: 71)
    show ?thesis
      apply (rule subsetI)
      apply (clarsimp simp add: conv_n_ps_eq 9 10 broadcast_def split: prod.split_asm)
      apply (drule (2) **)
      apply (drule (1) 73)+
      apply elims
      apply (intro image_eqI[rotated] CollectI exI conjI)
                    apply solve_triv+
      subgoal premises prems — Commited
        using prems(2) ‹set _ ⊆ {0..<n_ps}› by (auto dest: n_ps_rangeD simp: 9)
                  apply (erule maximalI; fail) — Maximal set
      by solve_triv+ (simp split: prod.split add: map_concat)
  qed
  subgoal
    apply (rule subsetI)
    apply (clarsimp simp:
        Simple_Network_Language.conv_t_def
        conv_n_ps_eq trans_conv_N_eq 9[symmetric] 10 broadcast_def map_concat)
    apply (drule (1) 71)
    apply (intros add: more_intros)
                  apply solve_triv+
                apply (subst comp_def; rule 71; fast elim: n_ps_rangeD; fail)
    subgoal premises prems for L s s' s'' aj p g f r l' gs fs rs ls' ps
      using prems(3,6) 9 by fastforce
              apply (erule maximalD)
                apply (solve_triv | blast)+
    done
  done

lemma conv_prod_ta:
  conv.prod_ta = Normalized_Zone_Semantics_Impl.conv_A prod_ta
  unfolding conv.prod_ta_def prod_ta_def
  unfolding conv.trans_prod_def trans_prod_def
  unfolding conv.prod_inv_def prod_inv_def
  unfolding conv.N_def N_def conv_n_ps_eq
  unfolding conv_A_def
  apply simp
  apply (rule conjI)
  subgoal
    unfolding image_Un
    by ((fo_rule arg_cong2)+; rule conv_trans_int conv_trans_bin conv_trans_broad)
  subgoal — Invariant
    unfolding conv.N_def N_def
    by (rule ext) (auto simp: 5 map_concat intro!: map_cong arg_cong[where f = concat])
  done

end

Fundamentals  definition clkp_set' ≡
   (⋃ A ∈ set automata. ⋃ g ∈ set (snd (snd (snd A))). collect_clock_pairs (snd g))
 ∪ (⋃ A ∈ set automata. ⋃ (l, b, g, _) ∈ set (fst (snd (snd A))). collect_clock_pairs g)
```

```
definition clk_set' where
  ‹clk_set' =
  fst ‘ clkp_set' ∪
  (⋃ A ∈ set automata. ⋃ (_, _, _, _, _, r, _) ∈ set (fst (snd (snd A))). set r)›

lemma (in −) default_map_of_in_listD:
  x ∈ ⋃ (set ‘ snd ‘ set invs) if x ∈ set (default_map_of [] invs l)
proof −
  have [] ≠ default_map_of [] invs l
    using that by force
  then have default_map_of [] invs l ∈ snd ‘ set invs
    by (metis (no_types) UNIV_I Un_insert_right range_default_map_of[of [] invs]
        image_eqI insertE subsetCE sup_bot.right_neutral)
  with that show ?thesis
    by blast
qed

lemma collect_clock_pairs_invsI:
  (a, b) ∈ ⋃ ((collect_clock_pairs o snd) ‘ set invs)
  if (a, b) ∈ collect_clock_pairs (default_map_of [] invs l)
  using that unfolding collect_clock_pairs_def by (auto dest!: default_map_of_in_listD)

lemma mem_trans_N_iff:
  t ∈ Simple_Network_Language.trans (N i) ⟷ t ∈ set (fst (snd (snd (automata ! i))))
  if i < n_ps
  unfolding N_eq[OF that] by (auto split: prod.splits simp: automaton_of_def trans_def)

lemma length_automata_eq_n_ps:
  length automata = n_ps
  unfolding n_ps_def by simp

lemma clkp_set'_subs:
  Timed_Automata.clkp_set prod_ta ⊆ clkp_set'
  unfolding Timed_Automata.clkp_set_def clkp_set'_def
proof (rule union_subsetI, goal_cases)
  case 1
  have ∗: False
    if i < n_ps L ∈ states
      (a, b) ∈ collect_clock_pairs (Simple_Network_Language.inv (N i) (L ! i))
      ∀ x∈set automata. ∀ x∈set (snd (snd (snd x))). (a, b) ∉ collect_clock_pairs (snd x)
    for a :: 'c and b :: int and L :: 's list and i :: nat
    using that
    apply (subst (asm) N_eq)
    apply assumption
    apply (inst_existentials automata ! i)
    unfolding automaton_of_def
    by (force dest!: nth_mem collect_clock_pairs_invsI
        split: prod.split_asm simp: inv_def Prod_TA_Defs.n_ps_def)
  from 1 show ?case
    unfolding Timed_Automata.collect_clki_def inv_of_prod prod_inv_def
    by (auto simp: collect_clock_pairs_concat intro: ∗)
next
  case 2
```

```
  then show ?case
    apply simp
    unfolding trans_prod_def Timed_Automata.collect_clkt_def
    apply safe
    subgoal
     unfolding trans_int_def by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)
    subgoal
      unfolding trans_bin_def
      by (fastforce
          simp: length_automata_eq_n_ps mem_trans_N_iff
          dest!: collect_clock_pairs_append_cases)
    subgoal
      unfolding trans_broad_def
      apply (clarsimp simp: length_automata_eq_n_ps mem_trans_N_iff)
      apply (drule collect_clock_pairs_append_cases)
      unfolding collect_clock_pairs_concat
      apply (clarsimp; safe)
      by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)+ — slow: 20s
    done
qed

lemma collect_clkvt_subs:
  collect_clkvt (trans_of prod_ta) ⊆
    (⋃ A ∈ set automata. ⋃ (_, _, _, _, _, r, _) ∈ set (fst (snd (snd A))). set r)
  apply simp
  unfolding collect_clkvt_def
  apply safe
  unfolding trans_prod_def
  subgoal
    apply simp
    unfolding trans_prod_def Timed_Automata.collect_clkt_def
    apply safe
    subgoal
      unfolding trans_int_def
      by (fastforce
          simp: length_automata_eq_n_ps mem_trans_N_iff
          dest!: collect_clock_pairs_append_cases)
    subgoal
      unfolding trans_bin_def
      by (fastforce
          simp: length_automata_eq_n_ps mem_trans_N_iff
          dest!: collect_clock_pairs_append_cases)
    subgoal
      unfolding trans_broad_def
      apply (clarsimp simp: length_automata_eq_n_ps mem_trans_N_iff)
      unfolding collect_clock_pairs_concat
      apply safe
      by (fastforce simp: length_automata_eq_n_ps mem_trans_N_iff)+ — slow: 20s
    done
  done

lemma clk_set'_subs: clk_set prod_ta ⊆ clk_set'
  using collect_clkvt_subs clkp_set'_subs unfolding clk_set'_def by auto
```

end

lemma (**in** *Prod_TA_Defs*) *finite_range_invI*:
finite (*range prod_inv*) **if** *assms*: $\forall\ i < n_ps.$ *finite* (*range* (*inv* (*N i*)))
proof −
let *?N* = $\bigcup$ (*range* ‘ *inv* ‘ *N* ‘ {*0..<n_ps*})
let *?X* = {*I*. *set I* ⊆ *?N* ∧ *length I* ≤ *n_ps*}
have *finite ?N*
using *assms* **by** *auto*
then have *finite ?X*
by (*rule finite_lists_length_le*)
moreover have *range prod_inv* ⊆ *concat* ‘ *?X* ∪ {[]}
proof
fix *x* **assume** *x* ∈ *range prod_inv*
then consider *L* **where** *x* = *concat* (*map* (λ*p*. (*inv* (*N p*)) (*L* ! *p*)) [*0..<n_ps*]) | *x* = []
unfolding *prod_inv_def* **by** (*auto split*: *if_split_asm*)
then show *x* ∈ *concat* ‘ *?X* ∪ {[]}
by (*cases*; *force*)
qed
ultimately show *?thesis* **by** − (*drule finite_subset*; *auto*)
qed

definition (**in** *Prod_TA_Defs*)
loc_set =
(⋃ {*fst* ‘ *trans* (*N p*) | *p*. *p* < *n_ps*} ∪
⋃ {(*snd o snd o snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *trans* (*N p*) | *p*. *p* < *n_ps*})

lemma (**in** *Prod_TA_Defs*) *states_loc_set*:
states ⊆ {*L*. *set L* ⊆ *loc_set* ∧ *length L* = *n_ps*}
unfolding *states_def loc_set_def*
apply (*intros add*: *more_intros*)
apply (*elims add*: *more_elims*)
apply (*drule mem_nth*)
apply *simp*
apply (*elims add*: *allE*, *assumption*)
apply (*simp split*: *prod.split_asm*)
apply (*erule disjE*; (*intros add*: *disjI1 disjI2 more_intros*, *solve_triv+*); *fail*)
by (*elims add*: *more_elims*)

lemma (**in** *Prod_TA_Defs*) *finite_states*:
assumes *finite_trans*: ∀ *p* < *n_ps*. *finite* (*Simple_Network_Language.trans* (*N p*))
shows *finite states*
proof −
have *states* ⊆ {*L*. *set L* ⊆ *loc_set* ∧ *length L* = *n_ps*}
by (*rule states_loc_set*)
also from *finite_trans* **have** *finite* ...
unfolding *loc_set_def* **by** (*intro finite_intros*) *auto*
finally show *?thesis* **.**
qed

context *Simple_Network_Impl*
begin

```
lemma trans_N_finite:
  assumes p < n_ps
  shows finite (Simple_Network_Language.trans (N p))
  using assms by (subst N_eq) (auto simp: automaton_of_def trans_def split: prod.split)

lemma states_finite:
  finite states
  by (intros add: finite_states trans_N_finite)

lemma bounded_finite:
  finite {s. bounded bounds s}
proof −
  let ?l = Min {fst (the (bounds x)) | x. x ∈ dom bounds}
  let ?u = Max {snd (the (bounds x)) | x. x ∈ dom bounds}
  have finite (dom bounds)
    unfolding bounds_def by simp
  then have {s. bounded bounds s} ⊆ {s. dom s = dom bounds ∧ ran s ⊆ {?l..?u}}
    unfolding bounded_def
    apply clarsimp
    apply (rule conjI)
    subgoal for s v
      unfolding ran_is_image
      apply clarsimp
      subgoal for x l u
        by (rule order.trans[where b = fst (the (bounds x))]; (rule Min_le)?; force)
      done
    subgoal for s v
      unfolding ran_is_image
      apply clarsimp
      subgoal for x l u
        by (rule order.trans[where b = snd (the (bounds x))]; (rule Max_ge)?; force)
      done
    done
  also from ‹finite (dom bounds)› have finite ...
    by (rule finite_set_of_finite_maps) blast
  finally show ?thesis .
qed

lemma trans_prod_finite:
  finite trans_prod
proof −
  have finite trans_int
  proof −
    have trans_int ⊆
      {((L, s), g, Internal a, r, (L', s')) | L s p l b g a f r l' s' L'.
        L ∈ states ∧ bounded bounds s ∧ p < n_ps ∧
        (l, b, g, Sil a, f, r, l') ∈ trans (N p) ∧
        bounded bounds s'
        ∧ L' = L[p := l']
      }
      unfolding trans_int_def by (force simp: L_len)
    also have finite ...
    proof −
      have finite {(a, b, c, d, e, f, g). (a, b, c, Sil d, e, f, g) ∈ trans (N p)}
```

```
    if p < n_ps for p
    using [[simproc add: finite_Collect]] that
    by (auto intro: trans_N_finite finite_vimageI injI)
  with states_finite bounded_finite show ?thesis
    by defer_ex
 qed
 finally show ?thesis .
qed
moreover have finite trans_bin
proof −
 have trans_bin ⊆
  {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
   L s p q l1 b1 g1 a f1 r1 l1' l2 b2 g2 f2 r2 l2' s'' L'.
    L ∈ states ∧ bounded bounds s ∧
    p < n_ps ∧ q < n_ps ∧
    (l1, b1, g1, In a, f1, r1, l1') ∈ trans (N p) ∧
    (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
    bounded bounds s'' ∧
    L' = L[p := l1', q := l2']
 }
  unfolding trans_bin_def by (fastforce simp: L_len)
 also have finite ...
 proof −
  have finite {(a, b, c, d, e, f, g). (a, b, c, In d, e, f, g) ∈ trans (N p)}
    if p < n_ps for p
    using [[simproc add: finite_Collect]] that
    by (auto intro: trans_N_finite finite_vimageI injI)
  moreover have finite {(a, b, c, e, f, g). (a, b, c, Out d, e, f, g) ∈ trans (N p)}
    if p < n_ps for p d
    using [[simproc add: finite_Collect]] that
    by (auto intro: trans_N_finite finite_vimageI injI)
  ultimately show ?thesis
    using states_finite bounded_finite by defer_ex
 qed
 finally show ?thesis .
qed
moreover have finite trans_broad
proof −
 define P where P ps ≡ set ps ⊆ {0..<n_ps} ∧ distinct ps for ps
 define Q where Q a n bs gs fs rs ≡
  (∀ p < n. ∃ q < n_ps. ∃ l l'. (l, bs ! p, gs ! p, In a, fs ! p, rs ! p, l') ∈ trans (N q)) ∧
      length bs = n ∧ length gs = n ∧ length fs = n ∧ length rs = n for a n bs gs fs rs
 define tag where tag x = True for x :: 's
 have Q_I: Q a (length ps) (map bs ps) (map gs ps) (map fs ps) (map rs ps)
  if set ps ⊆ {0..<n_ps}
    ∀ p∈set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)
  for ps :: nat list and L a bs gs fs rs ls'
  using that unfolding Q_def by (auto 4 4 dest!: nth_mem)
 have trans_broad ⊆
  {((L, s), g @ concat gs, Broad a, r @ concat rs, (L', s'')) |
  L s a p l b g f r l' ps bs gs fs rs L' s''.
   L ∈ states ∧ bounded bounds s ∧ a ∈ set broadcast ∧
   p < n_ps ∧
   (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
```

```
      P ps ∧
      Q a (length ps) bs gs fs rs ∧
      L' ∈ states ∧
      bounded bounds s'' ∧
      tag l'
    }
      unfolding trans_broad_def broadcast_def
      apply (rule subsetI)
      apply (elims add: more_elims)
      apply (intros add: more_intros)
              apply solve_triv+
           apply (simp add: L_len; fail)
          apply assumption
         apply (unfold P_def; intros; assumption)
        apply (rule Q_I; assumption)
      subgoal
        by (blast intro: state_preservation_updI state_preservation_fold_updI)
       apply assumption
      unfolding tag_def ..
    also have finite ...
    proof −
      have finite {(a, b, c, e, f, g). (a, b, c, Out d, e, f, g) ∈ trans (N p)}
        if p < n_ps for p d
        using [[simproc add: finite_Collect]] that
        by (auto intro: trans_N_finite finite_vimageI injI)
      moreover have finite {ps. P ps}
        unfolding P_def by (simp add: finite_intros)
      moreover have finite {(bs, gs, fs, rs). Q a n bs gs fs rs} (is finite ?S) for a n
      proof −
        let ?T = ⋃ (trans ‘ N ‘ {0..<n_ps})
        have ?S ⊆ {(bs, gs, fs, rs).
          (set bs ⊆ (λ(_,b,_). b) ‘ ?T ∧ length bs = n) ∧
          (set gs ⊆ (λ(_,_,g,_). g) ‘ ?T ∧ length gs = n) ∧
          (set fs ⊆ (λ(_,_,_,_,f,_). f) ‘ ?T ∧ length fs = n) ∧
          (set rs ⊆ (λ(_,_,_,_,_,r,_). r) ‘ ?T ∧ length rs = n)
        }
          unfolding Q_def
          apply safe
          apply (all ‹drule mem_nth; elims; drule spec; elims›)
          apply force+
          done
        also have finite ...
          using trans_N_finite by (intro finite_intros more_finite_intros) auto
        finally show ?thesis .
      qed
      ultimately show ?thesis
        using states_finite bounded_finite by defer_ex
    qed
    finally show ?thesis .
  qed
  ultimately show ?thesis
    by (simp add: trans_prod_def)
qed
```

lemma *prod_inv_finite*:
finite (*range prod_inv*)
apply (*intros add*: *finite_range_invI*)
unfolding *length_automata_eq_n_ps*[*symmetric*]
unfolding *N_def*
unfolding *automaton_of_def*
by (*auto intro*: *finite_range_default_map_of split*: *prod.split simp*: *inv_def*)

end

Collecting variables from expressions. **fun** *vars_of_bexp* **and** *vars_of_exp* **where**
vars_of_bexp (*not e*) = *vars_of_bexp e*
| *vars_of_bexp* (*and e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*bexp.or e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*imply e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*eq i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*le i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*lt i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*ge i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*gt i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp bexp.true* = {}
| *vars_of_exp* (*const c*) = {}
| *vars_of_exp* (*var x*) = {*x*}
| *vars_of_exp* (*if_then_else b e1 e2*) = *vars_of_bexp b* ∪ *vars_of_exp e1* ∪ *vars_of_exp e2*
| *vars_of_exp* (*binop _ e1 e2*) = *vars_of_exp e1* ∪ *vars_of_exp e2*
| *vars_of_exp* (*unop _ e*) = *vars_of_exp e*

definition (**in** *Prod_TA_Defs*)
var_set =
(⋃ *S* ∈ {(*fst* ∘ *snd*) ‘ *trans* (*N p*) | *p*. *p* < *n_ps*}. ⋃ *b* ∈ *S*. *vars_of_bexp b*) ∪
(⋃ *S* ∈ {(*fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *trans* (*N p*) | *p*. *p* < *n_ps*}.
⋃ *f* ∈ *S*. ⋃ (*x*, *e*) ∈ *set f*. {*x*} ∪ *vars_of_exp e*)

locale *Simple_Network_Impl_nat_defs* =
Simple_Network_Impl automata
for *automata* ::
(*nat list* × *nat list* × (*nat act*, *nat*, *nat*, *int*, *nat*, *int*) *transition list*
× (*nat* × (*nat*, *int*) *cconstraint*) *list*) *list* +
fixes *m* :: *nat* **and** *num_states* :: *nat* ⇒ *nat* **and** *num_actions* :: *nat*

locale *Simple_Network_Impl_nat* =
Simple_Network_Impl_nat_defs +
assumes *has_clock*: *m* > *0*
assumes *non_empty*: *0* < *length automata*

assumes *trans_num_states*:
∀ *i* < *n_ps*. *let* (_, _, *trans*, _) = (*automata* ! *i*) *in* ∀ (*l*, _, _, _, _, _, *l′*) ∈ *set trans*.
l < *num_states i* ∧ *l′* < *num_states i*
and *inv_num_states*:
∀ *i* < *n_ps*. *let* (_, _, _, *inv*) = (*automata* ! *i*) *in* ∀ (*x*, _) ∈ *set inv*. *x* < *num_states i*
assumes *var_set*:
∀ (_, _, *trans*, _) ∈ *set automata*. ∀ (_, _, _, _, *f*, _, _) ∈ *set trans*.
∀ (*x*, *upd*) ∈ *set f*. *x* < *n_vs* ∧ (∀ *i* ∈ *vars_of_exp upd*. *i* < *n_vs*)

$\forall (_, _, trans, _) \in set\ automata.\ \forall (_, b, _, _, _, _, _) \in set\ trans.$
$\forall i \in vars_of_bexp\ b.\ i < n_vs$
assumes *bounds*:
$\forall\ i < n_vs.\ fst\ (bounds'\ !\ i) = i$
assumes *action_set*:
$\forall a \in set\ broadcast.\ a < num_actions$
$\forall (_, _, trans, _) \in set\ automata.\ \forall (_, _, _, a, _, _, _) \in set\ trans.$
$pred_act\ (\lambda a.\ a < num_actions)\ a$
assumes *clock_set*:
$\forall (_, _, trans, _) \in set\ automata.\ \forall (_, _, g, _, _, r, _) \in set\ trans.$
$(\forall c \in set\ r.\ 0 < c \wedge c \leq m) \wedge$
$(\forall\ (c, x) \in collect_clock_pairs\ g.\ 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$

$\forall (_, _, _, inv) \in set\ automata.\ \forall (l, g) \in set\ inv.$
$(\forall\ (c, x) \in collect_clock_pairs\ g.\ 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$

assumes *broadcast_receivers*:
$\forall (_, _, trans, _) \in set\ automata.\ \forall (_, _, g, a, _, _, _) \in set\ trans.$
$case\ a\ of\ In\ a \Rightarrow a \in set\ broadcast \longrightarrow g = [] \mid _ \Rightarrow True$
begin

lemma *broadcast_receivers_unguarded*:
$\forall p{<}n_ps.\ \forall l\ b\ g\ a\ f\ r\ l'.$
$(l, b, g, In\ a, f, r, l') \in Simple_Network_Language.trans\ (N\ p) \wedge a \in set\ broadcast \longrightarrow g = []$
using *broadcast_receivers* **by** (*fastforce dest*: *nth_mem simp*: *n_ps_def mem_trans_N_iff*)

sublocale *conv*: *Prod_TA*
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
using *broadcast_receivers_unguarded*
by − (*standard*,
fastforce simp: *conv.broadcast_def Simple_Network_Language.conv_t_def conv_n_ps_eq trans_conv_N_eq*)

sublocale *TA_Start_No_Ceiling prod_ta init m*
proof *standard*
show *finite* (*trans_of prod_ta*)
using *trans_prod_finite* **by** *simp*
next
show *finite* (*range* (*inv_of prod_ta*))
using *prod_inv_finite* **by** *simp*
next
from *clk_set'_subs* **have** *clk_set prod_ta* $\subseteq$ *clk_set'* **.**
also have $\ldots \subseteq \{1..m\}$
using *clock_set* **unfolding** *clk_set'_def clkp_set'_def* **by** *force*
finally show *clk_set prod_ta* $\subseteq \{1..m\}$ **.**
next
from *clock_set* **have** $\forall (_, d){\in}clkp_set'.\ d \in \mathbb{N}$
unfolding *clkp_set'_def* **by** *force*
then show $\forall (_, d){\in}Timed_Automata.clkp_set\ prod_ta.\ d \in \mathbb{N}$
by (*auto dest*!: *subsetD*[*OF clkp_set'_subs*])
next
show $0 < m$
by (*rule has_clock*)

qed

end

context *Simple_Network_Impl*
begin

definition *sem* ≡ (*set broadcast*, *map* (*automaton_of o conv_automaton*) *automata*, *map_of bounds'*)

sublocale *sem*: *Prod_TA_sem sem* **.**

lemma *sem_N_eq*:
sem.N p = *automaton_of* (*conv_automaton* (*automata* ! *p*)) **if** ‹*p* < *n_ps*›
using *that* **unfolding** *sem.N_def n_ps_def* **unfolding** *sem_def fst_conv snd_conv*
by (*subst nth_map*) *auto*

end

inductive_cases *step_u_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* ∈ *states*
using *that* **unfolding** *states_def* **by** *simp*

end
theory *Simple_Network_Language_Impl_Refine*
imports *Simple_Network_Language_Impl*
begin

unbundle *no_library_syntax*
notation *fun_rel_syn* (**infixr** → *60*)
hide_const (**open**) *upd*

Expression evaluation **fun** *bval* :: ('*a* ⇒ '*b*) ⇒ ('*a*, '*b* :: *linorder*) *bexp* ⇒ *bool* **and** *eval* **where**
bval _ *bexp.true* ⟷ *True* |
bval s (*not e*) ⟷ ¬ *bval s e* |
bval s (*and e1 e2*) ⟷ *bval s e1* ∧ *bval s e2* |
bval s (*bexp.or e1 e2*) ⟷ *bval s e1* ∨ *bval s e2* |
bval s (*imply e1 e2*) ⟷ *bval s e1* ⟶ *bval s e2* |
bval s (*eq i x*) ⟷ *eval s i* = *eval s x* |

```
  bval s (le i x) ⟷ eval s i ≤ eval s x |
  bval s (lt i x) ⟷ eval s i < eval s x |
  bval s (ge i x) ⟷ eval s i ≥ eval s x |
  bval s (gt i x) ⟷ 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 ⟹ check_bexp s b b2 ⟹ b1 = b2
  and is_val_determ: is_val s e v1 ⟹ is_val s e v2 ⟹ v1 = v2
  by (induction b and e arbitrary: b1 b2 and v1 v2)
     (auto dest:  elim!: is_val_elims check_bexp_elims)+

lemma is_upd_determ:
  s1 = s2 if is_upd s f s1 is_upd s f s2
  using that unfolding is_upd_def
  by (clarsimp, fo_rule arg_cong)
     (smt
    case_prodE case_prodE' case_prod_conv is_val_determ list.rel_eq list_all2_swap list_all2_trans
     )

lemma is_upds_determ:
  s1 = s2 if is_upds s fs s1 is_upds s fs s2
  using that
  by (induction fs arbitrary: s) (auto 4 4 elim: is_upds.cases dest: is_upd_determ)

lemma check_bexp_bval:
  ∀x ∈ vars_of_bexp b. x ∈ dom s ⟹ check_bexp s b (bval (the o s) b)
and is_val_eval:
  ∀x ∈ vars_of_exp e. x ∈ dom s ⟹ is_val s e (eval (the o s) e)
  apply (induction b and e)
                    apply (simp; (subst eq_commute)?; rule check_bexp_is_val.intros; simp add:
dom_def)+
   apply ((subst eq_commute eval.simps)?; rule check_bexp_is_val.intros; simp add: dom_def)+
  done

lemma is_upd_dom:
  dom s' = dom s if is_upd s (x, e) s' x ∈ dom s
  using that unfolding is_upd_def by (auto split: if_split_asm)

lemma is_upds_dom:
  dom s' = dom s if is_upds s upds s' ∀(x, e) ∈ set upds. x ∈ dom s
  using that
  by (induction upds arbitrary: s)
     (erule is_upds.cases; auto simp: is_upd_dom split: prod.split_asm)+

definition
  mk_upd ≡ λ(x, e) s. s(x ↦ eval (the o s) e)

definition mk_upds ::
  ('a ⇒ ('b :: linorder) option) ⇒ ('a × ('a, 'b) exp) list ⇒ ('a ⇒ '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 ∈ vars_of_exp e. x ∈ dom s*
using *that(1) is_val_eval[OF that(2)]* **unfolding** *is_upd_def mk_upd_def* **by** *auto*

lemma *dom_mk_upd*:
dom s ⊆ dom (mk_upd upd s)
unfolding *mk_upd_def* **by** (*auto split*: *prod.split*)

lemma *is_upds_make_updsI*:
is_upds s upds (mk_upds s upds) **if** $\forall$ *(_, e) ∈ set upds.* $\forall$ *x ∈ vars_of_exp e. x ∈ dom s*
using *that*
proof (*induction upds arbitrary*: *s*)
case *Nil*
then show *?case*
by (*auto intro*!: *is_upds.intros simp*: *mk_upds_def*)
next
case (*Cons upd upds*)
let *?s = mk_upd upd s*
from *Cons(2)* **have** *is_upd s upd ?s*
by (*auto simp*: *dom_def intro*: *is_upd_make_updI*)
moreover have *is_upds ?s upds (mk_upds ?s upds)*
using *Cons.prems dom_mk_upd* **by** (*intro Cons.IH*) *force*
ultimately show *?case*
using *dom_mk_upd* **by** (*auto intro*!: *is_upds.intros simp*: *mk_upds_def*)
qed

Implementation auxiliaries **definition**
union_map_of xs =
fold (λ (*x*, *y*) *m*. *case m x of None ⇒ m(x ↦ [y]) | Some ys ⇒ m(x ↦ y # ys))* *xs Map.empty*

lemma *union_map_of_alt_def*:
union_map_of xs x = (let
ys = rev (map snd (filter (λ (*x′*, *y*). *x′ = x) xs))*
in if ys = [] then None else Some ys)
proof −
have *fold* (λ (*x*, *y*) *m*. *case m x of None ⇒ m(x ↦ [y]) | Some ys ⇒ m(x ↦ y # ys))* *xs m x*
= (let
ys = rev (map snd (filter (λ (*x′*, *y*). *x′ = x) xs))*
in
case m x of None ⇒ if ys = [] then None else Some ys | Some zs ⇒ Some (ys @ zs)) **for** *x m*
by (*induction xs arbitrary*: *m*; *clarsimp split*: *option.split*)
then show *?thesis*
unfolding *union_map_of_def* **by** *simp*
qed

lemma *in_union_map_ofI*:
∃ *ys. union_map_of xs x = Some ys ∧ y ∈ set ys* **if** *(x, y) ∈ set xs*
unfolding *union_map_of_alt_def Let_def* **using** *that set_filter*[*of* λ(*x′*, *y*). *x′ = x xs*] **by** *auto+*

lemma *in_union_map_ofD*:

$(x, y) \in set\ xs$ **if** *union_map_of xs x = Some ys y ∈ set ys*
using *that* **unfolding** *union_map_of_alt_def* **by** (*auto split*: *if_split_asm*)

Implementation of state set **context** *Simple_Network_Impl_nat_defs*
begin

definition
states_i i = $(\bigcup (l, e, g, a, r, u, l') \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))).\ \{l, l'\}$

lemma *states_mem_compute*[*code*]:
L ∈ states ⟷ *length L = n_ps* ∧ (∀ *i*<*n_ps*. *L ! i ∈ states_i i*)
unfolding *states_def states_i_def* **by** *simp* (*metis mem_trans_N_iff*)

lemma *states_mem_compute′*:
L ∈ states ⟷ *length L = n_ps* ∧ (∀ *i*<*n_ps*. *L ! i ∈ map states_i* [*0*..<*n_ps*] *! i*)
unfolding *states_mem_compute* **by** *simp*

end

context *Simple_Network_Impl_nat*
begin

Fundamentals **lemma** *mem_trans_N_iff*:
‹*t ∈ Simple_Network_Language.trans* (*N i*) ⟷ *t ∈ set* (*fst* (*snd* (*snd* (*automata ! i*))))›
if *i < n_ps*
unfolding *N_def fst_conv snd_conv*
unfolding *automaton_of_def*
unfolding *trans_def*
using *that* **by** (*cases automata ! i*) (*auto simp*: *length_automata_eq_n_ps*)

lemma *L_i_len*:
‹*L ! i < num_states i*› **if** *i < n_ps L ∈ states*
using *trans_num_states that*
unfolding *states_def* **by** (*auto 4 4 simp*: *mem_trans_N_iff*)

lemma *L_i_simp*:
‹[*0*..<*num_states i*] *! (L ! i) = L ! i*›
if *i < n_ps L ∈ states*
using *L_i_len*[*OF that*] **by** *simp*

lemma *action_setD*:
‹*pred_act* (λ*a′*. *a′ < num_actions*) *a*›
if ‹(*l, b, g, a, f, r, l′*) ∈ *Simple_Network_Language.trans* (*N p*)› ‹*p < n_ps*›
using *that action_set*
by (*cases automata ! p*)
(*subst* (*asm*) *mem_trans_N_iff*; *force dest*!: *nth_mem simp flip*: *length_automata_eq_n_ps*)

More precise state sets **definition**
states′ ≡ {(*L, s*). *L ∈ states* ∧ *dom s* = {*0*..<*n_vs*} ∧ *bounded bounds s*}

lemma *states′_states*[*intro, dest*]:
L ∈ states **if** (*L, s*) ∈ *states′*

using *that* **unfolding** *states'_def* **by** *auto*

lemma *states'_dom*[*intro*, *dest*]:
dom s = {*0*..<*n_vs*} **if** (*L*, *s*) ∈ *states'*
using *that* **unfolding** *states'_def* **by** *auto*

lemma *states'_bounded*[*intro*, *dest*]:
bounded bounds s **if** (*L*, *s*) ∈ *states'*
using *that* **unfolding** *states'_def* **by** *auto*

Implementation of invariants **definition** (**in** *Simple_Network_Impl_nat_defs*)
invs i ≡ *let m* = *default_map_of* [] (*snd* (*snd* (*snd* (*automata* ! *i*))));
m' = *map* (λ *j*. *m j*) [*0*..<*num_states i*]
in m'

definition (**in** *Simple_Network_Impl_nat_defs*)
invs1 ≡ *map* (λ *i*. *let m* = *default_map_of* [] (*snd* (*snd* (*snd* (*automata* ! *i*))));
m' = *map* (λ *j*. *m j*) [*0*..<*num_states i*]
in m') [*0*..<*n_ps*]

definition (**in** *Simple_Network_Impl_nat_defs*)
invs2 ≡ *IArray* (*map* (λ *i*. *let m* = *default_map_of* [] (*snd* (*snd* (*snd* (*automata* ! *i*))));
m' = *IArray* (*map* (λ *j*. *m j*) [*0*..<*num_states i*])
in m') [*0*..<*n_ps*])

lemma *refine_invs2*:
invs2 !! *i* !! *j* = *invs1* ! *i* ! *j* **if** *i* < *n_ps*
using *that* **unfolding** *invs2_def invs1_def* **by** *simp*

definition (**in** *Simple_Network_Impl_nat_defs*)
inv_fun ≡ λ(*L*, _).
concat (*map* (λ*i*. *invs1* ! *i* ! (*L* ! *i*)) [*0*..<*n_ps*])

lemma *refine_invs1*:
‹*invs1* ! *i* = *invs i*› **if** ‹*i* < *n_ps*›
using *that* **unfolding** *invs_def invs1_def* **by** *simp*

lemma *invs_simp*:
invs1 ! *i* ! (*L* ! *i*) = *Simple_Network_Language.inv* (*N i*) (*L* ! *i*)
if *i* < *n_ps L* ∈ *states*
using *that* **unfolding** *refine_invs1*[*OF* ‹*i* < _›] *invs_def N_def fst_conv snd_conv*
unfolding *inv_def*
by (*subst nth_map*;
clarsimp split: *prod.split simp*: *automaton_of_def length_automata_eq_n_ps L_i_len*)

lemma *inv_fun_inv_of'*:
(*inv_fun*, *inv_of prod_ta*) ∈ *inv_rel R states'* **if** *R* ⊆ *Id* ×$_r$ *S*
using *that*
unfolding *inv_rel_def*
unfolding *inv_fun_def*
unfolding *inv_of_prod prod_inv_def*
apply (*clarsimp simp*: *states'_def*)
apply (*rule arg_cong*[**where** *f* = *concat*])

apply (*auto simp add*: *invs_simp cong*: *map_cong*)
done

lemma *inv_fun_alt_def*:
inv_fun = (λ(*L*, _). *concat* (*map* (λ*i*. *invs2* !! *i* !! (*L* ! *i*)) [*0*..<*n_ps*]))
unfolding *inv_fun_def*
apply (*intro ext*)
apply (*clarsimp simp del*: *IArray.sub_def*)
apply (*fo_rule arg_cong*)
apply (*simp add*: *refine_invs2 del*: *IArray.sub_def*)
done

end

Implementation of transitions **context** *Simple_Network_Impl_nat_defs*
begin

definition
bounds_map ≡ *the* ∘ *map_of bounds'*

definition
check_bounded s =
(∀ *x* ∈ *dom s*. *fst* (*bounds_map x*) ≤ *the* (*s x*) ∧ *the* (*s x*) ≤ *snd* (*bounds_map x*))

Compute pairs of processes with committed initial locations from location vector.

definition
get_committed L =
List.map_filter (λ*p*.
let l = *L* ! *p in*
if l ∈ *set* (*fst* (*automata* ! *p*)) *then Some* (*p*, *l*) *else None*) [*0*..<*n_ps*]

Given a process and a location, return the corresponding transitions.

definition
trans_map i ≡
let m = *union_map_of* (*fst* (*snd* (*snd* (*automata* ! *i*)))) *in* (λ*j*.
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* (λ*p*. *int_trans_from_loc p* (*L* ! *p*) *L s*) [*0*..<*n_ps*])

definition
int_trans_from ≡ λ (*L*, *s*).
let pairs = *get_committed L in*
if pairs = []
then int_trans_from_all L s
else int_trans_from_vec pairs L s

definition
actions_by_state i ≡
fold (λ *t acc*. *acc*[*fst* (*snd* (*snd t*)) := (*i*, *t*) # (*acc* ! *fst* (*snd* (*snd t*)))])

definition
all_actions_by_state t L ≡
fold (λ *i*. *actions_by_state i* (*t i* (*L* ! *i*))) [*0*..<*n_ps*] (*repeat* [] *num_actions*)

definition
all_actions_from_vec t vec ≡
fold (λ(*p*, *l*). *actions_by_state p* (*t p l*)) *vec* (*repeat* [] *num_actions*)

definition
pairs_by_action L s OUT IN ≡
concat (
map (λ (*p*, *b1*, *g1*, *a1*, *f1*, *r1*, *l1*).
List.map_filter (λ (*q*, *b2*, *g2*, *a2*, *f2*, *r2*, *l2*).
if p = *q then None else*
let s' = *mk_upds* (*mk_upds s f1*) *f2 in*
if bval (*the o s*) *b1* ∧ *bval* (*the o s*) *b2* ∧ *check_bounded s'*
then Some (*g1* @ *g2*, *Bin a1*, *r1* @ *r2*, (*L*[*p* := *l1*, *q* := *l2*], *s'*))
else None
) *OUT*) *IN*)

definition

```
  trans_in_map i j ≡
    List.map_filter
      (λ (b, g, a, m, l'). case a of In a ⇒ Some (b, g, a, m, l') | _ ⇒ None)
    (trans_map i j)

definition
  trans_out_map i j ≡
    List.map_filter
      (λ (b, g, a, m, l'). case a of Out a ⇒ Some (b, g, a, m, l') | _ ⇒ None)
    (trans_map i j)

definition
  bin_actions = filter (λa. a ∉ set broadcast) [0..<num_actions]

lemma mem_bin_actions_iff:
  a ∈ set bin_actions ⟷ a ∉ local.broadcast ∧ a < num_actions
  unfolding bin_actions_def broadcast_def by auto

definition
  bin_trans_from ≡ λ(L, s).
    let
      pairs = get_committed L;
      In =  all_actions_by_state trans_in_map L;
      Out = all_actions_by_state trans_out_map L
    in
    if pairs = [] then
      concat (map (λa. pairs_by_action L s (Out ! a) (In ! a)) bin_actions)
    else
      let
        In2  = all_actions_from_vec trans_in_map pairs;
        Out2 = all_actions_from_vec trans_out_map pairs
      in
        concat (map (λa. pairs_by_action L s (Out ! a) (In2 ! a)) bin_actions)
      @ concat (map (λa. pairs_by_action L s (Out2 ! a) (In ! a)) bin_actions)


definition
  trans_in_broad_map i j ≡
    List.map_filter
      (λ (b, g, a, m, l').
      case a of In a ⇒ if a ∈ set broadcast then Some (b, g, a, m, l') else None | _ ⇒ None)
    (trans_map i j)

definition
  trans_out_broad_map i j ≡
    List.map_filter
      (λ (b, g, a, m, l').
      case a of Out a ⇒ if a ∈ set broadcast then Some (b, g, a, m, l') else None | _ ⇒ None)
    (trans_map i j)

definition
  actions_by_state' xs ≡
    fold (λ t acc. acc[fst (snd (snd t)) := t # (acc ! fst (snd (snd t)))])
      xs (repeat [] num_actions)
```

```
definition
  trans_out_broad_grouped i j ≡ 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..<n_ps])
  )
 [0..<num_actions])


end

lemma product_lists_empty: product_lists xss = [] ⟷ (∃ xs ∈ set xss. xs = []) for xss
  by (induction xss) auto

context Simple_Network_Impl_nat
begin

lemma broad_trans_from_alt_def:
  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
              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..<n_ps])
      )
    [0..<num_actions])
  else
    concat (
      map (λa.
        let
          ins_committed =
            List.map_filter (λ(p, _). if In ! p ! a ≠ [] then Some p else None) pairs
        in
        concat (map (λp.
          let
            outs = Out ! p ! a
          in if outs = [] then []
          else if
            (ins_committed = [p] ∨ ins_committed = []) ∧ ¬ list_ex (λ(q, _). q = p) pairs
          then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λx. map (λxs. x # xs) combs) outs));
              init = ([], Broad a, [], (L, s))
            in
            filter (λ (g, a, r, L, s). check_bounded s) (
              map (λcomb.
                  fold
                    (λ(q, _, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
                      (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
                    )
                    comb
                    init
              ) combs)
        )
        [0..<n_ps])
      )
    [0..<num_actions])

apply (rule eq_reflection)
unfolding broad_trans_from_def
unfolding filter_map_map_filter
unfolding Let_def
by (fo_rule
    arg_cong2[where f = map] arg_cong2[where f = List.map_filter] arg_cong HOL.refl |
    rule if_cong ext | auto split: if_split_asm)+
```

Correctness for implementations of primitives **lemma** *dom_bounds_eq*:
dom bounds = {*0..<n_vs*}

using *bounds* **unfolding** *bounds_def*
apply *simp*
unfolding *n_vs_def dom_map_of_conv_image_fst*
apply *safe*
apply (*force dest*: *mem_nth*; *fail*)
apply (*force dest*: *nth_mem*; *fail*)
done

lemma *check_bounded_iff*:
Simple_Network_Language.bounded bounds s $\longleftrightarrow$ *check_bounded s* **if** *dom s* = {*0*..<*n_vs*}
using *that*
unfolding *dom_bounds_eq*[*symmetric*]
unfolding *check_bounded_def Simple_Network_Language.bounded_def bounds_map_def bounds_def*
by *auto*

lemma *get_committed_mem_iff*:
(*p*, *l*) ∈ *set* (*get_committed L*) $\longleftrightarrow$ (*l* = *L* ! *p* ∧ *l* ∈ *committed* (*N p*) ∧ *p* < *n_ps*)
unfolding *get_committed_def*
unfolding *set_map_filter Let_def*
apply *clarsimp*
unfolding *N_def fst_conv snd_conv*
unfolding *committed_def*
by *safe*
((*subst nth_map* | *subst* (*asm*) *nth_map*);
auto split: *prod.splits simp*: *automaton_of_def length_automata_eq_n_ps*
)+

lemma *get_committed_empty_iff*:
(∀ *p* < *n_ps*. *L* ! *p* ∉ *committed* (*N p*)) $\longleftrightarrow$ *get_committed L* = []
apply *safe*
subgoal
proof (*rule ccontr*)
assume *prems*:
∀ *p*<*n_ps*. *L* ! *p* ∉ *committed* (*N p*) **and**
get_committed L ≠ []
then obtain *p l* **where** (*p*, *l*) ∈ *set* (*get_committed L*)
by (*metis length_greater_0_conv nth_mem old.prod.exhaust*)
from *this*[*unfolded get_committed_mem_iff*] *prems*(*1*)
show *False*
by *auto*
qed
subgoal for *p*
using *get_committed_mem_iff*[*of p L* ! *p L*] **by** *auto*
done

lemma *get_committed_distinct*: *distinct* (*get_committed L*)
unfolding *get_committed_def* **by** (*rule distinct_map_filterI*) (*auto simp*: *Let_def*)

lemma *is_upds_make_updsI2*:
is_upds s upds (*mk_upds s upds*)
if (*l*, *b*, *g*, *a*, *upds*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
dom s = {*0*..<*n_vs*}
using *that var_set*
by (*intro is_upds_make_updsI*, *subst* (*asm*) *mem_trans_N_iff*)

(*auto 4 5 simp flip*: *length_automata_eq_n_ps dest*!: *nth_mem*)

lemma *var_setD*:
 ∀(*x*, *upd*)∈*set f*. *x* < *n_vs* ∧ (∀*i*∈*vars_of_exp upd*. *i* < *n_vs*)
 if (*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
 using *var_set that*
 by (*force dest*: *nth_mem simp flip*: *length_automata_eq_n_ps simp*: *mem_trans_N_iff*)+

lemma *var_setD2*:
 ∀*i*∈*vars_of_bexp b*. *i* < *n_vs*
 if (*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
 using *var_set that*
 by (*force dest*: *nth_mem simp flip*: *length_automata_eq_n_ps simp*: *mem_trans_N_iff*)+

lemma *is_upds_dom2*:
 dom s' = {*0*..<*n_vs*} **if** *is_upds s f s'*
 (*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
 dom s = {*0*..<*n_vs*}
 unfolding *that*(*4*)[*symmetric*] **using** *that* **by** − (*drule* (*1*) *var_setD*, *erule is_upds_dom*, *auto*)

lemma *is_updsD*:
 s' = *mk_upds s f* **if** *is_upds s f s'*
 (*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
 dom s = {*0*..<*n_vs*}
proof −
 from *is_upds_make_updsI2*[*OF that*(*2*−)] **have** *is_upds s f* (*mk_upds s f*) **.**
 from *is_upds_determ*[*OF that*(*1*) *this*] **show** *?thesis* **.**
qed

lemma *is_upds_make_upds_concatI2*:
 is_upds s (*concat upds*) (*mk_upds s* (*concat upds*))
 if *dom s* = {*0*..<*n_vs*}
  ∀*upd* ∈ *set upds*. ∃*p* < *n_ps*. ∃*l b g a r l'*.
   (*l*, *b*, *g*, *a*, *upd*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*)
 using *that var_set*
 by (*intro is_upds_make_updsI*, *clarsimp*)
  (*smt* (*z3*) *atLeastLessThan_iff case_prod_conv domD var_setD zero_le*)

lemma *is_upds_concat_dom2*:
 assumes *is_upds s* (*concat upds*) *s'*
  and ∀*f* ∈ *set upds*. ∃ *p* < *n_ps*. ∃ *l b g a r l'*.
   (*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*)
  and *dom s* = {*0*..<*n_vs*}
 shows *dom s'* = *dom s*
 using *assms* **by** (*elim is_upds_dom*) (*auto dest*!: *var_setD*)

lemma *is_upds_dom3*:
 assumes *is_upds s* (*concat_map fs ps*) *s'*
  and ∀*p*∈*set ps*. (*L* ! *p*, *bs p*, *gs p*, *a*, *fs p*, *rs p*, *ls' p*) ∈ *trans* (*N p*)
  and *set ps* ⊆ {*0*..<*n_ps*}
  and *dom s* = {*0*..<*n_vs*}
 shows *dom s'* = *dom s*
 using *assms* **by** (*elim is_upds_concat_dom2*; *force*)

lemma *is_upds_make_updsI3*:
is_upds s (*concat_map fs ps*) (*mk_upds s* (*concat_map fs ps*))
if *dom s* = {*0*..<*n_vs*}
and ∀ *p*∈*set ps*. (*L* ! *p*, *bs p*, *gs p*, *a*, *fs p*, *rs p*, *ls*′ *p*) ∈ *trans* (*N p*)
and *set ps* ⊆ {*0*..<*n_ps*}
for *s* :: *nat* ⇒ *int option*
using *that* **by** (*elim is_upds_make_upds_concatI2*) *force*

lemma *is_upds_concatD*:
assumes
dom s = {*0*..<*n_vs*} **and**
∀ *p*∈*set ps*.
(*ls p*, *bs p*, *gs p*, *a*, *fs p*, *rs p*, *ls*′ *p*)
∈ *Simple_Network_Language.trans* (*N p*) **and**
set ps ⊆ {*0*..<*n_ps*} **and**
is_upds s (*concat_map fs ps*) *s*′
shows *s*′ = *mk_upds s* (*concat_map fs ps*)
proof −
let *?upds* = *concat_map fs ps*
have *is_upds s ?upds* (*mk_upds s ?upds*)
using *assms*(*1*−*3*) **by** (*intro is_upds_make_upds_concatI2*; *force*)
from *is_upds_determ*[*OF assms*(*4*) *this*] **show** *?thesis*
by (*simp add*: *fold_map comp_def*)
qed

context
notes [*simp*] = *length_automata_eq_n_ps*
begin

lemma *trans_mapI*:
assumes
(*L* ! *p*, *g*, *a*, *f*, *r*, *l*′) ∈ *Simple_Network_Language.trans* (*N p*)
p < *n_ps*
shows
(*g*, *a*, *f*, *r*, *l*′) ∈ *set* (*trans_map p* (*L* ! *p*))
using *assms* **unfolding** *trans_map_def N_def fst_conv snd_conv trans_def*
by (*subst* (*asm*) *nth_map*) (*auto dest*: *in_union_map_ofI split*: *prod.split_asm simp*: *automaton_of_def*)

lemma *trans_i_mapI*:
assumes
(*L* ! *p*, *b*, *g*, *Sil a*′, *f*, *r*, *l*′) ∈ *Simple_Network_Language.trans* (*N p*)
p < *n_ps*
shows
(*b*, *g*, *a*′, *f*, *r*, *l*′) ∈ *set* (*trans_i_map p* (*L* ! *p*))
using *assms* **unfolding** *trans_i_map_def set_map_filter* **by** (*fastforce dest*: *trans_mapI*)

lemma *trans_mapI*′:
assumes
(*l*, *b*, *g*, *a*, *f*, *r*, *l*′) ∈ *Simple_Network_Language.trans* (*N p*)
p < *n_ps*
shows

$(b, g, a, f, r, l') \in set\ (trans_map\ p\ l)$
using *assms* **unfolding** *trans_map_def N_def fst_conv snd_conv trans_def*
by (*subst* (*asm*) *nth_map*) (*auto dest*: *in_union_map_ofI split*: *prod.split_asm simp*: *automaton_of_def*)

lemma *trans_mapD*:
assumes
$(b, g, a, f, r, l') \in set\ (trans_map\ p\ l)$
$p < n_ps$
shows
$(l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)$
using *assms* **unfolding** *trans_map_def N_def fst_conv snd_conv trans_def*
by (*subst nth_map*) (*auto split*: *prod.split elim*: *in_union_map_ofD*[*rotated*] *simp*: *automaton_of_def*)

lemma *trans_map_iff*:
assumes
$p < n_ps$
shows
$(b, g, a, f, r, l') \in set\ (trans_map\ p\ l)$
$\longleftrightarrow (l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)$
using *trans_mapD trans_mapI'* ‹$p < n_ps$› **by** *auto*

lemma *trans_i_mapD*:
assumes
$(b, g, a', f, r, l') \in set\ (trans_i_map\ p\ (L\ !\ p))$
$p < n_ps$
shows
$(L\ !\ p, b, g, Sil\ a', f, r, l') \in Simple_Network_Language.trans\ (N\ p)$
using *assms* **unfolding** *trans_i_map_def set_map_filter*
by (*force split*: *act.split_asm intro*: *trans_mapD*)

An additional brute force method for forward-chaining of facts **method** *frules_all* =
repeat_rotate ‹*frules*›, *dedup_prems*

Internal transitions **lemma** *get_committed_memI*:
$(p, L\ !\ p) \in set\ (get_committed\ L)$ **if** $L\ !\ p \in committed\ (N\ p)\ p < n_ps$
using *that* **unfolding** *get_committed_mem_iff* **by** *simp*

lemma *check_bexp_bvalI*:
$bval\ (the\ o\ s)\ b$ **if** *check_bexp s b True*
$(l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)\ p < n_ps$
$dom\ s = \{0..<n_vs\}$
proof −
from *var_setD2*[*OF that*(*2*,*3*)] ‹$dom\ s = \{0..<n_vs\}$› **have** $check_bexp\ s\ b\ (bval\ (the\ o\ s)\ b)$
by (*intro check_bexp_bval*, *simp*)
with *check_bexp_determ that*(*1*) **show** *?thesis*
by *auto*
qed

lemma *check_bexp_bvalD*:
check_bexp s b True **if** $bval\ (the\ o\ s)\ b$
$(l, b, g, a, f, r, l') \in Simple_Network_Language.trans\ (N\ p)\ p < n_ps$

```
  dom s = {0..<n_vs}
proof −
  from var_setD2[OF that(2,3)] ‹dom s = {0..<n_vs}› have check_bexp s b (bval (the o s) b)
    by (intro check_bexp_bval, simp)
  with check_bexp_determ that(1) show ?thesis
    by auto
qed

lemmas [forward2] =
  trans_i_mapD
  trans_i_mapI
  get_committed_memI
lemmas [forward3] =
  is_upds_make_updsI2
lemmas [forward4] =
  is_updsD
  is_upds_dom2
  check_bexp_bvalI
  check_bexp_bvalD

lemma int_trans_from_correct:
  (int_trans_from, trans_int) ∈ transition_rel states'
  unfolding transition_rel_def
proof (safe del: iffI)
  note [more_elims] = allE
  fix L s g a r L' s' assume (L, s) ∈ states'
  then have L ∈ states dom s = {0..<n_vs} bounded bounds s
    by auto
  then have [simp]: length L = n_ps check_bounded s
    by (auto simp: check_bounded_iff)
  show (((L, s), g, a, r, L', s') ∈ trans_int)
    ⟷ ((g, a, r, L', s') ∈ set (int_trans_from (L, s)))
  proof (cases get_committed L = [])
    case True
    then have ∗: ((L, s), g, a, r, L', s') ∈ trans_int ⟷
      ((L, s), g, a, r, L', s') ∈ {((L, s), g, Internal a, r, (L', s')) | L s l b g f p a r l' L' s'.
        (l, b, g, Sil a, f, r, l') ∈ trans (N p) ∧
        (∀ p < n_ps. L ! p ∉ committed (N p)) ∧
        check_bexp s b True ∧
        L!p = l ∧ p < length L ∧ L' = L[p := l'] ∧ is_upds s f s' ∧
        L ∈ states ∧ bounded bounds s ∧ bounded bounds s'
      }
      unfolding get_committed_empty_iff[symmetric] trans_int_def by blast
    from True have ∗∗: int_trans_from (L, s) = int_trans_from_all L s
      unfolding int_trans_from_def by simp
    from ‹dom s = _› show ?thesis
      unfolding ∗ ∗∗ int_trans_from_all_def
      apply clarsimp
      unfolding int_trans_from_loc_def Let_def set_map_filter
      apply clarsimp
      apply safe
      subgoal for b f p a' l'
        apply frules
        unfolding check_bounded_iff by (intros; solve_triv)
```

```
    subgoal for p _ a' upds l'
      apply simp
      apply frules
      using ‹L ∈ states› ‹check_bounded s› True[folded get_committed_empty_iff]
      unfolding check_bounded_iff by (intros; solve_triv)
    done
  next
    case False
    then have ∗: ((L, s), g, a, r, L', s') ∈ trans_int ⟷
      ((L, s), g, a, r, L', s') ∈ {((L, s), g, Internal a, r, (L', s')) | L s l b g f p a r l' L' s'.
        (l, b, g, Sil a, f, r, l') ∈ trans (N p) ∧
        l ∈ committed (N p) ∧
        L!p = l ∧ p < length L ∧ check_bexp s b True ∧ L' = L[p := l'] ∧ is_upds s f s' ∧
        L ∈ states ∧ bounded bounds s ∧ bounded bounds s'
      }
      unfolding get_committed_empty_iff[symmetric] trans_int_def by blast
    from False have ∗∗: int_trans_from (L, s) = int_trans_from_vec (get_committed L) L s
      unfolding int_trans_from_def by simp
    from ‹dom s = _› ‹L ∈ states› show ?thesis
      unfolding ∗ ∗∗ int_trans_from_vec_def
      apply clarsimp
      unfolding int_trans_from_loc_def Let_def set_map_filter
      apply clarsimp
      apply safe
      subgoal for f p a' l'
        apply frules
        unfolding check_bounded_iff by (intros; solve_triv)
      subgoal for p _ a' upds l'
        unfolding get_committed_mem_iff
        apply (elims; simp)
        apply frules
        unfolding check_bounded_iff by (intros; solve_triv)
      done
  qed
qed

Mostly copied  lemma in_actions_by_stateI:
  assumes
    (b1, g1, a, r1) ∈ set xs a < length acc
  shows
    (q, b1, g1, a, r1) ∈ set (actions_by_state q xs acc ! a)
    ∧ a < length (actions_by_state q xs acc)
  using assms unfolding actions_by_state_def
  apply (induction xs arbitrary: acc)
   apply (simp; fail)
  apply simp
  apply (erule disjE)
   apply (rule fold_acc_preserv
      [where P = λ acc. (q, b1, g1, a, r1) ∈ set (acc ! a) ∧ a < length acc]
      )
    apply (subst list_update_nth_split; auto)
  by auto
```

```
lemma in_actions_by_state'I:
  assumes
    (b1, g1, a, r1) ∈ set xs a < num_actions
  shows
    (b1, g1, a, r1) ∈ set (actions_by_state' xs ! a)
    ∧ a < length (actions_by_state' xs)
proof −
  let ?f = (λt acc. acc[fst (snd (snd t)) := t # acc ! fst (snd (snd t))])
  have (b1, g1, a, r1) ∈ set (fold ?f xs acc ! a)
    ∧ a < length (fold ?f xs acc) if a < length acc for acc
    using assms(1) that
    apply (induction xs arbitrary: acc)
     apply (simp; fail)
    apply simp
    apply (erule disjE)
     apply (rule fold_acc_preserv
        [where P = λ acc. (b1, g1, a, r1) ∈ set (acc ! a) ∧ a < length acc]
        )
      apply (subst list_update_nth_split; auto)
    by auto
  with ‹a < _› show ?thesis
    unfolding actions_by_state'_def by simp
qed

lemma in_actions_by_state_preserv:
  assumes
    (q, b1, g1, a, r1) ∈ set (acc ! a) a < length acc
  shows
    (q, b1, g1, a, r1) ∈ set (actions_by_state y xs acc ! a)
    ∧ a < length (actions_by_state y xs acc)
  using assms unfolding actions_by_state_def
  apply −
  apply (rule fold_acc_preserv
      [where P = λ acc. (q, b1, g1, a, r1) ∈ set (acc ! a) ∧ a < length acc]
      )
   apply (subst list_update_nth_split; auto)
  by auto

lemma length_actions_by_state_preserv[simp]:
  shows length (actions_by_state y xs acc) = length acc
 unfolding actions_by_state_def by (auto intro: fold_acc_preserv simp: list_update_nth_split)

lemma in_all_actions_by_stateI:
  assumes
    a < num_actions q < n_ps (b1, g1, a, r1) ∈ set (M q (L ! q))
  shows
    (q, b1, g1, a, r1) ∈ set (all_actions_by_state M L ! a)
  unfolding all_actions_by_state_def
  apply (rule fold_acc_ev_preserv
      [where P = λ acc. (q, b1, g1, a, r1) ∈ set (acc ! a) and Q = λ acc. a < length acc,
        THEN conjunct1]
      )
      apply (rule in_actions_by_state_preserv[THEN conjunct1])
  using assms by (auto intro: in_actions_by_stateI[THEN conjunct1])
```

lemma *in_all_actions_from_vecI*:
assumes
a < *num_actions* (*b1*, *g1*, *a*, *r1*) ∈ *set* (*M q l*) (*q*, *l*) ∈ *set pairs*
shows
(*q*, *b1*, *g1*, *a*, *r1*) ∈ *set* (*all_actions_from_vec M pairs* ! *a*)
unfolding *all_actions_from_vec_def* **using** *assms*
by (*intro fold_acc_ev_preserv*
[**where** *P* = λ *acc*. (*q*, *b1*, *g1*, *a*, *r1*) ∈ *set* (*acc* ! *a*) **and** *Q* = λ *acc*. *a* < *length acc*,
THEN conjunct1])
(*auto intro*: *in_actions_by_stateI*[*THEN conjunct1*] *in_actions_by_state_preserv*[*THEN conjunct1*])

lemma *actions_by_state_inj*:
assumes *j* < *length acc*
shows ∀ (*q*, *a*) ∈ *set* (*actions_by_state i xs acc* ! *j*). (*q*, *a*) ∉ *set* (*acc* ! *j*) ⟶ *i* = *q*
unfolding *actions_by_state_def*
apply (*rule fold_acc_preserv*
[**where** *P* =
λ *acc*′. (∀ (*q*, *a*) ∈ *set* (*acc*′ ! *j*). (*q*, *a*) ∉ *set* (*acc* ! *j*) ⟶ *i* = *q*) ∧ *j* < *length acc*′,
THEN conjunct1])
subgoal for *x acc*
by (*cases fst* (*snd* (*snd x*)) = *j*; *simp*)
using *assms* **by** *auto*

lemma *actions_by_state_inj*′:
assumes *j* < *length acc* (*q*, *a*) ∉ *set* (*acc* ! *j*) (*q*, *a*) ∈ *set* (*actions_by_state i xs acc* ! *j*)
shows *i* = *q*
using *actions_by_state_inj*[*OF assms*(*1*)] *assms*(*2*−) **by** *fast*

lemma *in_actions_by_stateD*:
assumes
(*q*, *b*, *g*, *a*, *t*) ∈ *set* (*actions_by_state i xs acc* ! *j*) (*q*, *b*, *g*, *a*, *t*) ∉ *set* (*acc* ! *j*)
j < *length acc*
shows
(*b*, *g*, *a*, *t*) ∈ *set xs* ∧ *j* = *a*
using *assms* **unfolding** *actions_by_state_def*
apply −
apply (*drule fold_evD*
[**where** *y* = (*b*, *g*, *a*, *t*) **and** *Q* = λ *acc*′. *length acc*′ = *length acc*
and *R* = λ (_, _, *a*′, *t*). *a*′ = *j*]
)
apply *assumption*
apply (*subst* (*asm*) *list_update_nth_split*[*of j*]; *force*)
apply *simp*+
apply (*subst* (*asm*) *list_update_nth_split*[*of j*]; *force*)
by *auto*

lemma *in_actions_by_state*′*D*:
assumes
(*b*, *g*, *a*, *r*) ∈ *set* (*actions_by_state*′ *xs* ! *a*′) *a*′ < *num_actions*
shows
(*b*, *g*, *a*, *r*) ∈ *set xs* ∧ *a*′ = *a*
proof −

```
  let ?f = (λt acc. acc[fst (snd (snd t)) := t # acc ! fst (snd (snd t))])
  have (b, g, a, r) ∈ set xs ∧ a' = a
    if (b, g, a, r) ∈ set (fold ?f xs acc ! a') (b, g, a, r) ∉ set (acc ! a') a' < length acc
    for acc
    using that
    apply −
    apply (drule fold_evD
        [where y = (b, g, a, r) and Q = λ acc'. length acc' = length acc
          and R = λ (_, _, a, t). a = a']
        )
         apply ((subst (asm) list_update_nth_split[where j = a']) ?; solves auto)+
    done
  with assms show ?thesis
    unfolding actions_by_state'_def by auto
qed

lemma in_all_actions_by_stateD:
  assumes
    (q, b1, g1, a, r1) ∈ set (all_actions_by_state M L ! a') a' < num_actions
  shows
    (b1, g1, a, r1) ∈ set (M q (L ! q)) ∧ q < n_ps ∧ a' = a
  using assms
  unfolding all_actions_by_state_def
  apply −
  apply (drule fold_evD''[where y = q and Q = λ acc. length acc = num_actions])
     apply (simp; fail)
    apply (drule actions_by_state_inj'[rotated])
     apply (simp; fail)+
  apply safe
    apply (drule in_actions_by_stateD)
     apply assumption
    apply (rule fold_acc_preserv)
     apply (simp; fail)+
  subgoal premises prems
  proof −
    from prems(2) have q ∈ set [0..<n_ps] by auto
    then show ?thesis by simp
  qed
  by (auto intro: fold_acc_preserv dest!: in_actions_by_stateD)

lemma length_all_actions_by_state_preserv:
  length (all_actions_by_state M L) = num_actions
  unfolding all_actions_by_state_def by (auto intro: fold_acc_preserv)

lemma length_actions_by_state'_preserv:
  length (actions_by_state' xs) = num_actions
  unfolding actions_by_state'_def by (rule fold_acc_preserv; simp)

lemma all_actions_from_vecD:
  assumes
    (q, b1, g1, a, r1) ∈ set (all_actions_from_vec M pairs ! a') a' < num_actions
    distinct (map fst pairs)
  shows
    ∃ l. (q, l) ∈ set pairs ∧ (b1, g1, a, r1) ∈ set (M q l) ∧ a' = a
```

```
  using assms(1,2)
  unfolding all_actions_from_vec_def
  apply −
  apply (drule fold_evD2'[where
        y = (q, SOME l. (q, l) ∈ set pairs)
        and Q = λ acc. length acc = num_actions])
     apply (clarsimp; fail)
    apply clarsimp
    apply (drule actions_by_state_inj'[rotated])
     apply assumption
    apply assumption
   apply simp
  subgoal
    using assms(3) by (meson distinct_map_fstD someI_ex)
  apply (solves auto)+
  apply clarsimp
  apply (drule in_actions_by_stateD)
  apply (simp; fail)
   apply (rule fold_acc_preserv)
  apply (solves auto)+
  subgoal premises prems for ys zs
    using prems(4) by (subst ‹pairs = _›) auto
  done

lemma all_actions_from_vecD2:
  assumes
    (q, b1, g1, a, r1) ∈ set (all_actions_from_vec M pairs ! a') a' < num_actions
    (q, l) ∈ set pairs distinct (map fst pairs)
  shows
    (b1, g1, a, r1) ∈ set (M q l) ∧ a' = a
  using assms(1,2,3)
  unfolding all_actions_from_vec_def
  apply −
  apply (drule fold_evD2'[where y = (q, l) and Q = λ acc. length acc = num_actions])
     apply (clarsimp; fail)
    apply clarsimp
    apply (drule actions_by_state_inj'[rotated])
     apply assumption
    apply assumption
   apply simp
  subgoal
    using assms(4) by (meson distinct_map_fstD)
  by (auto intro: fold_acc_preserv dest!: in_actions_by_stateD)

Binary transitions  lemma bin_trans_from_correct:
  (bin_trans_from, trans_bin) ∈ transition_rel states'
  unfolding transition_rel_def
proof (safe del: iffI)
  fix L s g a r L' s' assume (L, s) ∈ states'
  then have L ∈ states dom s = {0..<n_vs} bounded bounds s
    by auto
  then have [simp]: length L = n_ps
    by auto
```

define *IN* **where** *IN = all_actions_by_state trans_in_map L*
define *OUT* **where** *OUT = all_actions_by_state trans_out_map L*
have *IN_I*:
(p, b, g, a′, f, r, l′) ∈ set (IN ! a′)
if *(L ! p, b, g, In a′, f, r, l′) ∈ Simple_Network_Language.trans (N p)*
p < n_ps a′ < num_actions
for *p b g a′ f r l′*
proof −
from *trans_mapI*[*OF that*(*1*,*2*)] **have** *(b, g, In a′, f, r, l′) ∈ set (trans_map p (L ! p))*
by *auto*
then have *(b, g, a′, f, r, l′) ∈ set (trans_in_map p (L ! p))*
unfolding *trans_in_map_def set_map_filter* **by** *(auto 4 7)*
with ‹*p < _*› ‹*a′ < _*› **show** *?thesis*
unfolding *IN_def* **by** *(intro in_all_actions_by_stateI)*
qed
have *OUT_I*:
(p, b, g, a′, f, r, l′) ∈ set (OUT ! a′)
if *(L ! p, b, g, Out a′, f, r, l′) ∈ Simple_Network_Language.trans (N p)*
p < n_ps a′ < num_actions
for *p b g a′ f r l′*
proof −
from *trans_mapI*[*OF that*(*1*,*2*)] **have** *(b, g, Out a′, f, r, l′) ∈ set (trans_map p (L ! p))*
by *auto*
then have *(b, g, a′, f, r, l′) ∈ set (trans_out_map p (L ! p))*
unfolding *trans_out_map_def set_map_filter* **by** *(auto 4 7)*
with ‹*p < _*› ‹*a′ < _*› **show** *?thesis*
unfolding *OUT_def* **by** *(intro in_all_actions_by_stateI)*
qed
have *IN_D*:
(L ! p, b, g, In a′, f, r, l′) ∈ Simple_Network_Language.trans (N p) ∧ p < n_ps ∧ a′ = a1
if *(p, b, g, a′, f, r, l′) ∈ set (IN ! a1) a1 < num_actions*
for *p b g a′ f r l′ a1*
using *that*
unfolding *IN_def*
apply −
apply *(drule in_all_actions_by_stateD)*
apply *assumption*
unfolding *trans_in_map_def set_map_filter*
apply *(clarsimp split*: *option.split_asm)*
apply *(auto split*: *act.split_asm dest*: *trans_mapD)*
done
have *OUT_D*:
(L ! p, b, g, Out a1, f, r, l′) ∈ Simple_Network_Language.trans (N p) ∧ p < n_ps
if *(p, b, g, a′, f, r, l′) ∈ set (OUT ! a1) a1 < num_actions*
for *p b g a′ f r l′ a1*
using *that*
unfolding *OUT_def*
apply −
apply *(drule in_all_actions_by_stateD)*
apply *assumption*
unfolding *trans_out_map_def set_map_filter*
apply *(clarsimp split*: *option.split_asm)*
apply *(auto split*: *act.split_asm dest*: *trans_mapD)*
done

```
have IN_p_num:
  p < n_ps
  if (p, b, g, a', f, r, l') ∈ set (IN ! a1) a1 < num_actions
  for p b g a' f r l' a1
 using that unfolding IN_def by (auto dest: in_all_actions_by_stateD split: option.split_asm)
have OUT_p_num:
  p < n_ps
  if (p, b, g, a', f, r, l') ∈ set (OUT ! a1) a1 < num_actions
  for p b g a' f r l' a1
 using that unfolding OUT_def by (auto dest: in_all_actions_by_stateD split: option.split_asm)
have actions_unique:
  a' = a1
  if (p, b, g, a', f, r, l') ∈ set (IN ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that unfolding IN_def trans_in_map_def set_map_filter
  by (auto dest: in_all_actions_by_stateD split: option.split_asm)
note [forward3] = OUT_I IN_I
note [forward2] = action_setD IN_D OUT_D
show (((L, s), g, a, r, L', s') ∈ trans_bin) =
      ((g, a, r, L', s') ∈ set (bin_trans_from (L, s)))
proof (cases get_committed L = [])
  case True
  with get_committed_empty_iff[of L] have ∀p<n_ps. L ! p ∉ committed (N p)
    by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_bin ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
      L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
      a ∉ local.broadcast ∧
      (l1, b1, g1, In a,  f1, r1, l1') ∈ trans (N p) ∧
      (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
      L!p = l1 ∧ L!q = l2 ∧ p < length L ∧ q < length L ∧ p ≠ q ∧
      check_bexp s b1 True ∧ check_bexp s b2 True ∧
      L' = L[p := l1', q := l2'] ∧ is_upds s f1 s' ∧ is_upds s' f2 s''
      ∧ L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }

    unfolding trans_bin_def by blast
  from True have **:
    bin_trans_from (L, s)
    = concat (map (λa. pairs_by_action L s (OUT ! a) (IN ! a)) bin_actions)
    unfolding bin_trans_from_def IN_def OUT_def by simp
  from ‹dom s = _› ‹L ∈ _› show ?thesis
    unfolding * **
    apply clarsimp
    unfolding pairs_by_action_def
    apply (clarsimp simp: set_map_filter Let_def)
    apply safe
    subgoal for _ s'' _ a' p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'
      apply clarsimp
      apply (inst_existentials a')
      subgoal
        apply frules
        apply simp
        apply frules_all
```

```
      unfolding check_bounded_iff by (intros; solve_triv)
    subgoal
      by (simp add: mem_bin_actions_iff; frules; simp)
    done
  subgoal
    unfolding mem_bin_actions_iff
    apply simp
    apply (erule conjE)
    apply frules
    apply elims
    apply frules_all
    apply frules_all
    apply elims
    apply intros
    using ‹bounded bounds s› unfolding check_bounded_iff[symmetric] by solve_triv+
  done
next
  case False
  with get_committed_empty_iff[of L] have ∃ p<n_ps. L ! p ∈ committed (N p)
    by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_bin ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
      L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
      a ∉ local.broadcast ∧
      (l1, b1, g1, In a,  f1, r1, l1') ∈ trans (N p) ∧
      (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
      (l1 ∈ committed (N p) ∨ l2 ∈ committed (N q)) ∧
      L!p = l1 ∧ L!q = l2 ∧ p < length L ∧ q < length L ∧ p ≠ q ∧
      check_bexp s b1 True ∧ check_bexp s b2 True ∧
      L' = L[p := l1', q := l2'] ∧ is_upds s f1 s' ∧ is_upds s' f2 s'' ∧
      L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }
    unfolding trans_bin_def
    by − (rule iffI; elims add: CollectE; intros add: CollectI; blast)
  let ?S1 =
    {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
      L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
        a ∉ local.broadcast ∧
        (l1, b1, g1, In a,  f1, r1, l1') ∈ trans (N p) ∧
        (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
        l1 ∈ committed (N p) ∧
        L!p = l1 ∧ L!q = l2 ∧ p < length L ∧ q < length L ∧ p ≠ q ∧
        check_bexp s b1 True ∧ check_bexp s b2 True ∧
        L' = L[p := l1', q := l2'] ∧ is_upds s f1 s' ∧ is_upds s' f2 s'' ∧
        L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }
  let ?S2 =
    {((L, s), g1 @ g2, Bin a, r1 @ r2, (L', s'')) |
      L s L' s' s'' a p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'.
        a ∉ local.broadcast ∧
        (l1, b1, g1, In a,  f1, r1, l1') ∈ trans (N p) ∧
        (l2, b2, g2, Out a, f2, r2, l2') ∈ trans (N q) ∧
        l2 ∈ committed (N q) ∧
        L!p = l1 ∧ L!q = l2 ∧
```

```
    p < length L ∧ q < length L ∧ p ≠ q ∧
    check_bexp s b1 True ∧ check_bexp s b2 True ∧
    L' = L[p := l1', q := l2'] ∧
    is_upds s f1 s' ∧ is_upds s' f2 s'' ∧
    L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
  }
have *: ((L, s), g, a, r, L', s') ∈ trans_bin ⟷
  ((L, s), g, a, r, L', s') ∈ ?S1 ∨ ((L, s), g, a, r, L', s') ∈ ?S2
  unfolding * by clarsimp (rule iffI; elims add: disjE; intros add: disjI1 disjI2 HOL.refl)
define pairs where pairs = get_committed L
define In2 where In2 = all_actions_from_vec trans_in_map pairs
define Out2 where Out2 = all_actions_from_vec trans_out_map pairs
have In2_I:
  (p, b, g, a', f, r, l') ∈ set (In2 ! a')
  if (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    p < n_ps a' < num_actions L ! p ∈ committed (N p)
  for p b g a' f r l'
proof −
  from ‹L ! p ∈ committed (N p)› ‹p < n_ps› have (p, L ! p) ∈ set pairs
    unfolding pairs_def get_committed_mem_iff by blast
  from trans_mapI[OF that(1,2)] have (b, g, In a', f, r, l') ∈ set (trans_map p (L ! p))
    by auto
  then have (b, g, a', f, r, l') ∈ set (trans_in_map p (L ! p))
    unfolding trans_in_map_def set_map_filter by (auto 4 7)
  with ‹p < _› ‹a' < _› ‹_ ∈ set pairs› show ?thesis
    unfolding In2_def by (intro in_all_actions_from_vecI)
qed
have Out2_I:
  (p, b, g, a', f, r, l') ∈ set (Out2 ! a')
  if (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
    p < n_ps a' < num_actions L ! p ∈ committed (N p)
  for p b g a' f r l'
proof −
  from ‹L ! p ∈ committed (N p)› ‹p < n_ps› have (p, L ! p) ∈ set pairs
    unfolding pairs_def get_committed_mem_iff by blast
  from trans_mapI[OF that(1,2)] have (b, g, Out a', f, r, l') ∈ set (trans_map p (L ! p))
    by auto
  then have (b, g, a', f, r, l') ∈ set (trans_out_map p (L ! p))
    unfolding trans_out_map_def set_map_filter by (auto 4 7)
  with ‹p < _› ‹a' < _› ‹_ ∈ set pairs› show ?thesis
    unfolding Out2_def by (intro in_all_actions_from_vecI)
qed
have distinct (map fst pairs)
  unfolding pairs_def get_committed_def distinct_map inj_on_def Let_def
  by (auto simp: set_map_filter intro!: distinct_map_filterI split: if_split_asm)
have in_pairsD: p < n_ps l = L ! p L ! p ∈ committed (N p)
  if (p, l) ∈ set pairs for p l
  using that using get_committed_mem_iff pairs_def by auto
have In2_D:
  (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p) ∧
  p < n_ps ∧ a' = a1 ∧ L ! p ∈ committed (N p)
  if (p, b, g, a', f, r, l') ∈ set (In2 ! a1) a1 < num_actions
  for p b g a' f r l' a1
  using that
```

unfolding *In2_def*
apply −
apply (*drule all_actions_from_vecD*)
apply *assumption*
apply (*rule* ‹*distinct* _›)
unfolding *trans_in_map_def set_map_filter*
apply (*clarsimp split*: *option.split_asm*)
apply (*auto dest*: *in_pairsD trans_mapD split*: *act.split_asm*)
done
have *Out2_D*:
(*L* ! *p*, *b*, *g*, *Out a′*, *f*, *r*, *l′*) ∈ *Simple_Network_Language.trans* (*N p*)
∧ *p* < *n_ps* ∧ *a′* = *a1* ∧ *L* ! *p* ∈ *committed* (*N p*)
if (*p*, *b*, *g*, *a′*, *f*, *r*, *l′*) ∈ *set* (*Out2* ! *a1*) *a1* < *num_actions*
for *p b g a′ f r l′ a1*
using *that*
unfolding *Out2_def*
apply −
apply (*drule all_actions_from_vecD*)
apply *assumption*
apply (*rule* ‹*distinct* _›)
unfolding *trans_out_map_def set_map_filter*
apply (*clarsimp split*: *option.split_asm*)
apply (*auto dest*: *in_pairsD trans_mapD split*: *act.split_asm*)
done
from *False* **have** ∗∗: *bin_trans_from* (*L*, *s*) =
concat (*map* (λ*a*. *pairs_by_action L s* (*OUT* ! *a*) (*In2* ! *a*)) *bin_actions*)
@ *concat* (*map* (λ*a*. *pairs_by_action L s* (*Out2* ! *a*) (*IN* ! *a*)) *bin_actions*)
unfolding *bin_trans_from_def IN_def OUT_def In2_def Out2_def pairs_def*
by (*simp add*: *Let_def*)
from ‹*dom s* = _› ‹*L* ∈ _› **have**
((*L*, *s*), *g*, *a*, *r*, *L′*, *s′*) ∈ *?S1* ⟷ (*g*, *a*, *r*, *L′*, *s′*) ∈
set (*concat* (*map* (λ*a*. *pairs_by_action L s* (*OUT* ! *a*) (*In2* ! *a*)) *bin_actions*))
apply *clarsimp*
unfolding *pairs_by_action_def*
apply (*clarsimp simp*: *set_map_filter Let_def*)
apply *safe*
subgoal for _ *s″* _ *a′ p q l1 b1 g1 f1 r1 l1′ l2 b2 g2 f2 r2 l2′*
apply *clarsimp*
apply (*inst_existentials a′*)
subgoal
supply [*forward4*] = *In2_I*
apply *frules_all*
apply *simp*
apply *frules_all*
unfolding *check_bounded_iff* **by** (*intros*; *solve_triv*)
subgoal
by (*simp add*: *mem_bin_actions_iff*; *frules*; *simp*)
done
subgoal
supply [*forward2*] = *In2_D OUT_D*
apply *simp*
unfolding *mem_bin_actions_iff*
apply (*erule conjE*)
apply *frules_all*

```
      apply elims
      apply frules_all
      apply elims
      apply simp
      apply frules_all
     using ‹bounded bounds s› unfolding check_bounded_iff[symmetric] by (intros; solve_triv)
    done
  moreover from ‹dom s = _› ‹L ∈ _› have
    ((L, s), g, a, r, L', s') ∈ ?S2 ⟷ (g, a, r, L', s')
    ∈ set (concat (map (λa. pairs_by_action L s (Out2 ! a) (IN ! a)) bin_actions))
    supply [forward2] = Out2_D In2_D
    supply [forward4] = Out2_I
    apply clarsimp
    unfolding pairs_by_action_def
    apply (clarsimp simp: set_map_filter Let_def)
    apply safe
    subgoal for _ s'' _ a' p q l1 b1 g1 f1 r1 l1' l2 b2 g2 f2 r2 l2'
      apply clarsimp
      apply (inst_existentials a')
      subgoal
        apply frules_all
        apply simp
        apply frules_all
        unfolding check_bounded_iff by (intros; solve_triv)
      subgoal
        unfolding mem_bin_actions_iff by frules_all simp
      done
    subgoal
      apply simp
      unfolding mem_bin_actions_iff
      apply (erule conjE)
      apply frules_all
      apply elims
      apply frules_all
      apply elims
      apply simp
      apply frules_all
     using ‹bounded bounds s› unfolding check_bounded_iff[symmetric] by (intros; solve_triv)
    done
  ultimately show ?thesis
    unfolding ∗ ∗∗ by simp
 qed
qed

Broadcast transitions  lemma make_combs_alt_def:
  make_combs p a xs ≡
    let
      ys =
        map (λ i. map (λt. (i, t)) (xs ! i ! a))
        (filter
          (λi. xs ! i ! a ≠ [] ∧ i ≠ p)
          [0..<n_ps])
    in if ys = [] then [] else product_lists ys
```

```
  apply (rule eq_reflection)
  unfolding make_combs_def
  apply (simp add: map_filter_def comp_def if_distrib[where f = the])
  apply intros
  apply (fo_rule arg_cong)
  apply auto
  done

lemma list_all2_fst_aux:
  map fst xs = ys if list_all2 (λx y. fst x = y) xs ys
  using that by (induction) auto

lemma broad_trans_from_correct:
  (broad_trans_from, trans_broad) ∈ transition_rel states'
  unfolding transition_rel_def
proof (safe del: iffI)
  fix L s g a r L' s' assume (L, s) ∈ states'
  then have L ∈ states dom s = {0..<n_vs} bounded bounds s
    by auto
  then have [simp]: length L = n_ps
    by auto
  define IN  where IN  = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps]
  define OUT where OUT = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps]
  define IN' where IN' = map (map (filter (λ (b, _). bval (the o s) b))) IN
  define OUT' where OUT' = map (map (filter (λ (b, _). bval (the o s) b))) OUT
  have IN_I:
    (b, g, a', f, r, l') ∈ set (IN ! p ! a')
    if (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
      p < n_ps a' < num_actions a' ∈ set broadcast
    for p b g a' f r l'
  proof −
    from trans_mapI[OF that(1,2)] ‹a' ∈ _› have
      (b, g, a', f, r, l') ∈ set (trans_in_broad_map p (L ! p))
      unfolding trans_in_broad_map_def set_map_filter by (auto 4 7)
    with ‹a' < _› have (b, g, a', f, r, l') ∈ set (trans_in_broad_grouped p (L ! p) ! a')
      unfolding trans_in_broad_grouped_def by (auto dest: in_actions_by_state'I)
    with ‹p < _› show ?thesis
      unfolding IN_def by auto
  qed
  have OUT_I:
    (b, g, a', f, r, l') ∈ set (OUT ! p ! a')
    if (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
      p < n_ps a' < num_actions a' ∈ set broadcast
    for p b g a' f r l'
  proof −
    from trans_mapI[OF that(1,2)] ‹a' ∈ _› have
      (b, g, a', f, r, l') ∈ set (trans_out_broad_map p (L ! p))
      unfolding trans_out_broad_map_def set_map_filter by (auto 4 7)
    with ‹a' < _› have (b, g, a', f, r, l') ∈ set (trans_out_broad_grouped p (L ! p) ! a')
      unfolding trans_out_broad_grouped_def by (auto dest: in_actions_by_state'I)
    with ‹p < _› show ?thesis
      unfolding OUT_def by auto
  qed
  have IN_D:
```

```
    (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
     ∧ a' = a1 ∧ a1 ∈ set broadcast
    if (b, g, a', f, r, l') ∈ set (IN ! p ! a1) a1 < num_actions p < n_ps
    for p b g a' f r l' a1
    using that unfolding IN_def trans_in_broad_grouped_def
    apply simp
    apply (drule in_actions_by_state'D)
    unfolding trans_in_broad_map_def set_map_filter
    by (auto split: option.split_asm) (auto split: act.split_asm if_split_asm dest: trans_mapD)
  have [simp]: length IN = n_ps length OUT = n_ps
    unfolding IN_def OUT_def by simp+
  have [simp]: length (IN ! p) = num_actions length (OUT ! p) = num_actions if p < n_ps for
p
    using that by (simp add:
    length_actions_by_state'_preserv trans_in_broad_grouped_def trans_out_broad_grouped_def
      IN_def OUT_def)+
  have OUT_D:
    (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
     ∧ a' = a1 ∧ a1 ∈ set broadcast
    if (b, g, a', f, r, l') ∈ set (OUT ! p ! a1) a1 < num_actions p < n_ps
    for p b g a' f r l' a1
    using that unfolding OUT_def trans_out_broad_grouped_def
    apply simp
    apply (drule in_actions_by_state'D)
    unfolding trans_out_broad_map_def set_map_filter
    by (auto split: option.split_asm) (auto split: act.split_asm if_split_asm dest: trans_mapD)
  have IN'_I:
    (b, g, a', f, r, l') ∈ set (IN' ! p ! a')
    if (b, g, a', f, r, l') ∈ set (IN ! p ! a') p < n_ps a' < num_actions
       check_bexp s b True for p b g a' f r l'
    using that ‹dom s = {0..<n_vs}› unfolding IN'_def
    by (auto dest!: IN_D[THEN conjunct1] elim: check_bexp_bvalI)
  have IN'_D:
    (L ! p, b, g, In a', f, r, l') ∈ Simple_Network_Language.trans (N p)
     ∧ a' = a1 ∧ a1 ∈ set broadcast ∧ check_bexp s b True
    if (b, g, a', f, r, l') ∈ set (IN' ! p ! a1) a1 < num_actions p < n_ps
    for p b g a' f r l' a1
   using that ‹dom s = {0..<n_vs}› unfolding IN'_def by (auto dest!: IN_D elim: check_bexp_bvalD)
  have OUT'_I:
    (b, g, a', f, r, l') ∈ set (OUT' ! p ! a')
    if (b, g, a', f, r, l') ∈ set (OUT ! p ! a') p < n_ps a' < num_actions
       check_bexp s b True for p b g a' f r l'
    using that ‹dom s = {0..<n_vs}› unfolding OUT'_def
    by (auto dest!: OUT_D[THEN conjunct1] elim: check_bexp_bvalI)
  have OUT'_D:
    (L ! p, b, g, Out a', f, r, l') ∈ Simple_Network_Language.trans (N p)
     ∧ a' = a1 ∧ a1 ∈ set broadcast ∧ check_bexp s b True
    if (b, g, a', f, r, l') ∈ set (OUT' ! p ! a1) a1 < num_actions p < n_ps
    for p b g a' f r l' a1
     using that ‹dom s = {0..<n_vs}› unfolding OUT'_def by (auto dest!: OUT_D elim:
check_bexp_bvalD)
  define make_trans where make_trans a p ≡
     let
       outs = OUT' ! p ! a
```

```
    in if outs = [] then []
    else
      let
        combs = make_combs p a IN';
        outs = map (λt. (p, t)) outs;
        combs = (
              if combs = [] then [[x]. x ← outs]
              else concat (map (λx. map (λxs. x # xs) combs) outs));
        init = ([], Broad a, [], (L, s))
      in
      filter (λ (g, a, r, L, s). check_bounded s) (
        map (λcomb.
            fold
              (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
                (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
              )
              comb
              init
        ) combs) for a p
{
  fix p ps bs gs a' fs rs ls'
  assume assms:
    ∀ p∈set ps.
        (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)
    ∀ q<n_ps. q ∉ set ps ∧ p ≠ q ⟶
        (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)
    p < n_ps set ps ⊆ {0..<n_ps} p ∉ set ps distinct ps sorted ps
    a' < num_actions a' ∈ set broadcast ∀ p ∈ set ps. check_bexp s (bs p) True
  define ys where ys = List.map_filter
      (λ i.
        if i = p then None
        else if IN' ! i ! a' = [] then None
        else Some (map (λt. (i, t)) (IN' ! i ! a'))
      )
      [0..<n_ps]
  have filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
    apply (rule filter_distinct_eqI)
    subgoal
      using assms(4-) by (simp add: sorted_distinct_subseq_iff)
    subgoal
      using assms(1,3-) by (auto 4 3 dest!: IN_I IN'_I)
    subgoal
      using assms(1,2,4,8-)
      apply -
      apply (rule ccontr)
      apply simp
      apply elims
      apply (drule hd_in_set)
      subgoal for x
        by (cases hd (IN' ! x ! a')) (fastforce dest!: IN'_D)
      done
    by auto
  then have length ys = length ps
    unfolding ys_def map_filter_def by simp
```

```
    have non_empty: ys ≠ [] if ps ≠ []
      using ‹_ = ps› ‹ps ≠ []› unfolding ys_def map_filter_def by simp
    have make_combsD:
      map (λp. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps ∈ set (make_combs p a' IN')
      if ps ≠ []
    proof −
      from assms(1,3−) have
        ∀ i<length ps. let p = ps ! i in (p, bs p, gs p, a', fs p, rs p, ls' p) ∈ set (ys ! i)
        unfolding ys_def Let_def map_filter_def
        apply (simp add: comp_def if_distrib[where f = the])
        apply (subst (2) map_cong)
         apply (rule HOL.refl)
        apply (simp; fail)
        apply (simp add: ‹_ = ps›)
        by (intros add: image_eqI[OF HOL.refl] IN_I IN'_I; simp add: subset_code(1))
      with non_empty[OF that] show ?thesis
        unfolding make_combs_def ys_def[symmetric] Let_def
        by (auto simp: ‹length ys = _› product_lists_set intro:list_all2_all_nthI)
    qed
    have make_combs_empty:
      make_combs p a' IN' = [] ⟷ ps = []
    proof (cases ps = [])
      case True
      then show ?thesis
       using ‹length ys = length ps› unfolding make_combs_def ys_def[symmetric] Let_def by
auto
    next
      case False
      then show ?thesis
        using make_combsD by auto
    qed
    note make_combsD make_combs_empty
  } note make_combsD = this(1) and make_combs_empty = this(2)
  have make_combs_emptyD: filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = []
    if make_combs p a' IN' = [] for p a'
    apply (rule filter_distinct_eqI)
    subgoal
      by auto
      subgoal
        by auto
      subgoal
        using that unfolding make_combs_alt_def
        by (auto simp: filter_empty_conv product_lists_empty split: if_split_asm)
      subgoal
        by simp
      done
  have make_combsI:
    ∃ ps bs gs fs rs ls'.
      (∀ p∈set ps.
       (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)) ∧
      (∀ q<n_ps. q ∉ set ps ∧ p ≠ q ⟶
       (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)) ∧
      set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧ ps ≠ [] ∧ a' ∈ set broadcast
      ∧ (∀ p ∈ set ps. check_bexp s (bs p) True)
```

```
    ∧ xs = map (λ p. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps
    ∧ filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
  if xs ∈ set (make_combs p a' IN') p < n_ps a' < num_actions
  for xs p a'
proof −
  define ps bs gs fs rs ls' where defs:
    ps  = map fst xs
    bs  = (λi. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ b)
    gs  = (λi. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ g)
    fs  = (λi. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ f)
    rs  = (λi. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ r)
    ls' = (λi. case the (map_of xs i) of (b, g, a, f, r, l') ⇒ l')
  have filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
    apply (rule filter_distinct_eqI)
    subgoal
      using that
      unfolding defs make_combs_alt_def Let_def
      by (auto simp: set_map_filter product_lists_set list_all2_map2 list.rel_eq
          dest: list_all2_map_fst_aux split: if_split_asm)
    subgoal
      using that unfolding defs make_combs_alt_def Let_def
      by (auto simp: set_map_filter product_lists_set dest!: list_all2_set1 split: if_split_asm)
    subgoal
      using that
      unfolding defs make_combs_alt_def Let_def
      by (auto
          simp: set_map_filter product_lists_set list_all2_map2 list.rel_eq
          dest!: map_eq_imageD list_all2_map_fst_aux split: if_split_asm)
    subgoal
      by simp
    done
  then have set ps ⊆ {0..<n_ps} p ∉ set ps distinct ps sorted ps
    by (auto intro: sorted_filter')
  have to_map: a' = a the (map_of xs q) = (b, g, a, r, f, l')
    if (q, b, g, a, r, f, l') ∈ set xs for b q g a r f l'
    using that
     apply −
    subgoal
      using ‹set ps ⊆ _› ‹a' < _› ‹xs ∈ _›
      by
        (simp
          add: make_combs_alt_def product_lists_set ‹_ = ps› list_all2_map2
          split: if_split_asm
          ) (auto 4 3 dest!: IN'_D list_all2_set1)
    subgoal
      using ‹distinct ps›
      by (subst (asm) map_of_eq_Some_iff[of xs q, symmetric]) (auto simp: defs)
    done
  from that have ∗: ∀ p∈set ps.
     (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ Simple_Network_Language.trans (N p)
    unfolding make_combs_alt_def ‹_ = ps›
    apply (clarsimp simp: set_map_filter product_lists_set split: if_split_asm)
    using ‹set ps ⊆ _› unfolding defs
    by (auto simp: comp_def list_all2_map2 list_all2_same to_map
```

```
      elim!: IN'_D[THEN conjunct1, rotated]
      )
  from that have ps ≠ [] a' ∈ set broadcast
  apply (simp_all add: make_combs_alt_def set_map_filter product_lists_set split: if_split_asm)
   using ‹_ = ps› ‹set ps ⊆ {0..<n_ps}›
   by (cases xs; auto dest: IN'_D simp: list_all2_Cons1)+
  with that have ∀ q<n_ps. q ∉ set ps ∧ p ≠ q ⟶
    (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)
   unfolding make_combs_alt_def
   by (auto 4 3 dest!: list_all2_set2 IN_I IN'_I
      simp: defs set_map_filter product_lists_set split: if_split_asm)
  have xs = map (λ p. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps
   apply (intro nth_equalityI)
    apply (simp add: defs; fail)
   subgoal for i
    by (cases xs ! i) (auto 4 4 simp: defs dest: to_map nth_mem)
   done
  have ∀ p∈set ps. check_bexp s (bs p) True
  proof
   fix q assume q ∈ set ps
   from ‹xs ∈ _› have distinct (map fst xs)
    unfolding make_combs_alt_def
    by (auto simp: product_lists_set list_all2_map2 to_map
          dest!: list_all2_fst_aux[OF list_all2_mono]
          split: if_split_asm)
   from ‹q ∈ set ps› ‹_ = ps› have IN' ! q ! a' ≠ [] q ≠ p q < n_ps
    by auto
   with that have the (map_of xs q) ∈ set (IN' ! q ! a')
    using set_map_of_compr[OF ‹distinct (map _ _)›] unfolding make_combs_alt_def
    apply (clarsimp simp: set_map_filter product_lists_set split: if_split_asm)
    apply (drule list_all2_set2)
    apply auto
    done
   then obtain a where (bs q, gs q, a, fs q, rs q, ls' q) ∈ set (IN' ! q ! a')
    unfolding defs by atomize_elim (auto split!: prod.splits)
   then show check_bexp s (bs q) True
    using IN'_D ‹a' < _› ‹set ps ⊆ _› ‹q ∈ set ps› by auto
  qed
  show ?thesis
   by (inst_existentials ps bs gs fs rs ls'; fact)
qed
let ?f = λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, L, s).
          (g1 @ g2, a, r1 @ r2, L[q := l2], mk_upds s f2)
have ***:
  fold ?f (map (λp. (p, bs p, gs p, a', fs p, rs p, ls' p)) ps) (g, a, r, L, s)
  = (g @ concat (map gs ps), a, r @ concat (map rs ps),
     fold (λp L. L[p := ls' p]) ps L, mk_upds s (concat_map fs ps))
   for ps bs gs a' fs rs ls' g a r L s
  by (induction ps arbitrary: g a r L s; simp add: mk_upds_def)
have upd_swap:
  (fold (λp L . L[p := ls' p]) ps L)[p := l'] = fold (λp L . L[p := ls' p]) ps (L[p := l'])
  if p ∉ set ps for ps ls' p l'
  using that by (induction ps arbitrary: L) (auto simp: list_update_swap)
have make_transI:
```

```
    a' < num_actions ∧ p < n_ps ∧
       (g1 @ concat (map gs ps), Broad a', r1 @ concat (map rs ps),
        (fold (λp L. L[p := ls' p]) ps L)[p := l'], s') ∈ set (make_trans a' p)
    if
      L ∈ states and
      g = g1 @ concat (map gs ps) and
      a = Broad a' and
      r = r1 @ concat (map rs ps) and
      L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] and
      a' ∈ set broadcast and
      (L ! p, b1, g1, Out a', f1, r1, l') ∈ trans (N p) and
      ∀ p∈set ps. (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)
      and
      ∀ q<n_ps. q ∉ set ps ∧ p ≠ q
        ⟶ (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True) and
      p < n_ps and
      set ps ⊆ {0..<n_ps} and
      p ∉ set ps and
      distinct ps and
      sorted ps and
      check_bexp s b1 True and
      ∀ p∈set ps. check_bexp s (bs p) True and
      is_upds s f1 s'' and
      is_upds s'' (concat_map fs ps) s' and
      Simple_Network_Language.bounded bounds s'
    for a' p b1 g1 bs gs ps r1 rs ls' l' f1 s'' fs
    using that ‹dom s = {0..<n_vs}›
    unfolding make_trans_def
    apply (clarsimp simp: set_map_filter Let_def split: prod.split)
    supply [forward2] = action_setD
      supply [forward4] = is_upds_dom2 is_upds_dom3 is_upds_concatD[rotated 3] OUT_I
OUT'_I
    apply frule2
    apply simp
    apply (rule conjI, rule impI)
    subgoal
      apply (subst (asm) make_combs_empty, (assumption | simp)+)
      apply frules_all
      apply (intro conjI)
        apply (solves auto)
       apply (intros add: more_intros)
         apply (solve_triv | intros add: more_intros UN_I)+
      subgoal
        unfolding comp_def by (auto elim!: is_upds.cases)
      by (auto simp: check_bounded_iff[symmetric] elim: is_upds_NilE)
    apply (rule impI)
    apply (frule make_combsD, simp, assumption+)
    subgoal
      by (subst (asm) make_combs_empty) (assumption | simp)+
    apply frules_all
    apply simp
    apply (intro conjI)
      apply (solves auto)
     apply (intros add: more_intros)
```

```
    apply (solve_triv | intros add: more_intros UN_I)+
   apply (simp add: *** upd_swap; fail)
  unfolding check_bounded_iff[symmetric] .
 have make_transD:
  ∃ s'' b ga f ra l' bs gs fs rs ls' ps.
      g = ga @ concat (map gs ps) ∧
      a = Broad a' ∧
      r = ra @ concat (map rs ps) ∧
      L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] ∧
      a' ∈ set broadcast ∧
      (L ! p, b, ga, Out a', f, ra, l') ∈ trans (N p) ∧
      (∀ p∈set ps. (L ! p, bs p, gs p, In a', fs p, rs p, ls' p) ∈ trans (N p)) ∧
      (∀ q<n_ps.
          q ∉ set ps ∧ p ≠ q ⟶
          (∀ b g f r l'. (L ! q, b, g, In a', f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)) ∧
      p < n_ps ∧
      set ps ⊆ {0..<n_ps} ∧
      p ∉ set ps ∧
      distinct ps ∧
      sorted ps ∧
      check_bexp s b True ∧ (∀ p∈set ps. check_bexp s (bs p) True) ∧
      is_upds s f s'' ∧ is_upds s'' (concat_map fs ps) s' ∧
      bounded bounds s' ∧
      filter (λi. IN' ! i ! a' ≠ [] ∧ i ≠ p) [0..<n_ps] = ps
  if
   L ∈ states and
   a' < num_actions and
   p < n_ps and
   (g, a, r, L', s') ∈ set (make_trans a' p)
  for a' p
  supply [forward2] = action_setD
  supply [forward3] = is_upds_make_updsI3[rotated] OUT'_D
   supply [forward4] = is_upds_dom2 is_upds_dom3 is_upds_concatD[rotated 3] OUT_I
OUT'_I
  using that ‹dom s = {0..<n_vs}›
  unfolding make_trans_def
  apply mini_ex
  apply (clarsimp simp: set_map_filter Let_def split: prod.split if_split_asm)
  subgoal for b a1 f l'
   apply (inst_existentials
   b :: (nat, int) bexp
   g :: (nat, int) acconstraint list
   f :: (nat × (nat, int) exp) list
   r :: nat list
   l'
   undefined :: nat ⇒ (nat, int) bexp
   undefined :: nat ⇒ (nat, int) acconstraint list
   undefined :: nat ⇒ (nat × (nat, int) exp) list
   undefined :: nat ⇒ nat list
   undefined :: nat ⇒ nat
   [] :: nat list)
             apply (solves ‹auto dest: OUT'_D›)+
   subgoal
     by (auto 4 3 simp: filter_empty_conv dest: bspec dest!: make_combs_emptyD OUT'_D
```

```
IN_I IN'_I)
          apply (solves ‹auto dest: OUT'_D›)+
      subgoal
        apply (inst_existentials s')
        subgoal is_upd
          by (auto intro: is_upds_make_updsI2 dest: OUT'_D)
        subgoal
          by simp (rule is_upds.intros)
        subgoal
          by (subst check_bounded_iff) (metis OUT'_D is_upds_dom2 is_upd)+
        subgoal
          by (rule make_combs_emptyD)
        done
      done
    subgoal for b1 g1 a1 r1 f1 l1' xs
      apply (drule make_combsI, assumption+)
      apply frules
      apply elims
      apply dedup_prems
      apply frules_all
      apply (simp add: *** )
      apply intros
                  apply solve_triv+
                 apply (erule upd_swap[symmetric]; fail)
                apply solve_triv+
               apply (erule bspec; assumption)
              apply (elims add: allE; intros?; assumption)
             apply solve_triv+
      subgoal for ps gs fs rs ls'
        apply (subst check_bounded_iff)
        subgoal
          apply (subst is_upds_dom3)
            apply (simp add: fold_map comp_def; fail)
           apply assumption+
          done
        subgoal
          by simp
        done
      apply solve_triv
      done
    done
  have make_trans_iff:
      (∃ s'' aa p b ga f ra l' bs gs fs rs ls' ps.
          g = ga @ concat (map gs ps) ∧
          a = Broad aa ∧
          r = ra @ concat (map rs ps) ∧
          L' = (fold (λp L. L[p := ls' p]) ps L)[p := l'] ∧
          aa ∈ set broadcast ∧
          (L ! p, b, ga, Out aa, f, ra, l') ∈ Simple_Network_Language.trans (N p) ∧
          (∀ p∈set ps.
              (L ! p, bs p, gs p, In aa, fs p, rs p, ls' p)
              ∈ Simple_Network_Language.trans (N p)) ∧
          (∀ q<n_ps.
              q ∉ set ps ∧ p ≠ q ⟶
```

```
          (∀ b g f r l'.
              (L ! q, b, g, In aa, f, r, l') ∉ trans (N q) ∨ ¬ check_bexp s b True)) ∧
        p < n_ps ∧
        set ps ⊆ {0..<n_ps} ∧
        p ∉ set ps ∧
        distinct ps ∧
        sorted ps ∧
        check_bexp s b True ∧ (∀ p∈set ps. check_bexp s (bs p) True) ∧
        is_upds s f s'' ∧
        is_upds s'' (concat_map fs ps) s' ∧
        bounded bounds s') =
    (∃ a'∈{0..<num_actions}.
        ∃ p∈{0..<n_ps}. (g, a, r, L', s') ∈ set (make_trans a' p)) (is ?l ⟷ ?r)
  if dom s = {0..<n_vs} L ∈ states
proof (intro iffI)
  assume ?l
  with that show ?r
    by elims (drule make_transI; (elims; intros)?; solve_triv)
next
  assume ?r
  with that show ?l
    apply elims
    subgoal for a' p
      apply simp
      apply (drule make_transD)
         apply assumption+
      apply elims
      apply intros
                    apply assumption+
      apply blast+
      done
    done
qed
show (((L, s), g, a, r, L', s') ∈ trans_broad) =
      ((g, a, r, L', s') ∈ set (broad_trans_from (L, s)))
proof (cases get_committed L = [])
  case True
  with get_committed_empty_iff[of L] have ∀ p<n_ps. L ! p ∉ committed (N p)
    by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_broad ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L', s'')) |
      L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
      a ∈ set broadcast ∧
      (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
      (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
      (∀ q < n_ps. q ∉ set ps ∧ p ≠ q ⟶
        ¬ (∃ b g f r l'. (L ! q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
      L!p = l ∧
      p < length L ∧ set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
      check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
      L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'] ∧
      is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
      L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }
```

```
    unfolding trans_broad_def broadcast_def[simplified]
    by (intro iffI; elims add: CollectE; intros add: CollectI) blast+
  from True have **:
    broad_trans_from (L, s)
    =  concat (
        map (λa.
          concat (map (λp. make_trans a p) [0..<n_ps])
        )
        [0..<num_actions]
      )
  unfolding broad_trans_from_alt_def IN_def OUT_def IN'_def OUT'_def Let_def make_trans_def
    by simp
  from ‹dom s = _› ‹L ∈ _› ‹bounded bounds s› show ?thesis
    unfolding * **
    apply simp
    apply (subst make_trans_iff[symmetric])
     apply simp+
    apply (intro iffI; elims; intros)
                    apply (solve_triv | blast)+
    done
next
  case False
  with get_committed_empty_iff[of L] have ¬ (∀ p<n_ps. L ! p ∉ committed (N p))
    by simp
  then have *: ((L, s), g, a, r, L', s') ∈ trans_broad ⟷ ((L, s), g, a, r, L', s') ∈
    {((L, s), g @ concat (map gs ps), Broad a, r @ concat (map rs ps), (L', s'')) |
      L s L' s' s'' a p l b g f r l' bs gs fs rs ls' ps.
      a ∈ set broadcast ∧
      (l, b, g, Out a, f, r, l') ∈ trans (N p) ∧
      (∀ p ∈ set ps. (L ! p, bs p, gs p, In a, fs p, rs p, ls' p) ∈ trans (N p)) ∧
      (l ∈ committed (N p) ∨ (∃ p ∈ set ps. L ! p ∈ committed (N p))) ∧
      (∀ q < n_ps. q ∉ set ps ∧ p ≠ q ⟶
        ¬ (∃ b g f r l'. (L ! q, b, g, In a, f, r, l') ∈ trans (N q) ∧ check_bexp s b True)) ∧
      L!p = l ∧
      p < length L ∧ set ps ⊆ {0..<n_ps} ∧ p ∉ set ps ∧ distinct ps ∧ sorted ps ∧
      check_bexp s b True ∧ (∀ p ∈ set ps. check_bexp s (bs p) True) ∧
      L' = (fold (λp L . L[p := ls' p]) ps L)[p := l'] ∧
      is_upds s f s' ∧ is_upds s' (concat_map fs ps) s'' ∧
      L ∈ states ∧ bounded bounds s ∧ bounded bounds s''
    }
    unfolding trans_broad_def broadcast_def[simplified]
    by (intro iffI; elims add: CollectE; intros add: CollectI) blast+
  have committed_iff:
    List.map_filter (λ(p, _). if IN' ! p ! a' = [] then None else Some p) (get_committed L) ≠
[p] ∧
    List.map_filter (λ(p, _). if IN' ! p ! a' = [] then None else Some p) (get_committed L) ≠ []
  ⟷ (∃ q<n_ps. IN' ! q ! a' ≠ [] ∧ q ≠ p ∧ L ! q ∈ committed (N q))
    for p a'
  proof −
    have *: xs ≠ [p] ∧ xs ≠ [] ⟷ (∃ x ∈ set xs. x ≠ p) if distinct xs for xs
      using that by auto (metis distinct.simps(2) distinct_length_2_or_more revg.elims)
    show ?thesis
      by (subst *)
```

```
      (auto
        intro: distinct_map_filterI get_committed_distinct
        simp: set_map_filter get_committed_mem_iff split: if_split_asm
        )
  qed
  from False have **:
    broad_trans_from (L, s)
    = concat (
      map (λa.
        let
          ins_committed =
           List.map_filter (λ(p, _). if IN' ! p ! a ≠ [] then Some p else None) (get_committed L)
        in
        concat (map (λp.
          if
            (ins_committed = [p] ∨ ins_committed = [])
            ∧ ¬ list_ex (λ (q, _). q = p) (get_committed L)
          then []
          else
            make_trans a p
        )
        [0..<n_ps])
      )
    [0..<num_actions])

  unfolding broad_trans_from_alt_def IN_def OUT_def IN'_def OUT'_def make_trans_def
    unfolding Let_def if_contract
    apply simp
     apply (fo_rule if_cong arg_cong2[where f = map] arg_cong[where f = concat] | rule
ext)+
       apply blast+
    done
  from ‹dom s = _› ‹L ∈ _› ‹bounded bounds s› show ?thesis
    unfolding * **
    apply (simp add: make_trans_iff)
    apply (intro iffI; elims)
    subgoal for s'a aa p ga f ra l' gs fs rs ls' ps — ?l ⟶ ?r
      apply (frule make_transI)
                    apply (assumption | blast)+
      apply elims
      apply intros
       apply (simp; fail)
      apply (simp add: Let_def)
      apply (erule disjE)
      subgoal — The process with the outgoing action label is committed
        using get_committed_mem_iff[of p L ! p L, simplified, symmetric]
        by (inst_existentials p) (auto simp add: list_ex_iff)
      apply (erule bexE)
      subgoal for q — One of the processes with an ingoing action label is committed
        apply (inst_existentials p)
         apply assumption
        apply (rule IntI)
         apply (simp; fail)
        apply simp
```

```
        unfolding committed_iff
        apply (rule disjI1; inst_existentials q; force dest!: IN_I IN'_I)
        done
      done
    subgoal for a' — ?r ⟶ ?l
      apply (clarsimp split: if_split_asm simp: Let_def)
      apply (drule make_transD[rotated 4])
         apply assumption+
      apply elims
      apply intros
                    apply assumption+
               apply (erule bspec; assumption)
      subgoal for p s'' g' f r' l' gs fs rs ls' ps
        unfolding committed_iff by (auto simp: get_committed_mem_iff list_ex_iff)
      apply blast+
      done
    done
  qed
qed
```

Refinement of the State Implementation **definition** *state_rel* :: (*nat* ⇀ *int*) ⇒ *int list* ⇒ *bool*
  where
  state_rel s xs ≡ *length xs* = *n_vs* ∧ *dom s* = {*0*..<*n_vs*} ∧ (∀ *i* < *n_vs*. *xs* ! *i* = *the* (*s i*))

definition *loc_rel* **where**
  loc_rel ≡ {((*L'*, *s'*), (*L*, *s*)) | *L s L' s'*. *L'* = *L* ∧ *length L* = *n_ps* ∧ *state_rel s s'*}

lemma *state_impl_abstract*:
  ∃ *L s*. ((*Li*, *si*), (*L*, *s*)) ∈ *loc_rel* **if** *length Li* = *n_ps length si* = *n_vs*
  using *that* **unfolding** *loc_rel_def state_rel_def*
  by (*inst_existentials Li* λ*i*. *if i* < *n_vs then Some* (*si* ! *i*) *else None*)(*auto split*: *if_split_asm*)

lemma *state_rel_left_unique*:
  l ∈ *states'* ⟹ (*li*, *l*) ∈ *loc_rel* ⟹ (*li'*, *l*) ∈ *loc_rel* ⟹ *li'* = *li*
  unfolding *loc_rel_def state_rel_def* **by** (*auto intro*: *nth_equalityI*)

lemma *state_rel_right_unique*:
  l ∈ *states'* ⟹ *l'* ∈ *states'* ⟹ (*li*, *l*) ∈ *loc_rel* ⟹ (*li*, *l'*) ∈ *loc_rel* ⟹ *l'* = *l*
  unfolding *loc_rel_def state_rel_def*
  apply *clarsimp*
  apply (*rule ext*)
  subgoal premises *prems* **for** *L s s'1 s'2 x*
  proof −
    show *s'1 x* = *s x*
    proof (*cases x* < *n_vs*)
      case *True*
      then have *x* ∈ *dom s'1 x* ∈ *dom s*
        using *prems* **by** *auto*
      with ‹*x* < *n_vs*› **show** *?thesis*
        using *prems*(*9*) **by** *auto*
    next
      case *False*

```
      then have x ∉ dom s'1 x ∉ dom s
        using prems by auto
      then show ?thesis
        by (auto simp: dom_def)
    qed
  qed
  done

end

end

fun bvali :: _ ⇒ (nat, 'b :: linorder) bexp ⇒ bool and evali where
  bvali s bexp.true = True |
  bvali s (not e) ⟷ ¬ bvali s e |
  bvali s (and e1 e2) ⟷ bvali s e1 ∧ bvali s e2 |
  bvali s (bexp.or e1 e2) ⟷ bvali s e1 ∨ bvali s e2 |
  bvali s (imply e1 e2) ⟷ bvali s e1 ⟶ bvali s e2 |
  bvali s (eq i x) ⟷ evali s i = evali s x |
  bvali s (le i x) ⟷ evali s i ≤ evali s x |
  bvali s (lt i x) ⟷ evali s i < evali s x |
  bvali s (ge i x) ⟷ evali s i ≥ evali s x |
  bvali s (gt i x) ⟷ 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 ⇒ (nat × (nat, int) exp) list ⇒ int list where
  mk_updsi s upds = fold (λ(x, upd) s. s[x := evali s upd]) upds s

context Simple_Network_Impl_nat_defs
begin

definition
  check_boundedi s =
    (∀ x < length s. fst (bounds_map x) ≤ s ! x ∧ s ! x ≤ snd (bounds_map x))

definition
  states'_memi ≡ λ(L, s). L ∈ states ∧ length s = n_vs ∧ check_boundedi s

definition
  int_trans_from_loc_impl 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_updsi s f in
        if bvali s b ∧ check_boundedi s' then Some (g, Internal a, r, (L[p := l'], s'))
        else None
    ) trans

definition
```

```
  int_trans_from_vec_impl pairs L s ≡
    concat (map (λ(p, l). int_trans_from_loc_impl p l L s) pairs)

definition
  int_trans_from_all_impl L s ≡
    concat (map (λp. int_trans_from_loc_impl p (L ! p) L s) [0..<n_ps])

definition
  int_trans_impl ≡ λ (L, s).
    let pairs = get_committed L in
    if pairs = []
    then int_trans_from_all_impl L s
    else int_trans_from_vec_impl pairs L s


definition
  pairs_by_action_impl L s OUT IN ≡
  concat (
    map (λ (p, b1, g1, a1, f1, r1, l1).
      List.map_filter (λ (q, b2, g2, a2, f2, r2, l2).
        if p = q then None else
          let s' = mk_updsi (mk_updsi s f1) f2 in
          if bvali s b1 ∧ bvali s b2 ∧ check_boundedi s'
          then Some (g1 @ g2, Bin a1, r1 @ r2, (L[p := l1, q := l2], s'))
          else None
    ) OUT) IN)


definition
  bin_trans_from_impl ≡ λ(L, s).
    let
      pairs = get_committed L;
      In =  all_actions_by_state trans_in_map L;
      Out = all_actions_by_state trans_out_map L
    in
    if pairs = [] then
      concat (map (λa. pairs_by_action_impl L s (Out ! a) (In ! a)) bin_actions)
    else
      let
        In2  = all_actions_from_vec trans_in_map pairs;
        Out2 = all_actions_from_vec trans_out_map pairs
      in
        concat (map (λa. pairs_by_action_impl L s (Out ! a) (In2 ! a)) bin_actions)
      @ concat (map (λa. pairs_by_action_impl L s (Out2 ! a) (In ! a)) bin_actions)


definition
  compute_upds init ≡ List.map_filter (λcomb.
  let (g, a, r, L', s) =
    fold
      (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
        (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_upds s f2))
      )
      comb
```

```
    init
  in if check_bounded s then Some (g, a, r, L', s) else None
)

definition
  compute_upds_impl init ≡ List.map_filter (λcomb.
  let (g, a, r, L', s) =
    fold
      (λ(q, b2, g2, a2, f2, r2, l2) (g1, a, r1, (L, s)).
        (g1 @ g2, a, r1 @ r2, (L[q := l2], mk_updsi s f2))
      )
      comb
      init
  in if check_boundedi s then Some (g, a, r, L', s) else None
)

definition trans_from where
  trans_from st = int_trans_from st @ bin_trans_from st @ broad_trans_from st

definition
  broad_trans_from_impl ≡ λ(L, s).
    let
      pairs = get_committed L;
      In  = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
      Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps];
      In  = map (map (filter (λ(b, _). bvali s b))) In;
      Out = map (map (filter (λ(b, _). bvali s b))) Out
    in
    if pairs = [] then
      concat (
        map (λa.
          concat (map (λp.
            let
              outs = Out ! p ! a
            in if outs = [] then []
            else
              let
                combs = make_combs p a In;
                outs = map (λt. (p, t)) outs;
                combs = (
                  if combs = [] then [[x]. x ← outs]
                  else concat (map (λx. map (λxs. x # xs) combs) outs));
                init = ([], Broad a, [], (L, s))
              in
                compute_upds_impl init combs
          )
          [0..<n_ps])
        )
      [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
          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_bvali:
  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
  by (induction b and e) (auto simp: state_rel_def)

lemma mk_upds_mk_updsi:
  state_rel (mk_upds s upds) (mk_updsi si upds)
  if assms: state_rel s si ∀ (_, e) ∈ set upds. ∀ x ∈ vars_of_exp e. x < n_vs
    ∀ (x, e) ∈ set upds. x < n_vs
proof −
  have upd_stepI: state_rel (mk_upd (x, e) s') (si'[x := evali si' e])
    if state_rel s' si' ∀ x ∈ vars_of_exp e. x < n_vs x < n_vs
    for s' si' x e
   using that assms unfolding mk_upd_def state_rel_def by (auto simp: state_rel_def eval_evali)
  from assms show ?thesis
  proof (induction upds arbitrary: s si)
    case Nil
```

```
    then show ?case
      by (simp add: mk_upds_def mk_updsi_def)
  next
    case (Cons upd upds)
    then show ?case
     by (simp add: mk_upds_def mk_updsi_def split: prod.splits) (rprem, auto intro!: upd_stepI)
  qed
qed

lemma check_bounded_check_boundedi:
  check_bounded s = check_boundedi si if state_rel s si
  using that unfolding check_bounded_def check_boundedi_def state_rel_def by auto

definition
  valid_upd ≡ λ(x, e). x < n_vs ∧ (∀ x ∈ vars_of_exp e. x < n_vs)

definition
  valid_check b ≡ (∀ x ∈ vars_of_bexp b. x < n_vs)

context includes lifting_syntax begin
notation rel_prod (infixr ×_R 56)

definition is_at_least_equality where
  is_at_least_equality R ≡ ∀ x y. R x y ⟶ x = y for R

named_theorems is_at_least_equality

lemma [is_at_least_equality]:
  is_at_least_equality (=)
  by (simp add: is_at_least_equality_def)

lemma [is_at_least_equality]:
  is_at_least_equality R if is_equality R for R
  using that by (simp add: is_at_least_equality_def is_equality_def)

lemma [is_at_least_equality]:
  is_at_least_equality (eq_onp P)
  by (simp add: is_at_least_equality_def eq_onp_def)

lemma is_at_least_equality_list_all2[is_at_least_equality]:
  is_at_least_equality (list_all2 R) if is_at_least_equality R for R
  using that unfolding is_at_least_equality_def
  by (auto simp: list.rel_eq dest: list_all2_mono[where Q = (=)])

lemma is_at_least_equality_rel_prod[is_at_least_equality]:
  is_at_least_equality (R1 ×_R R2)
  if is_at_least_equality R1 is_at_least_equality R2 for R1 R2
  using that unfolding is_at_least_equality_def by auto

lemma is_at_least_equality_cong1:
  (S1 ===> (=)) f f if is_at_least_equality S1 is_at_least_equality S2 for S1 f
  using that unfolding is_at_least_equality_def by (intro rel_funI) auto

lemma is_at_least_equality_cong2:
```

(*S1* ===> *S2* ===> (=)) *f f* **if** *is_at_least_equality S1 is_at_least_equality S2* **for** *S1 S2 f*
using *that* **unfolding** *is_at_least_equality_def* **by** (*intro rel_funI*) *auto*

lemma *is_at_least_equality_cong3*:
(*S1* ===> *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*
using *that* **unfolding** *is_at_least_equality_def* **by** (*intro rel_funI*) *force*

lemma *is_at_least_equality_Let*:
(*S* ===> ((=) ===> *R*) ===> *R*) *Let Let* **if** *is_at_least_equality S* **for** *R*
using *that* **unfolding** *is_at_least_equality_def*
by (*intro rel_funI*) (*erule Let_transfer*[*THEN rel_funD*, *THEN rel_funD*, *rotated*], *auto*)

lemma *map_transfer_length*:
fixes *R S n*
shows
((*R* ===> *S*)
===> (λ*x y*. *list_all2 R x y* ∧ *length x* = *n*)
===> (λ*x y*. *list_all2 S x y* ∧ *length x* = *n*))
map map
apply (*intro rel_funI conjI*)
apply (*erule list.map_transfer*[*THEN rel_funD*, *THEN rel_funD*], *erule conjunct1*)
apply *simp*
done

lemma *upt_0_transfer*:
(*eq_onp* (λ*x*. *x* = *0*) ===> *eq_onp* (λ*x*. *x* = *n*) ===> *list_all2* (*eq_onp* (λ*x*. *x* < *n*))) *upt upt* **for** *n*
apply (*intro rel_funI upt_transfer_upper_bound*[*THEN rel_funD*, *THEN rel_funD*])
apply (*assumption* | *erule eq_onp_to_eq*)+
done

lemma *upt_length_transfer*:
(*eq_onp* (λ*x*. *x* = *0*) ===> *eq_onp* (λ*x*. *x* = *n*)
===> (λ *x y*. *list_all2* (*eq_onp* (λ*x*. *x* < *n*)) *x y* ∧ *length x* = *n*)) *upt upt* **for** *n*
apply (*intro rel_funI conjI upt_0_transfer*[*THEN rel_funD*, *THEN rel_funD*], *assumption*+)
apply (*simp add*: *eq_onp_def*)
done

lemma *case_prod_transfer_strong*:
fixes *A B C*
assumes ⋀ *x y*. *A1 x y* ⟹ *A x y* ⋀ *x y*. *B1 x y* ⟹ *B x y*
shows ((*A* ===> *B* ===> *C*) ===> *A1* ×$_R$ *B1* ===> *C*) *case_prod case_prod*
apply (*intro rel_funI*)
apply *clarsimp*
apply (*drule assms*)+
apply (*drule* (*1*) *rel_funD*)+
apply *assumption*
done

lemma *concat_transfer_strong*:
fixes *A B C*
assumes ⋀*x y*. *A x y* ⟹ *B x y* ⋀ *x y*. *C x y* ⟹ *list_all2* (*list_all2 A*) *x y*
shows (*C* ===> *list_all2 B*) *concat concat*

apply (*intro rel_funI concat_transfer*[*THEN rel_funD*])
apply (*drule assms*)
apply (*erule list_all2_mono*)
apply (*erule list_all2_mono*)
apply (*erule assms*)
done

lemma *map_transfer_strong*:
fixes *A B C*
assumes $\bigwedge$*xs ys*. *C xs ys* $\Longrightarrow$ *list_all2 A xs ys*
shows ((*A* ===> *B*) ===> *C* ===> *list_all2 B*) *map map*
apply (*intro rel_funI*)
apply (*erule list.map_transfer*[*THEN rel_funD*, *THEN rel_funD*])
apply (*erule assms*)
done

lemma *list_update_transfer'*:
fixes *A* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *bool*
shows (*list_all2 A* ===> *eq_onp* (λ*i*. *i*<*n_ps*) ===> *A* ===> *list_all2 A*) *list_update list_update*
apply (*intro rel_funI*)
apply (*rule list_update_transfer*[*THEN rel_funD*, *THEN rel_funD*, *THEN rel_funD*])
apply (*auto simp*: *eq_onp_def*)
done

lemma *list_update_transfer''*:
fixes *A* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *bool* **and** *n*
shows ((λ *x y*. *list_all2 A x y* $\wedge$ *length x* = *n*) ===> *eq_onp* (λ*i*. *i*<*n*) ===> *A* ===> (λ *x y*. *list_all2 A x y* $\wedge$ *length x* = *n*)) *list_update list_update*
apply (*intro rel_funI conjI*)
subgoal
apply (*erule conjE*)
apply (*rule List.list_update_transfer*[*THEN rel_funD*, *THEN rel_funD*, *THEN rel_funD*])
apply (*assumption* | *elim eq_onp_to_eq*)+
done
apply *simp*
done

lemma *list_update_transfer'''*:
fixes *A* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *bool* **and** *n*
shows ((λ *x y*. *list_all2 A x y* $\wedge$ *length x* = *n*) ===> (=) ===> *A* ===> (λ *x y*. *list_all2 A x y* $\wedge$ *length x* = *n*)) *list_update list_update*
apply (*intro rel_funI conjI*)
subgoal
apply (*erule conjE*)
apply (*rule List.list_update_transfer*[*THEN rel_funD*, *THEN rel_funD*, *THEN rel_funD*])
apply (*assumption* | *elim eq_onp_to_eq*)+
done
apply *simp*
done

lemma *fold_transfer_strong*:
fixes *A B*
assumes $\bigwedge$*x y*. *A1 x y* $\Longrightarrow$ *A x y* $\bigwedge$*x y*. *B1 x y* $\Longrightarrow$ *B x y* $\bigwedge$*x y*. *B x y* $\Longrightarrow$ *B2 x y*

```
    ⋀x y. B x y ⟹ B3 x y
  shows ((A ===> B2 ===> B1) ===> list_all2 A1 ===> B ===> B3) fold fold
  apply (intro rel_funI, rule assms)
  apply (rule fold_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
    apply (intro rel_funI)
    apply (drule rel_funD, erule assms)
    apply (drule rel_funD, erule assms, erule assms)
   apply assumption+
  done

lemma bval_bvali_transfer[transfer_rule]:
  (state_rel ===> eq_onp valid_check ===> (=)) (λ s. bval (the o s)) bvali
  by (intro rel_funI) (auto simp: eq_onp_def valid_check_def state_rel_def intro!: bval_bvali)

lemma mk_upds_mk_updsi_transfer[transfer_rule]:
  (state_rel ===> list_all2 (eq_onp valid_upd) ===> state_rel) mk_upds mk_updsi
  apply (intro rel_funI)
  subgoal for x y upds upds'
    apply (subgoal_tac upds' = upds)
     apply simp
     apply (rule mk_upds_mk_updsi)
       apply assumption
    subgoal
      by (smt case_prodI2 case_prod_conv eq_onp_def list_all2_same valid_upd_def)
    subgoal
      by (smt case_prodE case_prod_conv eq_onp_def list_all2_same valid_upd_def)
    subgoal
      by (metis eq_onp_to_eq list.rel_eq_onp)
    done
  done

lemma check_bounded_transfer[transfer_rule]:
  (state_rel ===> (=)) check_bounded check_boundedi
  by (simp add: check_bounded_check_boundedi rel_funI)

lemma trans_map_transfer:
  (eq_onp (λi. i<n_ps) ===> (=) ===>
    list_all2 (
      eq_onp valid_check ×R (=) ×R eq_onp (pred_act (λx. x < num_actions))
      ×R list_all2 (eq_onp valid_upd) ×R (=)
   )) trans_map trans_map
  apply (intro rel_funI, simp add: eq_onp_def, intro list.rel_refl_strong)
  apply clarsimp
 apply (auto 4 4 dest!: trans_mapD dest: action_setD var_setD var_setD2 intro: list.rel_refl_strong
              simp: valid_upd_def valid_check_def
       )
  done

lemma trans_map_transfer':
  (eq_onp (λi. i<n_ps) ===> (=) ===>
    list_all2 (eq_onp valid_check ×R (=) ×R (=) ×R list_all2 (eq_onp valid_upd) ×R (=))
  ) trans_map trans_map
  apply (intro rel_funI, simp add: eq_onp_def, intro list.rel_refl_strong)
  apply clarsimp
```

apply (*intro conjI list.rel_refl_strong*)
apply (*auto 4 4 dest*: *var_setD trans_mapD var_setD2 simp*: *valid_upd_def valid_check_def*)
done

lemma *map_filter_transfer*[*transfer_rule*]:
((*S* ===> *rel_option R*) ===> *list_all2 S* ===> *list_all2 R*) *List.map_filter List.map_filter*
unfolding *map_filter_def*
apply (*clarsimp intro*!: *rel_funI*)
subgoal for *f g xs ys*
apply (*rule list.map_transfer*[*THEN rel_funD*, *THEN rel_funD*, *of* λ *x y*. *f x* ≠ *None* ∧ *S x y*])
apply (*rule rel_funI*)
subgoal for *a b*
apply (*cases f a*)
apply (*auto simp*: *option_rel_Some1 option_rel_Some2 dest*!: *rel_funD*)
done
subgoal
apply *rotate_tac*
apply (*induction rule*: *list_all2_induct*)
apply (*auto dest*: *rel_funD*)
done
done
done

lemma *trans_i_map_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===>
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ (=) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))
) *trans_i_map trans_i_map*
supply [*transfer_rule*] = *trans_map_transfer*′
unfolding *trans_i_map_def* **by** *transfer_prover*

lemma *int_trans_from_loc_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===> (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ===> *state_rel*
===> *list_all2*((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*))
int_trans_from_loc int_trans_from_loc_impl
supply [*transfer_rule*] = *list_update_transfer*′′
unfolding *int_trans_from_loc_def int_trans_from_loc_impl_def Let_def* **by** *transfer_prover*

lemma *n_ps_transfer*:
eq_onp (λ*x*. *x* = *n_ps*) *n_ps n_ps*
by (*simp add*: *eq_onp_def*)

lemma *zero_nat_transfer*:
(=) *0* (*0*::*nat*)
..

lemma *int_trans_from_all_transfer*[*transfer_rule*]:
((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ===> *state_rel*
===> *list_all2*((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*))
int_trans_from_all int_trans_from_all_impl
supply [*transfer_rule*] = *zero_nat_transfer n_ps_transfer*

unfolding *int_trans_from_all_def int_trans_from_all_impl_def Let_def*
by *transfer_prover*

lemma *int_trans_from_vec_transfer*[*transfer_rule*]:
(*list_all2* (*eq_onp* (λ*x*. *x* < *n_ps*) ×$_R$ (=)) ===> (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*)
===> *state_rel*
===> *list_all2*((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*))
int_trans_from_vec int_trans_from_vec_impl
unfolding *int_trans_from_vec_def int_trans_from_vec_impl_def Let_def*
by *transfer_prover*

private definition *R* **where** *R* ≡ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*)

lemma *get_committed_transfer*[*transfer_rule*]:
((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ===> *list_all2* (*eq_onp* (λ*x*. *x* < *n_ps*) ×$_R$ (=)))
get_committed get_committed
proof −
have [*transfer_rule*]:
R automata automata
unfolding *R_def* **by** (*simp add*: *n_ps_def list.rel_eq*)
show *?thesis*
supply [*transfer_rule*] = *zero_nat_transfer n_ps_transfer*
unfolding *get_committed_def*
unfolding *Let_def*
apply *transfer_prover_start*
using [[*goals_limit*=*15*]]
prefer *8*
apply *transfer_step*
prefer *8*
unfolding *R_def*
apply *transfer_step*+
apply *transfer_prover*
done
qed

lemma *eq_transfer*:
(*list_all2* (*eq_onp* (λ*x*. *x* < *n_ps*) ×$_R$ (=)) ===> *list_all2* (=) ===> (=)) (=) (=)
unfolding *eq_onp_def*
apply (*intro rel_funI*)
apply (*drule list_all2_mono*[**where** *Q* = (=)])
apply (*auto simp add*: *list.rel_eq rel_prod.simps*)
done

lemma *int_trans_from_transfer*:
((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*
===> *list_all2* ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ ((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*)))
int_trans_from int_trans_impl
supply [*transfer_rule*] = *eq_transfer*
unfolding *int_trans_impl_def int_trans_from_def Let_def*
by *transfer_prover*

lemma *pairs_by_action_transfer*[*transfer_rule*]:
 ((λ*x 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 ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n*) ×$_R$ *state_rel*))
 pairs_by_action pairs_by_action_impl
 supply [*transfer_rule*] = *list_update_transfer'''*
 unfolding *pairs_by_action_def pairs_by_action_impl_def* **by** *transfer_prover*

lemmas *rel_elims* =
 rel_prod.cases
 rel_funD

lemmas *rel_intros* =
 rel_funI

lemma *pairs_by_action_transfer'*:
 ((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n*) ===> *state_rel* ===>
  list_all2 (*B* ×$_R$ *eq_onp valid_check* ×$_R$ *C* ×$_R$ *D* ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ *E* ×$_R$ *F*) ===>
  list_all2 (*B* ×$_R$ *eq_onp valid_check* ×$_R$ *C* ×$_R$ *D* ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ *E* ×$_R$ *F*) ===>
  list_all2 ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n*) ×$_R$ *state_rel*))
 pairs_by_action pairs_by_action_impl
 if ⋀*x y*. *B x y* ⟹ *x* = *y*
  ⋀*x y*. *C x y* ⟹ *x* = *y* ⋀*x y*. *D x y* ⟹ *x* = *y*
  ⋀*x y*. *E x y* ⟹ *x* = *y* ⋀*x y*. *F x y* ⟹ *x* = *y*
 for *B C D E F*
 apply (*intro rel_funI*)
 apply (*rule pairs_by_action_transfer*[*THEN rel_funD*, *THEN rel_funD*, *THEN rel_funD*, *THEN rel_funD*])
  apply (*assumption* | *erule that list_all2_mono prod.rel_mono_strong*)+
 done

lemma *trans_in_map_transfer*[*transfer_rule*]:
 (*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=)
  ===> *list_all2* (
   eq_onp valid_check ×$_R$ (=) ×$_R$ *eq_onp* (λ*a*. *a* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))
) *trans_in_map trans_in_map*
 supply [*transfer_rule*] = *trans_map_transfer*[*folded act.rel_eq_onp*]
 unfolding *trans_in_map_def* **by** *transfer_prover*

lemma *trans_in_map_transfer*[*transfer_rule*]:
 (*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=)
  ===> *list_all2* (*eq_onp valid_check* ×$_R$ (=) ×$_R$ (=) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))
) *trans_in_map trans_in_map*
 unfolding *trans_in_map_def* **oops**

lemma *trans_out_map_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=)
===> *list_all2* (
eq_onp valid_check ×$_R$ (=) ×$_R$ *eq_onp* (λ*a*. *a* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)
)) *trans_out_map trans_out_map*
supply [*transfer_rule*] = *trans_map_transfer*[*folded act.rel_eq_onp*]
unfolding *trans_out_map_def* **by** *transfer_prover*

lemma *trans_out_map_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===> *list_all2* (
eq_onp valid_check ×$_R$ (=) ×$_R$ (=) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)))
trans_out_map trans_out_map
unfolding *trans_out_map_def* **oops**

lemma *actions_by_state_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i* < *n_ps*) ===>
list_all2 ((=) ×$_R$ (=) ×$_R$ *eq_onp* (λ*i*. *i* < *n*) ×$_R$ (=)) ===>
(λ *x y*. *list_all2* (*list_all2* (*eq_onp* (λ*i*. *i*<*n_ps*) ×$_R$ (=) ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *n*) ×$_R$ (=))) *x y* ∧ *length x* = *n*) ===>
(λ *x y*. *list_all2* (*list_all2* (*eq_onp* (λ*i*. *i*<*n_ps*) ×$_R$ (=) ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *n*) ×$_R$ (=))) *x y* ∧ *length x* = *n*)
)
actions_by_state actions_by_state **for** *n*
supply [*transfer_rule*] = *list_update_transfer''*
unfolding *actions_by_state_def* **by** *transfer_prover*

lemma *actions_by_state_transfer'*[*transfer_rule*]:
(
eq_onp (λ*i*. *i* < *n_ps*) ===>
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*i*. *i* < *n*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)) ===>
(λ *x y*. *list_all2* (*list_all2* (
eq_onp (λ*i*. *i*<*n_ps*) ×$_R$ *eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *n*)
×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))) *x y* ∧ *length x* = *n*) ===>
(λ *x y*. *list_all2* (*list_all2* (
eq_onp (λ*i*. *i*<*n_ps*) ×$_R$ *eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *n*)
×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))) *x y* ∧ *length x* = *n*)
)
actions_by_state actions_by_state
for *n*
supply [*transfer_rule*] = *list_update_transfer''*
unfolding *actions_by_state_def* **by** *transfer_prover*

lemma *transfer_consts*:
(*eq_onp* (λ*x*. *x* = *num_actions*)) *num_actions num_actions* (*eq_onp* (λ*x*. *x* = *0*)) (*0*::*nat*) *0*
(*eq_onp* (λ*x*. *x* = *n_ps*)) *n_ps n_ps*
by (*auto simp*: *eq_onp_def*)

lemma *all_actions_by_state_transfer*[*transfer_rule*]:

(
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===> *list_all2* (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*i*. *i* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)))

```
    ===>
    (λx y. list_all2 (=) x y ∧ length x = n_ps)
    ===>
    (λ x y. list_all2 (list_all2 (eq_onp (λi. i<n_ps) ×R eq_onp valid_check ×R (=) ×R eq_onp
(λi. i<num_actions) ×R list_all2 (eq_onp valid_upd) ×R (=))) x y ∧ length x = num_actions)
  )
  all_actions_by_state all_actions_by_state
 supply [transfer_rule] = map_transfer_length upt_0_transfer upt_length_transfer transfer_consts
    n_ps_transfer
  unfolding all_actions_by_state_def by transfer_prover

lemma all_actions_from_vec_transfer[transfer_rule]:

  (
    (eq_onp (λi. i<n_ps) ===> (=) ===> list_all2 (eq_onp valid_check ×R (=) ×R eq_onp
(λi. i < num_actions) ×R list_all2 (eq_onp valid_upd) ×R (=)))
    ===>
    list_all2 (eq_onp (λi. i < n_ps) ×R (=))
    ===>
    (λ x y. list_all2 (list_all2 (eq_onp (λi. i<n_ps) ×R eq_onp valid_check ×R (=) ×R eq_onp
(λi. i<num_actions) ×R list_all2 (eq_onp valid_upd) ×R (=))) x y ∧ length x = num_actions)
  )
  all_actions_from_vec all_actions_from_vec
  supply [transfer_rule] = map_transfer_length upt_length_transfer transfer_consts
  unfolding all_actions_from_vec_def all_actions_from_vec_def by transfer_prover

lemma bin_actions_transfer[transfer_rule]:
  (list_all2 (eq_onp (λx. x < num_actions))) bin_actions bin_actions
proof −
  have ∗: list_all2 (eq_onp (λx. x < num_actions)) [0..<num_actions] [0..<num_actions]
    by (rule list.rel_refl_strong) (auto simp: eq_onp_def)
  show ?thesis
    unfolding bin_actions_def
   by (rule filter_transfer[THEN rel_funD, THEN rel_funD, OF _ ∗]) (auto simp: eq_onp_def)
qed

lemma Let_transfer_bin_aux:
  ((λx y. list_all2 (list_all2
    (eq_onp (λi. i < n_ps) ×R eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R
          eq_onp (λi. i < num_actions) ×R list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R
(=))) x y
    ∧ length x = num_actions) ===>
   ((λx y. list_all2 (list_all2
    ((=) ×R eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R
        (=) ×R list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))) x y
    ∧ length x = num_actions) ===>
   list_all2
    (list_all2 (rel_acconstraint (=) (=)) ×R rel_label (=) ×R
     list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel)) ===>
  list_all2
    (list_all2 (rel_acconstraint (=) (=)) ×R rel_label (=) ×R
     list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel))
  Let Let
  unfolding Let_def
```

by (*intro rel_funI*, *drule rel_funD*)
(*auto simp*: *eq_onp_def elim*!: *list_all2_mono prod.rel_mono_strong*)

lemma *bin_trans_from_transfer*:
((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*
===> *list_all2* ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ ((λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*) ×$_R$ *state_rel*)))
bin_trans_from bin_trans_from_impl
unfolding *bin_trans_from_impl_def bin_trans_from_def*
supply [*transfer_rule*] =
map_transfer_length upt_length_transfer transfer_consts eq_transfer Let_transfer_bin_aux
by *transfer_prover*

lemma *trans_map_transfer''*:
((=) ===> (=) ===>
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (*pred_act* (λ*x*. *x* < *num_actions*)) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)))
trans_map trans_map
apply (*intro rel_funI*, *simp add*: *eq_onp_def*, *intro list.rel_refl_strong*)
apply *clarsimp*
apply (*auto 4 4 dest*!: *trans_mapD dest*: *action_setD var_setD intro*: *list.rel_refl_strong simp*: *valid_upd_def*)
oops

lemma *compute_upds_transfer*:
(
(*list_all2* (=) ×$_R$ (=) ×$_R$ *list_all2* (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n*) ×$_R$ *state_rel*)
===> *list_all2* (*list_all2*
((=) ×$_R$ *eq_onp valid_check* ×$_R$ *list_all2* (=) ×$_R$ (=) ×$_R$ *list_all2* (*eq_onp valid_upd*)
×$_R$ *list_all2* (=) ×$_R$ (=))) ===>
list_all2 (
list_all2 (=) ×$_R$ (=) ×$_R$ *list_all2* (=) ×$_R$ (λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n*) ×$_R$ *state_rel*
)) *compute_upds compute_upds_impl*
supply [*transfer_rule*] = *list_update_transfer'''*
unfolding *compute_upds_def compute_upds_impl_def* **by** *transfer_prover*

lemma *in_transfer*:
(*eq_onp* (λ*x*. *x* < *num_actions*) ===> (=) ===> (=)) (∈) (∈)
by (*intro rel_funI*, *rule member_transfer*[*of* (=), *THEN rel_funD*, *THEN rel_funD*])
(*auto simp*: *eq_onp_def rel_set_eq intro*: *bi_unique_eq*)

lemma *trans_in_broad_map_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===> *list_all2* (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)))
trans_in_broad_map trans_in_broad_map
supply [*transfer_rule*] = *trans_map_transfer*[*folded act.rel_eq_onp*] *in_transfer*
unfolding *trans_in_broad_map_def* **by** *transfer_prover*

lemma *trans_out_broad_map_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=) ===> *list_all2* (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)))
trans_out_broad_map trans_out_broad_map
supply [*transfer_rule*] = *trans_map_transfer*[*folded act.rel_eq_onp*] *in_transfer*
unfolding *trans_out_broad_map_def* **by** *transfer_prover*

We are using the "equality version" of parametricty for (!) here.

lemma *actions_by_state′_transfer*[*transfer_rule*]:
(*list_all2* (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))
===> (λ *x y*. *list_all2* (
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))) *x y*
∧ *length x* = *num_actions*
))
actions_by_state′ actions_by_state′
supply [*transfer_rule*] = *transfer_consts*
upt_length_transfer
map_transfer_length
list_update_transfer″
unfolding *actions_by_state′_def* **by** *transfer_prover*

lemma *trans_in_broad_grouped_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=)
===> (λ *x y*. *list_all2* (
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))) *x y*
∧ *length x* = *num_actions*
)) *trans_in_broad_grouped trans_in_broad_grouped*
unfolding *trans_in_broad_grouped_def* **by** *transfer_prover*

lemma *trans_out_broad_grouped_transfer*[*transfer_rule*]:
(*eq_onp* (λ*i*. *i*<*n_ps*) ===> (=)
===> (λ *x y*. *list_all2* (
list_all2 (*eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*x*. *x* < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))) *x y*
∧ *length x* = *num_actions*
)) *trans_out_broad_grouped trans_out_broad_grouped*
unfolding *trans_out_broad_grouped_def* **by** *transfer_prover*

lemma *make_combs_transfer*:
fixes *R*
assumes ⋀*x y*. *R x y* ⟹ *x* = *y*
shows
(*eq_onp* (λ*x*. *x* < *n_ps*)
===> *eq_onp* (λ*x*. *x* < *num_actions*)
===> (λ*x y*. *list_all2* (λ*x y*. *list_all2* (*list_all2 R*) *x y* ∧ *length x* = *num_actions*) *x y*
∧ *length x* = *n_ps*)
===> *list_all2* (*list_all2* (*eq_onp* (λ*x*. *x* < *n_ps*) ×$_R$ *R*)))
make_combs make_combs
proof −
have [*transfer_rule*]:
(*eq_onp* (λ*x*. *x* < *n_ps*) ===> *eq_onp* (λ*x*. *x* < *n_ps*) ===> (=)) (=) (=)

```
    (list_all2 R ===> list_all2 R ===> (=)) (=) (=)
    (list_all2 (list_all2 (eq_onp (λx. x < n_ps) ×_R R))
      ===> list_all2 (list_all2 (eq_onp (λx. x < n_ps) ×_R R)) ===> (=)) (=) (=)
    apply (simp_all add: eq_onp_def)
    subgoal
      by (smt assms list.rel_eq list_all2_mono rel_funI)
    subgoal
      by (smt assms fun.rel_eq list_all2_eq list_all2_mono rel_fun_mono rel_prod.cases)
    subgoal
      by (smt assms fun.rel_eq list_all2_eq list_all2_mono rel_fun_mono rel_prod.cases)
    done
  show ?thesis
    supply [transfer_rule] = upt_0_transfer transfer_consts
    unfolding make_combs_def by transfer_prover
qed

lemma broad_trans_from_alt_def2:
  broad_trans_from = (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 ∘ s) b))) In;
      Out = map (map (filter (λ(b, _). bval (the ∘ 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 init combs
          )
          [0..<n_ps])
        )
      [0..<num_actions])
    else
      concat (
        map (λa.
          let
            ins_committed =
              List.map_filter (λ(p, _). if In ! p ! a ≠ [] then Some p else None) pairs;
            always_committed = (length ins_committed > 1)
          in
```

```
        concat (map (λp.
          let
            outs = Out ! p ! a
          in if outs = [] then []
          else if
            ¬ always_committed ∧ (ins_committed = [p] ∨ ins_committed = [])
            ∧ ¬ list_ex (λ (q, _). q = p) pairs
          then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λx. map (λxs. x # xs) combs) outs));
              init = ([], Broad a, [], (L, s))
            in
              compute_upds init combs
        )
        [0..<n_ps])
      )
    [0..<num_actions])
  )

  unfolding broad_trans_from_def compute_upds_def
  apply (rule HOL.refl)
  done

lemma concat_length_transfer:
  ((λ x y. list_all2 (list_all2 A) x y ∧ length x = n) ===> list_all2 A) concat concat for A n
  by (intro rel_funI concat_transfer[THEN rel_funD], elim conjunct1)

lemma broad_trans_from_transfer:
  ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel
  ===> list_all2 ((=) ×R (=) ×R (=) ×R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R
state_rel)))
  broad_trans_from broad_trans_from_impl
proof −

  have compute_upds_impl_transfer[transfer_rule]:
    (list_all2 (=) ×R
      rel_label (eq_onp (λx. x < num_actions)) ×R
      list_all2 (=) ×R
      (λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel ===>
      list_all2
       (list_all2
         (eq_onp (λx. x < n_ps) ×R
          eq_onp valid_check ×R
          list_all2 (rel_acconstraint (=) (=)) ×R
          eq_onp (λx. x < num_actions) ×R
          list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))) ===>
      list_all2
       (list_all2 (=) ×R
        (=) ×R
```

```
      list_all2 (=) ×R (λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel))
    compute_upds compute_upds_impl
   apply (intro rel_funI)
   apply (rule compute_upds_transfer[THEN rel_funD, THEN rel_funD])
    apply (elim rel_prod.cases)
    apply (simp only:)
    apply (intro rel_prod.intros)
       apply assumption
   subgoal
     by (drule label.rel_mono_strong[of _ _ _ (=)]; simp add: eq_onp_def label.rel_eq)
      apply (simp; fail)+
   apply (elim list_all2_mono rel_prod.cases)
   apply (simp only:)
   apply (intro rel_prod.intros)
        apply (assumption | simp add: eq_onp_def acconstraint.rel_eq)+
   done

  have [is_at_least_equality]: is_equality (rel_acconstraint (=) (=))
    by (tactic ‹Transfer.eq_tac @{context} 1›)


  have eq_transfer1:
    (list_all2
        (eq_onp valid_check ×R  list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
        list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=)) ===>
    list_all2
    (eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x < num_actions)
×R
        list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))
    ===> (=)) (=) (=)

    by (intro is_at_least_equality_cong2 is_at_least_equality)

  let ?R = (eq_onp (λx. x < n_ps) ×R eq_onp valid_check ×R
        list_all2 (rel_acconstraint (=) (=)) ×R
         eq_onp (λx. x < num_actions) ×R list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R
(=))

  have eq_transfer5:
    (list_all2 (list_all2 ?R) ===> list_all2 (list_all2 ?R) ===> (=)) (=) (=)
    by (intro is_at_least_equality_cong2 is_at_least_equality)

  have eq_transfer2:
    (list_all2 (eq_onp (λx. x < n_ps)) ===> list_all2 (eq_onp (λx. x < n_ps)) ===> (=))
(=) (=)
    by (intro is_at_least_equality_cong2 is_at_least_equality)

  have eq_transfer3:
    (eq_onp (λx. x < n_ps) ===> eq_onp (λx. x < n_ps) ===> (=)) (=) (=)
    by (intro is_at_least_equality_cong2 is_at_least_equality)

  have eq_transfer4:
    (list_all2 (eq_onp (λx. x < n_ps) ×R (=)) ===>
```

```
    list_all2 (eq_onp (λx. x < n_ps) ×R (=)) ===> (=)) (=) (=)
   by (intro is_at_least_equality_cong2 is_at_least_equality)

  have make_combs_transfer:
   (eq_onp (λx. x < n_ps) ===>
      eq_onp (λx. x < num_actions) ===>
      (λx y. list_all2 (λx y. list_all2 (list_all2
          (eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
         list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=)))
        x y ∧ length x = num_actions) x y ∧ length x = n_ps) ===>
      list_all2 (list_all2 (eq_onp (λx. x < n_ps) ×R
          eq_onp valid_check ×R list_all2 (rel_acconstraint (=) (=)) ×R eq_onp (λx. x <
num_actions) ×R
        list_all2 (eq_onp valid_upd) ×R list_all2 (=) ×R (=))
      )
   ) make_combs make_combs
   apply (intro rel_funI)
   apply (rule make_combs_transfer[THEN rel_funD, THEN rel_funD, THEN rel_funD])
   subgoal
     apply (simp add: acconstraint.rel_eq list.rel_eq prod.rel_eq)
     apply (drule prod.rel_mono_strong[of _ _ _ _ (=) (=)], erule eq_onp_to_eq)
      apply (drule prod.rel_mono_strong[of _ _ _ _ (=) (=)])
        apply assumption
       apply (drule prod.rel_mono_strong[of _ _ _ _ (=) (=)], erule eq_onp_to_eq)
        apply (drule prod.rel_mono_strong[of _ _ _ _ (=) (=)])
         apply (drule list_all2_mono[of _ _ _ (=)], erule eq_onp_to_eq, simp only: list.rel_eq)
         apply assumption
        apply (drule (2) prod.rel_mono_strong[of _ _ _ _ (=) (=)];
              simp add: acconstraint.rel_eq list.rel_eq prod.rel_eq)+
     done
     apply (simp add: prod.rel_eq)+
   done
  have
  ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel
    ===> list_all2 (
        (=) ×R (=) ×R (=) ×R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel)))
  (
  λ(L, s).
     let
       pairs = [];
       In  = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
       Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps]
     in
       concat (
         map (λa.
           concat (map (λp.
             let
               outs = Out ! p ! a
             in if outs = [] then []
             else
               let
                 combs = make_combs p a In;
                 outs = map (λt. (p, t)) outs;
```

```
            combs = (
              if combs = [] then [[x]. x ← outs]
              else concat (map (λx. map (λxs. x # xs) combs) outs));
            init = ([], Broad a, [], (L, s))
          in
            compute_upds init combs
        )
        [0..<n_ps])
      )
    [0..<num_actions])
) (
λ(L, s).
  let
    pairs = [];
    In  = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
    Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps]
  in
    concat (
      map (λa.
        concat (map (λp.
          let
            outs = Out ! p ! a
          in if outs = [] then []
          else
            let
              combs = make_combs p a In;
              outs = map (λt. (p, t)) outs;
              combs = (
                if combs = [] then [[x]. x ← outs]
                else concat (map (λx. map (λxs. x # xs) combs) outs));
              init = ([], Broad a, [], (L, s))
            in
              compute_upds_impl init combs
        )
        [0..<n_ps])
      )
    [0..<num_actions]))

  supply [transfer_rule] =
    transfer_consts
    upt_0_transfer
    map_transfer_length
    upt_length_transfer
    make_combs_transfer
    compute_upds_transfer
    concat_length_transfer
    eq_transfer1
    eq_transfer2
    eq_transfer3
    eq_transfer5
  unfolding Let_def by transfer_prover
have
((λx y. list_all2 (=) x y ∧ length x = n_ps) ×R state_rel
  ===> list_all2 (
```

```
      (=) ×_R (=) ×_R (=) ×_R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R state_rel)))
(
λ(L, s).
    let
      pairs = get_committed L;
      In  = map (λp. trans_in_broad_grouped p (L ! p)) [0..<n_ps];
      Out = map (λp. trans_out_broad_grouped p (L ! p)) [0..<n_ps]
    in
  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 (λxs. map (λx. x # xs) combs) outs));
               init = ([], Broad a, [], (L, s))
             in
               compute_upds init combs
         )
         [0..<n_ps])
       )
     [0..<num_actions])
)
(
λ(L, s).
    let
      pairs = 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
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 (λxs. map (λx. x # xs) combs) outs));
              init = ([], Broad a, [], (L, s))
            in
              compute_upds_impl init combs
        )
        [0..<n_ps])
      )
    [0..<num_actions])
)
  supply [transfer_rule] =
    transfer_consts
    upt_0_transfer
    map_transfer_length
    upt_length_transfer
    make_combs_transfer
    compute_upds_transfer
    concat_length_transfer
    eq_transfer1
    eq_transfer2
    eq_transfer3
    eq_transfer5
  unfolding Let_def by transfer_prover

 show ?thesis
  supply [transfer_rule] =
    transfer_consts
    upt_0_transfer
    map_transfer_length
    upt_length_transfer
    make_combs_transfer
    compute_upds_transfer
    concat_length_transfer
    eq_transfer1
    eq_transfer2
    eq_transfer3
    eq_transfer4
    eq_transfer5
  unfolding broad_trans_from_alt_def2 broad_trans_from_impl_def Let_def by transfer_prover
qed

lemma trans_from_transfer:
  ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R state_rel
```

```
  ===> list_all2 ((=) ×_R (=) ×_R (=) ×_R ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R
state_rel)))
  trans_from trans_impl
 supply [transfer_rule] = int_trans_from_transfer bin_trans_from_transfer broad_trans_from_transfer
  unfolding trans_from_def trans_impl_def by transfer_prover

lemma list_all2_swap:
  list_all2 S ys xs if list_all2 R xs ys S = (λx y. R y x) for R S
  using list_all2_swap that by blast

lemma swap_rel_prod: (λ x y. (R ×_R S) y x) = (λx y. R y x) ×_R (λx y. S y x) for R S
  by (intro ext) auto

lemma swap_eq:
  (λx y. y = x) = (=)
  by auto

lemma trans_from_refine:
  (trans_impl, trans_from) ∈ fun_rel_syn loc_rel (list_rel (Id ×_r Id ×_r Id ×_r loc_rel))
proof −
  have [rel2p]:
    rel2p loc_rel = (λ x y. ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×_R state_rel) y x)
    unfolding loc_rel_def rel2p_def by (intro ext) (auto simp: list.rel_eq)
  have rel2p (fun_rel_syn
      {((L', s'), L, s) |L s L' s'. L' = L ∧ length L = n_ps ∧ state_rel s s'}
      (⟨Id ×_r Id ×_r Id ×_r loc_rel⟩list_rel))
    trans_impl trans_from
    unfolding rel2p rel2p_def state_rel_def
    by (intro rel_funI trans_from_transfer[THEN rel_funD, THEN list_all2_swap])
       (auto simp: list.rel_eq state_rel_def swap_rel_prod swap_eq)
  then show ?thesis
    unfolding rel2p_def relAPP_def loc_rel_def .
qed

lemma trans_from_correct:
  (trans_from, trans_prod) ∈ transition_rel states'
  using int_trans_from_correct bin_trans_from_correct broad_trans_from_correct
  unfolding trans_from_def trans_prod_def transition_rel_def by auto

lemma states'_alt_def:
  states' = {(L, s). L ∈ states ∧ bounded bounds s}
  unfolding states'_def bounded_def dom_bounds_eq by auto

lemma states'_alt_def2:
  states' = {(L, s). L ∈ states ∧ dom s = {0..<n_vs} ∧ check_bounded s}
proof −
  have states' = {(L, s). L ∈ states ∧ dom s = {0..<n_vs} ∧ bounded bounds s}
    unfolding states'_def bounded_def dom_bounds_eq by auto
  then show ?thesis
    by (auto simp: check_bounded_iff)
qed

lemma trans_prod_states'_inv:
  l' ∈ states' if (l, g, a, r, l') ∈ trans_prod l ∈ states'
```

using *that* **unfolding** *states′_alt_def*
by (*cases l′*) (*auto dest*: *trans_prod_bounded_inv trans_prod_states_inv*)

lemma *prod_ta_states′_inv*:
l′ ∈ *states′* **if** *prod_ta* ⊢ *l* $\longrightarrow^{g,a,r}$ *l′ l* ∈ *states′*
using *that* **by** *simp* (*rule trans_prod_states′_inv*)

lemma *dom_eq_transfer* [*transfer_rule*]:
(*state_rel* ===> (=)) (λ*s*. *dom s* = {*0*..<*n_vs*}) (λ*s*. *length s* = *n_vs*)
by (*rule rel_funI*) (*auto simp*: *state_rel_def*)

lemma *states′_memi_correct*:
(*states′_memi*, (λ*l*. *l* ∈ *states′*)) ∈ *loc_rel* → *bool_rel*
proof −
define *t* **where** *t s* ≡ *dom s* = {*0*..<*n_vs*} **for** *s* :: *nat* ⇀ *int*
define *ti* **where** *ti s* ≡ *length s* = *n_vs* **for** *s* :: *int list*
let *?R* = λ*x y*. (*eq_onp* (λ*L*. *length L* = *n_ps*) ×$_R$ *state_rel*) *y x*
note [*transfer_rule*] = *dom_eq_transfer*[*folded t_def ti_def*]
have [*p2rel*]: *p2rel* ((λ*x y*. (*eq_onp* (λ*L*. *length L* = *n_ps*) ×$_R$ *state_rel*) *y x*)) = *loc_rel*
by (*auto simp*: *eq_onp_def p2rel_def loc_rel_def*)
have ∗: (λ(*L*, *s*). *L* ∈ *states* ∧ *dom s* = {*0*..<*n_vs*} ∧ *check_bounded s*) = (λ*l*. *l* ∈ *states′*)
by (*intro ext*) (*auto simp*: *states′_alt_def2*)
have (((=) ×$_R$ *state_rel*) ===> (=))
(λ(*L*, *s*). *L* ∈ *states* ∧ *t s* ∧ *check_bounded s*) (λ(*L*, *s*). *L* ∈ *states* ∧ *ti s* ∧ *check_boundedi*
s)
by *transfer_prover*
then have
(((=) ×$_R$ *state_rel*) ===> (=))
(λ(*L*, *s*). *L* ∈ *states* ∧ *dom s* = {*0*..<*n_vs*} ∧ *check_bounded s*)
states′_memi
unfolding *t_def ti_def states′_memi_def* **.**
then have [*p2rel*]: (*?R* ===> (=)) *states′_memi* (λ*l*. *l* ∈ *states′*)
unfolding ∗ **by** (*intro rel_funI*) (*auto simp*: *eq_onp_def elim*!: *rel_funE*)
then have (*states′_memi*, (λ*l*. *l* ∈ *states′*)) ∈ *p2rel* (*?R* ===> (=))
unfolding *p2rel_def* **by** *simp*
then show *?thesis*
unfolding *p2rel* **.**
qed

end

end

end
theory *Simple_Network_Language_Model_Checking*
imports *Munta_Base.Temporal_Logics*
Simple_Network_Language_Impl_Refine
Munta_Model_Checker.UPPAAL_Model_Checking
begin

7 Product Bisimulation

no_notation *State_Networks.step_sn* (_ ⊢ ⟨_, _, _⟩ →_ ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*] *61*)

We have proved the necessary theorems already but we need to lift it to the case where delay and action transitions are strictly alternating.

inductive *step_u'* ::
  ('*a*, '*s*, '*c*, '*t* :: *time*, '*x*, '*v* :: *linorder*) *nta* ⇒ '*s list* ⇒ ('*x* ⇀ '*v*) ⇒ ('*c*, '*t*) *cval*
  ⇒ '*s list* ⇒ ('*x* ⇀ '*v*) ⇒ ('*c*, '*t*) *cval* ⇒ *bool*
(_ ⊢ ⟨_, _, _⟩ → ⟨_, _, _⟩ [*61*,*61*,*61*,*61*] *61*) **where**
  A ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L''*, *s''*, *u''*⟩ **if**
  A ⊢ ⟨*L*, *s*, *u*⟩ →$_{Del}$ ⟨*L'*, *s'*, *u'*⟩
  a ≠ *Del*
  A ⊢ ⟨*L'*, *s'*, *u'*⟩ →$_{a}$ ⟨*L''*, *s''*, *u''*⟩

inductive_cases *step'_elims*: *A* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩

inductive_cases *step_u'_elims*: *A* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩

theorem *Bisimulation_Invariant_strong_intro*:
  fixes *A* :: '*a* ⇒ '*a* ⇒ *bool*
    and *P* :: '*a* ⇒ *bool*
    and *B* :: '*a* ⇒ '*a* ⇒ *bool*
  assumes ⋀*a b*. *A a b* ⟹ *P a* ⟹ *B a b*
    and ⋀*a b*. *B a b* ⟹ *P a* ⟹ *A a b*
    and ⋀*a b*. *P a* ⟹ *A a b* ⟹ *P b*
  shows *Bisimulation_Invariant A B* (=) *P P*
  apply *standard*
  subgoal *A* **for** *a b a'*
    by (*blast intro*: *assms*)
  subgoal *B* **for** *a b a'*
    by (*blast intro*: *assms*)
  subgoal *C* **for** *a b*
    by (*rule assms*)
  subgoal *D* **for** *a b*
    by (*rule C*, *assumption*) (*rule assms*)
  done

context *Prod_TA_Defs*
begin

definition
  all_prop L s = (*L* ∈ *states* ∧ *bounded bounds s*)

lemma *all_prop_boundedD*[*dest*]:
  bounded bounds s **if** *all_prop L s*
  using *that* **unfolding** *all_prop_def* **..**

lemma *all_prop_statesD*[*dest*]:
  L ∈ *states* **if** *all_prop L s*
  using *that* **unfolding** *all_prop_def* **..**

end

context *Prod_TA*
begin

lemma *prod_action_step_not_eq_delay*:
a ≠ *Del* **if** *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →*a* ⟨(*L'*, *s'*), *u'*⟩
using *that*
apply *cases*
unfolding *trans_of_def*
unfolding *prod_ta_def trans_prod_def*
by (*auto simp*: *trans_int_def trans_bin_def trans_broad_def*)

end

locale *Prod_TA_urge* =
Prod_TA + **assumes** *urgency_removed*: ∀ *N* ∈ *set* (*fst* (*snd A*)). *urgent N* = {}
begin

lemma *prod_all_prop_inv*:
all_prop L' s' **if** *all_prop L s prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩
using *that* **unfolding** *all_prop_def*
by (*auto elim*: *bounded_inv states_inv simp*: *step_iff*[*symmetric*, *OF* _ _ *urgency_removed*])

lemma *prod_all_prop_inv'*:
all_prop L' s' **if** *all_prop L s prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩
using *that* **by** (*auto intro*: *prod_all_prop_inv elim*!: *step'_elims*)

interpretation *prod_bisim*:
Bisimulation_Invariant
(λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩)
(λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *A* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(=)
(λ (*L*, *s*, *u*). *all_prop L s*)
(λ (*L*, *s*, *u*). *all_prop L s*)
apply (*rule Bisimulation_Invariant_strong_intro*; *clarsimp*)
subgoal
by (*auto intro*: *step_u'.intros simp*: *all_prop_def*
dest: *action_sound prod_action_step_not_eq_delay delay_sound*[*OF* _ _ _ *urgency_removed*]
elim!: *step'_elims*)
subgoal
by (*auto 4 4 dest*: *prod_all_prop_inv action_complete elim*: *delay_complete elim*!: *step_u'_elims*)
subgoal
by (*rule prod_all_prop_inv'*)
done

lemmas *prod_bisim_intro* = *prod_bisim.Bisimulation_Invariant_axioms*

end

8 The Final Semantics

State formulas

datatype ('*n*, '*s*, '*a*, '*b*) *sexp* =
true |

```
  — Boolean connectives
  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 ⇒ 's list ⇒ ('a ⇒ 'b) ⇒ bool where
  check_sexp sexp.true _ _ ⟷ True |
  check_sexp (not e) L s ⟷ ¬ check_sexp e L s |
  check_sexp (and e1 e2) L s ⟷ check_sexp e1 L s ∧ check_sexp e2 L s |
  check_sexp (sexp.or e1 e2) L s ⟷ check_sexp e1 L s ∨ check_sexp e2 L s |
  check_sexp (imply e1 e2) L s ⟷ check_sexp e1 L s ⟶ check_sexp e2 L s |
  check_sexp (eq i x) L s ⟷ s i = x |
  check_sexp (le i x) L s ⟷ s i ≤ x |
  check_sexp (lt i x) L s ⟷ s i < x |
  check_sexp (ge i x) L s ⟷ s i ≥ x |
  check_sexp (gt i x) L s ⟷ s i > x |
  check_sexp (loc i x) L s ⟷ L ! i = x

fun locs_of_sexp :: ('n, 's, 'a, 'b) sexp ⇒ 'n set where
  locs_of_sexp (not e) = locs_of_sexp e |
  locs_of_sexp (and e1 e2) = locs_of_sexp e1 ∪ locs_of_sexp e2 |
  locs_of_sexp (sexp.or e1 e2) = locs_of_sexp e1 ∪ locs_of_sexp e2 |
  locs_of_sexp (imply e1 e2) = locs_of_sexp e1 ∪ locs_of_sexp e2 |
  locs_of_sexp (loc i x) = {i} |
  locs_of_sexp _ = {}

fun vars_of_sexp :: ('n, 's, 'a, 'b) sexp ⇒ 'a set where
  vars_of_sexp (not e) = vars_of_sexp e |
  vars_of_sexp (and e1 e2) = vars_of_sexp e1 ∪ vars_of_sexp e2 |
  vars_of_sexp (sexp.or e1 e2) = vars_of_sexp e1 ∪ vars_of_sexp e2 |
  vars_of_sexp (imply e1 e2) = vars_of_sexp e1 ∪ 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 ⇒ 'n set where
  locs_of_formula (formula.EX φ) = locs_of_sexp φ |
  locs_of_formula (EG φ) = locs_of_sexp φ |
  locs_of_formula (AX φ) = locs_of_sexp φ |
```

```
  locs_of_formula (AG φ) = locs_of_sexp φ |
  locs_of_formula (Leadsto φ ψ) = locs_of_sexp φ ∪ locs_of_sexp ψ

fun vars_of_formula :: ('n, 's, 'a, 'b) formula ⇒ 'a set where
  vars_of_formula (formula.EX φ) = vars_of_sexp φ |
  vars_of_formula (EG φ) = vars_of_sexp φ |
  vars_of_formula (AX φ) = vars_of_sexp φ |
  vars_of_formula (AG φ) = vars_of_sexp φ |
  vars_of_formula (Leadsto φ ψ) = vars_of_sexp φ ∪ vars_of_sexp ψ

fun hd_of_formula :: (nat, 's, 'a, 'b) formula ⇒ 's list ⇒ ('a ⇒ 'b :: linorder) ⇒ bool where
  hd_of_formula (formula.EX φ) L s = check_sexp φ L s |
  hd_of_formula (EG φ) L s = check_sexp φ L s |
  hd_of_formula (AX φ) L s = Not (check_sexp φ L s) |
  hd_of_formula (AG φ) L s = Not (check_sexp φ L s) |
  hd_of_formula (Leadsto φ _) L s = check_sexp φ L s

definition models (_,_ ⊨ _ [61,61] 61) where
  A,a0 ⊨ Φ ≡ (case Φ of
    formula.EX φ ⇒
      Graph_Defs.Ex_ev
        (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
        (λ (L, s, _). check_sexp φ L (the o s))
  | formula.EG φ ⇒
      Graph_Defs.Ex_alw
        (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
        (λ (L, s, _). check_sexp φ L (the o s))
  | formula.AX φ ⇒
      Graph_Defs.Alw_ev
        (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
        (λ (L, s, _). check_sexp φ L (the o s))
  | formula.AG φ ⇒
      Graph_Defs.Alw_alw
        (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
        (λ (L, s, _). check_sexp φ L (the o s))
  | formula.Leadsto φ ψ ⇒
      Graph_Defs.leadsto
        (λ (L, s, u) (L', s', u'). A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
        (λ (L, s, _). check_sexp φ L (the o s))
        (λ (L, s, _). check_sexp ψ 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 φ = PropC (λ(L, s, _). check_sexp φ 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 φ ψ) =
  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* $\lambda\ (L, s, u)\ (L', s', u').\ A \vdash \langle L, s, u\rangle \to \langle L', s', u'\rangle$ **.**

lemma *models_ctl_iff*:
$A,a_0 \models \Phi \longleftrightarrow$ *models_ctl* (*ctl_of* Φ) a_0
by (*cases* Φ) (*simp_all add*: *models_def prop_of_def leadsto_def*)

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* φ *L s* |
hd_of_formulai (*EG* φ) *L s* = *check_sexpi* φ *L s* |
hd_of_formulai (*AX* φ) *L s* = *Not* (*check_sexpi* φ *L s*) |
hd_of_formulai (*AG* φ) *L s* = *Not* (*check_sexpi* φ *L s*) |
hd_of_formulai (*Leadsto* φ _) *L s* = *check_sexpi* φ *L s*

lemma *all_prop_weak_cong*[*cong*]:
Prod_TA_Defs.all_prop A = *Prod_TA_Defs.all_prop A* **for** *A*
by(*rule HOL.refl*)
lemma *equiv'_weak_cong*[*cong*]:
Bisimulation_Invariant.equiv' R P Q = *Bisimulation_Invariant.equiv' R P Q* **for** *R P Q*
by (*rule HOL.refl*)
lemma *equiv'_weak_cong2*[*cong*]:
Simulation_Invariant.equiv' R P Q = *Simulation_Invariant.equiv' R P Q* **for** *R P Q*
by (*rule HOL.refl*)
lemma *models_ctl_weak_cong*[*cong*]:
Graph_Defs.models_ctl E = *Graph_Defs.models_ctl E* **for** *E*
by (*rule HOL.refl*)
lemma *models_path_weak_cong*[*cong*]:
Graph_Defs.models_path E = *Graph_Defs.models_path E* **for** *E*
by (*rule HOL.refl*)
lemma *models_state_weak_cong*[*cong*]:
Graph_Defs.models_state E = *Graph_Defs.models_state E* **for** *E*
by (*rule HOL.refl*)

lemma *models_ltl_weak_cong*[*cong*]:
Graph_Defs.models_ltlc E = *Graph_Defs.models_ltlc E* **for** *E*
by (*rule HOL.refl*)
lemma *models_weak_cong*[*cong*]:
models E = *models E*
for *E* **by** (*rule HOL.refl*)

9 Instantiating the Model Checking Locale

locale *Simple_Network_Impl_nat_urge* =
Simple_Network_Impl_nat + **assumes** *no_urgency*: ∀ (_, *U*, _, _) ∈ *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 L_0 :: *nat list*
and s_0 :: (*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*:
∀ *i* < *n_ps*. ∀ (*l*, *g*) ∈ *set* ((*snd o snd o snd*) (*automata* ! *i*)).
∀ (*x*, *m*) ∈ *collect_clock_pairs g*. *m* ≤ *int* (*k* ! *i* ! *l* ! *x*)
∀ *i* < *n_ps*. ∀ (*l*, _, *g*, _) ∈ *set* ((*fst o snd o snd*) (*automata* ! *i*)).
(∀ (*x*, *m*) ∈ *collect_clock_pairs g*. *m* ≤ *int* (*k* ! *i* ! *l* ! *x*))
and *k_resets*:
∀ *i* < *n_ps*. ∀ (*l*, *b*, *g*, *a*, *upd*, *r*, *l'*) ∈ *set* ((*fst o snd o snd*) (*automata* ! *i*)).
∀ *c* ∈ {*0*..<*m+1*} − *set r*. *k* ! *i* ! *l'* ! *c* ≤ *k* ! *i* ! *l* ! *c*
and *k_length*:
length k = *n_ps* ∀ *i* < *n_ps*. *length* (*k* ! *i*) = *num_states i*
∀ *xs* ∈ *set k*. ∀ *xxs* ∈ *set xs*. *length xxs* = *m* + *1*
and *k_0*:
∀ *i* < *n_ps*. ∀ *l* < *num_states i*. *k* ! *i* ! *l* ! *0* = *0*
and *inv_unambiguous*:
∀ (_, _, _, *inv*) ∈ *set automata*. *distinct* (*map fst inv*)
and s_0*_bounded*: *bounded bounds* (*map_of* s_0)
and L_0*_len*: *length* L_0 = *n_ps*
and L_0*_has_trans*: ∀ *i* < *n_ps*. L_0 ! *i* ∈ *fst* ‘ *set* ((*fst o snd o 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 ≡
if (*c* > *0* ∧ *c* ≤ *m*) ∧ *fst l* ∈ *states then Max* {*k* ! *i* ! (*fst l* ! *i*) ! *c* | *i* . *i* < *n_ps*} *else 0*

lemma (**in** −) *default_map_of_distinct*:
(*k*, *default_map_of x xs k*) ∈ *set xs* ∪ {(*k*, *x*)} **if** *distinct* (*map fst xs*)

unfolding *default_map_of_alt_def* **by** *clarsimp* (*simp add*: *map_of_eq_Some_iff*[*OF that*])

lemma *N_inv*:
(*L* ! *i*, *inv* (*N i*) (*L* ! *i*)) ∈ *set* ((*snd o snd o snd*) (*automata* ! *i*)) ∪ {(*L* ! *i*, [])}
if *i* < *n_ps*
unfolding *N_def comp_def fst_conv snd_conv inv_def*
using *that*
apply (*subst nth_map*)
apply (*simp add*: *n_ps_def*; *fail*)
apply (*clarsimp split*: *prod.split simp*: *automaton_of_def*)
subgoal for _ _ _ *xs*
using *default_map_of_distinct*[*of xs L* ! *i* []] *inv_unambiguous that*
by (*auto dest*!: *nth_mem simp*: *n_ps_def*)
done

lemma (**in** −) *subset_nat_0_atLeastLessThan_conv*:
S ⊆ {*0*..<*n*::*nat*} ⟷ (∀ *x* ∈ *S*. *x* < *n*)
by *auto*

lemma *k_ceiling_rule*:
m ≤ *int* (*k* ! *i* ! *l* ! *x*)
if *i* < *n_ps* (*l*, *b*, *g*, *xx*) ∈ *set* ((*fst o snd o snd*) (*automata* ! *i*))
(*x*, *m*) ∈ *collect_clock_pairs g*
for *i l x g xx*
using *that k_ceiling*(*2*) **by** *fastforce*

lemma *k_ceiling_1*:
∀ *s*. ∀ *L* ∈ *states*. ∀ (*x*,*m*) ∈ *clkp_set prod_ta* (*L*, *s*). *m* ≤ *k_fun* (*L*, *s*) *x*
proof *safe*
fix *L s c x*
assume ‹*L* ∈ *states*› ‹(*c*, *x*) ∈ *Closure.clkp_set prod_ta* (*L*, *s*)›
have *0* < *c c* ≤ *m*
proof −
from ‹(*c*, *x*) ∈ _› **have** (*c*, *x*) ∈ *Timed_Automata.clkp_set prod_ta*
unfolding *TA_clkp_set_unfold* **by** *auto*
with *clock_range* **show** *0* < *c c* ≤ *m*
by *auto*
qed
with ‹*L* ∈ _› **have** *k_fun* (*L*, *s*) *c* = *Max* {*k* ! *i* ! (*L* ! *i*) ! *c* | *i* . *i* < *n_ps*}
unfolding *k_fun_def* **by** *auto*
have *Max_aux*: *x* ≤ *int* (*Max* {*k* ! *i* ! (*L* ! *i*) ! *c* |*i*. *i* < *n_ps*})
if *x* ≤ *int* (*k* ! *p* ! (*L* ! *p*) ! *c*) *p* < *n_ps* **for** *p*
proof −
from ‹*p* < *n_ps* › **have** *k* ! *p* ! (*L* ! *p*) ! *c* ≤ *Max* {*k* ! *i* ! (*L* ! *i*) ! *c* |*i*. *i* < *n_ps*}
by (*intro Max_ge*) *auto*
with ‹*x* ≤ _› **show** *?thesis*
by *simp*
qed
from ‹(*c*, *x*) ∈ _› **show** ‹*x* ≤ *k_fun* (*L*, *s*) *c*›
unfolding *clkp_set_def*
proof *safe*
assume ‹(*c*, *x*) ∈ *Closure.collect_clki* (*inv_of prod_ta*) (*L*, *s*)›
then show ‹*x* ≤ *k_fun* (*L*, *s*) *c*›
using *k_ceiling*(*1*) **unfolding** *collect_clki_def* ‹*k_fun* (*L*, *s*) *c* = _›

```
    by (auto 0 8 dest: N_inv
        intro!: Max_aux simp: prod_inv_def collect_clock_pairs_def k_fun_def)
  next
   assume ‹(c, x) ∈ Closure.collect_clkt (trans_of prod_ta) (L, s)›
   then show ‹x ≤ k_fun (L, s) c›
    unfolding collect_clkt_def ‹k_fun (L, s) c = _›
    apply (clarsimp simp: trans_prod_def collect_clock_pairs_def k_fun_def)
    apply safe
    subgoal
     using k_ceiling(2) unfolding trans_int_def
     apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
     apply (fastforce intro!: Max_aux simp: collect_clock_pairs_def)
     done
    subgoal
     using k_ceiling(2) unfolding trans_bin_def
     apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
     apply (erule disjE)
      apply (force intro!: Max_aux simp: collect_clock_pairs_def)+
     done
    subgoal
     using k_ceiling(2) unfolding trans_broad_def
     apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
     apply (erule disjE)
      apply (fastforce intro!: Max_aux simp: collect_clock_pairs_def)
     apply (erule bexE)
     apply (force intro!: Max_aux simp: collect_clock_pairs_def) — slow: 60s
     done
    done
  qed
qed

lemma k_fun_mono':
  k_fun (L, s) c ≤ k_fun (L', s') c if
  ∀ i < n_ps. k ! i ! (L ! i) ! c ≤ k ! i ! (L' ! i) ! c L ∈ states L' ∈ states
  using that unfolding k_fun_def
  apply clarsimp
  apply (cases n_ps = 0)
   apply (simp; fail)
  apply (rule Max.boundedI)
   apply (simp; fail)
  apply blast
  apply safe
  subgoal for _ i
   by − (rule order.trans[where b = k ! i ! (L' ! i) ! c], auto intro: Max_ge)
  done

lemma k_fun_mono:
  ‹Max {k ! i ! (L ! i) ! c | i . i < n_ps} ≤ Max {k ! i ! (L' ! i) ! c | i . i < n_ps}›
  if ‹∀ i < n_ps. k ! i ! (L ! i) ! c ≤ k ! i ! (L' ! i) ! c›
  apply (cases n_ps = 0)
   apply (simp; fail)
  apply (rule Max.boundedI)
   apply (simp; fail)
  apply blast
```

apply *safe*
subgoal for _ *i*
using *that* **by** − (*rule order.trans*[**where** *b* = *k* ! *i* ! (*L'* ! *i*) ! *c*], *auto intro*: *Max_ge*)
done

lemma (**in** −) *fold_upds_aux1*:
fold (λ*p L*. *L*[*p* := *g p*]) *ps xs* ! *i* = *xs* ! *i* **if** ‹*i* ∉ *set ps*›
using *that* **by** (*induction ps arbitrary*: *xs*) *auto*

lemma (**in** −) *fold_upds_aux2*:
fold (λ*p L*. *L*[*p* := *g p*]) *ps xs* ! *i* = *g i* **if** ‹*distinct ps*› ‹*i* ∈ *set ps*› ‹*i* < *length xs*›
using *that* **by** (*induction ps arbitrary*: *xs*) (*auto simp*: *fold_upds_aux1*)

lemma (**in** −) *fold_upds_aux_length*:
length (*fold* (λ*p L*. *L*[*p* := *g p*]) *ps xs*) = *length xs*
by (*induction ps arbitrary*: *xs*) *auto*

lemma *prod_ta_step_statesD*:
assumes *prod_ta* ⊢ (*L*, *s*) $\longrightarrow^{g,a,r}$ (*L'*, *s'*)
shows *L* ∈ *states L'* ∈ *states*
using *assms state_set_states* **by** (*fastforce dest*: *state_setI1 state_setI2*)+

lemma *k_ceiling_2*:
∀ *l g a r l'*. ∀ *c* ≤ *m*. *prod_ta* ⊢ *l* $\longrightarrow^{g,a,r}$ *l'* ∧ *c* ∉ *set r* ⟶ *k_fun l' c* ≤ *k_fun l c*
proof *safe*
fix *L s g a r L' s' c*
assume *A*: ‹*c* ≤ *m*› ‹*prod_ta* ⊢ (*L*, *s*) $\longrightarrow^{g,a,r}$ (*L'*, *s'*)› ‹*c* ∉ *set r*›
then have *L* ∈ *states L'* ∈ *states*
by − (*rule prod_ta_step_statesD*, *assumption*)+
from *A* **have** ‹*Max* {*k* ! *i* ! (*L'* ! *i*) ! *c* | *i* . *i* < *n_ps*} ≤ *Max* {*k* ! *i* ! (*L* ! *i*) ! *c* | *i* . *i* < *n_ps*}›
apply *simp*
unfolding *trans_prod_def*
apply *safe*
subgoal
using *k_resets*
unfolding *trans_int_def*
apply *clarsimp*
apply (*rule k_fun_mono*)
apply (*clarsimp simp*: *mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv*)
subgoal for *b f p aa l' i*
by (*cases p* = *i*; *force simp add*: *L_len*)
done
subgoal
using *k_resets*
unfolding *trans_bin_def*
apply *clarsimp*
apply (*rule k_fun_mono*)
apply (*clarsimp simp*: *mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv*)
subgoal for _ _ *p q b1 g1 f1 r1 l1' b2 g2 f2 r2 l2' i*
by (*cases p* = *i*; *cases q* = *i*; *force simp add*: *L_len*)
done
subgoal
using *k_resets*
unfolding *trans_broad_def*

```
    apply clarsimp
    apply (rule k_fun_mono)
    apply (clarsimp simp: mem_trans_N_iff L_len subset_nat_0_atLeastLessThan_conv)
    subgoal premises prems for s'a aa p b ga f ra l' bs gs fs rs ls' ps i
    proof (cases p = i)
      case True
      with ‹p ∉ _› ‹i < _› ‹L ∈ states› have (fold (λp L. L[p := ls' p]) ps L)[p := l'] ! i = l'
        by (simp add: L_len fold_upds_aux_length)
      with prems ‹p = i› show ?thesis
        by (fastforce simp add: L_len)
    next
      case False
      then have ∗: (fold (λp L. L[p := ls' p]) ps L)[p := l'] ! i
        = fold (λp L. L[p := ls' p]) ps L ! i
        by simp
      show ?thesis
      proof (cases i ∈ set ps)
        case True
        then have ∗∗: fold (λp L. L[p := ls' p]) ps L ! i = ls' i
          using ‹distinct ps› ‹i < n_ps› ‹L ∈ states› by (auto simp: fold_upds_aux2)
        moreover have
          (L ! i, bs i, gs i, In aa, fs i, rs i, ls' i) ∈ set (fst (snd (snd (automata ! i))))
          using ‹p ≠ i› True prems by fast
        moreover have c∈{0..<Suc m} − set (rs i)
          using ‹p ≠ i› True prems by force
        ultimately show ?thesis
          using prems(2) ‹i < n_ps› by (auto 4 3 simp add: ∗)
      next
        case False
        with ‹p ≠ i› show ?thesis
          by (simp add: fold_upds_aux1)
      qed
    qed
    done
  done
with ‹L ∈ states› ‹L' ∈ states› ‹c ≤ m› show k_fun (L', s') c ≤ k_fun (L, s) c
  by (auto simp: k_fun_def)
qed



abbreviation F where F ≡ λ(L, s). hd_of_formula formula L (the o s)
abbreviation Fi ≡ λ(L, s). hd_of_formulai formula L s

lemma (in Simple_Network_Impl_nat) check_sexp_check_sexpi:
  check_sexp e L (the o s) ⟷ check_sexpi e L s'
  if state_rel s s' vars_of_sexp e ⊆ {0..<n_vs}
  using that unfolding state_rel_def by (induction e) auto

lemma (in Simple_Network_Impl_nat) hd_of_formula_hd_of_formulai:
  hd_of_formula φ L (the o s) ⟷ hd_of_formulai φ L s'
  if state_rel s s' vars_of_formula φ ⊆ {0..<n_vs}
  using that by (induction φ) (auto simp: check_sexp_check_sexpi)
```

lemma *F_Fi*:
F l $\longleftrightarrow$ *Fi l′* **if** (*l′*, *l*) ∈ *loc_rel*
using *vars_of_formula that* **unfolding** *loc_rel_def* **by** *clarsimp* (*erule hd_of_formula_hd_of_formulai*)

abbreviation $l_0 \equiv (L_0,\ map_of\ s_0)$
abbreviation $s_0 i \equiv map\ (the\ o\ map_of\ s_0)\ [0..<n_vs]$
abbreviation $l_0 i \equiv (L_0,\ s_0 i)$

lemma *state_rel_start*:
state_rel (*map_of* s_0) $s_0 i$
using s_0*_bounded* **unfolding** *state_rel_def bounded_def dom_bounds_eq* **by** *auto*

lemma *statesI*:
L ∈ *states* **if** *length L* = *n_ps* ∀ *i*<*n_ps*. *L* ! *i* ∈ *fst* ‘ *set* (*fst* (*snd* (*snd* (*automata* ! *i*))))
using *that* **unfolding** *states_def* **by** (*auto 4 3 simp*: *mem_trans_N_iff*[*symmetric*])

lemma L_0*_states*[*simp*, *intro*]:
L_0 ∈ *states*
using L_0*_has_trans* L_0*_len* **by** (*auto intro*: *statesI*)

lemma l_0*_states′*[*simp*, *intro*]:
l_0 ∈ *states′*
using *state_rel_start* s_0*_bounded* **unfolding** *states′_def state_rel_def* **by** *auto*

sublocale *reach*: *Reachability_Problem_Defs*
prod_ta
l_0
m
k_fun
F
.

lemma (**in** −) *collect_clkt_state_setI*:
assumes (*x*, *d*) ∈ *Closure.collect_clkt* (*trans_of A*) *l*
shows *l* ∈ *state_set* (*trans_of A*) *l* ∈ *Simulation_Graphs_TA.state_set A*
using *assms* **unfolding** *collect_clkt_def* **by** (*auto simp*: *state_set_def*)

lemma *clkp_set_statesD*:
fixes *x d*
assumes (*x*, *d*)∈*Closure.clkp_set prod_ta* (*L*, *s*)
shows *L* ∈ *states*
using *assms*
unfolding *clkp_set_def collect_clki_def*
apply *safe*
subgoal
apply *simp*
unfolding *prod_inv_def*
unfolding *collect_clock_pairs_def*
apply *auto*
done
apply (*drule collect_clkt_state_setI*)
using *state_set_states* **by** *auto*

sublocale *reach1*: *Reachability_Problem*
prod_ta
l_0
m
k_fun
F
apply *standard*
subgoal
apply *safe*
using *k_ceiling_1*
subgoal for *L s x m*
apply (*cases L* ∈ *states*)
apply *blast*
apply (*auto dest*: *clkp_set_statesD simp*: *k_fun_def*)
done
done
subgoal
apply *safe*
subgoal for *L s g a r L′ s′ c*
apply (*cases c* ≤ *m*)
using *k_ceiling_2*
apply *force*
apply (*auto simp*: *k_fun_def*)
done
done
subgoal
by (*simp add*: *k_fun_def*)
subgoal
by (*simp add*: *k_fun_def*)
done — slow: 60s

lemma *states′_superset*:
{l_0} ∪ *Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod* ⊆ *states′*
(**is** {l_0} ∪ *?S* ⊆ *states′*)
proof −
have *?S* ⊆ *states′*
proof *safe*
fix *L s*
assume (*L*, *s*) ∈ *?S*
then have *L* ∈ *states*
using *state_set_states*[*unfolded trans_of_prod state_set_eq*] **by** *blast*
moreover have *bounded bounds s*
using ‹(*L*, *s*) ∈ _›
unfolding *state_set_def*
unfolding *trans_prod_def*
unfolding *trans_int_def trans_bin_def trans_broad_def*
by *auto*
ultimately show (*L*, *s*) ∈ *states′*
by (*auto simp*: *states′_alt_def*)
qed
then show *?thesis*
by *simp*
qed

definition *k_i* ≡ *IArray* (*map* (*IArray o* (*map* (*IArray o map int*))) *k*)

definition
k_impl ≡ λ(*l*, _). *IArray* (*map* (λ *c*. *Max* {*k_i* !! *i* !! (*l* ! *i*) !! *c* | *i*. *i* < *n_ps*}) [*0*..<*m+1*])

lemma *Max_int_commute*:
int (*Max S*) = *Max* (*int* ‘ *S*) **if** *finite S S* ≠ {}
apply (*rule mono_Max_commute*)
apply *rule*
using *that* **by** *auto*

lemma (**in** *Simple_Network_Impl_nat*) *n_ps_gt_0*: *n_ps* > *0*
using *length_automata_eq_n_ps non_empty* **by** *auto*

lemma *statesD*:
L ! *i* ∈ *fst* ‘ *set* (*fst* (*snd* (*snd* (*automata* ! *i*))))
∨ *L* ! *i* ∈ (*snd o snd o snd o snd o snd o snd*) ‘ *set* (*fst* (*snd* (*snd* (*automata* ! *i*))))
if *L* ∈ *states length L* = *n_ps i* < *n_ps*
using *that* **unfolding** *states_def*
apply *safe*
apply −
apply (*elim allE impE*, *assumption*)
apply *safe*
apply (*force simp*: *mem_trans_N_iff*)+
done

lemma *k_impl_k_fun*:
k_impl (*L*, *s*) = *IArray* (*map* (*k_fun* (*L*, *s*)) [*0*..<*m+1*]) **if** *L* ∈ *states*
proof −
define *k_i2* **where** *k_i2 i c* = *k_i* !! *i* !! (*L* ! *i*) !! *c* **for** *i c*
have *k_i2_k*: *k_i2 i c* = *k* ! *i* ! (*L* ! *i*) ! *c* **if** *i* < *n_ps c* ≤ *m* **for** *i c*
proof −
have *i* < *length k*
by (*simp add*: *k_length*(*1*) *that*(*1*))
moreover have *L* ! *i* < *length* (*k* ! *i*)
using *L_i_len*[*OF* _ ‹*L* ∈ *states*›] *k_length*(*2*) ‹*i* < *n_ps*› **by** *auto*
moreover have *c* < *length* (*k* ! *i* ! (*L* ! *i*))
using *k_length*(*3*) ‹*c* ≤ *m*› ‹*i* < *length k*› ‹*L* ! *i* < *length* (*k* ! *i*)› **by** (*auto dest*: *nth_mem*)
ultimately show *?thesis*
unfolding *k_i2_def k_i_def* **by** *simp*
qed
have *k_impl_alt_def*: *k_impl* (*L*, *s*) = *IArray* (*map* (λ *c*. *Max* {*k_i2 i c* | *i*. *i* < *n_ps*})
[*0*..<*m+1*])
unfolding *k_impl_def k_i2_def* **by** *auto*
have *Max_cong*: *Max* {*f i* | *i*. *i* < *n_ps*} = *Max* {*g i* | *i*. *i* < *n_ps*}
if ⋀ *i*. *i* < *n_ps* ⟹ *f i* = *g i* **for** *f g* :: *nat* ⇒ *int*
by (*rule arg_cong*[**where** *f* = *Max*]) (*force simp*: *that*)
from *that n_ps_gt_0* **show** *?thesis*
unfolding *k_impl_alt_def*
unfolding *k_i2_def*[*symmetric*]
apply (*clarsimp simp*: *k_fun_def k_i2_k cong*: *Max_cong*)

```
    apply safe
    subgoal
      by (subst Max_int_commute; force simp: setcompr_eq_image image_comp comp_def)
    subgoal
      using k_0 L_i_len[OF _ ‹L ∈ states›] by (intro linorder_class.Max_eqI) auto
    subgoal
      by (subst Max_int_commute; force simp: setcompr_eq_image image_comp comp_def)
    done
qed


sublocale impl: Reachability_Problem_Impl
  where trans_fun = trans_from
  and inv_fun = inv_fun
  and F_fun = Fi
  and ceiling = k_impl
  and A = prod_ta
  and l₀ = l₀
  and l₀i = l₀i
  and F = PR_CONST F
  and n = m
  and k = k_fun
  and trans_impl = trans_impl
  and states' = states'
  and loc_rel = loc_rel
  apply standard


  subgoal
    unfolding trans_of_prod by (rule trans_from_correct)


  subgoal
    apply (rule trans_from_refine)
    done


  subgoal
    unfolding trans_of_prod
    by (rule set_mp[OF _ inv_fun_inv_of'[where R = loc_rel and S = {(s, s'). state_rel s' s}]])
       (auto simp: loc_rel_def)


  subgoal
    using states'_superset by simp


  subgoal
    using state_rel_start unfolding loc_rel_def by auto


  subgoal for l li li'
    unfolding trans_of_prod by (rule state_rel_left_unique)
```

subgoal for $l\ l'\ li$
unfolding *trans_of_prod* **by** (*rule state_rel_right_unique*)

subgoal
unfolding *inv_rel_def* **using** L_0*_states*
by (*auto simp*: *loc_rel_def state_rel_def reach.k'_def k_fun_def k_impl_k_fun*)

subgoal
unfolding *inv_rel_def* **by** (*clarsimp dest*!: *F_Fi*)

done — slow: 50s

end

no_notation *UPPAAL_Model_Checking.models* (_,_ ⊨_ _ [*61*,*61*] *61*)

context *Reachability_Problem_Impl*
begin

lemma *F_reachable_correct*:
op.F_reachable
$\longleftrightarrow$ ($\exists\, l'.\ \forall\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \longrightarrow (\exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u_0\rangle \rightarrow* \langle l',\ u'\rangle \wedge F\ l')$)
using *E_op''.E_from_op_reachability_check*[*symmetric*] *reachability_check_new*
unfolding *E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def*
unfolding *F_rel_def* **by** *auto*

lemma *E_op''_F_reachable_correct*:
E_op''.F_reachable
$\longleftrightarrow$ ($\exists\, l'.\ \forall\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \longrightarrow (\exists\ u'.\ conv_A\ A \vdash' \langle l_0,\ u_0\rangle \rightarrow* \langle l',\ u'\rangle \wedge F\ l')$)
using *E_op''.E_from_op_reachability_check*[*symmetric*] *reachability_check_new*
unfolding *E_op_F_reachable E_op''.F_reachable_def E_op''.reachable_def*
unfolding *F_rel_def* **by** *auto*

lemma *Ex_ev_impl_hnr*:
assumes $\forall\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \longrightarrow \neg\ deadlock\ (l_0,\ u_0)$
shows

(*uncurry0*
(*pw_impl* (*return* ∘ *fst*) *state_copy_impl tracei subsumes_impl* a_0*_impl F_impl succs_impl*
emptiness_check_impl),
uncurry0 (*SPEC* ($\lambda r.\ (r \longleftrightarrow (\forall\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \longrightarrow Ex_ev\ (\lambda(l,\ _).\ F\ l)\ (l_0,\ u_0)$))))))
∈ $unit_assn^k \rightarrow_a bool_assn$
proof −
interpret *Bisimulation_Invariant*
($\lambda(l,\ u)\ (l',\ u').\ conv_A\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle$)
($\lambda(l,\ u)\ (l',\ u').\ conv_A\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle$)
($\lambda(l,\ u)\ (l',\ u').\ l' = l \wedge (\forall\ c.\ c \in clk_set\ (conv_A\ A) \longrightarrow u\ c = u'\ c)$)
($\lambda_.\ True$) ($\lambda_.\ True$)

```
    apply (rule ta_bisimulation)
    done
  define spec where spec = (∀ u₀. (∀ c ∈ {1..n}. u₀ c = 0) ⟶ Ex_ev (λ(l, _). F l) (l₀, u₀))
  have ∗: E_op''.F_reachable ⟷ spec
    unfolding spec_def
    unfolding E_op''_F_reachable_correct
  proof safe
    fix l' :: ‹'s› and u₀ :: ‹nat ⇒ real›
    assume
      ‹∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ (∃ u'. conv_A A ⊢' ⟨l₀, u₀⟩ →∗ ⟨l', u'⟩ ∧ F l')› and
      ‹∀ c∈{1..n}. u₀ c = 0›
    then show ‹Ex_ev (λ(l, _). F l) (l₀, u₀)›
      using assms by (subst Ex_ev; unfold reaches_steps'[symmetric]) blast+
  next
    assume ‹∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ Ex_ev (λ(l, _). F l) (l₀, u₀)›
    then have Ex_ev (λ(l, _). F l) (l₀, λ_. 0)
      by auto
    then obtain l' u' where conv_A A ⊢' ⟨l₀, (λ_. 0)⟩ →∗ ⟨l', u'⟩ ∧ F l'
      apply (subst (asm) Ex_ev)
      using assms
      unfolding reaches_steps'[symmetric]
      by auto
    then show ‹∃ l'. ∀ u₀. (∀ c∈{1..n}. u₀ c = 0) ⟶ (∃ u'. conv_A A ⊢' ⟨l₀, u₀⟩ →∗ ⟨l', u'⟩ ∧
F l')›
    proof (inst_existentials l', safe, unfold reaches_steps'[symmetric])
      fix u₀ :: ‹nat ⇒ real›
      assume ‹reaches (l₀, λ_. 0) (l', u')› and ‹F l'› and ‹∀ c∈{1..n}. u₀ c = 0›
      then have equiv' (l₀, λ_. 0) (l₀, u₀)
        unfolding equiv'_def using clocks_I[of λ_. 0 u₀] by auto
      with ‹reaches _ _› ‹F l'› show ‹∃ u'. reaches (l₀, u₀) (l', u') ∧ F l'›
        by − (drule (1) bisim.A_B.simulation_reaches, unfold equiv'_def, auto)
    qed
  qed
  show ?thesis
   unfolding spec_def[symmetric] using pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
    by (sepref_to_hoare (sep_auto simp: ∗)
qed

end


context Simple_Network_Impl_nat_urge
begin

sublocale conv: Prod_TA_urge
  (set broadcast, map (Simple_Network_Language.conv_A o automaton_of) automata, map_of
bounds')
  by standard
    (use no_urgency in ‹auto simp: Simple_Network_Language.conv_A_def automaton_of_def
urgent_def›)

abbreviation A ≡ (set broadcast, map automaton_of automata, map_of bounds')

interpretation conv_eq_bisim:
```

Bisimulation_Invariant
(λ(*l*, *u*) (*l'*, *u'*). *conv_A prod_ta* ⊢' ⟨*l*, *u*⟩ → ⟨*l'*, *u'*⟩)
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *conv.prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩)
(λ((*L*, *s*), *u*) (*L'*, *s'*, *u'*). *L* = *L'* ∧ *u* = *u'* ∧ *s* = *s'*)
(λ((*L*, *s*), *u*). *conv.all_prop L s*)
(λ(*L*, *s*, *u*). *conv.all_prop L s*)
proof *goal_cases*
case *1*
interpret *Bisimulation_Invariant*
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *conv_A prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩)
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *conv.prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩)
(=)
(λ(*L*, *s*, *u*). *conv.all_prop L s*)
(λ(*L*, *s*, *u*). *conv.all_prop L s*)
by (*rule Bisimulation_Invariant_strong_intro*)
(*auto simp*: *conv_prod_ta elim*: *conv.prod_all_prop_inv'*)
show *?case*
by *standard* (*auto simp*: *conv_prod_ta elim*: *conv.prod_all_prop_inv'*)
qed

interpretation *Bisimulation_Invariant*
(λ(*l*, *u*) (*l'*, *u'*). *conv_A prod_ta* ⊢' ⟨*l*, *u*⟩ → ⟨*l'*, *u'*⟩)
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *Simple_Network_Language.conv A* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(λ((*L*, *s*), *u*) (*L'*, *s'*, *u'*). *L* = *L'* ∧ *u* = *u'* ∧ *s* = *s'*)
(λ((*L*, *s*), *u*). *conv.all_prop L s*)
(λ(*L*, *s*, *u*). *conv.all_prop L s*)
unfolding *conv_alt_def*
apply (*rule Bisimulation_Invariant_sim_replace*, *rule Bisimulation_Invariant_composition*)
apply (*rule conv_eq_bisim.Bisimulation_Invariant_axioms conv.prod_bisim_intro*)+
apply *auto*
done

lemmas *prod_bisim* = *Bisimulation_Invariant_axioms*

lemmas *deadlock_iff* = *deadlock_iff*

lemma *conv_all_prop*:
conv.all_prop = *all_prop*
unfolding *conv.all_prop_def all_prop_def* **by** *simp*

definition *prop_of2* **where**
prop_of2 φ = *PropC* (λ((*L*, *s*), _). *check_sexp* φ *L* (*the o s*))

fun *ctl_of2* **where**
ctl_of2 (*formula.EX* φ) = *ctl_formula.EX* (*prop_of2* φ)
| *ctl_of2* (*formula.EG* φ) = *ctl_formula.EG* (*prop_of2* φ)
| *ctl_of2* (*formula.AX* φ) = *ctl_formula.AX* (*prop_of2* φ)
| *ctl_of2* (*formula.AG* φ) = *ctl_formula.AG* (*prop_of2* φ)
| *ctl_of2* (*formula.Leadsto* φ ψ) =
ctl_formula.AG (*ctl_formula.ImpliesC* (*prop_of2* φ) (*ctl_formula.AX* (*prop_of2* ψ)))

lemma *models_correct*:
Simple_Network_Language.conv A,(*L*$_0$, *s*$_0$, *u*$_0$) ⊨ Φ = (*case* Φ *of*
formula.EX φ ⇒

```
    Graph_Defs.Ex_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
  | formula.EG φ ⇒
    Not o Graph_Defs.Alw_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). ¬ check_sexp φ L (the o s))
  | formula.AX φ ⇒
    Graph_Defs.Alw_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
  | formula.AG φ ⇒
    Not o Graph_Defs.Ex_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). ¬ check_sexp φ L (the o s))
  | formula.Leadsto φ ψ ⇒
    Graph_Defs.leadsto
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
      (λ ((L, s), _). check_sexp ψ L (the o s))
  ) ((L₀, s₀), u₀)
   if L₀ ∈ states Simple_Network_Language.bounded bounds s₀
proof −
  have ∗: ((Not ∘∘ case_prod) (λ(L, s) _. check_sexp φ L (the o s)))
    = (λ((L, s), _). ¬ check_sexp φ L (the o s))
    for φ by (auto simp: comp_def)
  have rel_ctl_formula compatible (ctl_of2 Φ) (ctl_of Φ)
    by (cases Φ; simp add: prop_of_def prop_of2_def rel_fun_def A_B.equiv'_def)
  moreover have A_B.equiv' ((L₀, s₀), u₀) (L₀, s₀, u₀)
    unfolding A_B.equiv'_def unfolding conv_all_prop all_prop_def using that by simp
  ultimately show ?thesis
    apply (subst models_ctl_iff)
    apply (subst CTL_compatible[THEN rel_funD, symmetric])
      apply assumption+
    apply (cases Φ; simp add:
        Graph_Defs.models_ctl.simps prop_of2_def
        Graph_Defs.Ex_alw_iff Graph_Defs.Alw_alw_iff Graph_Defs.leadsto_def ∗)
    done
qed

end

context Simple_Network_Impl_nat_ceiling_start_state — slow: 70s
begin

definition Alw_ev_checker where
  Alw_ev_checker = dfs_map_impl'
    (impl.succs_P_impl' Fi) impl.a₀_impl impl.subsumes_impl (return ∘ fst)
    impl.state_copy_impl

definition leadsto_checker where
  leadsto_checker ψ = do {
    r ← leadsto_impl
    impl.state_copy_impl (impl.succs_P_impl' (λ (L, s). ¬ check_sexpi ψ L s))
```

```
    impl.a₀_impl impl.subsumes_impl (return ∘ fst)
    impl.succs_impl' impl.emptiness_check_impl impl.F_impl
    (impl.Q_impl (λ (L, s). ¬ check_sexpi ψ L s))
    impl.tracei;
    return (¬ r)
  }

definition
  reachability_checker ≡
    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

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 u₀ ≡ (λ _. 0 :: real)

lemma all_prop_start:
  all_prop L₀ (map_of s₀)
  using L₀_states s₀_bounded unfolding all_prop_def ..

lemma deadlock_start_iff:
  Graph_Defs.deadlock
    (λ(L, s, u) (L', s', u'). Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩) (L₀,
(map_of s₀), u₀)
  ⟷ reach.deadlock (l₀, u₀)
  using all_prop_start by (subst deadlock_iff[symmetric]) (auto simp: conv_all_prop)

lemma F_Fi':
  check_sexp ψ L (the o s) ⟷ check_sexpi ψ L' s'
  if ((L', s'), (L, s)) ∈ loc_rel formula = Leadsto φ ψ
  using vars_of_formula that unfolding loc_rel_def by (auto elim!: check_sexp_check_sexpi)

theorem model_check':
```

```
 (uncurry0 model_checker,
  uncurry0 (
   SPEC (λr.
¬ Graph_Defs.deadlock
  (λ(L, s, u) (L', s', u').
   Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩) (L₀, (map_of s₀), u₀)
   ⟶ r = (Simple_Network_Language.conv A,(L₀, (map_of s₀), u₀) ⊨ formula)
   )
  )
 )
 ∈ unit_assn^k →_a bool_assn
proof −
 define protect where
  protect = ((λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩))

 have start: l₀ ∈ Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod
  if ¬ Graph_Defs.deadlock protect (l₀, λ_. 0)
  using that unfolding protect_def by (rule impl.init_state_in_state_set[simplified])

 interpret ta_bisim: Bisimulation_Invariant
  (λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(l, u) (l', u'). l' = l ∧ (∀ c. c ∈ clk_set (conv_A prod_ta) ⟶ u c = u' c))
  (λ_. True) (λ_. True)
  by (rule ta_bisimulation[of conv_A prod_ta])

 let ?φ1 = λφ. λ(p, _). case p of (L, s) ⇒ ¬ check_sexp φ L (the ∘ s)
 let ?φ2 = λφ. λ(p, _). case p of (L, s) ⇒ check_sexp φ L (the ∘ s)

 have start_sim:
  ta_bisim.A_B.equiv' (l₀, u) (l₀, λ_. 0) ta_bisim.A_B.equiv' (l₀, λ_. 0) (l₀, u)
  if ∀ c∈{Suc 0..m}. u c = 0 for u
  using impl.clocks_I[of u λ_. 0] that unfolding ta_bisim.A_B.equiv'_def by auto

 have compatibleI: φ a = φ b
  if ta_bisim.A_B.equiv' a b ⋀ l u u'. φ (l, u) = φ (l, u') for a b φ
  using that unfolding ta_bisim.A_B.equiv'_def by auto

 have bisims:
  (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ reach.Ex_ev (?φ1 φ) (l₀, u₀))
  ⟷ reach.Ex_ev (?φ1 φ) (l₀, u₀)
  (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ reach.Ex_ev (?φ2 φ) (l₀, u₀))
  ⟷ reach.Ex_ev (?φ2 φ) (l₀, u₀)
  (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ reach.Alw_ev (?φ1 φ) (l₀, u₀))
  ⟷ reach.Alw_ev (?φ1 φ) (l₀, u₀)
  (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ reach.Alw_ev (?φ2 φ) (l₀, u₀))
  ⟷ reach.Alw_ev (?φ2 φ) (l₀, u₀)
  (∀ u₀. (∀ c∈{Suc 0..m}. u₀ c = 0) ⟶ reach.leadsto (?φ2 φ) (?φ2 ψ) (l₀, u₀))
  ⟷ reach.leadsto (?φ2 φ) (?φ2 ψ) (l₀, u₀)
  for φ ψ
  apply safe
  apply (all ‹(solves auto)?›)
  subgoal for u
   using start_sim
```

```
    by (subst (asm) ta_bisim.Ex_ev_iff[of ?φ1 φ ?φ1 φ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Ex_ev_iff[of ?φ2 φ ?φ2 φ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Alw_ev_iff[of ?φ1 φ ?φ1 φ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Alw_ev_iff[of ?φ2 φ ?φ2 φ]) (erule compatibleI, auto)
  subgoal for u
    using start_sim
    by (subst (asm) ta_bisim.Leadsto_iff[of ?φ2 φ ?φ2 φ ?φ2 ψ ?φ2 ψ])
      ((erule compatibleI)?; auto)+
  done

have deadlock_bisim:
  (∀ u0. (∀ c∈{1..m}. u0 c = 0) ⟶ ¬ reach.deadlock (l0, u0))
  ⟷ ¬ Graph_Defs.deadlock protect (l0, λ_. 0)
  unfolding protect_def
  apply (safe; clarsimp)
  apply (drule start_sim(2))
  apply (subst (asm) ta_bisim.deadlock_iff)
  unfolding ta_bisim.A_B.equiv'_def
  by auto

have *****:
  return True = (return False ⤜ return o Not)
  by auto

show ?thesis
  unfolding deadlock_start_iff
  using models_correct[OF L0_states s0_bounded]
  unfolding protect_def[symmetric]
  apply (simp split del: if_split)
  apply sepref_to_hoare
  apply sep_auto
 unfolding model_checker_def Alw_ev_checker_def leadsto_checker_def reachability_checker_def
  apply (cases formula; simp)

  subgoal
    apply (cases Graph_Defs.deadlock protect (l0, λ_. 0))
    subgoal
      using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
      by (sep_auto elim!: cons_post_rule)
    subgoal
      using impl.Ex_ev_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def]
      by (sep_auto simp: pure_def protect_def bisims)
    done

  subgoal premises prems for φ
  proof −
    have *: (λ(l, u). ¬ F l) = ((λ(p, _). case p of (L, s) ⇒ ¬ check_sexp φ L (the ∘ s)))
      using prems by auto
```

```
  show ?thesis
    using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
    unfolding PR_CONST_def ∗ protect_def
    by (sep_auto elim!: cons_post_rule simp: pure_def bisims)
qed

subgoal premises prems for φ
proof −
  have ∗: (λ(l, u). ¬ F l) = ((λ(p, _). case p of (L, s) ⇒ check_sexp φ L (the ∘ s)))
    using prems by auto
  show ?thesis
    apply intros
    subgoal
      using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
      unfolding PR_CONST_def ∗ protect_def
      apply simp
      unfolding ∗∗∗∗∗
      apply (erule bind_rule)
      apply (sep_auto simp: pure_def bisims(2−))
      done
    subgoal
      using impl.Alw_ev_impl_hnr[to_hnr, unfolded hn_refine_def] prems start
      unfolding PR_CONST_def ∗ protect_def
      apply (sep_auto elim!: cons_post_rule simp: pure_def bisims(2−))
      done
    done
qed

subgoal
  apply (cases Graph_Defs.deadlock protect (l0, λ_. 0))
  subgoal
    using impl.pw_impl_hnr_F_reachable[to_hnr, unfolded hn_refine_def]
    by (sep_auto elim!: cons_post_rule)
  subgoal
    using impl.Ex_ev_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def]
    apply (sep_auto simp: pure_def protect_def bisims)
    done
  done

subgoal premises prems for φ ψ
proof −
  have ∗: (λ(l, u). F l) = ((λ(p, _). case p of (L, s) ⇒ check_sexp φ L (the ∘ s)))
    using prems by auto
  have ∗∗:(λ(L, s). ¬ check_sexpi ψ L s, λ(L, s). ¬ check_sexp ψ L (the ∘ s))
        ∈ inv_rel loc_rel states′
    unfolding trans_of_prod using prems by (auto simp: F_Fi′ inv_rel_def)
  have ∗∗∗∗: (λ(l, u). ¬ (case l of (L, s) ⇒ ¬ check_sexp ψ L (the ∘ s)))
  = (λ(l, u). (case l of (L, s) ⇒ check_sexp ψ L (the ∘ s)))
    by auto
  show ?thesis
    apply (cases reach.deadlock (l0, λ_. 0))
    subgoal
      using impl.leadsto_impl_hnr′[OF ∗∗, to_hnr, unfolded hn_refine_def]
      unfolding protect_def ‹formula = _› by (sep_auto simp: pure_def)
```

```
      using impl.leadsto_impl_hnr[unfolded deadlock_bisim, to_hnr, unfolded hn_refine_def,
OF **]
       prems start
      unfolding PR_CONST_def * protect_def by (sep_auto simp: pure_def bisims ****)
   qed
   done
qed
```

9.1 Extracting an Efficient Implementation

```
lemma reachability_checker_alt_def':
  reachability_checker ≡
    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.emptiness_check_impl;
        let trace = impl.tracei;
        pw_impl key copy trace sub start final succs empty
      };
      _ ← return ();
      return x
    }
  unfolding reachability_checker_def by simp

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
    }
  unfolding Alw_ev_checker_def by simp

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.emptiness_check_impl;
      trace = impl.tracei
    in
      leadsto_impl copy succs start sub key succs' empty final final' trace;
    return (¬ r)
  }
 unfolding leadsto_checker_def by simp

theorem k_impl_alt_def:
 k_impl = (λ(l, s). IArray (map (λc. MAX i ∈ {0..<n_ps}. k_i !! i !! (l ! i) !! c) [0..<m + 1]))
proof −
 have {i. i < p} = {0..<p} for p :: nat
  by auto
 then show ?thesis
  unfolding k_impl_def setcompr_eq_image by simp
qed

definition
 trans_map_inner ≡ map (λi. union_map_of (fst (snd (snd (automata ! i))))) [0..<n_ps]

lemma trans_map_alt_def:
 trans_map i j = (case (IArray trans_map_inner !! i) j of None ⇒ [] | Some xs ⇒ xs)
 if i < n_ps
 using that unfolding trans_map_inner_def trans_map_def by (auto simp: n_ps_def)

schematic_goal succs_impl_alt_def:
 impl.succs_impl ≡ ?impl
 unfolding impl.succs_impl_def
 apply (abstract_let impl.E_op''_impl E_op''_impl)
 unfolding impl.E_op''_impl_def fw_impl'_int
 apply (abstract_let trans_impl trans_impl)
 unfolding trans_impl_def
 apply (abstract_let int_trans_impl int_trans_impl)
 apply (abstract_let bin_trans_from_impl bin_trans_impl)
 apply (abstract_let broad_trans_from_impl broad_trans_impl)
 unfolding int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def
 apply (abstract_let trans_in_broad_grouped trans_in_broad_grouped)
 apply (abstract_let trans_out_broad_grouped trans_out_broad_grouped)
 apply (abstract_let trans_in_map trans_in_map)
 apply (abstract_let trans_out_map trans_out_map)
 apply (abstract_let int_trans_from_all_impl int_trans_from_all_impl)
 unfolding int_trans_from_all_impl_def
 apply (abstract_let int_trans_from_vec_impl int_trans_from_vec_impl)
 unfolding int_trans_from_vec_impl_def
 apply (abstract_let int_trans_from_loc_impl int_trans_from_loc_impl)
 unfolding int_trans_from_loc_impl_def
 apply (abstract_let trans_i_map trans_i_map)
 unfolding trans_out_broad_grouped_def trans_out_broad_map_def
 unfolding trans_in_broad_grouped_def trans_in_broad_map_def
 unfolding trans_in_map_def trans_out_map_def
 unfolding trans_i_map_def
 apply (abstract_let trans_map trans_map)
```

apply (*abstract_let inv_fun :: nat list × int list ⇒ _ inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)
apply (*abstract_let n_ps n_ps*)
by (*rule Pure.reflexive*)

schematic_goal *succs_P_impl_alt_def*:
impl.succs_P_impl Pi ≡ ?impl
if (*Pi*, *P*) ∈ *inv_rel loc_rel states'*
for *P Pi*
unfolding *impl.succs_P_impl_def*[*OF that*]
apply (*abstract_let impl.E_op''_impl E_op''_impl*)
unfolding *impl.E_op''_impl_def fw_impl'_int*
apply (*abstract_let trans_impl trans_impl*)
unfolding *trans_impl_def*
apply (*abstract_let int_trans_impl int_trans_impl*)
apply (*abstract_let bin_trans_from_impl bin_trans_impl*)
apply (*abstract_let broad_trans_from_impl broad_trans_impl*)
unfolding *int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def*
apply (*abstract_let trans_in_broad_grouped trans_in_broad_grouped*)
apply (*abstract_let trans_out_broad_grouped trans_out_broad_grouped*)
apply (*abstract_let trans_in_map trans_in_map*)
apply (*abstract_let trans_out_map trans_out_map*)
apply (*abstract_let int_trans_from_all_impl int_trans_from_all_impl*)
unfolding *int_trans_from_all_impl_def*
apply (*abstract_let int_trans_from_vec_impl int_trans_from_vec_impl*)
unfolding *int_trans_from_vec_impl_def*
apply (*abstract_let int_trans_from_loc_impl int_trans_from_loc_impl*)
unfolding *int_trans_from_loc_impl_def*
apply (*abstract_let trans_i_map trans_i_map*)
unfolding *trans_out_broad_grouped_def trans_out_broad_map_def*
unfolding *trans_in_broad_grouped_def trans_in_broad_map_def*
unfolding *trans_in_map_def trans_out_map_def*
unfolding *trans_i_map_def*
apply (*abstract_let trans_map trans_map*)
apply (*abstract_let inv_fun :: nat list × int list ⇒ _ inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)
apply (*abstract_let n_ps n_ps*)
by (*rule Pure.reflexive*)

lemmas *succs_P'_impl_alt_def* =
impl.succs_P_impl'_def[*OF impl.F_fun*, *unfolded succs_P_impl_alt_def*[*OF impl.F_fun*]]

schematic_goal *Alw_ev_checker_alt_def*:
Alw_ev_checker ≡ ?impl

unfolding *Alw_ev_checker_alt_def'*
unfolding *succs_P'_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *impl.F_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

lemma *ψ_compatibleI*:
(λ(*L*, *s*). ¬ *check_sexpi ψ L s*, λ(*L*, *s*). ¬ *check_sexp ψ L* (*the* ∘ *s*)) ∈ *inv_rel loc_rel states'*
if *formula* = *Leadsto φ ψ*
using *F_Fi'*[*OF _ that*] **unfolding** *inv_rel_def* **by** *auto*

lemma *Q_impl_alt_def*:
impl.Q_impl (λ(*L*, *s*). ¬ *check_sexpi ψ L s*) ≡
λ*xi*. *return* (*case xi of* (*a1*, *a2*) ⇒ (λ(*L*, *s*). ¬ *check_sexpi ψ L s*) *a1*)
if *formula* = *Leadsto φ ψ*
by (*intro impl.Q_impl_def*[**where** *Q* = λ(*L*, *s*). ¬ *check_sexp ψ L* (*the o s*)] *ψ_compatibleI*[*OF that*])

schematic_goal *leadsto_checker_alt_def*:
leadsto_checker ψ ≡ *?impl* **if** *formula* = *Leadsto φ ψ*
unfolding *leadsto_checker_alt_def'*
unfolding *Q_impl_alt_def*[*OF that*] *impl.F_impl_def*
unfolding *impl.succs_P_impl'_def*[*OF ψ_compatibleI*[*OF that*]]
unfolding *succs_P_impl_alt_def*[*OF ψ_compatibleI*[*OF that*]]
unfolding *impl.succs_impl'_def succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *impl.F_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

schematic_goal *reachability_checker_alt_def*:
reachability_checker ≡ *?impl*
unfolding *reachability_checker_alt_def'*
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *impl.F_impl_def*
unfolding *impl.subsumes_impl_def*

unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

end

concrete_definition *reachability_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.reachability_checker_alt_def*

concrete_definition *Alw_ev_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.Alw_ev_checker_alt_def*

concrete_definition *leadsto_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.leadsto_checker_alt_def*

context *Simple_Network_Impl_nat_ceiling_start_state* — slow: 100s
begin

lemma *model_checker_unfold_leadsto*:
model_checker = (
case formula of Simple_Network_Language_Model_Checking.formula.EX xa ⇒ *reachability_checker*
| *Simple_Network_Language_Model_Checking.formula.EG xa* ⇒
if PR_CONST F l_0 *then Alw_ev_checker else return False*
| *Simple_Network_Language_Model_Checking.formula.AX xa* ⇒
(*if PR_CONST F* l_0 *then Alw_ev_checker else return False*) ⤜ (λ*r. return* (¬ *r*))
| *Simple_Network_Language_Model_Checking.formula.AG xa* ⇒
reachability_checker ⤜ (λ*r. 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 L_0 s_0 *formula* ψ *show_clock show_state*)

unfolding *model_checker_def*
using *leadsto_checker.refine*[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*]
by (*simp split*: *formula.split*)

lemmas *model_checker_def_refined* = *model_checker_unfold_leadsto*[*unfolded*
reachability_checker.refine[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*]
Alw_ev_checker.refine[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*]
]

end

concrete_definition *model_checker* **uses**
Simple_Network_Impl_nat_ceiling_start_state.model_checker_def_refined

definition *precond_mc* **where**
precond_mc
show_clock show_state broadcast bounds′ automata m num_states num_actions k L_0 s_0 *formula*
≡
if Simple_Network_Impl_nat_ceiling_start_state
broadcast bounds′ automata m num_states num_actions k L_0 s_0 *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> precond_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 ⇒ ↑(¬ Simple_Network_Impl_nat_ceiling_start_state
      broadcast bounds automata m num_states num_actions k L₀ s₀ formula)
   >ₜ
proof −
  define A where A ≡ N broadcast automata bounds
  define check where check ≡ A,(L₀, map_of s₀, λ_ . 0) ⊨ formula
  note [sep_heap_rules] =
    Simple_Network_Impl_nat_ceiling_start_state.model_check'[
      to_hnr, unfolded hn_refine_def,
      of broadcast bounds automata m num_states num_actions k L₀ s₀ formula,
      unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
      folded A_def check_def has_deadlock_def,
      simplified
      ]
  show ?thesis
    unfolding A_def[symmetric] check_def[symmetric]
    unfolding precond_mc_def
    by (sep_auto simp: model_checker.refine[symmetric] pure_def)
qed

end
```

10 Deadlock Checking

```
theory Deadlock
  imports
   Timed_Automata.Timed_Automata Timed_Automata.CTL Difference_Bound_Matrices.DBM_Operations
    Timed_Automata.Normalized_Zone_Semantics
    Munta_Base.Normalized_Zone_Semantics_Impl
```

Munta_Base.Normalized_Zone_Semantics_Impl_Refine
Difference_Bound_Matrices.FW_More
Munta_Base.TA_Syntax_Bundles
begin

unbundle *no_library_syntax*

10.1 Notes

If we want to run the deadlock checker together with a reachability check for property P, we can:

- run a reachability check with $\neg$ *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 $\neg$ *check_deadlock* on the final set of states, and not intermediate subsumed ones, as the operation is expensive.

10.2 Abstract Reachability Checking

definition *zone_time_pre* :: ('*c*, ('*t*::*time*)) *zone* ⇒ ('*c*, '*t*) *zone*
(_$^{\downarrow}$ [*71*] *71*)
where
$Z^{\downarrow} = \{u \mid u\ d.\ (u \oplus d) \in Z \wedge d \geq (0::'t)\}$

definition *zone_set_pre* :: ('*c*, '*t*::*time*) *zone* ⇒ '*c list* ⇒ ('*c*, '*t*) *zone*
where
zone_set_pre Z r = $\{u\ .\ ([r \rightarrow (0::'t)]u) \in Z\}$

definition *zone_pre* :: ('*c*, '*t*::*time*) *zone* ⇒ '*c list* ⇒ ('*c*, '*t*) *zone*
where
zone_pre Z r = (*zone_set_pre Z r*)$^{\downarrow}$

lemma *zone_time_pre_mono*:
$A^{\downarrow} \subseteq B^{\downarrow}$ **if** $A \subseteq B$
using *that* **unfolding** *zone_time_pre_def* **by** *auto*

lemma *clock_set_split*:
P (([*r* → *0*]*u*) *x*) ⟷ (*x* ∉ *set r* ⟶ *P* (*u x*)) ∧ (*x* ∈ *set r* ⟶ *P 0*)
by (*cases x* ∈ *set r*) *auto*

context *Regions_TA*
begin

definition
check_deadlock l Z ≡ *Z* ⊆
⋃ {(*zone_set_pre* {*u*. *u* ⊢ *inv_of A l'*} *r* ∩ {*u*. *u* ⊢ *g*} ∩ {*u*. *u* ⊢ *inv_of A l*})$^{\downarrow}$ | *g a r l'*.

```
    A ⊢ l ⟶^{g,a,r} l'}

lemma V_zone_time_pre:
  x ∈ (Z ∩ V)^↓ if x ∈ Z^↓ x ∈ V
  using that unfolding zone_time_pre_def by (auto simp: V_def cval_add_def)

lemma check_deadlock_alt_def:
  check_deadlock l Z = (Z ⊆ ⋃ {
    (zone_set_pre ({u. u ⊢ inv_of A l'} ∩ V) r ∩ {u. ∀ x ∈ set r. u x ≥ 0}
      ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l})^↓ ∩ V
    | g a r l'. A ⊢ l ⟶^{g,a,r} l'}) (is _ = (?L ⊆ ?R)) if Z ⊆ V
proof −
  { fix g a r l' x
    assume t: A ⊢ l ⟶^{g,a,r} l'
    assume x: x ∈ (zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l})^↓
    assume x ∈ V
    let ?A = zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l}
    let ?B = zone_set_pre ({u. u ⊢ inv_of A l'} ∩ V) r ∩ {u. ∀ x ∈ set r. u x ≥ 0}
          ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l}
    from valid_abstraction have collect_clkvt (trans_of A) ⊆ X
      by (auto elim: valid_abstraction.cases)
    have ∗: 0 ≤ u c if c ∈ set r u ∈ V for c u
    proof −
      from t ‹c ∈ set r› have c ∈ collect_clkvt (trans_of A)
        unfolding collect_clkvt_def by force
      with ‹_ ⊆ X› have c ∈ X
        by auto
      with ‹u ∈ V› show ?thesis
        by (auto simp: V_def)
    qed
    have ∗∗: u ∈ zone_set_pre ({u. u ⊢ inv_of A l'} ∩ V) r
      if u ∈ zone_set_pre {u. u ⊢ inv_of A l'} r u ∈ V for u
      using that unfolding zone_set_pre_def V_def by (auto split: clock_set_split)
    from x ‹x ∈ V› have x ∈ (?A ∩ V)^↓
      by (rule V_zone_time_pre)
    moreover have y ∈ ?B if y ∈ ?A ∩ V for y
      using that by (auto intro: ∗ ∗∗)
    ultimately have x ∈ ?B^↓
      unfolding zone_time_pre_def by auto
  } note ∗ = this
  have zone_set_pre (Z ∩ V) r ⊆ zone_set_pre Z r for Z r
    unfolding zone_set_pre_def by auto
  with ‹Z ⊆ V› show ?thesis
    unfolding check_deadlock_def
    apply safe
    subgoal for x
      apply rotate_tac
      apply (drule subsetD, assumption)
      apply (drule subsetD, assumption)
      apply clarsimp
      apply (frule ∗, assumption+)
      subgoal for g a r l'
        by (inst_existentials
            (zone_set_pre ({u. u ⊢ inv_of A l'} ∩ V) r ∩ {u. ∀x∈set r. 0 ≤ u x} ∩ {u. u ⊢ g} ∩
```

```
        {u. u ⊢ inv_of A l})↓ ∩ V) auto
    done
  apply (drule subsetD, assumption)+
  apply safe
  subgoal for x X g a r l'
    by (drule
        subsetD[OF zone_time_pre_mono,
         where B1 = zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l},
          rotated]; force)
  done
qed

lemma step_trans1:
  assumes A ⊢_t ⟨l, u⟩ →_(g,a,r) ⟨l', u'⟩
  shows u ∈ zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} A ⊢ l ⟶^(g,a,r) l'
  using assms by (auto elim!: step_trans.cases simp: zone_set_pre_def)

lemma step_trans2:
  assumes u ∈ zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} A ⊢ l ⟶^(g,a,r) l'
  shows ∃ u'. A ⊢_t ⟨l, u⟩ →_(g,a,r) ⟨l', u'⟩
  using assms unfolding zone_set_pre_def by auto

lemma time_pre_zone:
  u ∈ (Z ∩ {u. u ⊢ inv_of A l})↓ if A ⊢ ⟨l, u⟩ →^d ⟨l', u'⟩ u' ∈ Z
  using that by (auto elim!: step_t.cases simp: zone_time_pre_def)

lemma time_pre_zone':
  ∃ d u'. u' ∈ Z ∧ A ⊢ ⟨l, u⟩ →^d ⟨l, u'⟩ if u ∈ (Z ∩ {u. u ⊢ inv_of A l})↓
  using that unfolding zone_time_pre_def by auto

lemma step_trans3:
  assumes A ⊢' ⟨l, u⟩ →^(g,a,r) ⟨l', u'⟩
  shows u ∈ (zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l})↓
        A ⊢ l ⟶^(g,a,r) l'
 using assms by (auto dest: step_trans1 time_pre_zone step_delay_loc elim: step_trans'.cases)

lemma step_trans4:
  assumes u ∈ (zone_set_pre {u. u ⊢ inv_of A l'} r ∩ {u. u ⊢ g} ∩ {u. u ⊢ inv_of A l})↓
        A ⊢ l ⟶^(g,a,r) l'
    shows ∃ u'. A ⊢' ⟨l, u⟩ →^(g,a,r) ⟨l', u'⟩
  using assms by (fast dest: time_pre_zone' step_trans2[rotated])

lemma check_deadlock_correct:
  check_deadlock l Z ⟷ (∀ u ∈ Z. ∃ l' u' g a r. A ⊢' ⟨l, u⟩ →^(g,a,r) ⟨l', u'⟩)
  unfolding check_deadlock_def
  apply safe
  subgoal for x
    using step_trans4 by blast
  subgoal for x
    using step_trans3 by fast
  done

lemma step'_step_trans'_iff:
```

$A \vdash' \langle l, u\rangle \to \langle l', u'\rangle \longleftrightarrow (\exists\, g\ a\ r.\ A \vdash' \langle l, u\rangle \to^{(g,a,r)} \langle l', u'\rangle)$
by (*metis prod_cases3 step'.cases step'.intros step_a.cases step_a.simps step_trans'.cases step_trans'.intros step_trans.cases step_trans.simps*)

lemma *check_deadlock_correct_step'*:
$check_deadlock\ l\ Z \longleftrightarrow (\forall\, u \in Z.\ \exists\, l'\ u'.\ A \vdash' \langle l, u\rangle \to \langle l', u'\rangle)$
using *check_deadlock_correct step'_step_trans'_iff* **by** *simp*

Unused lemma *delay_step_zone*:
$u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv_of\ A\ l\}$ **if** $A \vdash \langle l, u\rangle \to^{d} \langle l, u'\rangle\ u \in Z$
using *that* **by** (*auto elim*!: *step_t.cases simp*: *zone_delay_def*)

lemma *delay_step_zone'*:
$\exists\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \to^{d} \langle l, u'\rangle$ **if** $u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv_of\ A\ l\}$
using *that* **by** (*auto simp*: *zone_delay_def*)

lemma *delay_step_zone''*:
$(\exists\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \to^{d} \langle l, u'\rangle) \longleftrightarrow u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv_of\ A\ l\}$
using *delay_step_zone delay_step_zone'* **by** *blast*

lemma *delay_step_zone'''*:
$\{u' \mid u'\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \to^{d} \langle l, u'\rangle\} = Z^{\uparrow} \cap \{u.\ u \vdash inv_of\ A\ l\}$
using *delay_step_zone''* **by** *auto*

end

context *Regions_TA_Start_State*
begin

lemma *check_deadlock_deadlocked*:
$\neg\ check_deadlock\ l\ Z \longleftrightarrow (\exists\, u{\in}Z.\ sim.sim.deadlocked\ (l, u))$
unfolding *check_deadlock_correct_step' sim.sim.deadlocked_def* **by** *simp*

lemma *deadlock_check'*:
$(\exists\, x_0{\in}a_0.\ \exists\, l\ u.\ sim.sim.reaches\ x_0\ (l, u) \wedge sim.sim.deadlocked\ (l, u)) \longleftrightarrow$
$(\exists\, l\ Z.\ reaches\ (l_0, Z_0)\ (l, Z) \wedge \neg\ check_deadlock\ l\ Z)$
apply (*subst ta_reaches_ex_iff*)
subgoal for $l\ u\ u'\ R$
by (*rule sim_complete_bisim'.P1_deadlocked_compatible*[**where** $a = from_R\ l\ R$];
(*rule sim_complete_bisim'.P1_P1'*)*?*) (*auto intro*: *sim_complete_bisim'.P1_P1'*)
using *check_deadlock_deadlocked* **by** *auto*

lemma *deadlock_check*:
$(\exists\, x_0{\in}a_0.\ sim.sim.deadlock\ x_0) \longleftrightarrow (\exists\, l\ Z.\ reaches\ (l_0, Z_0)\ (l, Z) \wedge \neg\ check_deadlock\ l\ Z)$
unfolding *deadlock_check'*[*symmetric*] *sim.sim.deadlock_def* **by** *simp*

end

10.3 Operations

Subset inclusion check for federations on DBMs lemma
$S \subseteq R \longleftrightarrow S \cap -R = \{\}$

by *auto*

lemma
$A \subseteq B \cup C \longleftrightarrow A \cap -B \cap -C = \{\}$
by *auto*

lemma
$(A \cup B) \cap (C \cup D) = A \cap C \cup A \cap D \cup B \cap C \cup B \cap D$
by *auto*

lemma *Le_le_inf*[*intro*]:
Le (x :: _ :: *linordered_cancel_ab_monoid_add*) $\preceq \infty$
by (*auto intro*: *linordered_monoid.order.strict_implies_order*)

lemma *dbm_entry_val_mono*:
dbm_entry_val u a b e' **if** *dbm_entry_val u a b e e* $\leq e'$
using *that*
by *cases*
(*auto simp*: *DBM.less_eq intro*: *dbm_entry_val_mono1 dbm_entry_val_mono2 dbm_entry_val_mono3*
| *simp add*: *DBM.less_eq dbm_entry_val.simps dbm_le_def*
)+

definition *and_entry* ::
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 = ($\lambda i\ j$. *if* $i = a \wedge j = b$ *then min* (*M i j*) *e else M i j*)

lemma *and_entry_mono*:
and_entry a b e M i j $\leq$ *M i j*
by (*auto simp*: *and_entry_def*)

abbreviation *clock_to_option a* $\equiv$ (*if a* $>$ *0 then Some a 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* (λ(*i*, *j*). *map* (λ *M*. *and_entry j i* (*neg_dbm_entry* (*m i j*)) *M*) *xs*)
[(*i*, *j*). *i*←[*0*..<*Suc n*], *j*←[*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*:
$v\ c = c$ **if** $v\ c \leq n$
using *v_is_id that*
apply (*cases* $c = 0$)
apply (*simp*; *fail*)
apply (*cases* $c \leq n$; *auto*)
done

lemma *le_n*:
  $c \leq n$ **if** $v\ c \leq n$
  using *that v_is_id* **by** *auto*

lemma *gt_0*:
  $c > 0$ **if** $v\ c \leq n$
  using *that v_is_id* **by** *auto*

lemma *le_n_iff*:
  $v\ c \leq n \longleftrightarrow c \leq n \wedge c > 0$
  using *v_is_id* **by** *auto*

lemma *v_0*:
  $v\ c = 0 \longleftrightarrow \mathit{False}$
  using *v_is_id* **by** (*cases* $c > 0$; *simp*; *cases* $c > n$; *simp*)

lemma *surj_on*: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
  using *v_is_id* **by** *blast*

abbreviation *zone_of* ($[\![_]\!]$) **where** $\mathit{zone_of}\ M \equiv [M]_{v,n}$

abbreviation
  $\mathit{dbm_fed}\ S \equiv \bigcup\ m \in S.\ [\![m]\!]$

abbreviation
  $\mathit{dbm_list}\ xs \equiv \mathit{dbm_fed}\ (\mathit{set}\ xs)$

lemma *dbm_fed_singleton*:
  $\mathit{dbm_fed}\ \{m\} = [m]_{v,n}$
  by *auto*

lemma *dbm_list_single*:
  $\mathit{dbm_list}\ xs = [m]_{v,n}$ **if** $\mathit{set}\ xs = \{m\}$
  using *that* **by** *auto*

lemma *dbm_fed_superset_fold*:
  $S \subseteq \mathit{dbm_list}\ xs \longleftrightarrow \mathit{fold}\ (\lambda m\ S.\ S \cap -\ ([m]_{v,n}))\ xs\ S = \{\}$
proof (*induction xs arbitrary*: *S*)
  case *Nil*
  then show *?case*
    by *auto*
next
  case (*Cons m xs*)
  have $S \subseteq \mathit{dbm_list}\ (m\ \#\ xs) \longleftrightarrow S \cap -\ ([m]_{v,n}) \subseteq \mathit{dbm_list}\ xs$
    by *auto*
  moreover have $\ldots \longleftrightarrow \mathit{fold}\ (\lambda m\ S.\ S \cap -\ ([m]_{v,n}))\ xs\ (S \cap -\ ([m]_{v,n})) = \{\}$
    by *fact*
  ultimately show *?case*
    by *simp*
qed

lemma *dbm_fed_superset_fold'*:
  $\mathit{dbm_fed}\ S \subseteq \mathit{dbm_list}\ xs \longleftrightarrow \mathit{dbm_fed}\ (\mathit{fold}\ f\ xs\ S) = \{\}$ **if**
  $\bigwedge\ m\ S.\ m \in \mathit{set}\ xs \Longrightarrow \mathit{dbm_fed}\ (f\ m\ S) = \mathit{dbm_fed}\ S \cap -\ ([m]_{v,n})$

proof −
from *that* **have** *fold* (λ*m S*. *S* ∩ − ([*m*]$_{v,n}$)) *xs* (*dbm_fed S*) = *dbm_fed* (*fold f xs S*)
proof (*induction xs arbitrary*: *S*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a xs*)
from *Cons.prems* **have**
dbm_fed (*fold f xs* (*f a S*)) = *fold* (λ*m S*. *S* ∩ − ([*m*]$_{v,n}$)) *xs* (*dbm_fed* (*f a S*))
by − (*rule sym*, *rule Cons.IH*, *auto*)
then show *?case*
by (*simp add*: *Cons.prems*)
qed
then show *?thesis*
by (*simp add*: *dbm_fed_superset_fold*)
qed

lemma *dbm_fed_superset_fold″*:
dbm_list S ⊆ *dbm_list xs* ⟷ *dbm_list* (*fold f xs S*) = {} **if**
⋀ *m S*. *m* ∈ *set xs* ⟹ *dbm_list* (*f m S*) = *dbm_list S* ∩ − ([*m*]$_{v,n}$)
proof −
from *that* **have** *fold* (λ*m S*. *S* ∩ − ([*m*]$_{v,n}$)) *xs* (*dbm_list S*) = *dbm_list* (*fold f xs S*)
proof (*induction xs arbitrary*: *S*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons a xs*)
from *Cons.prems* **have**
dbm_list (*fold f xs* (*f a S*)) = *fold* (λ*m S*. *S* ∩ − ([*m*]$_{v,n}$)) *xs* (*dbm_list* (*f a S*))
by − (*rule sym*, *rule Cons.IH*, *auto*)
then show *?case*
by (*simp add*: *Cons.prems*)
qed
then show *?thesis*
by (*simp add*: *dbm_fed_superset_fold*)
qed

lemma *neg_inf*:
{*u*. ¬ *dbm_entry_val u a b e*} = {} **if** *e* = (∞ :: _ *DBMEntry*)
using *that* **by** *auto*

lemma *dbm_entry_val′_diff_shift*:
dbm_entry_val′ (*u* ⊕ *d*) *c1 c2* (*M c1 c2*) **if** *dbm_entry_val′ u c1 c2* (*M c1 c2*) *0* < *c1 0* < *c2*
using *that* **unfolding** *dbm_entry_val′_def cval_add_def*
by (*auto elim*!: *dbm_entry_val.cases intro*!: *dbm_entry_val.intros*)

lemma *dbm_entry_val_iff_bounded_Le1*:
dbm_entry_val u (*Some c1*) *None e* ⟷ *Le* (*u c1*) ≤ *e*
by (*cases e*) (*auto simp*: *any_le_inf*)

lemma *dbm_entry_val_iff_bounded_Le2*:
dbm_entry_val u None (*Some c2*) *e* ⟷ *Le* (− *u c2*) ≤ *e*

by (*cases e*) (*auto simp*: *any_le_inf*)

lemma *dbm_entry_val_iff_bounded_Le3*:
dbm_entry_val u (*Some c1*) (*Some c2*) *e* ⟷ *Le* (*u c1* − *u c2*) ≤ *e*
by (*cases e*) (*auto simp*: *any_le_inf*)

lemma *dbm_entry_val'_iff_bounded*:
dbm_entry_val' u c1 c2 e ⟷ *Le*((*if c1* > *0 then u c1 else 0*) − (*if c2* > *0 then u c2 else 0*))
≤ *e*
if *c1* > *0* ∨ *c2* > *0*
using *that* **unfolding** *dbm_entry_val'_def*
by (*auto simp*:
dbm_entry_val_iff_bounded_Le1 dbm_entry_val_iff_bounded_Le2 dbm_entry_val_iff_bounded_Le3
)

context
notes [*simp*] = *dbm_entry_val'_def*
begin

lemma *neg_entry'*:
{*u*. ¬ *dbm_entry_val' u a b e*} = {*u*. *dbm_entry_val' u b a* (*neg_dbm_entry e*)}
if *e* ≠ (∞ :: _ *DBMEntry*) *a* > *0* ∨ *b* > *0*

using *that* **by** (*cases e*; *cases a* > *0*; *cases b* > *0*; *auto 4 3 simp*: *le_minus_iff less_minus_iff*)

lemma *neg_unbounded*:
{*u*. ¬ *dbm_entry_val' u i j e*} = {} **if** *e* = (∞ :: _ *DBMEntry*)
using *that* **by** *auto*

lemma *and_entry_sound*:
u ⊢$_{v,n}$ *and_entry a b e M* **if** *dbm_entry_val' u a b e u* ⊢$_{v,n}$ *M*
using *that* **unfolding** *DBM_val_bounded_def*
by (*cases a*; *cases b*; *auto simp*: *le_n_iff v_is_id*(*1*) *min_def v_0 and_entry_def*)

lemma *DBM_val_bounded_mono*:
u ⊢$_{v,n}$ *M* **if** *u* ⊢$_{v,n}$ *M'* ∀ *i* ≤ *n*. ∀ *j* ≤ *n*. *M' i j* ≤ *M i j*
using *that* **unfolding** *DBM_val_bounded_def*
apply (*safe*; *clarsimp simp*: *le_n_iff v_is_id*(*1*) *DBM.less_eq*[*symmetric*])
apply *force*
apply (*blast intro*: *dbm_entry_val_mono*)+
done

lemma *and_entry_entry*:
dbm_entry_val' u a b e **if** *u* ⊢$_{v,n}$ *and_entry a b e M a* ≤ *n b* ≤ *n a* > *0* ∨ *b* > *0*
proof −
from *that* **have** *dbm_entry_val' u a b* (*min* (*M a b*) *e*)
unfolding *DBM_val_bounded_def* **by** (*fastforce simp*: *le_n_iff v_is_id*(*1*) *and_entry_def*)
then show *?thesis*
by (*auto intro*: *dbm_entry_val_mono*)
qed

lemma *and_entry_correct*:
[*and_entry a b e M*]$_{v,n}$ = [*M*]$_{v,n}$ ∩ {*u*. *dbm_entry_val' u a b e*}
if *a* ≤ *n b* ≤ *n a* > *0* ∨ *b* > *0*

unfolding *DBM_zone_repr_def* **using** *that*
by (*blast intro*: *and_entry_entry and_entry_sound DBM_val_bounded_mono and_entry_mono*)

lemma *dbm_list_Int_entry_iff_map*:
dbm_list xs ∩ {*u*. *dbm_entry_val′ u i j e*} = *dbm_list* (*map* (λ *m*. *and_entry i j e m*) *xs*)
if *i* ≤ *n j* ≤ *n i* > *0* ∨ *j* > *0*
unfolding *dbm_entry_val′_def*
by (*induction xs*;
simp add: *and_entry_correct*[*OF that*, *symmetric*, *unfolded dbm_entry_val′_def*] *Int_Un_distrib2*
)

context
fixes *m* :: *nat* ⇒ *nat* ⇒ (′*a* :: {*time*}) *DBMEntry*
assumes *Le 0* ≼ *m 0 0*
begin

private lemma *A*:
− ([*m*]*v*,*n*) =
(⋃ (*i*, *j*) ∈ {(*i*, *j*). *i* > *0* ∧ *j* > *0* ∧ *i* ≤ *n* ∧ *j* ≤ *n*}.
{*u*. ¬ *dbm_entry_val u* (*Some i*) (*Some j*) (*m i j*)})
∪ (⋃ *i* ∈ {*i*. *i* > *0* ∧ *i* ≤ *n*}. {*u*. ¬ *dbm_entry_val u* (*Some i*) *None* (*m i 0*)})
∪ (⋃ *j* ∈ {*i*. *i* > *0* ∧ *i* ≤ *n*}. {*u*. ¬ *dbm_entry_val u None* (*Some j*) (*m 0 j*)})
unfolding *DBM_zone_repr_def*
apply *safe*
subgoal for *u*
unfolding *DBM_val_bounded_def*
apply (*intro conjI impI allI*)
subgoal
by (*rule* ‹*Le 0* ≼ *m 0 0*›)
subgoal for *c*
by (*auto simp*: *le_n_iff v_is_id*(*1*))
subgoal for *c*
by (*auto simp*: *le_n_iff v_is_id*(*1*))
subgoal for *c1 c2*
by (*auto elim*: *allE*[**where** *x* = *c1*] *simp*: *le_n_iff v_is_id*(*1*))
done
unfolding *DBM_val_bounded_def* **by** (*simp add*: *le_n_iff v_is_id*(*1*))+

private lemma *B*:
S ∩ − ([*m*]*v*,*n*) =
(⋃ (*i*, *j*) ∈ {(*i*, *j*). *i* > *0* ∧ *j* > *0* ∧ *i* ≤ *n* ∧ *j* ≤ *n*}.
S ∩ {*u*. ¬ *dbm_entry_val u* (*Some i*) (*Some j*) (*m i j*)})
∪ (⋃ *i* ∈ {*i*. *i* > *0* ∧ *i* ≤ *n*}. *S* ∩ {*u*. ¬ *dbm_entry_val u* (*Some i*) *None* (*m i 0*)})
∪ (⋃ *j* ∈ {*i*. *i* > *0* ∧ *i* ≤ *n*}. *S* ∩ {*u*. ¬ *dbm_entry_val u None* (*Some j*) (*m 0 j*)})
by (*subst A*) *auto*

private lemma *UNION_cong*:
(⋃ *x* ∈ *S*. *f x*) = (⋃ *x* ∈ *T*. *g x*) **if** *S* = *T* ⋀ *x*. *x* ∈ *T* ⟹ *f x* = *g x*
by (*simp add*: *that*)

private lemma *1*:
S ∩ − ([*m*]*v*,*n*) =
(⋃ (*i*, *j*) ∈ {(*i*, *j*). (*i* > *0* ∨ *j* > *0*) ∧ *i* ≤ *n* ∧ *j* ≤ *n*}. *S* ∩ {*u*. ¬ *dbm_entry_val′ u i j* (*m i j*)})

proof −
  have ∗: $\{(i, j).\ (0 < i \vee 0 < j) \wedge i \leq n \wedge j \leq n\}$
  $= \{(i, j).\ 0 < i \wedge 0 < j \wedge i \leq n \wedge j \leq n\}$
  $\cup \{(i, j).\ 0 < i \wedge 0 = j \wedge i \leq n \wedge j \leq n\}$
  $\cup \{(i, j).\ 0 = i \wedge 0 < j \wedge i \leq n \wedge j \leq n\}$
    by *auto*
  show *?thesis*
    by (*simp only*: *B UN_Un* ∗) (*intro arg_cong2*[**where** $f = (\cup)$] *UNION_cong*; *force*)
qed

private lemma *UNION_remove*:
  $(\bigcup\ x \in S.\ f\ x) = (\bigcup\ x \in T.\ g\ x)$
  if $T \subseteq S$ $\bigwedge x.\ x \in T \Longrightarrow f\ x = g\ x$ $\bigwedge x.\ x \in S - T \Longrightarrow f\ x = \{\}$
  using *that* **by** *fastforce*

private lemma *2*:
  $(\bigcup (i, j) \in \{(i, j).(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}.\ S \cap \{u.\ \neg\ dbm_entry_val'\ u\ i\ j\ (m\ i\ j)\})$
 $= (\bigcup (i, j) \in \{(i, j).(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty\}.$
    $S \cap \{u.\ dbm_entry_val'\ u\ j\ i\ (neg_dbm_entry\ (m\ i\ j))\})$
  apply (*rule UNION_remove*)
    apply *force*
  subgoal for *x*
    by (*cases x*; *simp add*: *neg_entry'*[*simplified*])
  by *auto*

lemma *dbm_list_subtract*:
  $dbm_list\ xs \cap - ([m]_{v,n}) = dbm_list\ (dbm_minus\ n\ xs\ m)$
proof −
  have ∗:
    set $[(i, j).\ i \leftarrow [0..<Suc\ n], j \leftarrow [0..<Suc\ n], (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty]$
  $= \{(i, j).(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty\}$
    by (*auto simp del*: *upt.upt_Suc*)
  show *?thesis*
    unfolding *dbm_minus_def*
    apply (*subst set_concat*)
    apply (*subst set_map*)
    apply (*subst* ∗)
    apply (*subst 1*, *subst 2*)
    apply (*subst UN_UN_flatten*)
    apply (*subst UN_simps*)
    apply (*rule UNION_cong*[*OF HOL.refl*])
    apply (*simp split del*: *if_split split*: *prod.splits*)
    apply (*subst dbm_list_Int_entry_iff_map*[*simplified*])
      apply *auto*
    done
qed

end — Context for fixed DBM

end — Simplifier setup

lemma *dbm_list_empty_check*:
  $dbm_list\ xs = \{\} \longleftrightarrow list_all\ (\lambda m.\ [m]_{v,n} = \{\})\ xs$

unfolding *list_all_iff* **by** *auto*

lemmas *dbm_list_superset_op* =
dbm_fed_superset_fold''[*OF dbm_list_subtract*[*symmetric*], *unfolded dbm_list_empty_check*]

end

context *TA_Start_No_Ceiling*
begin

sublocale *dbm*: *Default_Nat_Clock_Numbering n v*
by *unfold_locales* (*auto simp*: *v_def*)

end

Down

Auxiliary lemma *dbm_entry_le_iff*:
$Le\ a \leq Le\ b \longleftrightarrow a \leq b$
$Le\ a \leq Lt\ b \longleftrightarrow a < b$
$Lt\ a \leq Le\ b \longleftrightarrow a \leq b$
$Lt\ a \leq Lt\ b \longleftrightarrow a \leq b$
$0 \leq Le\ a \longleftrightarrow 0 \leq a$
$0 \leq Lt\ a \longleftrightarrow 0 < a$
$Le\ a \leq 0 \longleftrightarrow a \leq 0$
$Lt\ a \leq 0 \longleftrightarrow a \leq 0$
$\infty \leq x \longleftrightarrow x = \infty$
$x \leq \infty \longleftrightarrow True$
proof −
show $\infty \leq x \longleftrightarrow x = \infty$
by (*cases x*; *auto*)
qed (*auto simp*: *any_le_inf DBM.neutral*)

lemma *dbm_entry_lt_iff*:
$Le\ a < Le\ b \longleftrightarrow a < b$
$Le\ a < Lt\ b \longleftrightarrow a < b$
$Lt\ a < Le\ b \longleftrightarrow a \leq b$
$Lt\ a < Lt\ b \longleftrightarrow a < b$
$0 < Le\ a \longleftrightarrow 0 < a$
$0 < Lt\ a \longleftrightarrow 0 < a$
$Le\ a < 0 \longleftrightarrow a < 0$
$Lt\ a < 0 \longleftrightarrow a \leq 0$
$x < \infty \longleftrightarrow x \neq \infty$
$\infty < x \longleftrightarrow False$
by (*auto simp*: *any_le_inf DBM.neutral DBM.less*)

lemmas [*dbm_entry_simps*] = *dbm_entry_le_iff*(*1*−*9*) *dbm_entry_lt_iff*(*1*−*9*)

lemma *Le_le_sum_iff*:
$Le\ (y :: _ :: time) \leq e \longleftrightarrow 0 \leq e + Le\ (-\ y)$
by (*cases e*) (*auto simp*: *DBM.add dbm_entry_le_iff*)

lemma *dense'*:

$\exists\, c{\geq}a.\ c \leq b$ **if** $a \leq b$ **for** $a :: _ :: time$
using *dense* ‹$a \leq b$› **by** *auto*

lemma *aux1*:
$-\ c \leq (a :: _ :: time)$ **if** $a \geq 0\ c \geq 0$
using *that* **using** *dual_order.trans neg_le_0_iff_le* **by** *blast*

lemma *dbm_entries_dense*:
$\exists\, d{\geq}0.\ Le\ (-\ d) \leq l \wedge Le\ (d :: _ :: time) \leq r$ **if** $0 \leq l\ l \leq r$
using *that* **by** (*cases l*; *cases r*; *auto simp*: *dbm_entry_le_iff intro*: *aux1*)

lemma *dbm_entries_dense'_aux*:
$\exists\, d{\geq}0.\ l + Le\ d \geq 0 \wedge 0 \leq r + Le\ (-\ d :: _ :: time)$ **if** $l \leq 0\ l + r \geq 0\ r \geq 0$
proof ((*cases l*; *cases r*), *goal_cases*)
case (*2 x1 x2*)
have $\exists\, d{\geq}0.\ 0 \leq x + d \wedge d < y$ **if** $x \leq 0\ 0 < x + y\ 0 < y$ **for** $x\ y :: 'a$
using *that* **by** (*metis add.right_inverse add_le_cancel_left leD leI linear*)
with *2 that* **show** *?case*
by (*auto simp*: *dbm_entry_le_iff DBM.add*)
next
case (*3 x1*)
have $\exists\, d{\geq}0.\ 0 \leq x + d$ **if** $x \leq 0$ **for** $x :: 'a$
using *that* **by** (*metis add.right_inverse eq_iff neg_0_le_iff_le*)
with *that 3* **show** *?case*
by (*auto simp*: *dbm_entry_le_iff DBM.add*)
next
case (*5 a b*)
have $\exists\, d{\geq}0.\ 0 < x + d \wedge d < y$ **if** $x \leq 0\ 0 < x + y$ **for** $x\ y :: 'a$
using *that* **by** (*smt add.right_inverse add_less_cancel_left leD le_less_trans linear neg_0_le_iff_le time_class.dense*)
with *5 that* **show** *?case*
by (*auto simp*: *dbm_entry_le_iff DBM.add*)
next
case (*6 x2*)
have $\exists\, d{\geq}0.\ 0 < x + d$ **if** $x \leq 0$ **for** $x :: 'a$
by (*metis add.inverse_neutral add_minus_cancel add_strict_increasing2 eq_iff less_le less_minus_iff non_trivial_neg not_less_iff_gr_or_eq*)
with *that 6* **show** *?case*
by (*auto simp*: *dbm_entry_le_iff DBM.add*)
qed (*use that* **in** ‹*auto simp*: *dbm_entry_le_iff DBM.add*›)

lemma *dbm_entries_dense'*:
$\exists\, d{\geq}0.\ l + Le\ d \geq 0 \wedge 0 \leq r + Le\ (-\ d :: _ :: time)$ **if** $l \leq 0\ l + r \geq 0$
proof −
from *that* **have** $r \geq 0$
by (*meson add_decreasing order_refl order_trans*)
with *that* **show** *?thesis*
by (*rule dbm_entries_dense'_aux*)
qed

lemma (**in** *time*) *non_trivial_pos*: $\exists\ x.\ x > 0$
by (*meson leD le_less_linear neg_le_0_iff_le non_trivial_neg*)

lemma *dbm_entries_dense_pos*:

```
  ∃ d>0. Le (d :: _ :: time) ≤ e if e > 0
  supply dbm_entry_simps[simp]
proof (cases e)
case (Le d)
  with that show ?thesis
    by auto
next
  case prems: (Lt d)
  with that have d > 0
    by auto
  from dense[OF this] obtain z where z > 0 z < d
    by auto
  then show ?thesis
    by (auto simp: prems)
next
  case prems: INF
  obtain d :: 'a where d > 0
    by atomize_elim (rule non_trivial_pos)
  then show ?thesis
    by (auto simp: prems)
qed

lemma le_minus_iff:
  − x ≤ (y :: _ :: time) ⟷ 0 ≤ y + x
  by (metis add.commute add.right_inverse add_le_cancel_left)

lemma lt_minus_iff:
  − x < (y :: _ :: time) ⟷ 0 < y + x
  by (metis add.commute add_less_cancel_right neg_eq_iff_add_eq_0)

context Default_Nat_Clock_Numbering
begin

lemma DBM_val_bounded_alt_def1:
  u ⊢v,n m ≡
    Le 0 ≼ m 0 0 ∧
    (∀ c. c > 0 ∧ c ≤ n ⟶
        dbm_entry_val u None (Some c) (m 0 c) ∧
        dbm_entry_val u (Some c) None (m c 0)) ∧
    (∀ c1 c2. c1 > 0 ∧ c1 ≤ n ∧ c2 > 0 ∧ c2 ≤ n ⟶ dbm_entry_val u (Some c1) (Some c2)
(m c1 c2))
  unfolding DBM_val_bounded_def by (rule eq_reflection) (auto simp: v_id le_n_iff)

lemma DBM_val_bounded_alt_def2:
  u ⊢v,n m ≡
    Le 0 ≤ m 0 0 ∧
    (∀ c1 c2. (c1 ≠ 0 ∨ c2 ≠ 0) ∧ c1 ≤ n ∧ c2 ≤ n ⟶ dbm_entry_val' u c1 c2 (m c1 c2))
  unfolding DBM_val_bounded_alt_def1 dbm_entry_val'_def DBM.less_eq
  by (rule eq_reflection; clarsimp; safe; blast)

lemma DBM_val_bounded_altI:
  assumes
    Le 0 ≤ m 0 0
    ⋀ c1 c2. (c1 ≠ 0 ∨ c2 ≠ 0) ∧ c1 ≤ n ∧ c2 ≤ n ⟹ dbm_entry_val' u c1 c2 (m c1 c2)
```

```
  shows
    u ∈ ⟦m⟧
  unfolding DBM_zone_repr_def DBM_val_bounded_alt_def2 using assms by auto

lemma dbm_entry_val'_delay1:
  dbm_entry_val' u c1 c2 (m c1 c2) if dbm_entry_val' (u ⊕ d) c1 c2 (m c1 c2) d ≥ 0 c1 > 0
  using that unfolding dbm_entry_val'_def
  by (cases m c1 c2)
     (auto 0 2
        dest: add_strict_increasing2 add_increasing intro!: dbm_entry_val.intros
        simp: cval_add_def
     )

lemma dbm_entry_val'_delay2:
  dbm_entry_val' u (0 :: nat) c2 (m c1 c2) if
  dbm_entry_val' (u ⊕ d) c1 c2 (m c1 c2) d ≥ 0
  c1 > 0 c2 > 0 c1 ≤ n c2 ≤ n
  ∀ c ≤ n. c > 0 ⟶ u c ≥ 0
proof −
  have − u c2 ≤ da
    if 0 ≤ d
       0 < c1
       0 < c2
       c1 ≤ n
       c2 ≤ n
       ∀ c≤n. 0 < c ⟶ 0 ≤ u c
       m c1 c2 = Le da
       u c1 − u c2 ≤ da
    for da :: 'a
    using that by (auto simp: algebra_simps le_minus_iff)
  moreover have − u c2 < da
    if 0 ≤ d
       0 < c1
       0 < c2
       c1 ≤ n
       c2 ≤ n
       ∀ c≤n. 0 < c ⟶ 0 ≤ u c
       m c1 c2 = Lt da
       u c1 − u c2 < da
    for da :: 'a
    using that by (auto simp: algebra_simps lt_minus_iff intro: dual_order.strict_trans2)
  ultimately show ?thesis
    using that  unfolding dbm_entry_val'_def
    by (auto elim!: dbm_entry_val.cases intro!: dbm_entry_val.intros simp: cval_add_def)
qed

lemma dbm_entry_val'_nonneg_bound:
  dbm_entry_val' u (0 :: nat) c (Le 0) if u c ≥ 0 c > 0
  using that unfolding dbm_entry_val'_def by auto

lemma neg_diag_empty_spec:
  ⟦M⟧ = {} if i ≤ n M i i < 0
  using that by (meson neg_diag_empty v_is_id(1))
```

lemma *in_DBM_D*:
  dbm_entry_val′ u c1 c2 (M c1 c2) **if** $u \in [\![M]\!]$ $c1 \neq 0 \vee c2 \neq 0$ $c1 \leq n$ $c2 \leq n$
  using *that* **unfolding** *zone_time_pre_def DBM_zone_repr_def DBM_val_bounded_alt_def2*
by *auto*

context
  fixes $M :: ('t::time)\ DBM$
  assumes $[\![M]\!] \neq \{\}$
begin

lemma *non_empty_diag_0_0*: $M\ 0\ 0 \geq 0$
  using ‹$[\![M]\!] \neq \{\}$› *neg_diag_empty_spec*[*of 0 M*] *leI* **by** *auto*

lemma *M_k_0*: $M\ k\ 0 \geq 0$ **if** $\forall\ u \in [\![M]\!].\ \forall\ c \leq n.\ c > 0 \longrightarrow u\ c \geq 0\ k \leq n$
proof (*cases* $k = 0$)
  case *True* **with** *non_empty_diag_0_0* **show** *?thesis*
    by *auto*
next
  case *False*
  from ‹$[\![M]\!] \neq \{\}$› **obtain** u **where** $u \in [\![M]\!]$
    by *auto*
  with *False that*(*1*) ‹$k \leq n$› **have** $u\ k \geq 0$
    by *auto*
  from ‹$u \in _$› ‹$k \neq 0$› ‹$k \leq n$› **have** *dbm_entry_val′ u k 0 (M k 0)*
    unfolding *DBM_zone_repr_def DBM_val_bounded_alt_def2* **by** *auto*
  with ‹$k \neq 0$› **have** *Le* $(u\ k) \leq M\ k\ 0$
    by (*simp add*: *dbm_entry_val′_iff_bounded*)
  with ‹$u\ k \geq 0$› **show** $M\ k\ 0 \geq 0$
    by (*cases M k 0*) (*auto simp*: *dbm_entry_le_iff*)
qed

lemma *non_empty_cycle_free*:
  cycle_free M n
  using ‹$[\![M]\!] \neq \{\}$› *non_empty_cycle_free v_is_id*(*1*) **by** *blast*

lemma *canonical_saturated_2*:
  assumes *Le* $r \leq M\ 0\ c$
    and *Le* $(-\ r) \leq M\ c\ 0$
    and *cycle_free M n*
    and *canonical M n*
    and $c \leq n$
    and $c > 0$
  obtains u **where** $u \in [\![M]\!]$ $u\ c = -\ r$
  using *assms v_0* **by** (*auto simp*: *v_is_id intro*: *canonical_saturated_2*[*of r M v c n*])

lemma *M_0_k*: $M\ 0\ k \leq 0$
  if *canonical M n* $M\ 0\ 0 \leq 0$ $\forall\ u \in [\![M]\!].\ \forall\ c \leq n.\ c > 0 \longrightarrow u\ c \geq 0\ k \leq n$
proof (*cases* $k = 0$)
  case *True*
  with ‹$M\ 0\ 0 \leq 0$› **show** *?thesis*
    by *auto*
next
  case *False*
  show *?thesis*

```
  proof (rule ccontr)
    assume ¬ M 0 k ≤ 0
    then have M 0 k > 0
      by auto
    from that(3) ‹k ≤ n› have M k 0 ≥ 0
      by (rule M_k_0)
    from ‹M 0 k > 0› obtain d where
      Le d ≤ M 0 k d > 0
      by (rule dbm_entries_dense_pos[elim_format]) auto
    with ‹M k 0 ≥ 0› have Le (−d) ≤ M k 0
      by (auto simp: dbm_entry_le_iff intro: order.trans[rotated])
    with ‹canonical M n› ‹Le d ≤ M 0 k› obtain u where
      u ∈ ⟦M⟧ u k = −d
      using v_0 False ‹k ≤ n›
      by − (rule canonical_saturated_2[of d], auto simp: non_empty_cycle_free)
    with ‹d > 0› that(3) False ‹k ≤ n› show False
      by fastforce
  qed
qed

end

end

Definition  definition
  down :: nat ⇒ ('t::linordered_cancel_ab_monoid_add) DBM ⇒ 't DBM
where
  down n M ≡
    λ i j. if i = 0 ∧ j > 0 then Min ({Le 0} ∪ {M k j | k. 1 ≤ k ∧ k ≤ n}) else M i j

Correctness  context Default_Nat_Clock_Numbering
begin

sublocale Alpha_defs {1..n} .

context
  fixes M :: ('t::time) DBM
begin

lemma down_complete: u ∈ ⟦down n M⟧ if u ∈ ⟦M⟧↓ ∀ c ≤ n. c > 0 ⟶ u c ≥ 0
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  with ‹u ∈ _› show ?case
    unfolding down_def zone_time_pre_def by (auto intro: non_empty_diag_0_0 simp: neutral[symmetric])
next
  case prems: (2 c1 c2)
  then consider c1 > 0 | c1 = 0 c2 > 0
    by auto
  then show ?case
  proof cases
    case 1
    with prems ‹u ∈ _› show ?thesis
```

```
      unfolding zone_time_pre_def down_def by (auto intro: dbm_entry_val'_delay1 dest:
in_DBM_D)
  next
    case 2
    from ‹u ∈ [[M]]↓› obtain d where d: 0 ≤ d u ⊕ d ∈ [[M]]
      unfolding zone_time_pre_def by auto
    let ?e = Min ({Le 0} ∪ {M k c2 |k. 1 ≤ k ∧ k ≤ n})
    have ?e ∈ {Le 0} ∪ {M k c2 |k. 1 ≤ k ∧ k ≤ n}
      by (intro Min_in) auto
    then consider ?e = Le 0 | k where ?e = M k c2 k > 0 k ≤ n
      by auto
    then show ?thesis
      using prems that(2) d 2 unfolding down_def
     by cases (auto intro: dbm_entry_val'_delay2 dbm_entry_val'_nonneg_bound in_DBM_D)
  qed
qed

lemma down_sound: u ∈ [[M]]↓ if u ∈ [[down n M]] canonical M n
proof −
  note [simp] = dbm_entry_simps and [intro] = order.trans add_right_mono
  from ‹u ∈ _› non_empty_diag_0_0[of down n M] have Le 0 ≤ M 0 0
    by (auto simp: down_def neutral)
  note ∗ = in_DBM_D[OF ‹u ∈ _›]
  define l where l = Min ({M 0 c + Le (u c)   | c. 0 < c ∧ c ≤ n} ∪ {Le 0})
    — maximum current violation of the future bounds
  define r where r = Min ({M c 0 + Le (− u c) | c. 0 < c ∧ c ≤ n} ∪ {∞})
    — slack for shifting upwards
  have 0 ≤ l + r l ≤ 0
  proof −
    have
      l ∈ {M 0 c + Le (u c)   | c. 0 < c ∧ c ≤ n} ∪ {Le 0}
      r ∈ {M c 0 + Le (− u c) | c. 0 < c ∧ c ≤ n} ∪ {∞}
      unfolding l_def r_def by (intro Min_in; simp)+
    from ‹l ∈ _› show l ≤ 0
      unfolding l_def by (auto intro: Min_le simp: DBM.neutral)
    from ‹l ∈ _› ‹r ∈ _› show 0 ≤ l + r
    proof (safe, goal_cases)
      case prems: (1 c1 c2)
      with ‹u ∈ _› have Le (u c2 − u c1) ≤ M c2 c1
        by (auto 0 2 dest: in_DBM_D simp: dbm_entry_val'_iff_bounded down_def)
      also from prems ‹canonical M n› have M c2 0 + M 0 c1 ≥ M c2 c1
        by auto
      finally have 0 ≤ M c2 0 + M 0 c1 + (Le (u c1) + Le (− u c2))
        by (simp add: DBM.add Le_le_sum_iff)
      then show ?case
        by (simp add: algebra_simps)
    next
      case (3 c)
      with ‹u ∈ _› have Le (u c) ≤ M c 0
        by (auto 0 2 dest: in_DBM_D simp: dbm_entry_val'_iff_bounded down_def)
      then show ?case
        by (auto simp: DBM.add Le_le_sum_iff)
    qed auto
  qed
```

```
  from dbm_entries_dense'[OF this(2,1)] obtain d where
    d ≥ 0 0 ≤ l + Le d 0 ≤ r + Le (− d)
    by auto
  have u ⊕ d ∈ ⟦M⟧
  proof (rule DBM_val_bounded_altI, goal_cases)
    case 1
    from ‹Le 0 ≤ M 0 0› show ?case .
  next
    case (2 c1 c2)
    with ∗ have ∗∗: dbm_entry_val' u c1 c2 (down n M c1 c2)
      by auto
    from 2 consider
      c1 ≤ n c2 ≤ n c1 > 0 c2 > 0
      | c1 = 0 c2 ≤ n c2 > 0 | c2 = 0 c1 ≤ n c1 > 0
      by auto
    then show ?case
    proof cases
      case 1
      then show ?thesis
        using ∗∗ unfolding down_def by (auto intro: dbm_entry_val'_diff_shift)
    next
      case 2
      then have l ≤ (M 0 c2 + Le (u c2))
        unfolding l_def by (auto intro: Min_le)
      with ‹0 ≤ l + Le d› have 0 ≤ M 0 c2 + Le (u c2) + Le d
        by auto
      with 2 show ?thesis
        unfolding down_def dbm_entry_val'_def
        by (cases M 0 c2)
          (auto 4 3 simp: cval_add_def DBM.add algebra_simps lt_minus_iff le_minus_iff)
    next
      case 3
      then have r ≤ M c1 0 + Le (− u c1)
        unfolding r_def by (auto intro: Min_le)
      with ‹0 ≤ r + Le (− d)› have 0 ≤ M c1 0 + Le (− u c1) + Le ( −d)
        by auto
      with 3 ∗∗ show ?thesis
        unfolding down_def dbm_entry_val'_def
        by (auto elim!: dbm_entry_val.cases simp: cval_add_def algebra_simps DBM.add)
    qed
  qed
  with ‹d ≥ 0› show ?thesis
    unfolding zone_time_pre_def cval_add_def by auto
qed

lemma down_canonical:
  canonical (down n M) n
  if assms: canonical M n ⟦M⟧ ≠ {} ∀ u ∈ ⟦M⟧. ∀ c ≤ n. c > 0 ⟶ u c ≥ 0 M 0 0 ≤ 0
proof −
  from non_empty_diag_0_0[OF ‹⟦M⟧ ≠ {}›] have M 0 0 ≥ 0 .
  with ‹M 0 0 ≤ 0› have M 0 0 = 0
    by auto
  note M_0_k = M_0_k[OF that(2,1,4,3)] and M_k_0 = M_k_0[OF that(2,3)]
  have Suc_0_le_iff: Suc 0 ≤ x ⟷ 0 < x for x
```

```
  by auto
 define S where S j = Min ({Le 0} ∪ {M k j |k. 1 ≤ k ∧ k ≤ n}) for j
 { fix j :: nat
  consider (0) S j = 0 ∀ i. 1 ≤ i ∧ i ≤ n ⟶ M i j ≥ 0
   | (entry) i where
     S j = M i j 0 < i i ≤ n M i j ≤ 0 ∀ k. 1 ≤ k ∧ k ≤ n ⟶ M i j ≤ M k j
   unfolding S_def neutral
   using Min_in[of {Le 0} ∪ {M k j |k. 1 ≤ k ∧ k ≤ n}]
   using Min_le[of {Le 0} ∪ {M k j |k. 1 ≤ k ∧ k ≤ n}]
   by (simp, safe) auto
 } note S_cases = this
 show ?thesis
  apply (intro allI impI; elim conjE)
  unfolding down_def S_def[symmetric]
  apply clarsimp
  apply safe
  subgoal premises prems for i j k
   using ‹M 0 0 ≥ 0› by (blast intro: add_increasing)
  subgoal for i j k
   using ‹canonical M n›
   by (cases rule: S_cases[of k]; cases rule: S_cases[of j])
     (auto intro: order.trans simp: Suc_0_le_iff)
  subgoal premises prems for i j k
  proof −
   from prems have M 0 k ≤ S k
   apply (cases rule: S_cases[of k])
   subgoal
    using M_0_k[of k] by auto
   subgoal for i'
   using M_0_k ‹canonical M n› by (metis add.left_neutral add_right_mono dual_order.trans)
   done
  from ‹canonical M n› prems have M i k ≤ M i 0 + M 0 k
   by auto
  also from ‹_ ≤ S k› have ... ≤ M i 0 + S k
   by (simp add: add_left_mono)
  finally show ?thesis .
 qed
 subgoal for i j k
  using ‹canonical M n› ‹M 0 0 ≤ 0›
  by (smt M_k_0 S_cases add_increasing Orderings.order.trans)
 apply (use ‹canonical M n› in simp; fail)+
 subgoal for i j k
  using ‹canonical M n› ‹M 0 0 ≤ 0›
  by (smt M_k_0 S_cases add_increasing Orderings.order.trans)
 apply (use ‹canonical M n› in simp_all)
 done
qed

end

end

Free
```

Definition definition
free :: *nat* ⇒ ('*t*::*linordered_cancel_ab_monoid_add*) *DBM* ⇒ *nat* ⇒ '*t DBM*
where
free n M x ≡
λ i j. if i = *x* ∧ *j* ≠ *x then* ∞ *else if i* ≠ *x* ∧ *j* = *x then M i 0 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 ≡ *repair_pair n* (*and_entry a b e M*) *a b*

definition
restrict_zero n M x ≡
let
M1 = *and_entry x 0* (*Le 0*) *M*;
M2 = *and_entry 0 x* (*Le 0*) *M1*
in repair_pair n M2 x 0

definition
pre_reset n M x ≡ *free n* (*restrict_zero n M x*) *x*

definition
pre_reset_list n M r ≡ *fold* (*λ x M. pre_reset n M x*) *r M*

Auxiliary lemma *repair_pair_characteristic*:
assumes *canonical_subs n I M*
and *I* ⊆ {*0..n*}
and *a* ≤ *n b* ≤ *n*
shows *canonical_subs n* (*I* ∪ {*a*,*b*}) (*repair_pair n M a b*) ∨ (∃ *i*≤*n. repair_pair n M a b i i* < *0*)
proof −
from *fwi_characteristic*[*OF assms*(*1*,*2*,*4*)] **have**
canonical_subs n (*I* ∪ {*b*}) (*FWI M n b*) ∨ (∃ *i*≤*n. FWI M n b i i* < *0*)
by *auto*
then show *?thesis*
proof
assume *canonical_subs n* (*I* ∪ {*b*}) (*FWI M n b*)
from *fwi_characteristic*[*OF this* _ ‹*a* ≤ *n*›] *assms*(*2*) ‹*b* ≤ *n*› **show** *?thesis*
unfolding *repair_pair_def* **by** *simp*
next
assume ∃ *i*≤*n. FWI M n b i i* < *0*
then have ∃ *i*≤*n. repair_pair n M a b i i* < *0*
unfolding *repair_pair_def*
apply *safe*
subgoal for *i*
apply (*inst_existentials i*)
apply *assumption*
apply (*frule FWI_mono*[**where** *M* = *FWI M n b* **and** *k* = *a*])
apply *auto*
done
done
then show *?thesis* **..**

qed
qed

lemma *repair_pair_mono*:
assumes $i \leq n$
and $j \leq n$
shows *repair_pair n M a b i j* $\leq$ *M i j*
unfolding *repair_pair_def* **by** (*auto intro*: *FWI_mono assms order.trans*)

context *Default_Nat_Clock_Numbering*
begin

lemmas *FWI_zone_equiv* = *FWI_zone_equiv*[*OF surj_on, symmetric*]

lemma *repair_pair_zone_equiv*:
⟦*repair_pair n M a b*⟧ = ⟦*M*⟧ **if** $a \leq n$ $b \leq n$
using *that* **unfolding** *repair_pair_def* **by** (*simp add*: *FWI_zone_equiv*)

context
fixes *c1 c2 c x* :: *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* ⟷ $e \geq 0$ **if** $c > 0$
using *that* **by** (*cases e*) *auto*

lemma *dbm_entry_val'_inf*: *dbm_entry_val' u c1 c2* ∞ ⟷ *True*
unfolding *dbm_entry_val'_def* **by** *auto*

lemma *dbm_entry_val'_reset_1*:
dbm_entry_val' (*u*(*x* := *d*)) *x c e* ⟷ *dbm_entry_val' u 0 c* (*e* + *Le* (−*d*))
if $d \geq 0$ $c \neq x$ $c > 0$ $x > 0$
using *that* ‹$d \geq 0$› **by** (*cases e*) *auto*

lemma *dbm_entry_val'_reset_2*:
dbm_entry_val' (*u*(*x* := *d*)) *c x e* ⟷ *dbm_entry_val' u c* (*0* :: *nat*) (*e* + *Le d*)
if $d \geq 0$ $c \neq x$ $c > 0$ $x > 0$
using *that* ‹$d \geq 0$› **by** (*cases e*) *auto*

lemma *dbm_entry_val'_reset_2'*:
dbm_entry_val' (*u*(*x* := *d*)) *0 x e* ⟷ *Le* (− *d*) ≤ *e* **if** $d \geq 0$ $x > 0$
using *that* ‹$d \geq 0$› **by** (*cases e*) *auto*

lemma *dbm_entry_val'_reset_3*:
dbm_entry_val' (*u*(*x* := *d*)) *c1 c2 e* ⟷ *dbm_entry_val' u c1 c2 e* **if** $c1 \neq x$ $c2 \neq x$ **for** *e*
using *that* **unfolding** *dbm_entry_val'_def* **by** (*cases e*) *auto*

end

Correctness context
fixes *M* :: (′*t*::*time*) *DBM*
begin

```
lemma free_complete: u(x := d) ∈ ⟦free n M x⟧
  if assms: u ∈ ⟦M⟧ d ≥ 0 x > 0 ∀ c ≤ n. M c c ≥ 0
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  with ‹_ ∈ ⟦M⟧› show ?case
    unfolding free_def by (auto simp: neutral[symmetric] intro: non_empty_diag_0_0)
next
  case prems: (2 c1 c2)
  then have c1 ≤ n c2 ≤ n
    by auto
  note [simp] = dbm_entry_simps
  have ∗: Le (u c1) ≤ M c1 0 + Le d if c1 > 0
  proof −
    from ‹_ ∈ ⟦M⟧› ‹c1 > 0› ‹c1 ≤ n› have Le (u c1) ≤ M c1 0
      by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
    with ‹d ≥ 0› show ?thesis
      by (simp add: algebra_simps add_increasing)
  qed
  have dbm_entry_val' (u(x := d)) c1 x (M c1 0) if c1 ≠ x
  proof (cases c1 = 0)
    case True
    with that show ?thesis
     using assms(4) ‹d ≥ 0› by (auto intro: order.trans[rotated] simp: dbm_entry_val'_reset_2')
  next
    case False
    with that ‹x > 0› show ?thesis
     by (subst dbm_entry_val'_reset_2[OF ‹d ≥ 0›]) (auto simp: dbm_entry_val'_iff_bounded
∗)
  qed
  with prems in_DBM_D[OF ‹_ ∈ ⟦M⟧›] that(4) show ?case
   by (auto simp: free_def dbm_entry_val'_diag_iff dbm_entry_val'_inf dbm_entry_val'_reset_3)
qed

lemma free_sound: ∃ d ≥ 0. u(x := d) ∈ ⟦M⟧ u x ≥ 0
  if assms: u ∈ ⟦free n M x⟧ x > 0 x ≤ n canonical M n M 0 x ≤ 0 M 0 0 ≤ 0
proof −
  define l where l = Min ({M c x + Le (− u c) | c. 0 < c ∧ c ≤ n ∧ c ≠ x} ∪ {M 0 x})
  define r where r = Min ({M x c + Le (u c)   | c. 0 < c ∧ c ≤ n ∧ c ≠ x} ∪ {M x 0})
  from non_empty_diag_0_0 ‹u ∈ _› ‹x > 0› have 0 ≤ M 0 0
    unfolding free_def by fastforce
  note [simp] = dbm_entry_simps and [intro] = order.trans add_right_mono
  have 0 ≤ l + r l ≤ 0
  proof −
    have
      l ∈ {M c x + Le (− u c)   | c. 0 < c ∧ c ≤ n ∧ c ≠ x} ∪ {M 0 x}
      r ∈ {M x c + Le (u c)     | c. 0 < c ∧ c ≤ n ∧ c ≠ x} ∪ {M x 0}
      unfolding l_def r_def by (intro Min_in; simp)+
    from ‹l ∈ _› ‹M 0 x ≤ 0› show l ≤ 0
      unfolding l_def by − (rule order.trans[rotated], auto intro: Min_le simp: DBM.neutral)
    from ‹l ∈ _› ‹r ∈ _› show 0 ≤ l + r
    proof (safe, goal_cases)
      case prems: (1 c1 c2)
      with ‹canonical M n› ‹x ≤ n› have M c1 x + M x c2 ≥ M c1 c2
        by auto
```

```
    from prems ‹u ∈ _› have Le (u c1 − u c2) ≤ M c1 c2
      unfolding free_def by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
    with ‹M c1 c2 ≤ M c1 x + M x c2› have Le (u c1 − u c2) ≤ M c1 x + M x c2
      by auto
    then have 0 ≤ M c1 x + M x c2 + (Le (u c2) + Le (− u c1))
      by (simp add: DBM.add Le_le_sum_iff)
    then show ?case
      by (simp add: algebra_simps)
  next
    case prems: (2 c)
    from prems ‹u ∈ _› have Le (u c) ≤ M c 0
      unfolding free_def by (auto 0 2 simp: dbm_entry_val'_iff_bounded dest: in_DBM_D)
    also from prems ‹canonical M n› ‹x ≤ n› have ... ≤ M c x + M x 0
      by auto
    finally show ?case
      by (simp add: algebra_simps Le_le_sum_iff)
  next
    case prems: (3 c)
    with ‹u ∈ _› ‹x > 0› have Le (− u c) ≤ M 0 c
      unfolding free_def by (auto simp: dbm_entry_val'_iff_bounded dest: in_DBM_D[of _
_ 0 c])
    also from prems ‹canonical M n› ‹x ≤ n› have ... ≤ M 0 x + M x c
      by auto
    finally show ?case
      by (simp add: algebra_simps Le_le_sum_iff)
  next
    case 4
    from ‹0 ≤ M 0 0› ‹canonical M n› ‹x ≤ n› show ?case
      by auto
  qed
qed
from dbm_entries_dense'[OF this(2,1)] obtain d where
  d ≥ 0 0 ≤ l + Le d 0 ≤ r + Le (− d)
  by auto
have u(x := d) ∈ ⟦M⟧
proof (rule DBM_val_bounded_altI, goal_cases)
  case 1
  from ‹0 ≤ M 0 0› show ?case unfolding DBM.neutral .
next
  case prems: (2 c1 c2)
  then have ∗∗: dbm_entry_val' u c1 c2 (free n M x c1 c2)
    by (auto intro: in_DBM_D[OF ‹u ∈ _›])
  show ?case
  proof −
    have Le (d − u c2) ≤ M x c2
      if 0 < x c1 = x c2 ≠ x x ≤ n c2 ≤ n 0 < c2
    proof −
      from that have r ≤ M x c2 + Le (u c2)
        unfolding r_def by (intro Min_le) auto
      with ‹0 ≤ r + _› have 0 ≤ M x c2 + Le (u c2) + Le (− d)
        by auto
      moreover have Le (d − u c2) ≤ M x c2 ⟷ 0 ≤ M x c2 + Le (u c2) + Le (− d)
        by (cases M x c2) (auto simp: DBM.add algebra_simps)
      ultimately show Le (d − u c2) ≤ M x c2
```

```
      by simp
    qed
    moreover have Le d ≤ M x 0
      if c1 = x 0 < x x ≤ n c2 = 0
    proof −
      from ‹0 ≤ r + _› have Le d ≤ r
        by (simp add: Le_le_sum_iff)
      also have r ≤ M x 0
        unfolding r_def by auto
      finally show Le d ≤ M x 0 .
    qed
    moreover have Le (u c1 − d) ≤ M c1 x
      if 0 < x c1 ≤ n x ≤ n c1 ≠ x c2 = x 0 < c1 Le (u c1 − u x) ≤ M c1 0
    proof −
      from that have l ≤ M c1 x + Le (− u c1)
        unfolding l_def by (intro Min_le) auto
      with ‹0 ≤ l + Le d› have 0 ≤ M c1 x + Le (− u c1) + Le d
        by auto
      moreover have Le (u c1 − d) ≤ M c1 x ⟷ 0 ≤ M c1 x + Le (− u c1) + Le d
        by (cases M c1 x) (auto simp: DBM.add algebra_simps)
      ultimately show Le (u c1 − d) ≤ M c1 x
        by simp
    qed
    moreover have Le (− d) ≤ M 0 x
      if x ≤ n 0 < x c2 = x c1 = 0 Le (− u x) ≤ M 0 0
    proof −
      from ‹0 ≤ l + Le d› have Le (− d) ≤ l
        by (simp add: Le_le_sum_iff)
      also have l ≤ M 0 x
        unfolding l_def by auto
      finally show Le (− d) ≤ M 0 x .
    qed
    ultimately show ?thesis
      using prems ‹x > 0› **
      by (auto simp: dbm_entry_val'_iff_bounded free_def split: if_split_asm)
  qed
 qed
 with ‹d ≥ 0› show ∃ d≥0. u(x := d) ∈ ⟦M⟧
  by auto
 from ‹x > 0› ‹x ≤ n› have dbm_entry_val' u 0 x (free n M x 0 x)
  by (auto intro: in_DBM_D[OF ‹u ∈ _›])
 with ‹0 < x› have Le (− u x) ≤ M 0 0
  by (auto simp: free_def dbm_entry_val'_iff_bounded)
 with ‹M 0 0 ≤ 0› have Le (− u x) ≤ 0
  by blast
 then show 0 ≤ u x
  by auto
qed

lemma free_correct:
 ⟦free n M x⟧ = {u(x := d) | u d. u ∈ ⟦M⟧ ∧ d ≥ 0}
 if x > 0 x ≤ n ∀ c ≤ n. M c c ≥ 0 ∀ u ∈ ⟦M⟧. u x ≥ 0 canonical M n
   M 0 x ≤ 0 M 0 0 ≤ 0
 using that
```

```
  apply safe
  subgoal for u'
    apply (frule free_sound, assumption+)
    apply (frule free_sound(2), assumption+)
    apply (erule exE)
    subgoal for d
      by (inst_existentials u'(x := d) u' x; simp)
    done
  subgoal for u d
    by (auto intro: free_complete)
  done

lemma pre_reset_correct_aux:
  {u. (u(x := (0::'t))) ∈ [[M]]} ∩ {u. u x ≥ 0} = {u(x := d) | u d. u ∈ [[M]] ∧ u x = 0 ∧ d ≥ 0}
  apply safe
  subgoal for u
    by (inst_existentials u(x := (0::'t)) u x) auto
  apply clarsimp
  subgoal for u d
    by (subgoal_tac u = u(x := 0)) auto
  by auto

lemma restrict_zero_correct:
  [[restrict_zero n M x]] = {u. u ∈ [[M]] ∧ u x = 0} if 0 < x x ≤ n
  using that unfolding restrict_zero_def
  by (auto simp: repair_pair_zone_equiv and_entry_correct dbm_entry_val'_iff_bounded
      dbm_entry_simps)

lemma restrict_zero_canonical:
  canonical (restrict_zero n M x) n ∨ check_diag n (uncurry (restrict_zero n M x))
  if canonical M n x ≤ n
proof −
  from ‹x ≤ n› have ∗: {0..n} − {0, x} ∪ {x, 0} = {0..n}
    by auto
  define M1 and M2 where M1 = and_entry x 0 (Le 0) M M2 = and_entry 0 x (Le 0) M1
  from ‹canonical M n› have canonical_subs n {0..n} M
    unfolding canonical_alt_def .
  with ∗ have canonical_subs n ({0..n} − {0, x}) M2
   unfolding and_entry_def M1_M2_def canonical_subs_def by (auto simp: min.coboundedI1)
  from repair_pair_characteristic[OF this, of x 0] ‹x ≤ n› have
    canonical (repair_pair n M2 x 0) n ∨ check_diag n (uncurry (repair_pair n M2 x 0))
    unfolding canonical_alt_def check_diag_def ∗ neutral by auto
  then show ?thesis
    unfolding restrict_zero_def M1_M2_def Let_def .
qed

end
```

10.4 Structural Properties

```
lemma free_canonical:
  canonical (free n M x) n if canonical M n M x x ≥ 0
  unfolding free_def using that by (auto simp: add_increasing2 any_le_inf)
```

lemma *free_diag*:
free n M x i i = M i i
unfolding *free_def* **by** *auto*

lemma *check_diag_free*:
check_diag n (*uncurry* (*free n M x*)) **if** *check_diag n* (*uncurry M*)
using *that* **unfolding** *check_diag_def* **by** (*auto simp*: *free_diag*)

lemma
$\forall\, i{\le}n.$ (*free n M x*) *i i* $\le$ *0* **if** $\forall\, i{\le}n.$ *M i i* $\le$ *0*
using *that* **by** (*auto simp*: *free_diag*)

lemma *canonical_nonneg_diag_non_empty*:
assumes *canonical M n* $\forall\, i{\le}n.$ *0* $\le$ *M i i*
shows $[M]_{v,n} \neq \{\}$
using *v_0* **by** (*intro canonical_nonneg_diag_non_empty*[*OF assms*]) *force*

lemma *V_structuralI*:
$[\![M]\!] \subseteq V$ **if** $\forall\ i \le n.\ i > 0 \longrightarrow M\ 0\ i \le 0$
using *that*
unfolding *V_def*
proof *safe*
fix *u i* **assume** $u \in [\![M]\!]$ $i \in \{1..n\}$
then have *Suc 0* $\le$ *i i* $\le$ *n* **by** *simp*+
from *in_DBM_D*[*OF* ‹*u* ∈ _›, *of 0 i*] ‹_ ≤ *i*› ‹*i* ≤ *n*› **have** *dbm_entry_val′ u 0 i* (*M 0 i*)
by *auto*
with ‹_ ≤ *i*› ‹*i* ≤ *n*› **have** *Le* (− *u i*) ≤ *M 0 i*
by (*auto simp*: *dbm_entry_val′_iff_bounded*)
also from *that* ‹_ ≤ *i*› ‹*i* ≤ *n*› **have** ... ≤ *0*
by *simp*
finally show *0* ≤ *u i*
by (*auto simp*: *dbm_entry_simps*)
qed

lemma *canonical_V_non_empty_iff*:
assumes *canonical M n M 0 0* ≤ *0*
shows $[\![M]\!] \subseteq V \wedge [\![M]\!] \neq \{\} \longleftrightarrow (\forall\ i \le n.\ i > 0 \longrightarrow M\ 0\ i \le 0) \wedge (\forall\ i \le n.\ M\ i\ i \ge 0)$
proof (*safe*, *goal_cases*)
case (*1 u i*)
with ‹*M 0 0* ≤ *0*› **show** *?case*
unfolding *V_def* **by** − (*rule M_0_k*[*OF* _ ‹*canonical M n*›], *auto*)
next
case (*2 x i*)
then show *?case*
using *neg_diag_empty_spec*[*of i M*] **by** *fastforce*
next
case *prems*: (*3 u*)
then show *?case*
by (*auto dest*: *subsetD*[*OF V_structuralI*])
next
case *4*
with *canonical_nonneg_diag_non_empty*[*OF* ‹*canonical M n*›] **show** *?case*
by *simp*
qed

lemma
assumes $(\forall\ i \leq n.\ i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ M\ i\ i \geq 0)\ M\ 0\ 0 \leq 0\ x > 0$
shows $(\forall\ i \leq n.\ i > 0 \longrightarrow \mathit{free}\ n\ M\ x\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ \mathit{free}\ n\ M\ x\ i\ i \geq 0)$
using *assms* **by** (*auto simp*: *free_def*)

lemma
assumes $(\forall\ i \leq n.\ i > 0 \longrightarrow M\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ M\ i\ i \geq 0) \Longrightarrow$
$(\forall\ i \leq n.\ i > 0 \longrightarrow f\ M\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ f\ M\ i\ i \geq 0)$
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* $(\mathit{uncurry}\ M) \Longrightarrow \mathit{check_diag}\ n\ (\mathit{uncurry}\ (f\ M))$
assumes $[\![M]\!] \subseteq V$
shows $[\![f\ M]\!] \subseteq V$
proof (*cases* $[\![M]\!] = \{\}$)
case *True*
then have *check_diag n* (*uncurry M*)
using *canonical_nonneg_diag_non_empty*[*OF* ‹*canonical M n*›] **by** (*force simp*: *neutral check_diag_def*)
then have *check_diag n* (*uncurry* $(f\ M)$)
by (*rule check_diag*)
then have $[\![f\ M]\!] = \{\}$
by (*auto dest*: *neg_diag_empty_spec simp*: *check_diag_def neutral*)
then show *?thesis*
by *auto*
next
case *False*
with ‹$[\![M]\!] \subseteq V$› *canonical_V_non_empty_iff*[*OF* ‹*canonical M n*› ‹$M\ 0\ 0 \leq 0$›] **have**
$(\forall\, i{\leq}n.\ 0 < i \longrightarrow M\ 0\ i \leq 0) \wedge (\forall\, i{\leq}n.\ 0 \leq M\ i\ i)$
by *auto*
then have $(\forall\ i \leq n.\ i > 0 \longrightarrow f\ M\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ f\ M\ i\ i \geq 0)$
by (*rule assms*(*1*))
with ‹*canonical* $(f\ M)$ *n*› **have** $[\![f\ M]\!] \subseteq V \wedge [\![f\ M]\!] \neq \{\}$
using ‹$f\ M\ 0\ 0 \leq 0$› **by** (*subst canonical_V_non_empty_iff*) (*auto simp*: *free_diag*)
then show *?thesis* **..**
qed

lemma
$[\![\mathit{free}\ n\ M\ x]\!] \subseteq V$ **if** *assms*: $x > 0$ *canonical M n* $M\ 0\ 0 \leq 0\ 0 \leq M\ x\ x\ [\![M]\!] \subseteq V$
proof (*cases* $[\![M]\!] = \{\}$)
case *True*
then obtain *i* **where** $M\ i\ i < 0\ i \leq n$
using *canonical_nonneg_diag_non_empty*[*OF* ‹*canonical M n*›] **by** *atomize_elim force*
then have *free n M x i i* < 0
by (*auto simp*: *free_diag*)
with ‹$i \leq n$› **have** $[\![\mathit{free}\ n\ M\ x]\!] = \{\}$
by (*intro neg_diag_empty_spec*)
then show *?thesis*
by *auto*
next
case *False*
with ‹$[\![M]\!] \subseteq V$› *canonical_V_non_empty_iff*[*OF that*(*2*,*3*)] **have**
$(\forall\, i{\leq}n.\ 0 < i \longrightarrow M\ 0\ i \leq 0) \wedge (\forall\, i{\leq}n.\ 0 \leq M\ i\ i)$
by *auto*
with *that* **have** $(\forall\ i \leq n.\ i > 0 \longrightarrow \mathit{free}\ n\ M\ x\ 0\ i \leq 0) \wedge (\forall\ i \leq n.\ \mathit{free}\ n\ M\ x\ i\ i \geq 0)$

```
    by (auto simp: free_def)
  moreover have canonical (free n M x) n
    apply (rule free_canonical)
     apply fact
    apply fact
    done
  ultimately have ⟦free n M x⟧ ⊆ V ∧ ⟦free n M x⟧ ≠ {}
    using ‹M 0 0 ≤ 0› by (subst canonical_V_non_empty_iff) (auto simp: free_diag)
  then show ?thesis ..
qed

lemma
  down n M i i = M i i
  unfolding down_def by auto

lemma
  assumes (∀ i ≤ n. i > 0 ⟶ M 0 i ≤ 0) ∧ (∀ i ≤ n. M i i ≥ 0) M 0 0 ≤ 0 x > 0
  shows (∀ i ≤ n. i > 0 ⟶ down n M 0 i ≤ 0) ∧ (∀ i ≤ n. down n M i i ≥ 0)
  using assms by (auto simp: down_def neutral)

lemma check_diag_empty:
  ⟦M⟧ = {} if check_diag n (uncurry M)
  using check_diag_empty[of n v uncurry M] that v_is_id by auto

lemma restrict_zero_mono:
  restrict_zero n M x i j ≤ M i j if i ≤ n j ≤ n
  unfolding restrict_zero_def
  by simp (rule ‹i ≤ n› ‹j ≤ n› repair_pair_mono and_entry_mono order.trans)+

lemma restrict_zero_diag:
  check_diag n (uncurry (restrict_zero n M x)) if check_diag n (uncurry M)
  using that unfolding check_diag_def neutral[symmetric]
  by (elim exE conjE) (frule restrict_zero_mono[where M = M and x = x], auto)

lemma pre_reset_correct:
  ⟦pre_reset n M x⟧ = {u. (u(x := (0::'t::time))) ∈ ⟦M⟧} ∩ {u. u x ≥ 0}
  if x > 0 x ≤ n canonical M n ∨ check_diag n (uncurry M) M 0 x ≤ 0 M 0 0 ≤ 0
proof −
  have check_diag: ?thesis if A: check_diag n (uncurry (restrict_zero n M x))
  proof −
    from A have check_diag n (uncurry (pre_reset n M x))
      unfolding pre_reset_def by (rule check_diag_free)
    then have ⟦pre_reset n M x⟧ = {}
      by (rule check_diag_empty)
    from A have ⟦restrict_zero n M x⟧ = {}
      by (rule check_diag_empty)
    then have {u. (u(x := (0::'t::time))) ∈ ⟦M⟧} ∩ {u. u x ≥ 0} = {}
      using ‹0 < x› ‹x ≤ n› by (auto simp: restrict_zero_correct)
    with ‹⟦pre_reset n M x⟧ = {}› show ?thesis
      by simp
  qed
  from that(3) show ?thesis
  proof
    assume canonical M n
```

```
    from restrict_zero_canonical[OF ‹canonical M n› ‹x ≤ n›] have
      canonical (restrict_zero n M x) n ∨ check_diag n (uncurry (restrict_zero n M x))
      (is ?A ∨ ?B) .
    then consider ?A ¬ ?B | ?B
      by blast
    then show ?thesis
    proof cases
      case 1
      assume ?A ¬ ?B
      moreover from ‹¬ ?B› have ∀ c≤n. 0 ≤ restrict_zero n M x c c
        unfolding check_diag_def by (auto simp: DBM.neutral)
      moreover have ∀ u∈⟦restrict_zero n M x⟧. 0 ≤ u x
        by (simp add: restrict_zero_correct that)
      moreover from ‹x ≤ n› ‹M 0 x ≤ 0› have restrict_zero n M x 0 x ≤ 0
        by (blast intro: order.trans restrict_zero_mono)
      moreover from ‹x ≤ n› ‹M 0 0 ≤ 0› have restrict_zero n M x 0 0 ≤ 0
        by (blast intro: order.trans restrict_zero_mono)
      ultimately show ?thesis
        using that
        by (auto simp: pre_reset_correct_aux restrict_zero_correct free_correct pre_reset_def)
    next
      assume ?B then show ?thesis
        by (rule check_diag)
    qed
  next
    assume check_diag n (uncurry M)
    then have check_diag n (uncurry (restrict_zero n M x))
      by (rule restrict_zero_diag)
    then show ?thesis
      by (rule check_diag)
  qed
qed

lemma zone_set_pre_Cons:
  zone_set_pre ⟦M⟧ (x # r) = zone_set_pre {u. (u(x := (0::'t::time))) ∈ ⟦M⟧} r
  unfolding zone_set_pre_def by auto

lemma pre_reset_list_Cons:
  pre_reset_list n M (x # r) = pre_reset_list n (pre_reset n M x) r
  unfolding pre_reset_list_def by simp

lemma pre_reset_diag:
  check_diag n (uncurry (pre_reset n M x)) if check_diag n (uncurry M)
  using that unfolding pre_reset_def by (intro check_diag_free restrict_zero_diag)

lemma free_canonical':
  canonical (free n (M :: (_ :: time) DBM) x) n ∨ check_diag n (uncurry (free n M x))
  if canonical M n ∨ check_diag n (uncurry M) x ≤ n
  by (smt check_diag_def check_diag_free dbm_entry_le_iff(5) free_canonical leI
        order_mono_setup.refl order_trans that uncurry_apply
    )

lemma pre_reset_canonical':
  canonical (pre_reset n (M :: (_ :: time) DBM) x) n ∨ check_diag n (uncurry (pre_reset n M
```

```
x))
  if canonical M n ∨ check_diag n (uncurry M) x ≤ n
  using that(1)
proof standard
  assume canonical M n
  with ‹x ≤ n› have
    canonical (restrict_zero n M x) n ∨ check_diag n (uncurry (restrict_zero n M x))
    by (intro restrict_zero_canonical)
  with ‹x ≤ n› show ?thesis
    unfolding pre_reset_def by (intro free_canonical')
next
  assume check_diag n (uncurry M)
  from pre_reset_diag[OF this] show ?thesis ..
qed

lemma pre_reset_list_correct:
  ⟦pre_reset_list n M r⟧ = zone_set_pre ⟦M⟧ r ∩ {u. ∀ x ∈ set r. u x ≥ 0}
  if ∀ x ∈ set r. x > 0 ∧ x ≤ n
    canonical M n ∨ check_diag n (uncurry M) ∀ x ∈ set r. M 0 x ≤ 0 M 0 0 ≤ 0
  using that
  apply (induction r arbitrary: M)
   apply (simp add: zone_set_pre_def pre_reset_list_def)
  subgoal premises prems for x r M
    apply (subst zone_set_pre_Cons)
    apply (subst pre_reset_list_Cons)
    apply (subst prems)
        prefer 5
        apply (subst pre_reset_correct)
            prefer 6
    subgoal
      unfolding zone_set_pre_def by (cases x ∈ set r) auto
    using prems(2−) apply (solves auto)+
    subgoal
      using prems(2−) by (intro pre_reset_canonical'; auto)
    subgoal
      unfolding pre_reset_def free_def using prems(2−)
      by (auto 4 3 intro: order.trans restrict_zero_mono)
    subgoal
      unfolding pre_reset_def free_def using prems(2−)
      by (auto 4 3 intro: order.trans restrict_zero_mono)
    done
  done

end
```

Computes *dbm_list xs* − ⟦*M*⟧ = *dbm_list xs* ∩ (− ⟦*M*⟧) by negating each entry of *M* and intersecting it with each member of *xs*.

```
definition
  dbm_minus_canonical n xs M =
   [and_entry_repair n j i (neg_dbm_entry (M i j)) M'.
     (i, j) ← [(i, j).
        i←[0..<Suc n], j←[0..<Suc n], (i > 0 ∨ j > 0) ∧ i ≤ n ∧ j ≤ n ∧ M i j ≠ ∞],
     M'     ← xs
   ]
```

Same as *dbm_minus_canonical* but filters out empty DBMs.

definition
dbm_minus_canonical_check n xs M =
filter (λ*M*. ¬ *check_diag n* (*uncurry M*)) (*dbm_minus_canonical n xs M*)

Checks whether ⟦*M*⟧ − *dbm_list xs* = {}.

definition
dbm_subset_fed n M xs ≡
let xs = *filter* (λ*M*. ¬ *check_diag n* (*uncurry M*)) *xs in*
list_all (λ *M*. *check_diag n* (*uncurry M*)) (*fold* (λ*m S*. *dbm_minus_canonical n S m*) *xs* [*M*])

definition
dbm_subset_fed_check n M xs ≡
let
xs = *filter* (λ*M*. ¬ *check_diag n* (*uncurry M*)) *xs*;
is_direct_subset = *list_ex* (λ *M′*. *dbm_subset′ n* (*uncurry M*) (*uncurry M′*)) *xs*
in is_direct_subset ∨
list_all (λ*M*. *check_diag n* (*uncurry M*)) (*fold* (λ*m S*. *dbm_minus_canonical_check n S m*) *xs* [*M*])

definition *canonical′ n M* ≡ *canonical M n* ∨ *check_diag n* (*uncurry M*)

lemma *canonical′I*:
canonical′ n (*f M*) **if**
canonical′ n M
canonical M n ⟹ *canonical′ n* (*f M*) *check_diag n* (*uncurry M*) ⟹ *check_diag n* (*uncurry* (*f M*))
using *that* **unfolding** *canonical′_def* **by** *metis*

lemma *check_diag_repair_pair*:
assumes *check_diag n* (*uncurry M*)
shows *check_diag n* (*uncurry* (*repair_pair n M i j*))
using *assms repair_pair_mono*[**where** *M* = *M* **and** *a* = *i* **and** *b* = *j*] **unfolding** *check_diag_def* **by** *force*

lemma *check_diag_and_entry*:
assumes *check_diag n* (*uncurry M*)
shows *check_diag n* (*uncurry* (*and_entry a b e M*))
using *assms* **unfolding** *check_diag_def*
apply (*elim exE*)
subgoal for *i*
using *and_entry_mono*[**where** *M* = *M* **and** *a* = *a* **and** *b* = *b* **and** *e* = *e*, *of i i*] **by** *auto*
done

lemma *canonical′_and_entry_repair*:
canonical′ n (*and_entry_repair n i j e M*) **if** *canonical′ n M i* ≤ *n j* ≤ *n*
using *that*(*1*)
proof (*rule canonical′I*)
assume *canonical M n*
from ‹*i* ≤ *n*› ‹*j* ≤ *n*› **have** ∗: {*0*..*n*} − {*i*, *j*} ∪ {*i*, *j*} = {*0*..*n*}
by *auto*
define *M1* **where** *M1* = *and_entry i j e M*
from ‹*canonical M n*› **have** *canonical_subs n* {*0*..*n*} *M*

unfolding *canonical_alt_def* **.**
with ∗ **have** *canonical_subs n* ({*0..n*} − {*i*, *j*}) *M1*
unfolding *and_entry_def M1_def canonical_subs_def* **by** (*auto simp*: *min.coboundedI1*)
from *repair_pair_characteristic*[*OF this*, *of i j*] ‹*i* ≤ *n*› ‹*j* ≤ *n*› **have**
canonical′ n (*repair_pair n M1 i j*)
unfolding *canonical′_def*
unfolding *canonical_alt_def check_diag_def* ∗ *neutral* **by** *auto*
then show *?thesis*
unfolding *and_entry_repair_def M1_def Let_def* **.**
next
assume *check_diag n* (*uncurry M*)
then show *check_diag n* (*uncurry* (*and_entry_repair n i j e M*))
unfolding *and_entry_repair_def* **by** (*intro check_diag_repair_pair check_diag_and_entry*)
qed

lemma *dbm_minus_canonical_canonical′*:
∀ *M* ∈ *set* (*dbm_minus_canonical n xs m*). *canonical′ n M* **if** ∀ *M* ∈ *set xs*. *canonical′ n M*
using *that* **unfolding** *dbm_minus_canonical_def*
by (*auto split*: *if_split_asm intro*: *canonical′_and_entry_repair*)

lemma *dbm_minus_canonical_check_canonical′*:
∀ *M* ∈ *set* (*dbm_minus_canonical_check n xs m*). *canonical′ n M* **if** ∀ *M* ∈ *set xs*. *canonical′ n M*
using *dbm_minus_canonical_canonical′*[*OF that*] **unfolding** *dbm_minus_canonical_check_def*
by *auto*

10.5 Correctness of *dbm_subset_fed*

Misc **lemma** *list_all_iffI*:
assumes ∀ *x* ∈ *set xs*. ∃ *y* ∈ *set ys*. *P x* ⟷ *Q y*
and ∀ *y* ∈ *set ys*. ∃ *x* ∈ *set xs*. *P x* ⟷ *Q y*
shows *list_all P xs* ⟷ *list_all Q ys*
using *assms* **unfolding** *list_all_def* **by** *blast*

lemma *list_all_iff_list_all2I*:
assumes *list_all2* (λ *x y*. *P x* ⟷ *Q y*) *xs ys*
shows *list_all P xs* ⟷ *list_all Q ys*
using *assms* **by** (*intro list_all_iffI list_all2_set1 list_all2_set2*)

lemma *list_all2_mapI*:
assumes *list_all2* (λ *x y*. *P* (*f x*) (*g y*)) *xs ys*
shows *list_all2 P* (*map f xs*) (*map g ys*)
using *assms* **by** (*simp only*: *list.rel_map*)

context *Default_Nat_Clock_Numbering*
begin

lemma *canonical_empty_zone*:
$[M]_{v,n} = \{\}$ ⟷ (∃ *i*≤*n*. *M i i* < 0) **if** *canonical M n*
using *v_0 that surj_on* **by** (*intro canonical_empty_zone*) *auto*

lemma *check_diag_iff_empty*:
check_diag n (*uncurry M*) ⟷ ⟦*M*⟧ = {} **if** *canonical′ n M*
proof (*safe*, *goal_cases*)

```
  case 2
  show ?case
  proof −
    from that
    show ?thesis
      unfolding canonical'_def
    proof
      assume canonical M n
      from canonical_empty_zone[OF this] ‹⟦M⟧ = {}› have
        ∃ i≤n. M i i < 0
        by auto
      then show ?thesis unfolding check_diag_def neutral
        by auto
    next
      assume check_diag n (uncurry M)
      then show ?thesis unfolding check_diag_def neutral by auto
    qed
  qed
qed (auto dest: check_diag_empty)

lemma and_entry_repair_zone_equiv:
  ⟦and_entry_repair n a b e M⟧ = ⟦and_entry a b e M⟧ if a ≤ n b ≤ n
  unfolding and_entry_repair_def using that by (rule repair_pair_zone_equiv)

lemma dbm_minus_rel:
  assumes list_all2 (λx y. ⟦x⟧ = ⟦y⟧) ms ms'
  shows list_all2 (λx y. ⟦x⟧ = ⟦y⟧) (dbm_minus n ms m) (dbm_minus_canonical n ms' m)
  unfolding dbm_minus_def dbm_minus_canonical_def
  apply (rule concat_transfer[unfolded rel_fun_def, rule_format])
  apply (rule list_all2_mapI)
  apply (rule list.rel_refl_strong)
  apply (auto 4 3
      intro: list_all2_mapI list_all2_mono[OF assms]
      simp: and_entry_repair_zone_equiv and_entry_correct split: if_split_asm
      )
  done

lemma dbm_minus_canonical_fold_canonical':
  ∀ M ∈ set (fold (λm S. dbm_minus_canonical n S m) xs ms). canonical' n M
  if ∀ M ∈ set ms. canonical' n M for ms and xs :: ('t ::time) DBM list
  using that by (induction xs arbitrary: ms) (auto dest: dbm_minus_canonical_canonical')

lemma not_check_diag_nonnegD:
  M i i ≥ 0 if ¬ check_diag n (uncurry M) i ≤ n
  using that unfolding check_diag_def by (auto simp: DBM.less_eq[symmetric] neutral)

theorem dbm_subset_fed_correct:
  fixes xs :: (nat ⇒ nat ⇒ ('t ::time) DBMEntry) list
    and S :: (nat ⇒ nat ⇒ 't DBMEntry) list
  assumes canonical' n M
  shows ⟦M⟧ ⊆ (⋃ m∈set xs. ⟦m⟧) ⟷ dbm_subset_fed n M xs
proof −
  have ∗: list_all2 (λx y. ⟦x⟧ = ⟦y⟧)
    (fold (λm S. dbm_minus n S m) xs ms)
```

```
    (fold (λm S. dbm_minus_canonical n S m) xs ms')
    if list_all2 (λx y. ⟦x⟧ = ⟦y⟧) ms ms' for ms ms' and xs :: 't DBM list
    using that
  proof (induction xs arbitrary: ms ms')
    case Nil
    then show ?case
      by simp
  next
    case prems: (Cons a xs)
    from this(2) show ?case
      by simp (intro prems(1) dbm_minus_rel)
  qed
  let ?xs = filter (λM. ¬ check_diag n (uncurry M)) xs
  have ∗: list_all2 (λx y. ⟦x⟧ = ⟦y⟧)
     (fold (λm S. dbm_minus n S m) ?xs [M])
     (fold (λm S. dbm_minus_canonical n S m) ?xs [M])
    by (rule ∗) simp
  have ∗∗:(⋃ m∈set xs. ⟦m⟧) = (⋃ m∈set (filter (λM. ¬ check_diag n (uncurry M)) xs). ⟦m⟧)
    by (auto simp: check_diag_empty)
  show ?thesis
    apply (subst ∗∗)
    apply (subst dbm_list_superset_op[where S = [M], simplified])
    subgoal
      by (auto dest: not_check_diag_nonnegD simp: neutral[symmetric] DBM.less_eq)
    subgoal
      unfolding dbm_subset_fed_def Let_def
      using dbm_minus_canonical_fold_canonical'[of [M]] ‹canonical' n M›
    by (intro list_all_iff_list_all2I list.rel_mono_strong[OF ∗])(auto dest: check_diag_iff_empty)
    done
qed

lemma dbm_minus_canonical_check_fed_equiv:
  dbm_list (dbm_minus_canonical_check n S m) = dbm_list (dbm_minus_canonical n S m)
  unfolding dbm_minus_canonical_check_def by (auto simp: check_diag_empty)

lemma dbm_minus_canonical_dbm_minus:
  dbm_list (dbm_minus_canonical n xs m) = dbm_list (dbm_minus n xs m)
  using dbm_minus_rel[of xs xs m] unfolding list_all2_same
  by (force dest: list_all2_set1 list_all2_set2)

lemma dbm_minus_canonical_fed_equiv:
  dbm_list (dbm_minus_canonical n xs m) = dbm_list (dbm_minus_canonical n xs' m)
  if dbm_list xs = dbm_list xs' 0 ≤ m 0 0
  unfolding dbm_minus_canonical_dbm_minus
  using that by (auto simp: neutral dbm_list_subtract[symmetric] DBM.less_eq)

theorem dbm_subset_fed_correct':
  fixes xs :: (nat ⇒ nat ⇒ ('t ::time) DBMEntry) list
    and S :: (nat ⇒ nat ⇒ 't DBMEntry) list
  assumes canonical' n M
  shows ⟦M⟧ ⊆ (⋃ m∈set xs. ⟦m⟧) ⟷ (
    let xs = filter (λM. ¬ check_diag n (uncurry M)) xs in
    list_all (λ M. check_diag n (uncurry M)) (fold (λm S. dbm_minus_canonical_check n S m)
xs [M]))
```

```
proof −
 have canonical: ∀ M ∈ set (fold (λm S. dbm_minus_canonical_check n S m) xs ms). canonical' n M
   if ∀ M ∈ set ms. canonical' n M for ms and xs :: 't DBM list
  using that by (induction xs arbitrary: ms) (auto dest: dbm_minus_canonical_check_canonical')
 have ∗: dbm_list (fold (λm S. dbm_minus_canonical_check n S m) xs ms) =
   dbm_list (fold (λm S. dbm_minus_canonical n S m) xs ms')
   if dbm_list ms = dbm_list ms' ∀ m ∈ set xs. m 0 0 ≥ 0 for ms ms' and xs :: 't DBM list
   using that
 proof (induction xs arbitrary: ms ms')
   case Nil
   then show ?case
     by simp
 next
   case (Cons a xs)
   from Cons.prems show ?case
     by − (simp, rule Cons.IH,
      auto intro!: dbm_minus_canonical_fed_equiv simp add: dbm_minus_canonical_check_fed_equiv
        )
 qed
 define xs' where xs' = filter (λM. ¬ check_diag n (uncurry M)) xs
 have ∗: dbm_list (fold (λm S. dbm_minus_canonical_check n S m) xs' [M]) =
       dbm_list (fold (λm S. dbm_minus_canonical n S m) xs' [M])
   by (auto intro!: ∗ dest: not_check_diag_nonnegD simp: xs'_def)
 have ∗∗: list_all (λ M. check_diag n (uncurry M)) xs ⟷ dbm_list xs = {}
   if ∀ M ∈ set xs. canonical' n M for xs :: 't DBM list
  using that by (metis (mono_tags, lifting) Ball_set SUP_bot_conv(2) check_diag_iff_empty)
 show ?thesis
   unfolding
     dbm_subset_fed_correct[OF ‹canonical' _ _›] dbm_subset_fed_def
     xs'_def[symmetric] Let_def
   apply (subst ∗∗)
    defer
    apply (subst ∗∗)
   using assms by (auto intro!: dbm_minus_canonical_fold_canonical' canonical simp: ∗)
qed

lemma subset_if_pointwise_le:
 ⟦M⟧ ⊆ ⟦M'⟧ if pointwise_cmp (≤) n M M'
 using that by (simp add: DBM.less_eq DBM_le_subset pointwise_cmp_def subsetI)

theorem dbm_subset_fed_check_correct:
 fixes xs :: (nat ⇒ nat ⇒ ('t ::time) DBMEntry) list
   and S :: (nat ⇒ nat ⇒ 't DBMEntry) list
 assumes canonical' n M
 shows ⟦M⟧ ⊆ (⋃m∈set xs. ⟦m⟧) ⟷ dbm_subset_fed_check n M xs
proof −
 define xs' where xs' = filter (λM. ¬ check_diag n (uncurry M)) xs
 define is_direct_subset where
   is_direct_subset = list_ex (λ M'. dbm_subset' n (uncurry M) (uncurry M')) xs'
 have ⟦M⟧ ⊆ (⋃m∈set xs. ⟦m⟧) if is_direct_subset
 proof −
   from that have ∃ m ∈ set xs'. ⟦M⟧ ⊆ ⟦m⟧
     unfolding is_direct_subset_def list_ex_iff dbm_subset'_def
```

by (*auto intro*!: *subset_if_pointwise_le*)
then show *?thesis*
unfolding *xs'_def* **by** *auto*
qed
then show *?thesis*
apply (*cases is_direct_subset*; *simp add*:
dbm_subset_fed_check_def is_direct_subset_def dbm_subset_fed_def xs'_def[*symmetric*]
)
unfolding *dbm_subset_fed_correct*[
OF ‹*canonical' n M*›, *of xs*, *unfolded Let_def dbm_subset_fed_def*, *folded xs'_def*, *symmetric*
]
unfolding *dbm_subset_fed_correct'*[
OF ‹*canonical' n M*›, *of xs*, *folded xs'_def*, *unfolded Let_def*, *symmetric*
]
..
qed

end

10.6 Refined DBM Operations

definition
V_dbm = (λ *i j*. *if i* = *j then* 0 *else if i* = 0 ∧ *j* > 0 *then* 0 *else* ∞)

definition *and_entry_upd* ::
nat ⇒ *nat* ⇒ *int DBMEntry* ⇒ *int DBM'* ⇒ *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 ≡
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* (λ(*i*, *j*). *map* (λ *M*. *and_entry_repair_upd n j i* (*neg_dbm_entry* (*m* (*i*, *j*))) *M*) *xs*)
[(*i*, *j*). *i*←[0..<*Suc n*], *j*←[0..<*Suc n*], (*i* > 0 ∨ *j* > 0) ∧ *i* ≤ *n* ∧ *j* ≤ *n* ∧ *m* (*i*, *j*) ≠ ∞])

definition
dbm_minus_canonical_check_upd n xs M =
filter (λ*M*. ¬ *check_diag n M*) (*dbm_minus_canonical_upd n xs M*)

definition
dbm_subset_fed_upd n M xs ≡
let xs = *filter* (λ*M*. ¬ *check_diag n M*) *xs*;
is_direct_subset = *list_ex* (λ*M'*. *dbm_subset' n M M'*) *xs*
in is_direct_subset ∨
list_all (λ *M*. *check_diag n M*) (*fold* (λ*m S*. *dbm_minus_canonical_check_upd n S m*) *xs* [*M*])

lemma *list_all_filter_neg*:
list_all P (*filter* (λ*x*. ¬ *P x*) *xs*) ⟷ (*filter* (λ*x*. ¬ *P x*) *xs*) = []

by (*metis Cons_eq_filterD list_all_simps*(2) *list_pred_cases*)

lemma *dbm_subset_fed_upd_alt_def*:
dbm_subset_fed_upd n M xs ≡
let xs = *filter* (λ*M*. ¬ *check_diag n M*) *xs*
in if xs = [] *then check_diag n M*
else if list_ex (λ*M'*. *dbm_subset' n M M'*) *xs then True*
else fold (λ*m S*. *dbm_minus_canonical_check_upd n S m*) *xs* [*M*] = []
unfolding *dbm_subset_fed_upd_def short_circuit_conv* **using** *list_last*
by (*intro eq_reflection*;*force simp*: *list_all_filter_neg dbm_minus_canonical_check_upd_def Let_def*)

definition
V_dbm' n = (λ(*i*, *j*). (*if i* = *j* ∨ *i* = 0 ∧ *j* > 0 ∨ *i* > *n* ∨ *j* > *n then* 0 *else* ∞))

definition
down_upd :: *nat* ⇒ _ *DBM'* ⇒ _ *DBM'*
where
down_upd n M ≡ λ(*i*, *j*).
if i = 0 ∧ *j* > 0 ∧ *i* ≤ *n* ∧ *j* ≤ *n then Min* ({*Le* 0} ∪ {*M* (*k*, *j*) | *k*. 1 ≤ *k* ∧ *k* ≤ *n*}) *else M* (*i*, *j*)

definition
restrict_zero_upd n M x ≡
let
M1 = *and_entry_upd x* 0 (*Le* 0) *M*;
M2 = *and_entry_upd* 0 *x* (*Le* 0) *M1*
in Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair n M2 x 0

definition
free_upd :: *nat* ⇒ _ *DBM'* ⇒ *nat* ⇒ _ *DBM'*
where
free_upd n M x ≡
λ (*i*, *j*). *if i* = *x* ∧ *j* ≠ *x* ∧ *i* ≤ *n* ∧ *j* ≤ *n*
then ∞ *else if i* ≠ *x* ∧ *j* = *x* ∧ *i* ≤ *n* ∧ *j* ≤ *n then M* (*i*, 0) *else M* (*i*, *j*)

definition
pre_reset_upd n M x ≡ *free_upd n* (*restrict_zero_upd n M x*) *x*

definition
pre_reset_list_upd n M r ≡ *fold* (λ *x M*. *pre_reset_upd n M x*) *r M*

10.7 Transferring Properties

context
includes *lifting_syntax*
begin

lemma *neg_dbm_entry_transfer*[*transfer_rule*]:
(*rel_DBMEntry ri* ===> *rel_DBMEntry ri*) *neg_dbm_entry neg_dbm_entry*
by (*auto elim*!: *DBMEntry.rel_cases intro*!: *rel_funI*)

lemma *fold_min_transfer*:
((*list_all2* (*rel_DBMEntry ri*)) ===> *rel_DBMEntry ri* ===> *rel_DBMEntry ri*) (*fold min*)

```
(fold min)
  by transfer_prover

context
  fixes n :: nat
begin

definition
  RI2 M D ≡ RI n (uncurry M) D

lemma and_entry_transfer[transfer_rule]:
  ((=) ===> (=) ===> rel_DBMEntry ri ===> RI2 ===> RI2) and_entry and_entry_upd
  unfolding and_entry_def and_entry_upd_def
  unfolding rel_fun_def eq_onp_def RI2_def
  by (auto intro: min_ri_transfer[unfolded rel_fun_def, rule_format])

lemma FWI_transfer[transfer_rule]:
  (RI2 ===> eq_onp (λx. x = n) ===> eq_onp (λx. x < Suc n) ===> RI2) FWI FWI' (is
?A)
and FW_transfer[transfer_rule]:
  (RI2 ===> eq_onp (λx. x = n) ===> RI2) FW FW' (is ?B)
proof −
  define RI' where
    RI' = (eq_onp (λ x. x < Suc n) ===> eq_onp (λ x. x < Suc n) ===> rel_DBMEntry ri)
  have RI_iff: RI' M M' ⟷ RI n (uncurry M) (uncurry M') for M M'
    unfolding RI'_def rel_fun_def by auto
  { fix M D k assume RI n (uncurry M) D k < Suc n
    then have RI' M (curry D)
      by (simp add: RI_iff)
    with ‹k < Suc n› have RI' (FWI M n k) (FWI (curry D) n k)
      unfolding FWI_def
      by (intro fwi_RI_transfer[of n, folded RI'_def, unfolded rel_fun_def, rule_format])
        (auto simp: eq_onp_def)
    then have RI n (uncurry (FWI M n k)) (uncurry (FWI (curry D) n k))
      by (simp add: RI_iff)
  } note ∗ = this
  show ?A
    unfolding FWI'_def
    unfolding rel_fun_def
    unfolding RI2_def
    apply clarsimp
    apply (subst (asm) (3) eq_onp_def)
    apply (subst (asm) (3) eq_onp_def)
    apply clarsimp
    by (intro ∗)
  { fix M D assume RI n (uncurry M) D
    then have RI' M (curry D)
      by (simp add: RI_iff)
    then have RI' (FW M n) (FW (curry D) n)
      by (intro FW_RI_transfer[of n, folded RI'_def, unfolded rel_fun_def, rule_format])
        (auto simp: eq_onp_def)
    then have RI n (uncurry (FW M n)) (FW' D n)
      by (simp add: RI_iff FW'_def)
  } note ∗ = this
```

```
  show ?B
    unfolding rel_fun_def
    unfolding RI2_def
    apply clarsimp
    apply (subst (asm) (3) eq_onp_def)
    apply clarsimp
    by (intro *)
qed

lemma repair_pair_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> RI2 ===> eq_onp (λx. x < Suc n) ===> eq_onp (λx. x < Suc
n) ===> RI2)
    repair_pair Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair
 unfolding repair_pair_def Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair_def
  by transfer_prover

lemma and_entry_transfer_weak:
  (eq_onp (λx. x < Suc n) ===> eq_onp (λx. x < Suc n) ===> rel_DBMEntry ri ===>
RI2 ===> RI2)
    and_entry and_entry_upd
  using and_entry_transfer unfolding rel_fun_def eq_onp_def by auto

lemma and_entry_repair_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> eq_onp (λx. x < Suc n) ===> eq_onp (λx. x < Suc n) ===>
rel_DBMEntry ri
    ===> RI2 ===> RI2) and_entry_repair and_entry_repair_upd

  supply [transfer_rule] = and_entry_transfer_weak
  unfolding and_entry_repair_def and_entry_repair_upd_def by transfer_prover

lemma dbm_minus_canonical_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> list_all2 RI2 ===> RI2 ===> list_all2 RI2)
    dbm_minus_canonical dbm_minus_canonical_upd

  unfolding dbm_minus_canonical_def dbm_minus_canonical_upd_def
  apply (intro rel_funI)
  apply (rule concat_transfer[unfolded rel_fun_def, rule_format])
  apply (rule list.map_transfer[unfolded rel_fun_def, rule_format,
       where Rb = rel_prod (eq_onp (λx. x < Suc n)) (eq_onp (λx. x < Suc n))])
   apply clarsimp
  subgoal
    apply (rule list_all2_mapI)
    apply (erule list_all2_mono)
    apply (rule and_entry_repair_transfer[unfolded rel_fun_def, rule_format])
       apply assumption+
    subgoal
      unfolding RI2_def rel_fun_def
      by (auto intro: neg_dbm_entry_transfer[unfolded rel_fun_def, rule_format])
    apply assumption
    done
  subgoal premises prems for n1 n2 xs ys M D
  proof −
    have [simp]: D (i, j) ≠ ∞ ⟷ M i j ≠ ∞ if i < Suc n j < Suc n for i j
       using prems that by (auto 4 3 simp: eq_onp_def rel_fun_def RI2_def elim!: DBMEn-
```

```
try.rel_cases)
    from prems have [simp]: n1 = n n2 = n
      by (auto simp: eq_onp_def)
    let ?a = (concat
      (map (λi. concat
                (map (λj. if (0 < i ∨ 0 < j) ∧
                             i ≤ n ∧ j ≤ n ∧ M i j ≠ ∞
                          then [(i, j)] else [])
                  [0..<Suc n]))
        [0..<Suc n]))
    let ?b = (concat
      (map (λi. concat
                (map (λj. if (0 < i ∨ 0 < j) ∧
                             i ≤ n ∧ j ≤ n ∧ D (i, j) ≠ ∞
                          then [(i, j)] else [])
                  [0..<Suc n]))
        [0..<Suc n]))
    have ?b = ?a
      by (auto intro!: arg_cong[where f = concat] simp del: upt_Suc)
    then show ?thesis
      by (simp del: upt_Suc add: list_all2_same eq_onp_def)
  qed
  done

lemma check_diag_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> RI2 ===> (=)) (λn M. check_diag n (uncurry M)) check_diag
  unfolding RI2_def rel_fun_def check_diag_def
  by (auto 0 5 dest: neutral_RI simp: eq_onp_def less_Suc_eq_le neutral[symmetric])

lemma dbm_minus_canonical_check_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> list_all2 RI2 ===> RI2 ===> list_all2 RI2)
    dbm_minus_canonical_check dbm_minus_canonical_check_upd

  unfolding dbm_minus_canonical_check_def dbm_minus_canonical_check_upd_def by transfer_prover

lemma le_rel_DBMEntry_iff:
  a ≤ b ⟷ x ≤ y if rel_DBMEntry ri a x rel_DBMEntry ri b y
  using that by (auto elim!: DBMEntry.rel_cases simp: dbm_entry_simps ri_def)

lemma dbm_subset'_transfer[transfer_rule]:
  (eq_onp (λx. x = n) ===> RI2 ===> RI2 ===> (=))
    (λ n M M'. dbm_subset' n (uncurry M) (uncurry M')) dbm_subset'
  unfolding RI2_def rel_fun_def dbm_subset'_def
  using le_rel_DBMEntry_iff by (clarsimp simp: eq_onp_def pointwise_cmp_def less_Suc_eq_le)
meson

lemma dbm_subset_fed_transfer:
  (eq_onp (λx. x = n) ===> RI2 ===> list_all2 RI2 ===> (=)) dbm_subset_fed_check
dbm_subset_fed_upd
  unfolding dbm_subset_fed_check_def dbm_subset_fed_upd_def by transfer_prover

lemma V_dbm_transfer[transfer_rule]:
  RI2 V_dbm (V_dbm' n)
```

unfolding *V_dbm_def V_dbm'_def RI2_def* **by** (*auto simp*: *rel_fun_def neutral eq_onp_def zero_RI*)

lemma *down_transfer*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *RI2*) *down down_upd*
unfolding *down_def down_upd_def*
apply (*clarsimp simp*: *rel_fun_def RI2_def eq_onp_def*, *goal_cases*)
subgoal premises *prems* **for** *M D x*
proof −
have *A*: (*insert* (*Le 0*) {*M k x* |*k*. *Suc 0* ≤ *k* ∧ *k* ≤ *n*})
= (*set* (*Le 0* # [*M k x*. *k* ← [*1*..<*Suc n*]]))
by *auto*
have *B*: *insert* (*Le 0*) {*D* (*k*, *x*) |*k*. *Suc 0* ≤ *k* ∧ *k* ≤ *n*}
= (*set* (*Le 0* # [*D* (*k*, *x*). *k* ← [*1*..<*Suc n*]]))
by *auto*
show *?thesis*
unfolding *A B Min.set_eq_fold*
apply (*rule fold_min_transfer*[*unfolded rel_fun_def*, *rule_format*])
unfolding *list.rel_map list_all2_same* **using** *prems* **by** (*auto simp*: *zero_RI*)
qed
done

lemma *free_upd*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> (=) ===> *RI2*) *free free_upd*
unfolding *free_def free_upd_def* **by** (*auto simp*: *RI2_def rel_fun_def eq_onp_def*)

lemma *pre_reset_transfer*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *eq_onp* (λ*x*. *x* < *Suc n*) ===> *RI2*) *pre_reset pre_reset_upd*
proof −
have [*transfer_rule*]: *eq_onp* (λ*x*. *x* = *n*) *n n*
by (*simp add*: *eq_onp_def*)
note [*transfer_rule*] = *and_entry_transfer_weak*
have [*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *eq_onp* (λ*x*. *x* < *Suc n*) ===> *RI2*) *free free_upd*
using *free_upd* **unfolding** *rel_fun_def eq_onp_def* **by** *blast*
show *?thesis*
unfolding *pre_reset_def pre_reset_upd_def*
unfolding *restrict_zero_def restrict_zero_upd_def*
by *transfer_prover*
qed

lemma *pre_reset_list_transfer*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *list_all2* (*eq_onp* (λ*x*. *x* < *Suc n*)) ===> *RI2*)
pre_reset_list pre_reset_list_upd
unfolding *pre_reset_list_def pre_reset_list_upd_def* **by** *transfer_prover*

end
end

definition *unbounded_dbm* **where**
unbounded_dbm ≡ λ *i j*. *if i* = *j then 0 else* ∞

lemma *canonical_unbounded_dbm*:
canonical unbounded_dbm n
by (*auto simp*: *unbounded_dbm_def any_le_inf*)

lemma *diag_unbounded_dbm*:
unbounded_dbm i i = 0
unfolding *unbounded_dbm_def* **by** *simp*

lemma *down_diag*:
down n M i i = M i i
unfolding *down_def* **by** *auto*

definition
abstr_FW n cc M v ≡ FW (*abstr cc M v*) *n*

definition
abstr_FW_upd n cc M ≡ FW' (*abstr_upd cc M*) *n*

lemma *abstr_mono*:
abstr cc M v i j ≤ M i j
by (*subst abstr.simps*, *induction cc arbitrary*: *M*) (*auto intro*: *order.trans abstra_mono*)

lemma *abstr_FW_mono*:
abstr_FW n cc M v i j ≤ M i j **if** *i ≤ n j ≤ n*
unfolding *abstr_FW_def* **by** (*blast intro*: *that abstr_mono fw_mono order.trans*)

lemma *abstr_FW_diag_preservation*:
∀ k≤n. abstr_FW n cc M v k k ≤ 0 **if** *∀ k≤n. M k k ≤ 0*
using *that* **by** (*blast intro*: *abstr_FW_mono order.trans*)

lemma *FW_canonical*:
canonical' n (*FW M n*)
unfolding *canonical'_def* **using** *FW_canonical*[*of n M*] **by** (*simp add*: *check_diag_def neutral*)

lemma *abstr_FW_canonical*:
canonical' n (*abstr_FW n cc M v*)
unfolding *abstr_FW_def* **by** (*rule FW_canonical*)

lemma *down_check_diag*:
check_diag n (*uncurry* (*down n M*)) **if** *check_diag n* (*uncurry M*)
using *that* **unfolding** *check_diag_def down_def* **by** *force*

context *Default_Nat_Clock_Numbering*
begin

lemma *clock_numbering*:
∀ c. v c > 0 ∧ (*∀ x. ∀ y. v x ≤ n ∧ v y ≤ n ∧ v x = v y ⟶ x = y*)
by (*metis neq0_conv v_0 v_id*)

lemma *V_dbm_correct*:
⟦*V_dbm*⟧ = *V*
unfolding *V_def DBM_zone_repr_def DBM_val_bounded_alt_def2*
by (*auto simp*: *dbm_entry_val'_iff_bounded V_dbm_def dbm_entry_simps*)

lemma *abstr_correct*:
 $[\![abstr\ cc\ M\ v]\!] = [\![M]\!] \cap \{u.\ u \vdash cc\}$ **if** $\forall\, c{\in}collect_clks\ cc.\ 0 < c \wedge c \leq n$
 apply (*rule dbm_abstr_zone_eq2*)
 subgoal
  by (*rule clock_numbering*)
 subgoal
  using *that* **by** (*auto simp*: *v_is_id*)
 done

lemma *unbounded_dbm_correct*:
 $[\![unbounded_dbm]\!] = UNIV$
 unfolding *DBM_zone_repr_def DBM_val_bounded_alt_def2 unbounded_dbm_def neutral*
 by (*simp add*: *dbm_entry_val'_iff_bounded any_le_inf*)

lemma *abstr_correct'*:
 $[\![abstr\ cc\ unbounded_dbm\ v]\!] = \{u.\ u \vdash cc\}$ **if** $\forall\, c{\in}collect_clks\ cc.\ 0 < c \wedge c \leq n$
 by (*simp add*: *unbounded_dbm_correct abstr_correct*[*OF that*] *del*: *abstr.simps*)

lemma *abstr_FW_correct*:
 $[\![abstr_FW\ n\ cc\ M\ v]\!] = [\![M]\!] \cap \{u.\ u \vdash cc\}$ **if** $\forall\, c{\in}collect_clks\ cc.\ 0 < c \wedge c \leq n$
 unfolding *abstr_FW_def* **by** (*subst FW_zone_equiv*[*symmetric*]; *intro surj_on abstr_correct that*)

lemma *abstr_FW_correct'*:
 $[\![abstr_FW\ n\ cc\ unbounded_dbm\ v]\!] = \{u.\ u \vdash cc\}$ **if** $\forall\, c{\in}collect_clks\ cc.\ 0 < c \wedge c \leq n$
 by (*simp add*: *unbounded_dbm_correct abstr_FW_correct*[*OF that*])

lemma *down_V*:
 $[\![down\ n\ M]\!] \subseteq V$
 by (*rule V_structuralI*) (*auto simp*: *down_def neutral intro*: *Min_le*)

lemma *down_correct'*:
 $[\![down\ n\ M]\!] = [\![M]\!]^{\downarrow} \cap V$ **if** *canonical M n*
 apply *safe*
 subgoal for *u*
  by (*erule down_sound*, *rule that*)
 subgoal for *u*
  using *down_V* **by** *blast*
 subgoal for *u*
  unfolding *V_def* **by** (*erule down_complete*) *simp*
 done

lemma *down_correct*:
 $[\![down\ n\ M]\!] = [\![M]\!]^{\downarrow} \cap V$ **if** *canonical' n M*
 using *that* **unfolding** *canonical'_def*
 apply *standard*
 subgoal
  by (*rule down_correct'*)
 by (*frule down_check_diag*) (*simp add*: *check_diag_empty zone_time_pre_def*)

lemma *pre_reset_diag_preservation*:
 pre_reset n M x i i $\leq M\ i\ i$ **if** $i \leq n$

unfolding *pre_reset_def* **by** (*auto simp add*: *free_diag restrict_zero_mono that*)

lemma *pre_reset_list_diag*:
pre_reset_list n M r i i ≤ *M i i* **if** *i* ≤ *n*
apply (*induction r arbitrary*: *M*)
apply (*simp add*: *pre_reset_list_def*; *fail*)
apply (*simp add*: *pre_reset_list_Cons*, *blast intro*: *pre_reset_diag_preservation that order.trans*)
done

context
includes *lifting_syntax*
begin

lemma *abstra_upd_abstra*:
abstra_upd ac M (*i*, *j*) = *abstra ac* (*curry M*) *v i j* **if** *0* < *constraint_clk ac i* ≤ *n j* ≤ *n*
using *that* **by** (*cases ac*) (*auto simp*: *le_n_iff v_is_id v_0*)

lemma *abstra_transfer*[*transfer_rule*]:
(*rel_acconstraint* (*eq_onp* (λ *x*. *0* < *x* ∧ *x* < *Suc n*)) *ri* ===> *RI2 n* ===> *RI2 n*)
(λ *cc M*. *abstra cc M v*) *abstra_upd*
apply (*intro rel_funI*)
apply (*subst RI2_def*)
apply (*intro rel_funI*)
apply (*elim rel_prod.cases*)
apply (*simp only*:)
apply (*subst abstra_upd_abstra*)
by (*auto 4 3 simp*: *eq_onp_def RI2_def rel_fun_def*
intro!: *min_ri_transfer*[*unfolded rel_fun_def*, *rule_format*]
elim!: *acconstraint.rel_cases*
)

lemma *abstr_transfer*[*transfer_rule*]:
(*list_all2* (*rel_acconstraint* (*eq_onp* (λ *x*. *0* < *x* ∧ *x* < *Suc n*)) *ri*) ===> *RI2 n* ===> *RI2 n*)
(λ *cc M*. *abstr cc M v*) *abstr_upd*
unfolding *abstr.simps abstr_upd_def* **by** *transfer_prover*

lemma *abstr_FW_transfer*[*transfer_rule*]:
(*list_all2* (*rel_acconstraint* (*eq_onp* (λ *x*. *0* < *x* ∧ *x* < *Suc n*)) *ri*) ===> *RI2 n* ===> *RI2 n*)
(λ *cc M*. *abstr_FW n cc M v*) (*abstr_FW_upd n*)
proof −
have [*transfer_rule*]: *eq_onp* (λ*x*. *x* = *n*) *n n*
by (*simp add*: *eq_onp_def*)
show *?thesis*
unfolding *abstr_FW_def abstr_FW_upd_def* **by** *transfer_prover*
qed

end

end

lemma *RI2_trivial_transfer*[*transfer_rule*]: (*RI2 n*) (*curry* (*conv_M M*)) *M*
unfolding *RI2_def rel_fun_def* **by** (*auto simp*: *eq_onp_def*)

end
theory *Deadlock_Impl*
imports *Deadlock Munta_Base.Abstract_Term*
begin

Misc **lemma** *constraint_clk_conv_ac*:
constraint_clk (*conv_ac ac*) = *constraint_clk ac*
by (*cases ac*; *auto*)

lemma *constraint_clk_conv_cc*:
collect_clks (*conv_cc cc*) = *collect_clks cc*
by (*auto simp*: *collect_clks_def constraint_clk_conv_ac image_def*)

lemma *atLeastLessThan_alt_def*:
{*a*..<*b*} = {*k*. *a* ≤ *k* ∧ *k* < *b*}
by *auto*

lemma *atLeastLessThan_Suc_alt_def*:
{*a*..<*Suc b*} = {*k*. *a* ≤ *k* ∧ *k* ≤ *b*}
by *auto*

lemma (**in** *Graph_Defs*) *deadlock_if_deadlocked*:
deadlock y **if** *deadlocked y*
using *that* **unfolding** *deadlock_def* **by** *auto*

10.8 Functional Refinement

Elementary list operations **lemma** *map_conv_rev_fold*:
map f xs = *rev* (*fold* (λ *a xs*. *f a* # *xs*) *xs* [])
proof −
have *fold* (λ *a xs*. *f a* # *xs*) *xs ys* = *rev* (*map f xs*) @ *ys* **for** *ys*
by (*induction xs arbitrary*: *ys*) *auto*
then show *?thesis*
by *simp*
qed

lemma *concat_map_conv_rev_fold*:
concat (*map f xs*) = *rev* (*fold* (λ *xs ys*. *rev* (*f xs*) @ *ys*) *xs* [])
proof −
have *rev* (*fold* (λ *xs ys*. *rev* (*f xs*) @ *ys*) *xs ys*) = *rev ys* @ *List.maps f xs* **for** *ys*
by (*induction xs arbitrary*: *ys*) (*auto simp*: *maps_simps*)
then show *?thesis*
by (*simp add*: *concat_map_maps*)
qed

lemma *concat_conv_fold_rev*:
concat xss = *fold* (@) (*rev xss*) []
using *fold_append_concat_rev*[*of rev xss*] **by** *simp*

lemma *filter_conv_rev_fold*:
filter P xs = *rev* (*fold* (λ*x xs*. *if P x then x* # *xs else xs*) *xs* [])
proof −
have *rev ys* @ *filter P xs* = *rev* (*fold* (λ*x xs*. *if P x then x* # *xs else xs*) *xs ys*) **for** *ys*

by (*induction xs arbitrary*: *ys*) (*auto*, *metis revg.simps*(*2*) *revg_fun*)
from *this*[*symmetric*] **show** *?thesis*
by *simp*
qed

DBM operations definition
free_upd1 n M c =
(*let*
M1 = *fold* (λ*i M*. *if i* ≠ *c then* (*M*((*i*, *c*) := *op_mtx_get M* (*i*, *0*))) *else M*) [*0*..<*Suc n*] *M*;
M2 = *fold* (λ*i M*. *if i* ≠ *c then M*((*c*,*i*) := ∞) *else M*) [*0*..<*Suc n*] *M1*
in
M2
)

definition
pre_reset_upd1 n M x ≡ *free_upd1 n* (*restrict_zero_upd n M x*) *x*

definition
pre_reset_list_upd1 n M r ≡ *fold* (λ *x M*. *pre_reset_upd1 n M x*) *r M*

definition
upd_pairs xs = *fold* (λ(*p*,*q*) *f*. *f*(*p*:=*q*)) *xs*

lemma *upd_pairs_Nil*:
upd_pairs [] *f* = *f*
unfolding *upd_pairs_def* **by** *simp*

lemma *upd_pairs_Cons*:
upd_pairs ((*p*, *q*) # *xs*) *f* = *upd_pairs xs* (*f*(*p*:=*q*))
unfolding *upd_pairs_def* **by** *simp*

lemma *upd_pairs_no_upd*:
assumes *p* ∉ *fst* ‘ *set xs*
shows *upd_pairs xs f p* = *f p*
using *assms* **by** (*induction xs arbitrary*: *f*) (*auto simp*: *upd_pairs_Nil upd_pairs_Cons*)

lemma *upd_pairs_upd*:
assumes (*p*, *y*) ∈ *set xs distinct* (*map fst xs*)
shows *upd_pairs xs f p* = *y*
using *assms* **proof** (*induction xs arbitrary*: *f*)
case *Nil*
then show *?case*
by (*simp add*: *upd_pairs_Nil*)
next
case (*Cons x xs*)
then show *?case*
by (*cases x*) (*auto simp add*: *upd_pairs_Cons upd_pairs_no_upd*)
qed

lemma *upd_pairs_append*:
upd_pairs (*xs* @ *ys*) = *upd_pairs ys o upd_pairs xs*
unfolding *upd_pairs_def fold_append* **..**

lemma *upd_pairs_commute_single*:
upd_pairs xs (*f*(*a* := *b*)) = (*upd_pairs xs f*)(*a* := *b*) **if** *a* ∉ *fst* ‘ *set xs*
using *that* **by** (*induction xs arbitrary*: *f*) (*auto simp*: *upd_pairs_Nil upd_pairs_Cons fun_upd_twist*)

lemma *upd_pairs_append′*:
upd_pairs (*xs* @ *ys*) = *upd_pairs xs* ∘ *upd_pairs ys* **if** *fst* ‘ *set xs* ∩ *fst* ‘ *set ys* = {}
using *that*
proof (*induction xs*)
case *Nil*
then show *?case*
by (*simp add*: *upd_pairs_Nil*)
next
case (*Cons a xs*)
then show *?case*
by (*cases a*) (*auto simp add*: *upd_pairs_Nil upd_pairs_Cons upd_pairs_commute_single*)
qed

definition
upd_pairs′ xs = *fold* (λ(*p*,*q*) *f*. *f*(*p*:=*f q*)) *xs*

lemma *upd_pairs′_Nil*:
upd_pairs′ [] *f* = *f*
unfolding *upd_pairs′_def* **by** *simp*

lemma *upd_pairs′_Cons*:
upd_pairs′ ((*p*, *q*) # *xs*) *f* = *upd_pairs′ xs* (*f*(*p*:=*f q*))
unfolding *upd_pairs′_def* **by** *simp*

lemma *upd_pairs′_upd_pairs*:
assumes *fst* ‘ *set xs* ∩ *snd* ‘ *set xs* = {}
shows *upd_pairs′ xs f* = *upd_pairs* (*map* (λ(*p*, *q*). (*p*, *f q*)) *xs*) *f*
using *assms*
proof (*induction xs arbitrary*: *f*)
case *Nil*
then show *?case*
by (*simp add*: *upd_pairs_Nil upd_pairs′_Nil*)
next
case (*Cons x xs*)
obtain *a b* **where** [*simp*]: *x* = (*a*, *b*)
by (*cases x*)
from *Cons.prems* **have** ∗:
map (λ(*p*, *q*). (*p*, *if q* = *a then f b else f q*)) *xs* = *map* (λ(*p*, *q*). (*p*, *f q*)) *xs*
by *auto*
from *Cons* **show** *?case*
by (*auto simp add*: *upd_pairs_Cons upd_pairs′_Cons* ∗)
qed

lemma *upd_pairs_map*:
upd_pairs (*map f xs*) = *fold* (λ*pq g*. *let* (*p*,*q*) = *f pq in g*(*p*:=*q*)) *xs*
unfolding *upd_pairs_def fold_map*
by (*intro arg_cong2*[**where** *f* = *fold*] *ext*) (*auto simp*: *comp_def split*: *prod.split*)

lemma *down_upd_alt_def*:

```
  down_upd n M =
  upd_pairs ([((0, j), fold min [M (k, j). k ← [1..<Suc n]] (Le 0)). j ← [1..<Suc n]]) M
proof −
  define xs where
    xs = [((0::nat, j), Min ({Le 0} ∪ {M (k, j) | k. 1 ≤ k ∧ k ≤ n})). j ← [1..<Suc n]]
  have distinct (map fst xs)
    unfolding xs_def by (auto simp add: map_concat comp_def distinct_map)
  have ∗: fold min [M (k, j). k←[1..<Suc n]] (Le 0) = Min ({Le 0} ∪ {M (k, j) |k. 1 ≤ k ∧ k
≤ n})
    for j
    by (subst Min.set_eq_fold[symmetric])
       (auto simp: atLeastLessThan_Suc_alt_def simp del: upt_Suc intro: arg_cong[where f =
Min])
  show ?thesis
    unfolding ∗ xs_def[symmetric]
  by (intro ext, auto simp add: down_upd_def)
     (solves ‹subst upd_pairs_upd[OF _ ‹distinct _›] upd_pairs_no_upd,
      auto simp: image_def xs_def›)+
qed

lemma down_upd_alt_def1:
  down_upd n M =
  fold (λj M. let (p,q) = ((0, j), fold min [M (k, j). k ← [1..<Suc n]] (Le 0)) in M(p:=q))
  [1..<Suc n] M
proof −
  have ∗:
    fold (λj M.  let (p,q) = ((0,j), fold min [M (k, j). k ← [1..<Suc n]] (Le 0)) in M(p:=q))  xs
M
  = fold (λj M'. let (p,q) = ((0,j), fold min [M (k, j). k ← [1..<Suc n]] (Le 0)) in M'(p:=q)) xs
M
   for xs
  proof (induction xs arbitrary: M)
    case Nil
    then show ?case
      by simp
  next
    case (Cons x xs)
    then show ?case
      by (auto 4 7 intro: arg_cong2[where f = min] fold_cong)
  qed
  then show ?thesis
    unfolding down_upd_alt_def upd_pairs_map ..
qed

lemma
  free_upd n M c =
  upd_pairs ([((c,i), ∞). i ← [0..<Suc n], i ≠ c] @ [((i,c), M(i,0)). i ← [0..<Suc n], i ≠ c]) M
  if c ≤ n
proof −
  let ?xs1 = λn. [((c,i), ∞). i ← [0..<Suc n], i ≠ c]
  let ?xs2 = λn. [((i,c), M(i,0)). i ← [0..<Suc n], i ≠ c]
  define xs where xs = ?xs1 n @ ?xs2 n
  have distinct (map fst xs)
    unfolding xs_def
```

```
    apply (clarsimp simp del: upt_Suc simp add: map_concat comp_def if_distrib split: if_split)
     apply (auto split: if_split_asm simp: distinct_map inj_on_def intro!: distinct_concat)
    done
  from ‹c ≤ n› show ?thesis
    unfolding xs_def[symmetric]
  by (intro ext, auto simp: free_upd_def)
     (solves ‹subst upd_pairs_upd[OF _ ‹distinct (map fst xs)›] upd_pairs_no_upd,
      auto simp: xs_def›)+
qed

lemma free_upd_alt_def1:
  free_upd n M c = (let
    M1 = upd_pairs' ([((i,c), (i,0)). i ← [0..<Suc n], i ≠ c]) M;
    M2 = upd_pairs ([((c,i), ∞). i ← [0..<Suc n], i ≠ c]) M1
  in
    M2
  ) (is _ = ?r)
  if 0 < c c ≤ n
proof −
  let ?xs1 = λn. [((c,i), ∞). i ← [0..<Suc n], i ≠ c]
  let ?xs2 = λn. [((i,c), M(i,0)). i ← [0..<Suc n], i ≠ c]
  define xs where xs = ?xs1 n @ ?xs2 n
  let ?t = upd_pairs xs M
  have distinct (map fst xs)
    unfolding xs_def
    apply (clarsimp simp del: upt_Suc simp add: map_concat comp_def if_distrib split: if_split)
     apply (auto split: if_split_asm simp: distinct_map inj_on_def intro!: distinct_concat)
    done
  from ‹c ≤ n› have free_upd n M c = ?t
    by (intro ext, auto simp: free_upd_def)
      (solves ‹subst upd_pairs_upd[OF _ ‹distinct _›] upd_pairs_no_upd, auto simp: xs_def›)+
  also have ... = ?r
    unfolding xs_def
    apply (subst upd_pairs_append')
    subgoal
      by auto
    apply (subst upd_pairs'_upd_pairs)
    subgoal
      using ‹c > 0› by auto
    apply (simp del: upt_Suc add: map_concat)
    apply (intro arg_cong2[where f = upd_pairs] arg_cong[where f = concat])
    by (simp del: upt_Suc)+
  finally show ?thesis .
qed

lemma free_upd_free_upd1:
  free_upd n M c = free_upd1 n M c if c > 0 c ≤ n
proof −
  let ?x1 = λxs. upd_pairs' ([((i,c), (i,0)). i ← xs, i ≠ c]) M
  let ?x2 = λxs. fold (λi M. if i ≠ c then (M((i, c) := M(i, 0))) else M) xs M
  have ∗: ?x1 xs = ?x2 xs for xs
    by (induction xs arbitrary: M) (auto simp: upd_pairs'_Nil upd_pairs'_Cons)
  let ?y1 = λxs. upd_pairs ([((c,i), ∞). i ← xs, i ≠ c])
  let ?y2 = fold (λi M. if i ≠ c then M((c,i) := ∞) else M)
```

have $**$: *?y1 xs M = ?y2 xs M* **for** *xs M*
by (*induction xs arbitrary*: *M*) (*auto simp*: *upd_pairs_Nil upd_pairs_Cons*)
show *?thesis*
unfolding *free_upd_alt_def1*[*OF that*] *free_upd1_def op_mtx_get_def* $*$ $**$ **..**
qed

lemma *free_upd_alt_def*:
free_upd n M c =
fold (λ*i M*. *if i* ≠ *c then* (*M*((*c*,*i*) := ∞, (*i*, *c*) := *M*(*i*, *0*))) *else M*) [*0*..<*Suc n*] *M*
if *c* ≤ *n*
oops

lemma *pre_reset_upd_pre_reset_upd1*:
pre_reset_upd n M c = *pre_reset_upd1 n M c* **if** *c* > *0 c* ≤ *n*
unfolding *pre_reset_upd_def pre_reset_upd1_def free_upd_free_upd1*[*OF that*] **..**

lemma *pre_reset_list_upd_pre_reset_list_upd1*:
pre_reset_list_upd n M cs = *pre_reset_list_upd1 n M cs* **if** ∀ *c* ∈ *set cs*. *0* < *c* ∧ *c* ≤ *n*
unfolding *pre_reset_list_upd_def pre_reset_list_upd1_def*
using *that* **by** (*intro fold_cong*; *simp add*: *pre_reset_upd_pre_reset_upd1*)

lemma *pre_reset_list_transfer'*[*transfer_rule*]:
includes *lifting_syntax* **shows**
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2 n* ===> *list_all2* (*eq_onp* (λ*x*. *0* < *x* ∧ *x* < *Suc n*)) ===>
RI2 n)
pre_reset_list pre_reset_list_upd1
unfolding *rel_fun_def*
apply *safe*
apply (*subst pre_reset_list_upd_pre_reset_list_upd1*[*symmetric*])
subgoal
unfolding *list.rel_eq_onp* **by** (*auto simp*: *eq_onp_def list_all_iff*)
by (*rule pre_reset_list_transfer*[*unfolded rel_fun_def*, *rule_format*])
(*auto simp*: *eq_onp_def list_all2_iff*)

10.9 Imperative Refinement

context
fixes *n* :: *nat*
notes [*id_rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref_import_param*] = *IdI*[*of n*]
begin

interpretation *DBM_Impl n* **.**

sepref_definition *down_impl* **is**
RETURN o PR_CONST (*down_upd n*) :: *mtx_assn*^{d} →$_a$ *mtx_assn*
unfolding *down_upd_alt_def1 upd_pairs_map PR_CONST_def*
unfolding *Let_def prod.case*
unfolding *fold_map comp_def*
unfolding *neutral*[*symmetric*]
by *sepref*

context
notes [*map_type_eqs*] = *map_type_eqI*[*of TYPE*(*nat* $*$ *nat* ⇒ *'e*) *TYPE*(*'e i_mtx*)]

begin

sepref_register
abstr_upd FW'' n Normalized_Zone_Semantics_Impl_Semantic_Refinement.repair_pair n and_entry_upd free_upd1 n restrict_zero_upd n pre_reset_upd1 n

end

lemmas [*sepref_fr_rules*] = *abstr_upd_impl.refine fw_impl_refine_FW''*

sepref_definition *abstr_FW_impl* **is**
uncurry (*RETURN oo PR_CONST* (*abstr_FW_upd n*)) ::
(*list_assn* (*acconstraint_assn clock_assn id_assn*))^k $*_a$ *mtx_assn*^{d} $\rightarrow_a$ *mtx_assn*
unfolding *abstr_FW_upd_def FW''_def*[*symmetric*] *PR_CONST_def* **by** *sepref*

sepref_definition *free_impl* **is**
uncurry (*RETURN oo PR_CONST* (*free_upd1 n*)) ::
$[\lambda(_, i).\ i{\leq}n]_a$ *mtx_assn*^{d} $*_a$ *nat_assn*^{k} $\rightarrow$ *mtx_assn*
unfolding *free_upd1_def op_mtx_set_def*[*symmetric*] *PR_CONST_def* **by** *sepref*

sepref_definition *and_entry_impl* **is**
uncurry2 (*uncurry* ($\lambda x.$ *RETURN ooo and_entry_upd x*)) ::
$[\lambda(((i, j),_),_).\ i{\leq}n \wedge j \leq n]_a$ *nat_assn*^{k} $*_a$ *nat_assn*^{k} $*_a$ *id_assn*^{k} $*_a$ *mtx_assn*^{d} $\rightarrow$ *mtx_assn*
unfolding *and_entry_upd_def* **by** *sepref*

lemmas [*sepref_fr_rules*] = *and_entry_impl.refine*

sepref_definition *restrict_zero_impl* **is**
uncurry (*RETURN oo PR_CONST* (*restrict_zero_upd n*)) ::
$[\lambda(_, i).\ i{\leq}n]_a$ *mtx_assn*^{d} $*_a$ *nat_assn*^{k} $\rightarrow$ *mtx_assn*
unfolding *restrict_zero_upd_def PR_CONST_def* **by** *sepref*

lemmas [*sepref_fr_rules*] = *free_impl.refine restrict_zero_impl.refine*

sepref_definition *pre_reset_impl* **is**
uncurry (*RETURN oo PR_CONST* (*pre_reset_upd1 n*)) ::
$[\lambda(_, i).\ i{\leq}n]_a$ *mtx_assn*^{d} $*_a$ (*clock_assn*)^k $\rightarrow$ *mtx_assn*
unfolding *pre_reset_upd1_def PR_CONST_def* **by** *sepref*

lemmas [*sepref_fr_rules*] = *pre_reset_impl.refine*

sepref_definition *pre_reset_list_impl* **is**
uncurry (*RETURN oo PR_CONST* (*pre_reset_list_upd1 n*)) ::
mtx_assn^d $*_a$ (*list_assn* (*clock_assn*))^k $\rightarrow_a$ *mtx_assn*
unfolding *pre_reset_list_upd1_def PR_CONST_def* **by** *sepref*

sepref_definition *and_entry_repair_impl* **is**
uncurry2 (*uncurry* ($\lambda x.$ *RETURN ooo PR_CONST* (*and_entry_repair_upd n*) *x*)) ::
$[\lambda(((i, j),_),_).\ i{\leq}n \wedge j \leq n]_a$ *nat_assn*^{k} $*_a$ *nat_assn*^{k} $*_a$ *id_assn*^{k} $*_a$ *mtx_assn*^{d} $\rightarrow$ *mtx_assn*
unfolding *and_entry_repair_upd_def PR_CONST_def* **by** *sepref*

private definition
V_dbm'' = V_dbm' n

We use *V_dbm''* because we cannot register *V_dbm'* twice with the refinement framework: we view it as a function first, and as a DBM later.

lemma *V_dbm'_alt_def*:
V_dbm' n = op_amtx_new (Suc n) (Suc n) (V_dbm'')
unfolding *V_dbm''_def* **by** *simp*

notation *fun_rel_syn* (**infixr** $\rightarrow$ *60*)

We need the custom rule here because *V_dbm'* is a higher-order constant

lemma [*sepref_fr_rules*]:
(uncurry0 (return V_dbm''), uncurry0 (RETURN (PR_CONST (V_dbm''))))
$\in$ *unit_assn^k* $\rightarrow_a$ *pure (nat_rel* $\times_r$ *nat_rel* $\rightarrow$ *Id)*
by *sepref_to_hoare sep_auto*

sepref_register *V_dbm'' :: nat* $\times$ *nat* $\Rightarrow$ *_ DBMEntry*

Necessary to solve side conditions of *op_amtx_new*

lemma *V_dbm''_bounded*:
mtx_nonzero V_dbm'' $\subseteq$ *{0..<Suc n}* $\times$ *{0..<Suc n}*
unfolding *mtx_nonzero_def V_dbm''_def V_dbm'_def neutral* **by** *auto*

We need to pre-process the lemmas due to a failure of *TRADE*

lemma *V_dbm''_bounded_1*:
(a, b) $\in$ *mtx_nonzero V_dbm''* $\Longrightarrow$ *a < Suc n*
using *V_dbm''_bounded* **by** *auto*

lemma *V_dbm''_bounded_2*:
(a, b) $\in$ *mtx_nonzero V_dbm''* $\Longrightarrow$ *b < Suc n*
using *V_dbm''_bounded* **by** *auto*

lemmas [*sepref_opt_simps*] = *V_dbm''_def*

sepref_definition *V_dbm_impl* **is**
uncurry0 (RETURN (PR_CONST (V_dbm' n))) :: *unit_assn^k* $\rightarrow_a$ *mtx_assn*
supply *V_dbm''_bounded_1*[*simp*] *V_dbm''_bounded_2*[*simp*]
using *V_dbm''_bounded*
apply (*subst V_dbm'_alt_def*)
unfolding *PR_CONST_def* **by** *sepref*

This form of 'type casting' is used to assert a bound on the natural number.

private definition *make_clock* :: *nat* $\Rightarrow$ *nat* **where** [*sepref_opt_simps*]:
make_clock x = x

lemma *make_clock*[*sepref_import_param*]:
(make_clock, make_clock) $\in$ $[\lambda i.\ i \leq n]_f$ *nat_rel* $\rightarrow$ *nbn_rel (Suc n)*
unfolding *make_clock_def* **by** (*rule frefI*) *simp*

private definition *get_entries m* =
[(make_clock i, make_clock j).
i$\leftarrow$*[0..<Suc n], j*$\leftarrow$*[0..<Suc n],* $(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ (i, j) \neq \infty$*]*

private lemma *get_entries_alt_def*:
get_entries m = [(make_clock i, make_clock j).

$i \leftarrow [0..<Suc\ n],\ j \leftarrow [0..<Suc\ n],\ (i > 0 \vee j > 0) \wedge m\ (i, j) \neq \infty]$
unfolding *get_entries_def* **by** (*intro arg_cong*[**where** $f = concat$] *map_cong*) *auto*

private definition
upd_entry i j M m = *and_entry_repair_upd n j i* (*neg_dbm_entry* (*m* (*i*, *j*))) (*op_mtx_copy M*)

private definition
upd_entries i j m = *map* (λ *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* **by** *simp*

lemmas [*sepref_fr_rules*] = *and_entry_repair_impl.refine*

sepref_definition *upd_entry_impl* **is**
uncurry2 (*uncurry* (λx. *RETURN ooo PR_CONST upd_entry x*)) ::
$[\lambda(((i, j),_),_).\ i \leq n \wedge j \leq n]_a$
nat_assn^k $*_a$ *nat_assn*^{k} $*_a$ *mtx_assn*^{k} $*_a$ *mtx_assn*^{k} $\rightarrow$ *mtx_assn*
unfolding *upd_entry_def PR_CONST_def* **by** *sepref*

This is to ensure that the refinement initially infers the right 'type' for list arguments.

lemma [*intf_of_assn*]:
intf_of_assn AA TYPE($'aa$) $\Longrightarrow$ *intf_of_assn* (*list_assn*(*AA*)) *TYPE*($'aa$ *list*)
by (*rule intf_of_assnI*)

lemmas [*sepref_fr_rules*] = *upd_entry_impl.refine*

sepref_definition *upd_entries_impl* **is**
uncurry2 (*uncurry* (λx. *RETURN ooo PR_CONST upd_entries x*)) ::
$[\lambda(((i, j),_),_).\ i \leq n \wedge j \leq n]_a$
nat_assn^k $*_a$ *nat_assn*^{k} $*_a$ *mtx_assn*^{k} $*_a$ (*list_assn mtx_assn*)^k $\rightarrow$ *list_assn mtx_assn*
unfolding *upd_entries_def PR_CONST_def*
unfolding *map_conv_rev_fold rev_conv_fold*
supply [*sepref_fr_rules*] = *HOL_list_empty_hnr_aux*
by *sepref*

lemma [*sepref_import_param*]: ((=), (=)) $\in$ *Id*$\rightarrow$*Id*$\rightarrow$*Id* **by** *simp*

sepref_definition *get_entries_impl* **is**
RETURN o PR_CONST get_entries ::
mtx_assn^k $\rightarrow_a$ *list_assn* ((*clock_assn*) $\times_a$ (*clock_assn*))
unfolding *get_entries_alt_def PR_CONST_def*
unfolding *map_conv_rev_fold*

unfolding *concat_conv_fold_rev*
supply [*sepref_fr_rules*] = *HOL_list_empty_hnr_aux*
by *sepref*

lemmas [*sepref_fr_rules*] = *upd_entries_impl.refine get_entries_impl.refine*

private lemma *dbm_minus_canonical_upd_alt_def*:
dbm_minus_canonical_upd n xs m =
concat (*map* (λ*ij*. *map* (λ *M*.
and_entry_repair_upd n (*snd ij*) (*fst ij*) (*neg_dbm_entry* (*m* (*fst ij*, *snd ij*))) (*op_mtx_copy M*))
xs) (*get_entries m*))
unfolding *dbm_minus_canonical_upd_def op_mtx_copy_def get_entries_def split_beta make_clock_def*
by *simp*

sepref_definition *dbm_minus_canonical_impl* **is**
uncurry (*RETURN oo PR_CONST* (*dbm_minus_canonical_check_upd n*)) ::
(*list_assn mtx_assn*)^k ∗$_a$ *mtx_assn*^{k} →$_a$ *list_assn mtx_assn*
unfolding *dbm_minus_canonical_check_upd_def dbm_minus_canonical_upd_alt_def PR_CONST_def*
unfolding *upd_entry_def*[*symmetric*] *upd_entries_def*[*symmetric*]
unfolding *filter_conv_rev_fold concat_map_conv_rev_fold rev_conv_fold*
supply [*sepref_fr_rules*] = *HOL_list_empty_hnr_aux*
by *sepref*

lemmas [*sepref_fr_rules*] = *dbm_minus_canonical_impl.refine*

sepref_definition *dbm_subset_fed_impl* **is**
uncurry (*RETURN oo PR_CONST* (*dbm_subset_fed_upd n*)) ::
mtx_assn^d ∗$_a$ (*list_assn mtx_assn*)^d →$_a$ *bool_assn*
unfolding *dbm_subset_fed_upd_alt_def PR_CONST_def*
unfolding *list_ex_foldli filter_conv_rev_fold*
unfolding *short_circuit_conv*
supply [*sepref_fr_rules*] = *HOL_list_empty_hnr_aux*
by *sepref*

end

10.10 Implementation for a Concrete Automaton

context *TA_Impl*
begin

sublocale *TA_Impl_Op*
where *loc_rel* = *loc_rel* **and** *f* = *PR_CONST E_op''* **and** *op_impl* = *E_op''_impl*
unfolding *PR_CONST_def* **by** *standard* (*rule E_op''_impl.refine*)

end

context *TA_Impl*
begin

definition
check_deadlock_dbm l M = *dbm_subset_fed_upd n M* (
[*down_upd n* (*abstr_FW_upd n* (*inv_of_A l*) (*abstr_FW_upd n g*

```
    (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of_A l') (V_dbm' n)) r)
   )). (g, a, r, l') ← trans_fun l
  ]
 )

abbreviation zone_of ([[_]]) where zone_of M ≡ [curry (conv_M M)]_{v,n}

theorem check_deadlock_dbm_correct:
  TA.check_deadlock l [[M]] = check_deadlock_dbm l M if
  [[M]] ⊆ V l ∈ states' Deadlock.canonical' n (curry (conv_M M))
proof −

0. Setup & auxiliary facts

  include lifting_syntax
  note [simp del] = And.simps abstr.simps
  have inv_of_simp: inv_of (conv_A A) l = conv_cc (inv_of A l) for l
    using inv_fun unfolding inv_rel_def b_rel_def fun_rel_def conv_A_def
    by (force split: prod.split simp: inv_of_def)

  have trans_funD: l' ∈ states
    collect_clks (inv_of A l) ⊆ clk_set A collect_clks (inv_of A l') ⊆ clk_set A
    collect_clks g ⊆ clk_set A set r ⊆ clk_set A
    if (g, a, r, l') ∈ set (trans_fun l) for g a r l'
    subgoal
      using ‹l ∈ states'› that trans_impl_states by auto
    subgoal
      by (metis collect_clks_inv_clk_set)
    subgoal
      by (metis collect_clks_inv_clk_set)
    subgoal
      using that ‹l ∈ _› by (intro collect_clocks_clk_set trans_impl_trans_of)
    subgoal
      using that ‹l ∈ _› by (intro reset_clk_set trans_impl_trans_of)
    done

1. Transfer to most abstract DBM operations

  have [transfer_rule]:
    (eq_onp (λ (g, a, r, l'). (g, a, r, l') ∈ set (trans_fun l)) ===> RI2 n)
      (λ(g, a, r, l'). down n
          (abstr_FW n (conv_cc (inv_of A l))
            (abstr_FW n (conv_cc g)
              (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v))
      (λ(g, a, r, l'). down_upd n
          (abstr_FW_upd n (inv_of A l)
            (abstr_FW_upd n g (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of A l') (V_dbm'
n)) r))
          ))
     (is (_ ===> RI2 n) (λ (g, a, r, l'). ?f g a r l') (λ (g, a, r, l'). ?g g a r l'))
  proof −
    {
    fix g a r l' assume (g, a, r, l') ∈ set (trans_fun l)
    have ∗: rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri (conv_ac ac) ac
      if constraint_clk ac > 0 constraint_clk ac ≤ n for ac
      using that by (cases ac) (auto simp: eq_onp_def ri_def)
```

```
  have ∗: list_all2 (rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri) (conv_cc g) g
    if collect_clks g ⊆ clk_set A for g
    unfolding list_all2_map1 using that clock_range
    by (auto simp: collect_clks_def intro!: ∗ list.rel_refl_strong)
  have [transfer_rule]:
    list_all2 (rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri) (conv_cc g) g
    list_all2 (rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri)
      (conv_cc (inv_of A l)) (inv_of A l)
    list_all2 (rel_acconstraint (eq_onp (λx. 0 < x ∧ x < Suc n)) ri)
      (conv_cc (inv_of A l')) (inv_of A l')
    by (intro ∗ trans_funD[OF ‹(g, a, r, l') ∈ set (trans_fun l)›])+
  have [transfer_rule]:
    list_all2 (eq_onp (λx. 0 < x ∧ x < Suc n)) r r
    using trans_funD(5)[OF ‹(g, a, r, l') ∈ set (trans_fun l)›] clock_range
    by (auto 4 3 dest: bspec subsetD simp: eq_onp_def list_all2_same)
  have RI2 n (?f g a r l') (?g g a r l')
    by transfer_prover
  } then show ?thesis
  by (intro rel_funI, clarsimp simp: eq_onp_def)
qed
have [transfer_rule]:
  (eq_onp (λl'. l' = l) ===> list_all2 (eq_onp (λ(g,a,r,l'). (g,a,r,l') ∈ set (trans_fun l))))
    trans_fun trans_fun

  unfolding rel_fun_def eq_onp_def by (auto intro: list.rel_refl_strong)
note [transfer_rule] = dbm_subset_fed_transfer
have [transfer_rule]: eq_onp (λl'. l' = l) l l (eq_onp (λx. x < Suc n)) n n
  by (auto simp: eq_onp_def)
have ∗:
  (dbm_subset_fed_check n (curry (conv_M M))
      (map (λ(g, a, r, l').
             down n
       (abstr_FW n (conv_cc (inv_of A l))
         (abstr_FW n (conv_cc g)
           (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v))
       (trans_fun l))) =
  (dbm_subset_fed_upd n M
      (map (λ(g, a, r, l').
             down_upd n
       (abstr_FW_upd n (inv_of A l)
        (abstr_FW_upd n g (pre_reset_list_upd1 n (abstr_FW_upd n (inv_of A l') (V_dbm'
n)) r))
       ))
      (trans_fun l)))

  by transfer_prover
```

2. Semantic argument establishing equivalences between zones and DBMs

```
  have ∗∗:
    [down n (abstr_FW n (conv_cc (inv_of A l)) (abstr_FW n (conv_cc g)
        (pre_reset_list n (abstr_FW n (conv_cc (inv_of A l')) V_dbm v) r) v) v)]_{v,n}
  = (zone_set_pre ({u. u ⊢ inv_of (conv_A A) l'} ∩ V) r ∩ {u. ∀ x ∈ set r. u x ≥ 0}
      ∩ {u. u ⊢ conv_cc g} ∩ {u. u ⊢ inv_of (conv_A A) l})^↓ ∩ V
    if (g, a, r, l') ∈ set(trans_fun l) for g a r l'
```

proof −
from *trans_funD*[*OF* ‹(*g*, *a*, *r*, *l′*) ∈ *set* (*trans_fun l*)›] **have** *clock_conditions*:
∀ *c*∈*collect_clks* (*conv_cc* (*inv_of A l*)). 0 < *c* ∧ *c* ≤ *n*
∀ *c*∈*collect_clks* (*conv_cc* (*inv_of A l′*)). 0 < *c* ∧ *c* ≤ *n*
∀ *c*∈*collect_clks* (*conv_cc g*). 0 < *c* ∧ *c* ≤ *n*
∀ *c*∈*set r*. 0 < *c* ∧ *c* ≤ *n*
using ‹*l* ∈ *states′*› *clock_range* **by** (*auto simp*: *constraint_clk_conv_cc*)
have *structural_conditions*:
abstr_FW n (*conv_cc* (*inv_of A l′*)) *V_dbm v* 0 0 ≤ 0
∀ *x*∈*set r*. *abstr_FW n* (*map conv_ac* (*inv_of A l′*)) *V_dbm v* 0 *x* ≤ 0
subgoal
using *abstr_FW_diag_preservation*[*of n V_dbm conv_cc* (*inv_of A l′*) *v*]
by (*simp add*: *V_dbm_def*)
subgoal
using ‹∀ *c*∈*set r*. 0 < *c* ∧ *c* ≤ *n*›
by (*auto 4 3 simp*: *V_dbm_def intro*: *abstr_FW_mono order.trans*)
done
note *side_conditions* = *clock_conditions structural_conditions*
abstr_FW_canonical[*unfolded canonical′_def*] *abstr_FW_canonical*
show *?thesis*
apply (*subst dbm.down_correct*, *rule side_conditions*)
apply (*subst dbm.abstr_FW_correct*, *rule side_conditions*)
apply (*subst dbm.abstr_FW_correct*, *rule side_conditions*)
apply (*subst dbm.pre_reset_list_correct*, (*rule side_conditions*)+)
apply (*subst dbm.abstr_FW_correct*, (*rule side_conditions*)+)
apply (*simp add*: *inv_of_simp dbm.V_dbm_correct atLeastLessThanSuc_atLeastAtMost Int_commute*)
done
qed
have ∗∗:
(⋃ *x*∈*set* (*trans_fun l*).
dbm.zone_of
(*case x of* (*g*, *a*, *r*, *l′*) ⇒
down n
(*abstr_FW n* (*conv_cc* (*inv_of A l*))
(*abstr_FW n* (*conv_cc g*)
(*pre_reset_list n* (*abstr_FW n* (*conv_cc* (*inv_of A l′*)) *V_dbm v*) *r*) *v*) *v*)))
= (⋃ (*g*,*a*,*r*,*l′*)∈*set* (*trans_fun l*).
((*zone_set_pre* ({*u*. *u* ⊢ *inv_of* (*conv_A A*) *l′*} ∩ *V*) *r* ∩ {*u*. ∀ *x* ∈ *set r*. *u x* ≥ 0}
∩ {*u*. *u* ⊢ *conv_cc g*} ∩ {*u*. *u* ⊢ *inv_of* (*conv_A A*) *l*})↓ ∩ *V*)
)

by (*auto simp*: ∗∗)

3. Putting it all together

have *transD*: ∃ *g′*. (*g′*, *a*, *r*, *l′*) ∈ *set* (*trans_fun l*) ∧ *g* = *conv_cc g′*
if *conv_A A* ⊢ *l* ⟶^*g*,*a*,*r* *l′* **for** *g a r l′*
using *trans_of_trans_impl*[*OF* _ ‹*l* ∈ *states′*›] *that*
unfolding *trans_of_def conv_A_def* **by** (*auto 5 0 split*: *prod.split_asm*)
have *transD2*:
conv_A A ⊢ *l* ⟶^*conv_cc g*,*a*,*r* *l′* **if** (*g*, *a*, *r*, *l′*) ∈ *set* (*trans_fun l*) **for** *g a r l′*
using *trans_impl_trans_of*[*OF that* ‹*l* ∈ *states′*›]
unfolding *trans_of_def conv_A_def* **by** (*auto 4 3 split*: *prod.split*)
show *?thesis*

```
  unfolding TA.check_deadlock_alt_def[OF ‹_ ⊆ V›] check_deadlock_dbm_def inv_of_A_def
∗[symmetric]
    apply (subst dbm.dbm_subset_fed_check_correct[symmetric, OF that(3)])
    apply (simp add: ∗∗)
    apply safe
    subgoal for x
      by (auto 4 5 elim!: transD[elim_format] dest: subsetD)
    subgoal for x
      apply (drule subsetD, assumption)
      apply clarsimp
      subgoal for g a r l'
        apply (inst_existentials
          (zone_set_pre ({u. u ⊢ inv_of (conv_A A) l'} ∩ V) r ∩ {u. ∀x∈set r. 0 ≤ u x}
           ∩ {u. u ⊢ conv_cc g} ∩ {u. u ⊢ inv_of (conv_A A) l})↓ ∩ V conv_cc g a r l')
        apply (auto dest: transD2)
        done
      done
    done
qed

lemmas [sepref_fr_rules] =
  V_dbm_impl.refine abstr_FW_impl.refine pre_reset_list_impl.refine
  down_impl.refine dbm_subset_fed_impl.refine

sepref_definition check_deadlock_impl is
  uncurry (RETURN oo PR_CONST check_deadlock_dbm) ::
  location_assn^k ∗a (mtx_assn n)^d →a bool_assn
  unfolding check_deadlock_dbm_def PR_CONST_def
  unfolding case_prod_beta
  unfolding map_conv_rev_fold
  apply (rewrite HOL_list.fold_custom_empty)
  by sepref

end
```

10.11 Checking a Property on the Reachable Set

```
locale Worklist_Map2_Impl_check =
  Worklist_Map2_Impl_finite +
  fixes Q :: 'a ⇒ bool
  fixes Qi
  assumes Q_refine: (Qi,RETURN o PR_CONST Q) ∈ A^d →a bool_assn
  and F_False: F = (λ _. False)
  and Q_mono: ⋀ a b. a ≼ b ⟹ ¬ empty a ⟹ reachable a ⟹ reachable b ⟹ Q a ⟹ 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 (λb. ¬b)
      (λ passed' _.
        do {
```

```
        passed' ← SPEC (λl. set l = passed');
        nfoldli passed' (λb. ¬b)
          (λv' _.
            if Q v' then RETURN True else RETURN False
          )
          False
      }
    )
    False;
  RETURN r
}

lemma check_passed_correct:
  check_passed ≤ SPEC (λ r. (r ⟷ (∃ a. reachable a ∧ ¬ empty a ∧ Q a)))
proof −
  have [simp]: F_reachable = False
    unfolding F_reachable_def F_False Search_Space_Defs.F_reachable_def by simp
  define outer_inv where outer_inv passed done todo r ≡
    set done ∪ set todo = ran passed ∧
    (¬ r ⟶ (∀ S ∈ set done. ∀ x ∈ S. ¬ Q x)) ∧
    (r ⟶ (∃ a. reachable a ∧ ¬ empty a ∧ Q a))
   for passed :: 'c ⇒ 'a set option and done todo and r :: bool
  define inner_inv where inner_inv passed done todo r ≡
    set done ∪ set todo = passed ∧
    (¬ r ⟶ (∀ x ∈ set done. ¬ Q x)) ∧
    (r ⟶ (∃ a. reachable a ∧ ¬ empty a ∧ Q a))
   for passed :: 'a set and done todo and r :: bool
  show ?thesis
    supply [refine_vcg] = pw_algo_map2_correct
    unfolding check_passed_def
    apply (refine_rcg refine_vcg)
    subgoal
      by auto
    subgoal for _ brk_false passed range_list
      apply (rule nfoldli_rule[where I = outer_inv passed])


      subgoal
        unfolding outer_inv_def
        by auto


        apply (refine_rcg refine_vcg)
      subgoal for current done todo r xs
        apply clarsimp
        apply (rule nfoldli_rule[where I = inner_inv current])

        subgoal
          unfolding inner_inv_def
          by auto


        subgoal for p x l1 l2 r'
          by (clarsimp simp: inner_inv_def outer_inv_def map_set_rel_def)
```

(*metis Sup_insert Un_insert_left insert_iff subset_Collect_conv*)

subgoal for *p l1 l2 r'*
unfolding *inner_inv_def outer_inv_def*
by (*metis append.assoc append_Cons append_Nil set_append*)

subgoal for *p r'*
unfolding *inner_inv_def outer_inv_def*
by *clarsimp*
done

subgoal for *l1 l2 r*
unfolding *outer_inv_def*
by *auto*

subgoal for *r*
unfolding *outer_inv_def*
apply *clarsimp*
apply (*elim allE impE*)
apply (*intro conjI*; *assumption*)
apply *safe*
apply (*drule Q_mono*, *assumption*+)
unfolding *map_set_rel_def*
by *auto*
done
done
qed

schematic_goal *pw_algo_map2_refine*:
(*?x*, *uncurry0* (*PR_CONST pw_algo_map2*)) ∈
unit_assn^k →$_a$ *bool_assn* ×$_a$ *hm.hms_assn'* *K* (*lso_assn A*)
unfolding *PR_CONST_def hm.hms_assn'_id_hms_assn*[*symmetric*] **by** (*rule pw_algo_map2_impl.refine_raw*)

sepref_register *pw_algo_map2*

sepref_register *PR_CONST Q*

sepref_thm *check_passed_impl* **is**
uncurry0 check_passed :: *unit_assn*^{k} →$_a$ *bool_assn*
supply [*sepref_fr_rules*] = *pw_algo_map2_refine ran_of_map_impl.refine lso_id_hnr Q_refine*
using *pure_K left_unique_K right_unique_K*
unfolding *check_passed_def*
apply (*rewrite* **in** *Q PR_CONST_def*[*symmetric*])
unfolding *hm.hms_fold_custom_empty*
unfolding *list_of_set_def*[*symmetric*]
by *sepref*

concrete_definition (**in** −) *check_passed_impl*
uses *Worklist_Map2_Impl_check.check_passed_impl.refine_raw* **is** (*uncurry0 ?f*,_)∈_

lemma *check_passed_impl_hnr*:
(*uncurry0* (*check_passed_impl succsi* a_0*i Fi Lei emptyi keyi copyi tracei Qi*),
uncurry0 (*RETURN* (∃ *a*. *reachable a* ∧ ¬ *empty a* ∧ *Q a*)))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
using
check_passed_impl.refine[
OF Worklist_Map2_Impl_check_axioms,
FCOMP check_passed_correct[*THEN Id_SPEC_refine*, *THEN nres_relI*]
]
by (*simp add*: *RETURN_def*)

end

10.12 Complete Deadlock Checker

Miscellaneous Properties **context** *E_From_Op_Bisim*
begin

More general variant of *?F_rel* = (λ(*l*, *D*). *?F l* ∧ ¬ *check_diag n D*) ⟹ (∃ *l′ D′*. *E*** *a*$_0$ (*l′*, *D′*) ∧ *?F_rel* (*l′*, *D′*)) = (∃ *l′ D′*. *E_from_op*** *a*$_0$ (*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*** *a*$_0$ (*l′*, *D′*) ∧ *P* (*l′*, *D′*)) ⟷ (∃ *l′ D′*. *E_from_op*** *a*$_0$ (*l′*, *D′*) ∧ *P* (*l′*, *D′*))
by (*rule E_E*$_1$*_steps_equiv*[*OF E_E_from_op_step E_from_op_E_step E_from_op_wf_state*];
(*assumption* | *rule assms*))

end

context *Regions_TA*
begin

lemma *check_deadlock_anti_mono*:
check_deadlock l Z **if** *check_deadlock l Z′ Z* ⊆ *Z′*
using *that* **unfolding** *check_deadlock_def* **by** *auto*

lemma *global_clock_numbering*:
global_clock_numbering A v n
using *valid_abstraction clock_numbering_le clock_numbering* **by** *blast*

Variant of ⟦(∀ *c*. *0* < *v c* ∧ (∀ *x y*. *v x* ≤ *n* ∧ *v y* ≤ *n* ∧ *v x* = *v y* ⟶ *x* = *y*)) ∧ (∀ *c*∈*clk_set ?A*. *v c* ≤ *n*) ∧ (∀ *k*≤*n*. *0* < *k* ⟶ (∃ *c*. *v c* = *k*)); *Closure.valid_abstraction ?A X* (λ*x xa*. *real* (*k x xa*))⟧ ⟹ *Bisimulation_Invariant* (λ(*l*, *Z*) (*l′*, *Z′*). *?A* ⊢ ⟨*l*, *Z*⟩ ⇝$_\beta$ ⟨*l′*, *Z′*⟩ ∧ *Z′* ≠ {}) (λ(*l*, *D*) (*l′*, *D′*). ∃ *a*. *?A* ⊢′ ⟨*l*, *D*⟩ ⇝$_{\mathcal{N}a}$ ⟨*l′*, *D′*⟩ ∧ [*D′*]$_{v,n}$ ≠ {}) (λ(*l*, *Z*) (*l′*, *D*). *l* = *l′* ∧ *Z* = [*D*]$_{v,n}$) (λ_. *True*) (λ(*l*, *y*). *local.valid_dbm y*) without emptiness.

lemma *bisim*:
Bisimulation_Invariant
(λ (*l*, *Z*) (*l′*, *Z′*). *A* ⊢ ⟨*l*, *Z*⟩ ⇝$_\beta$ ⟨*l′*, *Z′*⟩)
(λ (*l*, *D*) (*l′*, *D′*). ∃ *a*. *A* ⊢′ ⟨*l*, *D*⟩ ⇝$_{\mathcal{N}(a)}$ ⟨*l′*, *D′*⟩)
(λ (*l*, *Z*) (*l′*, *D*). *l* = *l′* ∧ *Z* = [*D*]$_{v,n}$)
(λ _. *True*) (λ (*l*, *D*). *valid_dbm D*)
proof (*standard*, *goal_cases*)
— $\beta \Rightarrow \mathcal{N}$
case (*1 a b a′*)

```
  then show ?case
    by (blast elim: norm_beta_complete1[OF global_clock_numbering valid_abstraction])
next
  — 𝒩 ⇒ β
  case (2 a a' b')
  then show ?case
    by (blast intro: norm_beta_sound''[OF global_clock_numbering valid_abstraction])
next
  — β invariant
  case (3 a b)
  then show ?case
    by simp
next
  — 𝒩 invariant
  case (4 a b)
  then show ?case
    by (auto intro: valid_dbm_step_z_norm''[OF global_clock_numbering valid_abstraction])
qed

lemma steps_z_beta_reaches:
  reaches (l, Z) (l', Z') if A ⊢ ⟨l, Z⟩ ↝β* ⟨l', Z'⟩ Z' ≠ {}
  using that
proof (induction (l', Z') arbitrary: l' Z' rule: rtranclp_induct)
  case base
  then show ?case
    by blast
next
  case (step y l'' Z'')
  obtain l' Z' where [simp]: y = (l', Z')
    by force
  from step.prems step.hyps(2) have Z' ≠ {}
    by (clarsimp simp: step_z_beta'_empty)
  from step.prems step.hyps(3)[OF ‹y = _› this] step.hyps(2) show ?case
    including graph_automation_aggressive by auto
qed

lemma reaches_steps_z_beta_iff:
  reaches (l, Z) (l', Z') ⟷ A ⊢ ⟨l, Z⟩ ↝β* ⟨l', Z'⟩ ∧ Z' ≠ {} if Z ≠ {}
  apply safe
  subgoal
    including graph_automation_aggressive by (induction rule: rtranclp_induct) auto
  using that by (auto dest: steps_z_beta_reaches elim: rtranclp.cases)

end

lemma dbm_int_init_dbm:
  dbm_int (curry init_dbm) n
  unfolding init_dbm_def by auto

context TA_Start
begin

lemma wf_dbm_canonical'D:
  Deadlock.canonical' n (curry (conv_M D)) if wf_dbm D
```

using *that* **unfolding** *wf_dbm_def Deadlock.canonical'_def check_diag_conv_M_iff* **by** *simp*

lemma *subsumes_dbm_subsetD*:
dbm_subset n M M' **if** *subsumes n* (*l*, *M*) (*l'*, *M'*) ¬ *check_diag n M*
using *that* **by** (*cases l = l'*) (*auto simp*: *subsumes_simp_1 subsumes_simp_2*)

lemma *subsumes_loc_eqD*:
l = l' **if** *subsumes n* (*l*, *M*) (*l'*, *M'*) ¬ *check_diag n M*
using *that* **by** (*cases l = l'*) (*auto simp*: *subsumes_simp_1 subsumes_simp_2*)

lemma *init_dbm_zone*:
[*curry* (*init_dbm* :: *real DBM'*)]$_{v,n}$ = {*u*. ∀ *c*∈{*1*..*n*}. *u c* = *0*}
using *init_dbm_semantics* **by** *blast*

lemma *not_check_deadlock_mono*:
(*case a of* (*l*, *M*) ⇒ ¬ *TA.check_deadlock l* (*dbm.zone_of* (*curry* (*conv_M M*)))) ⟹
a ∼ *b* ⟹ *wf_state a* ⟹ *wf_state b* ⟹
case b of (*l*, *M*) ⇒ ¬ *TA.check_deadlock l* (*dbm.zone_of* (*curry* (*conv_M M*)))
unfolding *TA.check_deadlock_def state_equiv_def dbm_equiv_def* **by** *auto*

lemma *not_check_deadlock_non_empty*:
Z ≠ {} **if** ¬ *TA.check_deadlock l Z*
using *that* **unfolding** *TA.check_deadlock_def* **by** *auto*

end

context *TA_Impl*
begin

lemma *op_E_from_op_iff*:
op.E_from_op = *E_op''.E_from_op*
unfolding *PR_CONST_def* **..**

lemmas *reachable_states* = *reachable_states*[*unfolded op_E_from_op_iff*]

lemma *E_op''_states*:
l' ∈ *states* **if** *E_op''.E_from_op* (*l*, *M*) (*l'*, *M'*) *l* ∈ *states*
using *that* **by** (*force simp*: *a*$_0$*_def state_set_def E_op''.E_from_op_def*)

Instantiating the Checker Algorithm **corollary** *check_deadlock_dbm_correct'*:
assumes *l* ∈ *states' wf_state* (*l*, *M*)
shows *TA.check_deadlock l* (*dbm.zone_of* (*curry* (*conv_M M*))) = *check_deadlock_dbm l M*
apply (*rule check_deadlock_dbm_correct*)
using *assms*
unfolding *wf_state_def Deadlock.canonical'_def prod.case*
apply (*blast dest*: *wf_dbm_altD*)
apply (*rule assms*)
using *assms*
unfolding *wf_state_def Deadlock.canonical'_def prod.case*
apply −
apply (*drule wf_dbm_altD*(*1*))
unfolding *canonical'_conv_M_iff check_diag_conv_M_iff* **by** *simp*

corollary *check_deadlock_dbm_correct''*:
assumes *E_op''.E_from_op*** a_0 (l, M)
shows *TA.check_deadlock l* (*dbm.zone_of* (*curry* (*conv_M M*))) = *check_deadlock_dbm l M*
using *assms*
apply −
apply (*rule check_deadlock_dbm_correct'*)
apply (*drule reachable_states*, *use states'_states* **in** *auto*; *fail*)
unfolding *wf_state_def prod.case* **apply** (*erule E_op''.reachable_wf_dbm*)
done

lemma *not_check_deadlock_non_empty*:
$Z \neq \{\}$ **if** ¬ *TA.check_deadlock l Z*
using *that* **unfolding** *TA.check_deadlock_def* **by** *auto*

sepref_register *check_deadlock_dbm* :: *'s* ⇒ *int DBMEntry i_mtx* ⇒ *bool*

sepref_definition *check_deadlock_neg_impl* **is**
RETURN o (λ(*l*, *M*). ¬ *check_deadlock_dbm l M*) :: (*location_assn* ×$_a$ (*mtx_assn n*))^d →$_a$ *bool_assn*
supply [*sepref_fr_rules*] = *check_deadlock_impl.refine*
by *sepref*

lemma *not_check_deadlock_compatible*:
assumes
(*case a of* (*l*, *Z*) ⇒ λ(*l'*, *D'*). *l'* = *l* ∧ *dbm.zone_of* (*curry* (*conv_M D'*)) = *Z*) *b*
case b of (*l*, *M*) ⇒ *l* ∈ *states'* ∧ *wf_state* (*l*, *M*)
shows
(*case a of* (*l*, *Z*) ⇒ ¬ *TA.check_deadlock l Z*) = (*case b of* (*l*, *M*) ⇒ ¬ *check_deadlock_dbm l M*)
using *assms* **by** (*auto simp*: *check_deadlock_dbm_correct'*[*symmetric*])

lemma *deadlock_check_diag*:
¬ *check_diag n M* **if** ¬ *check_deadlock_dbm l M E_op''.E_from_op*** a_0 (l, M)
using *that*(*1*)
apply (*subst* (*asm*) *check_deadlock_dbm_correct'*[*symmetric*])
subgoal
using *reachable_states that*(*2*) *states'_states* **by** *auto*
subgoal
unfolding *wf_state_def* **using** *E_op''.reachable_wf_dbm*[*OF that*(*2*)] **by** *simp*
using *canonical_check_diag_empty_iff* **by** (*blast dest*: *not_check_deadlock_non_empty*)

lemma *norm_final_bisim*:
Bisimulation_Invariant
(λ(*l*, *D*) (*l'*, *D'*). ∃ *a*. *step_z_norm''* (*conv_A A*) *l D a l' D'*)
E_op''.E_from_op
(λ (*l*, *M*) (*l'*, *D'*). *l'* = *l* ∧ [*curry* (*conv_M D'*)]$_{v,n}$ = [*M*]$_{v,n}$)
(λ(*l*, *y*). *valid_dbm y*) *wf_state*
by (*rule*
Bisimulation_Invariant_sim_replace[*OF*
Bisimulation_Invariant_composition[*OF*
step_z_norm''_step_impl'_equiv[*unfolded step_impl'_E*] *E_op''.E_from_op_bisim*
]
])

(*auto simp add*: *state_equiv_def dbm_equiv_def*)

interpretation *bisim*:
Bisimulation_Invariant
$\lambda(l, Z)\ (l', Z').$ *step_z_beta'* (*conv_A A*) *l Z l' Z'*
$\lambda a\ b.$ *op.E_from_op a b*
$\lambda(l, Z)\ (l', D').\ l' = l \wedge [curry\ (conv_M\ D')]_{v,n} = Z$
$\lambda_.$ *True* $\lambda(l, M).\ l \in states \wedge wf_state\ (l, M)$
unfolding *op_E_from_op_iff*
by (*rule Bisimulation_Invariant_strengthen_post'*,
(*rule Bisimulation_Invariant_composition*[*OF TA.bisim norm_final_bisim*]
Bisimulation_Invariant_sim_replace
)+) (*auto 4 3 dest*: *E_op''_states wf_dbm_D*(*3*) *simp*: *wf_state_def*)

lemmas *beta_final_bisim* = *bisim.Bisimulation_Invariant_axioms*

definition
is_start_in_states = (*trans_fun* $l_0 \neq []$)

lemma *is_start_in_states*:
$l_0 \in$ *Simulation_Graphs_TA.state_set A* **if** *is_start_in_states*
proof −
from *that* **obtain** *g a r l'* **where** (*g*, *a*, *r*, *l'*) ∈ *set* (*trans_fun* l_0)
by (*cases hd* (*trans_fun* l_0) *rule*: *prod_cases4*)
(*auto dest*: *hd_in_set simp*: *is_start_in_states_def*)
from *trans_impl_trans_of*[*OF this*] *states'_states* **have** $A \vdash l_0 \longrightarrow^{g,a,r} l'$
by *simp*
then show *?thesis*
unfolding *Simulation_Graphs_TA.state_set_def trans_of_def* **by** *auto*
qed

lemma *deadlocked_if_not_is_start_in_states*:
deadlocked (l_0, Z_0) **if** ¬ *is_start_in_states*
proof −
have ∗: *False* **if** $A \vdash l_0 \longrightarrow^{g,a,r} l'$ **for** *g a r l'*
using *trans_of_trans_impl*[*OF that*] ‹¬ _› *states'_states* **unfolding** *is_start_in_states_def*
by *auto*
{ fix *l g2 a2 r2 l'* **assume** *A*: $conv_A\ A \vdash l \longrightarrow^{g2,a2,r2} l'$
obtain *g1 a1 r1* **where** ∗∗: $A \vdash l \longrightarrow^{g1,a1,r1} l'$
using *A* **unfolding** *trans_of_def conv_A_def* **by** (*cases A*) *force*
then have ∃ *g1 a1 r1*. $A \vdash l \longrightarrow^{g1,a1,r1} l'$
by *auto*
} note ∗∗ = *this*
show *?thesis*
unfolding *deadlocked_def*
apply (*rule ccontr*)
apply *clarsimp*
apply (*cases rule*: *step'.cases*, *assumption*)
apply (*cases rule*: *step_a.cases*, *assumption*)
apply (*auto 4 3 elim*: ∗ *dest*: ∗∗ *step_delay_loc*)
done
qed

lemma *deadlock_if_not_is_start_in_states*:

deadlock (l_0, Z_0) **if** ¬ *is_start_in_states*
unfolding *deadlock_def* **using** *deadlocked_if_not_is_start_in_states*[*OF that*] **by** *blast*

sepref_definition *is_start_in_states_impl* **is**
uncurry0 (*RETURN is_start_in_states*) :: *unit_assn*^{k} →$_a$ *bool_assn*
unfolding *is_start_in_states_def* **by** *sepref*

Turning this into a Hoare triple:

thm *is_start_in_states_impl.refine*[*to_hnr*, *unfolded hn_refine_def*, *simplified*]
lemma *is_start_in_states_impl_hoare*:
<*emp*> *is_start_in_states_impl*
<λ*r*. ↑ ((*r* ⟶ l_0 ∈ *Simulation_Graphs_TA.state_set A*)
∧ (¬*r* ⟶ *deadlocked* (l_0, λ_ . 0))
)>$_t$
by (*sep_auto*
intro: *deadlocked_if_not_is_start_in_states*
simp: *mod_star_conv is_start_in_states*[*simplified*]
heap: *is_start_in_states_impl.refine*[*to_hnr*, *unfolded hn_refine_def*, *simplified*]
)

lemma *deadlock_if_deadlocked*:
deadlock y **if** *deadlocked y*
using *that* **unfolding** *deadlock_def* **by** (*inst_existentials y*) *auto*

lemma *is_start_in_states_impl_hoare′*:
<*emp*> *is_start_in_states_impl*
<λ*r*. ↑ ((*r* ⟶ l_0 ∈ *Simulation_Graphs_TA.state_set A*)
∧ (¬*r* ⟶ (∃ u_0. (∀ *c*∈{*1*..*n*}. u_0 *c* = *0*) ∧ *deadlock* (l_0, u_0))))
)>$_t$
by (*rule cons_post_rule*[*OF is_start_in_states_impl_hoare*]) (*auto intro*: *deadlock_if_deadlocked*)

end

context *Reachability_Problem_Impl*
begin

context
assumes l_0*_in_A*: l_0 ∈ *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*. ∀ *c*∈{*1*..*n*}. *u c* = *0*}
apply *standard*
subgoal
by (*intro* l_0*_state_set* l_0*_in_A*)
subgoal
unfolding *V′_def*
apply *safe*
subgoal for *x*
unfolding *V_def* **by** *auto*
apply (*inst_existentials curry init_dbm* :: *real DBM*)
apply (*simp add*: *init_dbm_zone*)
by (*rule dbm_int_init_dbm*)
subgoal

by *auto*
done

interpretation *bisim*:
Bisimulation_Invariant
$\lambda(l, Z)\ (l', Z').$ *step_z_beta'* (*conv_A A*) $l\ Z\ l'\ Z'$
$\lambda a\ b.$ *op.E_from_op a b*
$\lambda(l, Z)\ (l', D').\ l' = l \wedge [curry\ (conv_M\ D')]_{v,n} = Z$
$\lambda_.$ *True* $\lambda(l, M).\ l \in states \wedge wf_state\ (l, M)$
by (*rule beta_final_bisim*)

lemma *check_deadlock*:
$(\exists\, a.\ E_op''.E_from_op^{**}\ a_0\ a \wedge$
$\neg\ (case\ a\ of\ (l, M) \Rightarrow check_diag\ n\ M) \wedge (case\ a\ of\ (l, M) \Rightarrow \neg\ check_deadlock_dbm\ l\ M))$
$\longleftrightarrow (\exists\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \wedge deadlock\ (l_0, u_0))$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof −

TA.reaches corresponds to non-empty steps of *step_z_beta'*

bisim.A.reaches corresponds to steps of *step_z_beta'*

E_op''.reaches a_0 corresponds to steps of *op.E_from_op* (*E_op''*)

have *?r* $\longleftrightarrow (\exists\, l\ Z.$ *TA.reaches* $(l_0, \{u.\ \forall\, c{\in}\{1..n\}.\ u\ c = 0\})\ (l, Z) \wedge \neg$*TA.check_deadlock l Z*)
using *TA.deadlock_check* **unfolding** *TA.a0_def from_R_def* **by** *simp*
also have
$\ldots \longleftrightarrow (\exists\, l\ Z.$ *bisim.A.reaches* $(l_0, \{u.\ \forall\, c{\in}\{1..n\}.\ u\ c = 0\})\ (l, Z) \wedge \neg$*TA.check_deadlock l Z*)
apply *safe*
subgoal for *l Z*
by (*subst* (*asm*) *TA.reaches_steps_z_beta_iff*) *auto*
subgoal for *l Z*
by (*subst TA.reaches_steps_z_beta_iff*) (*auto dest*: *not_check_deadlock_non_empty*)
done
also have $\ldots \longleftrightarrow (\exists\, l\ M.$ *E_op''.E_from_op*** $a_0\ (l, M) \wedge \neg$ *check_deadlock_dbm l M*)
using *bisim.reaches_ex_iff*[**where**
$\varphi = \lambda\ (l, Z).\ \neg$*TA.check_deadlock l Z* **and** $\psi = \lambda(l, M).\ \neg$ *check_deadlock_dbm l M*,
OF not_check_deadlock_compatible, *of* $(l_0, \{u.\ \forall\, c{\in}\{1..n\}.\ u\ c = 0\})\ a_0$
]
using *wf_state_init init_dbm_zone states'_states* **unfolding** *a0_def* **by** *force*
also have $\ldots \longleftrightarrow$ *?l*
by (*auto 4 4 dest*: *deadlock_check_diag*)
finally show *?thesis* **..**
qed

lemma *check_deadlock'*:
$(\nexists\, a.\ E_op''.E_from_op^{**}\ a_0\ a \wedge$
$\neg\ (case\ a\ of\ (l, M) \Rightarrow check_diag\ n\ M) \wedge (case\ a\ of\ (l, M) \Rightarrow \neg\ check_deadlock_dbm\ l\ M))$
$\longleftrightarrow (\forall\, u_0.\ (\forall\, c \in \{1..n\}.\ u_0\ c = 0) \longrightarrow \neg\ deadlock\ (l_0, u_0))$ (**is** *?l* $\longleftrightarrow$ *?r*)
using *check_deadlock* **by** *auto*

context
assumes $F = (\lambda\ _.\ False)$
begin

```
interpretation Worklist_Map2_Impl_check
  op.E_from_op a₀ F_rel subsumes n succs λ (l, M). check_diag n M subsumes'
  λ (l, M). F l state_assn'
 succs_impl a₀_impl F_impl subsumes_impl emptiness_check_impl fst return o fst state_copy_impl
  tracei location_assn λ(l, M). ¬ check_deadlock_dbm l M check_deadlock_neg_impl
  apply standard
  subgoal
    using check_deadlock_neg_impl.refine unfolding PR_CONST_def .
  subgoal
    unfolding F_rel_def unfolding ‹F = _› by auto
  subgoal for a b
    apply (clarsimp simp: E_op''.reachable_def)
    apply (subst (asm) check_deadlock_dbm_correct''[symmetric], assumption)
    apply (subst (asm) check_deadlock_dbm_correct''[symmetric], assumption)
    apply (frule subsumes_loc_eqD, assumption)
    apply (drule subsumes_dbm_subsetD, assumption)
    apply (drule dbm_subset_conv)
    apply (subst (asm) dbm_subset_correct''[symmetric])
   by (auto dest: TA.check_deadlock_anti_mono E_op''.reachable_wf_dbm simp: E_op''.reachable_def)
  done

lemmas check_passed_impl_hnr =
  check_passed_impl_hnr[unfolded op.reachable_def, unfolded op_E_from_op_iff check_deadlock]

end

end

definition
  check_passed_impl_start ≡ do {
    r1 ← is_start_in_states_impl;
    if r1 then do {
     r2 ← check_passed_impl succs_impl a₀_impl F_impl subsumes_impl emptiness_check_impl
           (return ∘ fst) state_copy_impl tracei check_deadlock_neg_impl;
            return r2
    }
    else return True
  }

lemma check_passed_impl_start_hnr:
  (uncurry0 check_passed_impl_start,
    uncurry0 (RETURN (∃ u₀. (∀ c∈{1..n}. u₀ c = 0) ∧ deadlock (l₀, u₀)))
  ) ∈ unit_assn^k →_a bool_assn if F = (λ_. False)
proof −
  define has_deadlock where has_deadlock ≡ ∃ u₀. (∀ c∈{1..n}. u₀ c = 0) ∧ deadlock (l₀, u₀)
  note [sep_heap_rules] =
    check_passed_impl_hnr[
      OF _ that, to_hnr, unfolded hn_refine_def, folded has_deadlock_def, simplified
    ]
    is_start_in_states_impl_hoare'[folded has_deadlock_def, simplified]
  show ?thesis
    unfolding has_deadlock_def[symmetric] check_passed_impl_start_def
    by sepref_to_hoare (sep_auto simp: mod_star_conv)
```

qed

end

context *Reachability_Problem_precompiled*
begin

lemma *F_is_False_iff*:
PR_CONST F = (λ_. *False*) ⟷ *final* = []
unfolding *F_def* **by** (*cases final*; *simp*; *metis*)

lemma *F_impl_False*: *F_impl* = (λ_. *return False*) **if** *final* = []
using *that* **unfolding** *F_impl_def* **unfolding** *final_fun_def List.member_def* **by** *auto*

definition *deadlock_checker* **where**
deadlock_checker ≡
let
key = *return* ∘ *fst*;
sub = *subsumes_impl*;
copy = *state_copy_impl*;
start = a_0*_impl*;
final = (λ_. *return False*);
succs = *succs_impl*;
empty = *emptiness_check_impl*;
P = *check_deadlock_neg_impl*;
trace = *tracei*
in do {
r1 ← *is_start_in_states_impl*;
if r1 then do {
r2 ← *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* (∃ u_0. (∀ *c*∈{*1..m*}. u_0 *c* = *0*) ∧ *deadlock* (*0*, u_0))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
unfolding *deadlock_checker_def Let_def*
using *check_passed_impl_start_hnr*[
unfolded F_is_False_iff F_impl_False[*OF assms*] *check_passed_impl_start_def*,
OF assms] **.**

schematic_goal *deadlock_checker_alt_def*:
deadlock_checker ≡ *?impl*
unfolding *deadlock_checker_def*
unfolding *succs_impl_def*
unfolding *E_op''_impl_def abstr_repair_impl_def abstra_repair_impl_def*
unfolding *start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *check_deadlock_neg_impl_def check_deadlock_impl_def*

```
  unfolding is_start_in_states_impl_def
   apply (abstract_let IArray.sub (IArray (map (IArray o map int) k)) k)
   apply (abstract_let inv_fun inv_fun)
   apply (abstract_let trans_impl trans)
   unfolding inv_fun_def[abs_def] trans_impl_def[abs_def]
   apply (abstract_let IArray inv inv_ia)
   apply (abstract_let IArray trans_map trans_map)
   unfolding trans_map_def label_def
   unfolding init_dbm_impl_def a₀_impl_def
   unfolding subsumes_impl_def
   unfolding emptiness_check_impl_def
   unfolding state_copy_impl_def
  by (rule Pure.reflexive)

end

concrete_definition deadlock_checker_impl
  uses Reachability_Problem_precompiled.deadlock_checker_alt_def

export_code deadlock_checker_impl in SML_imp module_name TA

definition [code]:
  check_deadlock n m k I T ≡
    if Reachability_Problem_precompiled n m k I T
    then deadlock_checker_impl m k I T ⤜ (λx. return (Some x))
    else return None

theorem deadlock_check:
  (uncurry0 (check_deadlock n m k I T),
    uncurry0 (
      RETURN (
       if (Reachability_Problem_precompiled n m k I T)
       then Some (
           if
            ∃ u₀. (∀ c ∈ {1..m}. u₀ c = 0) ∧
            Graph_Defs.deadlock
              (λ(l, u) (l', u').
                 (conv_A (Reachability_Problem_precompiled_defs.A n I T)) ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
              (0, u₀)
           then True
           else False
         )
       else None
      )
    )
  ) ∈ unit_assn^k →ₐ id_assn
proof −
  define reach_check where
    reach_check =
    (∃ u₀. (∀ c ∈ {1..m}. u₀ c = 0) ∧
             Graph_Defs.deadlock
           (λ(l, u) (l', u').
              (conv_A (Reachability_Problem_precompiled_defs.A n I T)) ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
           (0, u₀))
```

note [*sep_heap_rules*] = *Reachability_Problem_precompiled.deadlock_checker_hnr*[*OF _ HOL.refl*,
to_hnr, *unfolded hn_refine_def*, *rule_format*,
of n m k I T,
unfolded reach_check_def[*symmetric*]
]
show *?thesis*
unfolding *reach_check_def*[*symmetric*]
by *sepref_to_hoare*
(*sep_auto simp*: *deadlock_checker_impl.refine*[*symmetric*] *mod_star_conv check_deadlock_def*)
qed

end
theory *Deadlock_Checking*
imports *Deadlock_Impl Munta_Model_Checker.UPPAAL_Model_Checking*
begin

Standard Deadlock Checker Implementation **context** *Reachability_Problem_Impl*
begin

definition *deadlock_checker* **where**
deadlock_checker ≡
let
key = *return* ∘ *fst*;
sub = *subsumes_impl*;
copy = *state_copy_impl*;
start = a_0*_impl*;
final = (λ_. *return False*);
succs = *succs_impl*;
empty = *emptiness_check_impl*;
P = *check_deadlock_neg_impl*;
trace = *tracei*
in do {
r1 ← *is_start_in_states_impl*;
if r1 then do {
r2 ← *check_passed_impl succs start final sub empty key copy trace P*;
return r2
}
else return True
}

interpretation *ta_bisim*: *Bisimulation_Invariant*
(λ(*l*, *u*) (*l'*, *u'*). *conv_A A* ⊢' ⟨*l*, *u*⟩ → ⟨*l'*, *u'*⟩)
(λ(*l*, *u*) (*l'*, *u'*). *conv_A A* ⊢' ⟨*l*, *u*⟩ → ⟨*l'*, *u'*⟩)
(λ(*l*, *u*) (*l'*, *u'*). *l'* = *l* ∧ (∀ *c*. *c* ∈ *clk_set* (*conv_A A*) ⟶ *u c* = *u'* *c*))
(λ_. *True*) (λ_. *True*)
by (*rule ta_bisimulation*[*of conv_A A*])

lemma *deadlock_zero_clock_val_iff*:
(∃ u_0. (∀ *c*∈{*1..n*}. u_0 *c* = *0*) ∧ *deadlock* (l_0, u_0)) ⟷ *deadlock* (l_0, λ_. *0*)
apply *safe*
subgoal for *u*
using *clocks_I*[*of* λ_. *0 u*] **by** (*subst ta_bisim.deadlock_iff*; *simp*)

by *auto*

context
assumes *F_fun_False*: $\bigwedge x$. *F_fun x* = *False* **and** *F_False*: *F* = (λ_. *False*)
begin
lemma *F_impl_is_False*:
F_impl = (λ_. *return False*)
unfolding *F_impl_def F_fun_False* **by** *simp*

lemma *deadlock_checker_correct*:
(*uncurry0 deadlock_checker*, *uncurry0* (*Refine_Basic.RETURN* (*deadlock* (l_0, λ_. *0*))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
unfolding
deadlock_checker_def Let_def F_impl_is_False[*symmetric*]
check_passed_impl_start_def[*symmetric*] *deadlock_zero_clock_val_iff*[*symmetric*]
using *check_passed_impl_start_hnr*[*OF F_False*] **.**

lemmas *deadlock_checker_hoare* = *deadlock_checker_correct*[*to_hnr*, *unfolded hn_refine_def*, *simplified*]

end

end

context *UPPAAL_Reachability_Problem_precompiled′*
begin

- *equiv.defs.states′*
- *EA* is *EA*, the state automaton constructed from UPPAAL network (*equiv*/*N*)
- *A* is *A*, the product automaton constructed from UPPAAL network (*equiv*/*N*)

context
fixes φ
assumes *formula_is_false*: *formula* = *formula.EX* (*and* φ (*not* φ))
begin

lemma *F_is_FalseI*:
shows *PR_CONST* (λ(*x*, *y*). *F x y*) = (λ_. *False*)
unfolding *F_def* **by** (*subst formula_is_false*) *auto*

lemma *final_fun_is_False*:
final_fun a = *False*
unfolding *final_fun_def* **by** (*subst formula_is_false*) *auto*

lemmas *deadlock_checker_hoare* = *impl.deadlock_checker_hoare*[
OF final_fun_is_False F_is_FalseI, *folded deadlock_start_iff has_deadlock_def*]

end

schematic_goal *deadlock_checker_alt_def*:
impl.deadlock_checker ≡ *?impl*

unfolding *impl.deadlock_checker_def impl.succs_impl_def*
unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *k_impl_alt_def*
unfolding *impl.check_deadlock_neg_impl_def impl.check_deadlock_impl_def*
unfolding *impl.is_start_in_states_impl_def*
apply (*abstract_let k_i ceiling*)
apply (*abstract_let inv_fun* :: (_ × *int list* ⇒ _) *inv*)
apply (*abstract_let trans_fun trans*)
unfolding *impl.init_dbm_impl_def impl.*a_0*_impl_def*
unfolding *impl.F_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

schematic_goal *deadlock_checker_alt_def_refined*:
impl.deadlock_checker ≡ *?impl*
unfolding *deadlock_checker_alt_def*
unfolding *fw_impl'_int*
unfolding *inv_fun_def trans_fun_def trans_s_fun_def trans_i_fun_def*
unfolding *trans_i_from_impl*
unfolding *runf_impl runt_impl check_g_impl pairs_by_action_impl check_pred_impl*
apply (*abstract_let IArray* (*map IArray inv*) *inv_array*)
apply (*abstract_let IArray* (*map IArray trans_out_map*) *trans_out*)
apply (*abstract_let IArray* (*map IArray trans_in_map*) *trans_in*)
apply (*abstract_let IArray* (*map IArray trans_i_map*) *trans_internal*)
apply (*abstract_let IArray bounds bounds_array*)
apply (*abstract_let PF PF*)
apply (*abstract_let PT PT*)
unfolding *PF_alt_def PT_alt_def*
apply (*abstract_let PROG' PROG'*)
unfolding *PROG'_def*
apply (*abstract_let length prog len_prog*)
apply (*abstract_let IArray* (*map* (*map_option stripf*) *prog*) *prog_f*)
apply (*abstract_let IArray* (*map* (*map_option stript*) *prog*) *prog_t*)
apply (*abstract_let IArray prog prog_array*)
unfolding *all_actions_by_state_impl*
apply (*abstract_let* [*0..<p*] *p_ran*)
apply (*abstract_let* [*0..<na*] *num_actions_ran*)
apply (*abstract_let* {*0..<p*} *num_processes_ran*)
apply (*abstract_let* [*0..<m+1*] *num_clocks_ran*)
by (*rule Pure.reflexive*)

end

concrete_definition *deadlock_checker* **uses**
UPPAAL_Reachability_Problem_precompiled'.deadlock_checker_alt_def_refined

definition
precond_dc
num_processes num_clocks clock_ceiling max_steps I T prog bounds program s_0 *num_actions*
≡
if UPPAAL_Reachability_Problem_precompiled'

```
    num_processes num_clocks max_steps I T prog bounds program s₀ num_actions clock_ceiling
   then
     deadlock_checker
     num_processes num_clocks max_steps I T prog bounds program s₀ num_actions clock_ceiling
     ⤜ (λx. return (Some x))
   else return None

theorem deadlock_check:
  <emp>
     precond_dc p m k max_steps I T prog bounds P s₀ na
   <λ Some r ⇒ ↑(
      UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s₀ na k ∧
      r = has_deadlock (conv (N p I P T prog bounds)) max_steps (repeat 0 p, s₀, λ_ . 0))
   | None ⇒ ↑(¬ UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds
P s₀ na k)
   >ₜ
proof −
  define A where A ≡ conv (N p I P T prog bounds)
 note [sep_heap_rules] = UPPAAL_Reachability_Problem_precompiled'.deadlock_checker_hoare[
     OF _ HOL.refl,
     of p m max_steps I T prog bounds P s₀ na k,
     unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
     folded A_def
     ]
  show ?thesis
    unfolding A_def[symmetric] precond_dc_def
    by (sep_auto simp:
      deadlock_checker.refine[symmetric] UPPAAL_Reachability_Problem_precompiled_defs.init_def

        mod_star_conv has_deadlock_def)
qed

export_code 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 (_ [] _ [40, 40] 41)
```

Helpful methods and theorems to work with tags:

```
lemmas TAG_cong = arg_cong[where f = TAG t for t]

lemma TAG_mp:
  assumes TAG t x x ⟹ y
  shows TAG t y
  using assms unfolding TAG_def by blast
```

lemma *TAG_mp'*:
assumes *TAG t x x* $\Longrightarrow$ *y*
shows *y*
using *assms* **unfolding** *TAG_def* **by** *blast*

lemmas *all_mono_rule* = *all_mono*[*THEN mp*, *OF impI*, *rotated*]

lemma *imp_mono_rule*:
assumes *P1* $\longrightarrow$ *P2*
and *Q1* $\Longrightarrow$ *P1*
and *Q1* $\Longrightarrow$ *P2* $\Longrightarrow$ *Q2*
shows *Q1* $\longrightarrow$ *Q2*
using *assms* **by** *blast*

lemma *Ball_mono*:
assumes $\forall x \in S.\ P\ x\ \bigwedge x.\ x \in S \Longrightarrow P\ x \Longrightarrow Q\ x$
shows $\forall x \in S.\ Q\ x$
using *assms* **by** *blast*

method *prop_monos* =
erule all_mono_rule
| *erule* (*1*) *imp_mono_rule*
| *erule disj_mono*[*rule_format*, *rotated 2*]

locale *Prod_TA_Defs_urge* =
fixes *A* :: ('*a*, '*s*, '*c*, '*t* :: {*zero*}, '*x*, '*v* :: *linorder*) *nta* **and** *urge* :: '*c*
begin

definition
add_reset $\equiv \lambda(C,\ U,\ T,\ I).$
$(C,\ \{\},\ (\lambda(l,\ b,\ g,\ a,\ f,\ r,\ l').\ (l,\ b,\ g,\ a,\ f,\ urge\ \#\ r,\ l'))\ `\ T,\ I)$

definition
add_inv $\equiv \lambda(C,\ U,\ T,\ I).\ (C,\ U,\ T,$
$(\lambda l.\ if\ l \in U\ then\ acconstraint.LE\ urge\ 0\ \#\ I\ l\ else\ I\ l))$

definition *A_urge* $\equiv$
$(fst\ A,\ map\ add_reset\ (map\ add_inv\ (fst\ (snd\ A))),\ snd\ (snd\ A))$

definition
N_urge i $\equiv$ *map add_reset* (*map add_inv* (*fst* (*snd A*))) ! *i*

end

locale *Prod_TA_sem_urge* =
Prod_TA_Defs_urge A urge + *Prod_TA_sem A*
for *A* :: ('*a*, '*s*, '*c*, '*t* :: {*zero*, *time*}, '*x*, '*v* :: *linorder*) *nta* **and** *urge* :: '*c* +
assumes *urge_not_in_invariants*:
$\forall (_,\ _,\ _,\ I) \in set\ (fst\ (snd\ A)).\ \forall l.\ urge \notin constraint_clk\ `\ set\ (I\ l)$
assumes *urge_not_in_guards*:
$\forall (_,\ _,\ T,\ _) \in set\ (fst\ (snd\ A)).\ \forall (l,\ b,\ g,\ a,\ f,\ r,\ l') \in T.\ urge \notin constraint_clk\ `\ set\ g$
assumes *urge_not_in_resets*:

$\forall (_,\ _,\ T,\ _) \in set\ (fst\ (snd\ A)).\ \forall (l,\ b,\ g,\ a,\ f,\ r,\ l') \in T.\ urge \notin set\ r$
begin

lemma *N_urge_simp*:
N_urge i = add_reset (add_inv (N i)) **if** *i < n_ps*
using *that* **unfolding** *N_urge_def N_def* **by** (*simp add*: *n_ps_def*)

lemma *inv_add_reset*:
inv (add_reset A') = inv A'
unfolding *inv_def add_reset_def* **by** (*simp split*: *prod.splits*)

lemma *inv_add_inv*:
inv (add_inv (C, U, T, I)) = (λl. if l ∈ U then acconstraint.LE urge 0 # I l else I l)
unfolding *add_inv_def inv_def* **by** *simp*

definition
is_urgent L ≡ ∃p<n_ps. L ! p ∈ urgent (N p)

lemma *map_nth'*:
map ((!) xs) [0..<n] = xs **if** *n = length xs*
using *that List.map_nth* **by** *simp*

lemma *A_urge_split*:
A_urge = (broadcast, map N_urge [0..<n_ps], bounds)
unfolding *broadcast_def N_urge_def bounds_def n_ps_def A_urge_def* **by** (*cases A*)(*simp add*: *map_nth'*)

lemma *inv_add_inv'*:
inv (add_inv A') =
(λl. if l ∈ urgent A' then acconstraint.LE urge 0 # inv A' l else inv A' l)
apply (*cases A'*)
apply (*simp add*: *inv_add_inv urgent_def*)
unfolding *inv_def*
apply *auto*
done

lemma *A_split'*:
A = (fst A, fst (snd A), snd (snd A))
by *auto*

lemma *is_urgent_simp*:
(∃p<n_ps. L ! p ∈ urgent (map N [0..<n_ps] ! p)) ⟷ is_urgent L
unfolding *is_urgent_def* **by** *auto*

lemma *urgent_N_urge*:
urgent (N_urge p) = {} **if** *p < n_ps*
using *that* **unfolding** *N_def N_urge_def n_ps_def urgent_def add_reset_def add_inv_def*
by (*auto split*: *prod.split_asm*)

lemma *committed_N_urge*:
committed (N_urge p) = committed (N p) **if** *p < n_ps*
using *that* **unfolding** *N_def N_urge_def n_ps_def committed_def add_reset_def add_inv_def*
by (*auto split*: *prod.split*)

lemma *is_urgent_simp2*:
(∃ *p*<*n_ps*. *L* ! *p* ∈ *urgent* (*map N_urge* [*0*..<*n_ps*] ! *p*)) ⟷ *False*
unfolding *is_urgent_def* **by** (*auto simp*: *urgent_N_urge*)

lemma *inv_add_invI1*:
u ⊢ *inv* (*add_inv* (*N p*)) (*L* ! *p*) **if** *u* ⊢ *inv* (*N p*) (*L* ! *p*) ¬ *is_urgent L p* < *n_ps*
using *that* **unfolding** *inv_add_inv′ is_urgent_def* **by** *simp*

lemma *inv_add_invI2*:
u ⊢ *inv* (*add_inv* (*N p*)) (*L* ! *p*) **if** *u* ⊢ *inv* (*N p*) (*L* ! *p*) *u urge* ≤ *0 p* < *n_ps*
using *that* **unfolding** *inv_add_inv′ is_urgent_def* **by** (*auto elim*: *guard_consI*[*rotated*])

lemma *inv_add_invD*:
u ⊢ *inv A′ l* **if** *u* ⊢ *inv* (*add_inv A′*) *l*
using *that* **unfolding** *inv_add_inv′* **by** (*auto split*: *if_split_asm simp*: *clock_val_def*)

lemma *inv_N_urge*:
inv (*N_urge p*) = *inv* (*add_inv* (*N p*)) **if** *p* < *n_ps*
using *that* **by** (*simp add*: *N_urge_simp inv_add_reset*)

lemma *N_urge_trans_simp*:
trans (*N_urge i*) = (λ(*l*, *b*, *g*, *a*, *f*, *r*, *l′*). (*l*, *b*, *g*, *a*, *f*, *urge* # *r*, *l′*)) ‘ (*trans* (*N i*))
if *i* < *n_ps*
unfolding *N_urge_simp*[*OF that*] *add_inv_def add_reset_def trans_def* **by** (*simp split*: *prod.splits*)

lemma *trans_N_urgeD*:
(*l*, *b*, *g*, *a*, *f*, *urge* # *r*, *l′*) ∈ *trans* (*N_urge p*)
if (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N p*) *p* < *n_ps*
using *that* **by** (*force simp add*: *N_urge_trans_simp*)

lemma *trans_N_urgeE*:
assumes (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N_urge p*) *p* < *n_ps*
obtains (*l*, *b*, *g*, *a*, *f*, *tl r*, *l′*) ∈ *trans* (*N p*) *r* = *urge* # *tl r*
using *assms* **by** (*force simp add*: *N_urge_trans_simp*)

lemma *urge_not_in_inv*:
urge ∉ *constraint_clk* ‘ *set* (*inv* (*N p*) *l*) **if** *p* < *n_ps*
using *urge_not_in_invariants that* **unfolding** *N_def inv_def n_ps_def* **by** (*fastforce dest*!
:*nth_mem*)

lemma *urge_not_in_guards′*:
assumes (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N p*) *p* < *n_ps*
shows *urge* ∉ *constraint_clk* ‘ *set g*
using *urge_not_in_guards assms* **unfolding** *N_def trans_def n_ps_def* **by** (*fastforce dest*!
:*nth_mem*)

lemma *urge_not_in_resets′*:
assumes (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N p*) *p* < *n_ps*
shows *urge* ∉ *set r*
using *urge_not_in_resets assms* **unfolding** *N_def trans_def n_ps_def* **by** (*fastforce dest*!
:*nth_mem*)

lemma *clk_upd_clock_val_a_simp*:
u(*c* := *d*) ⊢$_a$ *ac* ⟷ *u* ⊢$_a$ *ac* **if** *c* ≠ *constraint_clk ac*

using *that* **by** (*cases ac*) *auto*

lemma *clk_upd_clock_val_simp*:
u(*c* := *d*) ⊢ *cc* ⟷ *u* ⊢ *cc* **if** *c* ∉ *constraint_clk* ‘ *set cc*
using *that* **unfolding** *clock_val_def list_all_def*
by (*force simp*: *image_def clk_upd_clock_val_a_simp*)

lemma *inv_N_urgeI*:
assumes *u* ⊢ *inv* (*N p*) *l p* < *n_ps*
shows *u*(*urge* := *0*) ⊢ *inv* (*N_urge p*) *l*
using *assms urge_not_in_inv*[*of p*]
by (*auto simp*: *inv_N_urge inv_add_inv′ clk_upd_clock_val_simp intro*!: *guard_consI*)

lemma *inv_N_urge_updI*:
assumes *u* ⊢ *inv* (*N p*) *l p* < *n_ps*
shows *u*(*urge* := *d*) ⊢ *inv* (*N p*) *l*
using *assms urge_not_in_inv*[*of p*] **by** (*auto simp*: *clk_upd_clock_val_simp*)

lemma *inv_N_urge_upd_simp*:
assumes *p* < *n_ps*
shows *u*(*urge* := *d*) ⊢ *inv* (*N p*) *l* ⟷ *u* ⊢ *inv* (*N p*) *l*
using *assms urge_not_in_inv*[*of p*] **by** (*auto simp*: *clk_upd_clock_val_simp*)

lemma *inv_N_urge_updD*:
assumes *u*(*urge* := *d*) ⊢ *inv* (*N p*) *l p* < *n_ps*
shows *u* ⊢ *inv* (*N p*) *l*
using *assms urge_not_in_inv*[*of p*] **by** (*auto simp*: *clk_upd_clock_val_simp*)

lemma *inv_N_urgeD*:
assumes *u* ⊢ *inv* (*N_urge p*) *l p* < *n_ps*
shows *u*(*urge* := *d*) ⊢ *inv* (*N p*) *l*
using *assms urge_not_in_inv*[*of p*]
by (*auto simp*: *inv_N_urge inv_add_inv′ clk_upd_clock_val_simp split*: *if_split_asm elim*: *guard_consE*)

lemma *inv_N_urge_urges*:
assumes ∀ *p*<*n_ps*. *u*(*urge* := *0*) ⊕ *d* ⊢ *inv* (*N_urge p*) (*L* ! *p*) *is_urgent L*
shows *d* ≤ *0*
proof −
from ‹*is_urgent L*› **obtain** *p* **where** *p* < *n_ps L* ! *p* ∈ *urgent* (*N p*)
unfolding *is_urgent_def* **by** *auto*
then have *acconstraint.LE urge 0* ∈ *set* (*inv* (*N_urge p*) (*L* ! *p*))
by (*simp add*: *inv_N_urge inv_add_inv′*)
with *assms* ‹*p* < *n_ps* › **have** *u*(*urge* := *0*) ⊕ *d* ⊢$_a$ *acconstraint.LE urge 0*
unfolding *clock_val_def list_all_iff* **by** *auto*
then show *?thesis*
by (*auto simp*: *cval_add_def*)
qed

lemma *trans_urge_upd_iff*:
assumes (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N p*) *p* < *n_ps*
shows *u*(*urge* := *d*) ⊢ *g* ⟷ *u* ⊢ *g*
using *assms urge_not_in_guards′* **by** (*auto simp*: *clk_upd_clock_val_simp*)

lemma *cval_add_0*[*simp*]:
u ⊕ (*0* :: *'tt* :: *time*) = *u*
unfolding *cval_add_def* **by** *simp*

lemma *clock_val_reset_delay*:
u(*c* := *0*) ⊕ (*d* :: *'tt* :: *time*) = (*u* ⊕ *d*)(*c* := *d*)
unfolding *cval_add_def* **by** *auto*

lemma *clock_set_upd_simp*:
[*r* → *d*]*u*(*c* := *d'*) = [*r* → *d*]*u* **if** *c* ∈ *set r*
using *that*
apply (*induction r arbitrary*: *u*)
apply *auto*
oops

lemma *fun_upd_twist2*:
f(*a* := *x*, *b* := *x*) = *f*(*b* := *x*, *a* := *x*)
by *auto*

lemma *clock_set_upd_simp*:
[*c* # *r* → *d*]*u*(*c* := *d'*) = [*c* # *r* → *d*]*u*
apply (*induction r arbitrary*: *u*)
apply *simp*
apply *simp*
apply (*subst fun_upd_twist2*)
apply *auto*
done

lemma *clock_set_commute_single*:
[*r* → *d*]*u*(*c* := *d'*) = ([*r* → *d*]*u*)(*c* := *d'*) **if** *c* ∉ *set r*
using *that* **by** (*induction r*) *auto*

lemma *clock_set_upd_simp2*:
([*xs* @ *c* # *ys* → *d*]*u*)(*c* := *d'*) = ([*xs* @ *ys* → *d*]*u*)(*c* := *d'*)
apply (*induction xs*)
apply (*solves auto*)
apply (*intro ext*)
subgoal for *a xs x*
by (*drule fun_cong*[**where** *x* = *x*]) *auto*
done

lemma *clock_set_upd_simp3*:
([*xs* → *d*]*u*)(*c* := *d'*) = ([*filter* (λ*x*. *x* ≠ *c*) *xs* → *d*]*u*)(*c* := *d'*)
apply (*induction xs*)
apply (*solves auto*)
apply (*rule ext*)
apply (*clarsimp*, *safe*)
apply *clarsimp*
subgoal for *xs x*
by (*drule fun_cong*[**where** *x* = *x*]) *auto*
subgoal for *a xs x*
by (*drule fun_cong*[**where** *x* = *x*]) *auto*
done

interpretation *urge*: *Prod_TA_sem A_urge* **.**

lemma *urge_n_ps_eq*:
urge.n_ps = *n_ps*
unfolding *urge.n_ps_def n_ps_def* **unfolding** *A_urge_def* **by** *simp*

lemma *urge_N_simp*[*simp*]:
urge.N = *N_urge*
unfolding *urge.N_def N_urge_def* **unfolding** *A_urge_def* **by** *simp*

lemma *urge_states_eq*:
urge.states = *states*
unfolding *states_def urge.states_def urge_n_ps_eq*
by (*auto simp add*: *N_urge_trans_simp split*: *prod.splits*)

interpretation *Bisimulation_Invariant*
$\lambda(L,\ s,\ u)\ (L',\ s',\ u').$ *step_u' A L s u L' s' u'*
$\lambda(L,\ s,\ u)\ (L',\ s',\ u').$ *step_u' A_urge L s u L' s' u'*
$\lambda(L,\ s,\ u)\ (L',\ s',\ u').\ L' = L \wedge s' = s \wedge u' = u(urge := 0)$
$\lambda(L,\ s,\ u).\ L \in states\ \lambda(L,\ s,\ u).\ True$
proof −
~~*case (A A s u L' s' u') then show ?case*~~
have *A_urge* $\vdash \langle L,\ s,\ u\ (urge := 0)\rangle \rightarrow \langle L',\ s',\ u'(urge := 0)\rangle$
if *steps*: $A \vdash \langle L,\ s,\ u\rangle \rightarrow \langle L',\ s',\ u'\rangle\ L \in states$
for *L* :: *'s list*
and *s* :: *'x* ⇒ *'v option*
and *u* :: *'c* ⇒ *'t*
and *L'* :: *'s list*
and *s'* :: *'x* ⇒ *'v option*
and *u'* :: *'c* ⇒ *'t*
using *that*
proof *cases*
case *steps*: (*1 L'' s'' u'' a*)
have ∗: $(u \oplus d)(urge := (u \oplus d)\ urge - u\ urge) = u(urge := 0) \oplus d$ **for** *d*
by (*auto simp*: *cval_add_def*)
from *steps*(*1*) ‹*L* ∈ *states*› **have** *L''* ∈ *states*
by (*rule states_inv*)
then have [*simp*]: *length L''* = *n_ps*
by (*rule states_lengthD*)
from *steps*(*1*) **have**
A_urge $\vdash \langle L,\ s,\ u(urge := 0)\rangle \rightarrow_{Del} \langle L'',\ s'',\ u''(urge := u''\ urge - u\ urge)\rangle$
apply *cases*
apply (*subst A_urge_split*)
apply (*subst* (*asm*) *A_split*)
apply (*drule sym*)
apply *simp*
unfolding *TAG_def*[*of ''urgent''*]
apply (*simp add*: *is_urgent_simp* ∗)
apply (*rule step_u.intros*)
apply *tag*
apply *simp*
subgoal
by (*cases is_urgent L*)
(*auto 4 3*

```
      intro: inv_N_urge_updI inv_add_invI1 inv_add_invI2
      simp: inv_N_urge clock_val_reset_delay)
  by (tag, simp add: is_urgent_simp2)+
moreover from steps(3,2) have
  A_urge ⊢ ⟨L'', s'', u''(urge := u'' urge − u urge)⟩ →a ⟨L', s', u'(urge := 0)⟩
  apply (cases; subst A_urge_split; subst (asm) A_split)
    apply (all ‹drule sym›)
  subgoal delay
    by simp
  subgoal internal premises prems[tagged] for l b g a' f r l' N p B broadcast
    using prems apply tag usingT ''range''
    apply simp
    apply (rule step_u.intros)
    preferT ‹TRANS ''silent''›
      apply (solves ‹tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp›)
    preferT ‹''committed''›
      apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
    preferT ‹''bexp''›
      apply (tag, assumption)
    preferT ‹''guard''›
      apply (solves ‹tag ''guard'' TRANS _; auto simp: trans_urge_upd_iff›)
    preferT ‹''new valuation''›
      apply (tag, unfold clock_set.simps(2)[symmetric] clock_set_upd_simp, simp; fail)
    by (tag; auto intro: inv_N_urgeI)+
  subgoal binary premises prems[tagged]
    for a' broadcast' l1 b1 g1 f1 r1 l1' N' p l2 b2 g2 f2 r2 l2' q s'' B
    using prems apply tag
    usingT RECV ''range'' SEND ''range''
    apply simp
    apply (rule step_u.intros)
    preferT ‹''not broadcast''›
      apply (tag; simp; fail)
    preferT ‹TRANS ''in''›
      apply (solves ‹tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp›)
    preferT ‹TRANS ''out''›
      apply (solves ‹tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp›)
    preferT ‹''committed''›
      apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
    preferT ‹''bexp''›
      apply (tag, assumption)
    preferT ‹''bexp''›
      apply (tag, assumption)
    preferT ‹''guard''›
      apply (solves ‹tag ''guard'' TRANS _; auto simp: trans_urge_upd_iff›)
    preferT ‹''guard''›
      apply (solves ‹tag ''guard'' TRANS _; auto simp: trans_urge_upd_iff›)
    preferT ‹''new valuation''›
      apply (solves ‹tag, simp, unfold clock_set.simps(2)[symmetric] clock_set_upd_simp,
        simp add: clock_set_upd_simp2›)
    by (solves ‹tag; auto intro: inv_N_urgeI; simp›)+
  subgoal broadcast premises prems[tagged]
    for a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s'' B
    using prems apply tag
    usingT SEL ''range'' SEND ''range''
```

```
      apply (simp add: subset_nat_0_atLeastLessThan_conv)
      apply (rule step_u.intros)
      preferT ‹''broadcast''›
        apply (tag; simp; fail)
      preferT ‹TRANS ''out''›
        apply (solves ‹tag, simp, drule (1) trans_N_urgeD, subst nth_map, simp, simp›)
      preferT ‹TRANS ''in''›
        apply (solves ‹tag, auto 4 3 dest: trans_N_urgeD›)
      preferT ‹''committed''›
        apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
      preferT ‹''bexp''›
        apply (tag, assumption)
      preferT ‹''bexp''›
        apply (tag, assumption)
      preferT ‹''guard''›
        apply (solves ‹tag ''guard'' TRANS _; auto simp: trans_urge_upd_iff›)
      preferT ‹''guard''›
        apply (solves ‹tag ''guard'' TRANS _; auto simp: trans_urge_upd_iff›)
      preferT ‹''maximal''›
        apply (all ‹tag; auto intro: inv_N_urgeI; fail | succeed›)
      preferT ‹''maximal''›
        apply (solves ‹tag; auto; erule (1) trans_N_urgeE, force simp: trans_urge_upd_iff›)
      preferT ‹''new valuation''› subgoal
        apply tag
        apply clarsimp
        unfolding clock_set.simps(2)[symmetric] clock_set_upd_simp
        apply simp
        apply (subst (2) clock_set_upd_simp3)
        apply (subst clock_set_upd_simp3)
        apply (simp add: filter_concat comp_def)
        done
      done
    done
    ultimately show A_urge ⊢ ⟨L, s, u(urge := 0)⟩ → ⟨L', s', u'(urge := 0)⟩
      using ‹a ≠ _› by (intro step_u'.intros)
  qed
next case (2 L s u L' s' u')
  moreover have ∃ ba. A ⊢ ⟨L, s, u⟩ → ⟨L', s', ba⟩ ∧ u' = ba(urge := 0)
    if A_urge ⊢ ⟨L, s, u (urge := 0)⟩ → ⟨L', s', u'⟩
      and L ∈ states
    for L :: 's list
      and s :: 'x ⇒ 'v option
      and u :: 'c ⇒ 't
      and L' :: 's list
      and s' :: 'x ⇒ 'v option
      and u' :: 'c ⇒ 't
    using that
  proof cases
    case steps: (1 L'' s'' u'' a)
    have ∗: (u(urge := 0) ⊕ d)(urge := (u(urge := 0) ⊕ d) urge + u urge) = u ⊕ d for d
      unfolding cval_add_def by auto
    from steps(1) ‹L ∈ states› have L'' ∈ states
      by (rule urge.states_inv[unfolded urge_states_eq])
    then have [simp]: length L'' = n_ps
```

```
  by (rule states_lengthD)
have 1: ([r→0]u'') urge = 0 if r = urge # xs for r xs
  by (subst that) simp
have 2: filter (λx. x ≠ urge) r = filter (λx. x ≠ urge) (tl r) if r = urge # tl r for r
  by (subst that) simp
from steps(3,2) have u' urge = 0
  apply (cases; subst (asm) A_urge_split)
    apply (all ‹drule sym›)
    apply (simp; fail)
  subgoal delay
    by (tag− ''new valuation'' TRANS _ ''range'')
      (solves ‹auto elim!: trans_N_urgeE simp: 1›)
  subgoal binary for a' broadcast' l1 b1 g1 f1 r1 l1' N p l2 b2 g2 f2 r2 l2' q s'' B
    by (tag− ''new valuation'' TRANS _ RECV ''range'')
      (auto elim!: trans_N_urgeE simp: 1[of _ tl r1 @ r2])
  subgoal broadcast for a' broadcast' l b g f r l' N p ps bs gs fs rs ls' s'a B
    by (tag− ''new valuation'' TRANS _ SEND ''range'')
      (auto elim!: trans_N_urgeE simp: 1[of _ tl r @ concat (map rs ps)])
  done
have **:
  ([r→0]u'')(urge := u'' urge + u urge) = ([tl r→0]u'')(urge := u'' urge + u urge)
  if r = urge # tl r for r
  by (subst that) simp
from steps(1) have A ⊢ ⟨L, s, u⟩ →Del ⟨L'', s'', u''(urge := u'' urge + u urge)⟩
  apply cases
  apply (subst (asm) A_urge_split)
  apply (subst A_split)
  apply (drule sym)
  apply simp
  unfolding TAG_def[of ''urgent'']
  apply (simp add: is_urgent_simp *)
  apply (rule step_u.intros)
  subgoal
  by tag (auto dest!: inv_add_invD inv_N_urge_updD simp: inv_N_urge clock_val_reset_delay)
    apply (tag, assumption)
   apply (tag− ''target invariant'' ''nonnegative'')
   apply (auto simp add: is_urgent_simp2 is_urgent_simp dest: inv_N_urge_urges)
  done
moreover from steps(3,2) have
  A ⊢ ⟨L'', s'', u''(urge := u'' urge + u urge)⟩ →a ⟨L', s', u'(urge := u'' urge + u urge)⟩
  apply (cases; subst (asm) A_urge_split; subst A_split)
    apply (all ‹drule sym›)
  subgoal delay
    by simp
  subgoal internal premises prems[tagged] for l b g a' f r l' N p B broadcast
    using prems apply del_tags
    usingT ''range''
    apply simp
    apply (rule step_u.intros)
    preferT ‹TRANS ''silent''›
      apply (solves ‹tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp›)
    preferT ‹''committed''›
      apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
    preferT ‹''bexp''›
```

```
    apply (tag; assumption)
  preferT ‹''guard''›
    apply (solves ‹tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff›)
  preferT ‹''target invariant''›
    apply (all ‹tag; auto intro: inv_N_urgeD; fail | succeed›)
  preferT ‹''new valuation''›
    apply (solves ‹tag TRANS _, simp, erule (1) trans_N_urgeE,
      subst clock_set_commute_single, rule urge_not_in_resets', (simp add: **)+›)
  done
subgoal binary premises prems[tagged]
  for a' broadcast' l1 b1 g1 f1 r1 l1' Na p l2 b2 g2 f2 r2 l2' q s'' B
  using prems apply del_tags
  usingT RECV ''range'' SEND ''range''
  apply simp
  apply (rule step_u.intros)
  preferT ‹''not broadcast''› apply (tag; simp; fail)
  preferT ‹TRANS ''in''›
    apply (solves ‹tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp›)
  preferT ‹TRANS ''out''›
    apply (solves ‹tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp›)
  preferT ‹''committed''›
    apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
  preferT ‹''bexp''›
    apply (tag; assumption)
  preferT ‹''bexp''›
    apply (tag; assumption)
  preferT ‹''guard''›
    apply (solves ‹tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff›)
  preferT ‹''guard''›
    apply (solves ‹tag TRANS _; auto intro: trans_N_urgeE simp: trans_urge_upd_iff›)
  preferT ‹''new valuation''› subgoal premises prems[tagged]
    using prems apply tag
    apply (tag TRANS _, simp, erule (1) trans_N_urgeE, erule (1) trans_N_urgeE)
    apply (subst clock_set_upd_simp3)
    apply (subst clock_set_commute_single)
     apply (simp; rule conjI; erule (1) urge_not_in_resets'; fail)
    apply (rule arg_cong)
    subgoal premises prems
      using urge_not_in_resets'[OF prems(7) ‹p < n_ps›]
            urge_not_in_resets'[OF prems(9) ‹q < n_ps›]
      by (subst ‹r1 = _›, subst ‹r2 = _›, simp) (subst filter_True, auto)+
    done
  by (solves ‹tag; auto intro: inv_N_urgeD›)+
subgoal broadcast premises prems[tagged]
  for a' broadcast' l b g f r l' N' p ps bs gs fs rs ls' s2 B
  using prems apply del_tags
  usingT SEL ''range'' SEND ''range''
  apply (simp add: subset_nat_0_atLeastLessThan_conv)
  apply (rule step_u.intros)
  preferT ‹TRANS ''out''›
    — instantiates schematics
    apply (solves ‹tag, simp, erule (1) trans_N_urgeE, subst nth_map, simp, simp›)
  preferT ‹TRANS ''in''›
    apply (tag, auto 4 3 elim: trans_N_urgeE; fail) — instantiates schematics
```

```
      preferT ‹''broadcast''›
        apply (tag; simp; fail)
      preferT ‹''committed''›
        apply (solves ‹tag, subst nth_map, simp, simp add: committed_N_urge›)
      preferT ‹''guard''›
        apply (solves ‹tag TRANS _; auto elim: trans_N_urgeE simp: trans_urge_upd_iff›)
      preferT ‹''guard''› subgoal guards
        by (tag TRANS _; auto elim!: trans_N_urgeE simp: trans_urge_upd_iff)
      preferT ‹''maximal''› subgoal maximal
        by (tag, force simp: trans_urge_upd_iff dest: trans_N_urgeD)
      preferT ‹''new valuation''› subgoal new_valuation premises prems[tagged]
        apply (tag TRANS _)
        using prems apply tag
        apply (subst [[get_tagged ‹''new valuation''›]])
        apply (subst clock_set_upd_simp3)
        apply (simp add: filter_concat comp_def)
        apply (subst clock_set_commute_single[symmetric], (simp; fail))
        apply (rule arg_cong)
        apply (fo_rule arg_cong2)
        subgoal
          by (auto simp: 2 urge_not_in_resets' filter_id_conv elim!: trans_N_urgeE)
        subgoal
          apply (fo_rule arg_cong)
          apply (fastforce elim: trans_N_urgeE simp: 2 urge_not_in_resets' filter_id_conv)+
          done
        done
      by (solves ‹tag; auto intro: inv_N_urgeD›)+
    done
  ultimately show ?thesis
    using ‹a ≠ _› ‹u' urge = 0› by (intros add: step_u'.intros) auto
qed
moreover have ab ∈ states
  if a ∈ states
    and A ⊢ ⟨a, aa, b⟩ → ⟨ab, ac, ba⟩
  for a :: 's list
    and aa :: 'x ⇒ 'v option
    and b :: 'c ⇒ 't
    and ab :: 's list
    and ac :: 'x ⇒ 'v option
    and ba :: 'c ⇒ 't
  using that by (elim step_u'_elims states_inv)
ultimately show
  Bisimulation_Invariant
   (λ(L, s, u) (L', s', u').
      A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λ(L, s, u) (L', s', u').
      A_urge ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λ(L, s, u) (L', s', u').
      L' = L ∧ s' = s ∧ u' = u(urge := 0))
   (λ(L, s, u). L ∈ states) (λ(L, s, u). True)
  by − (standard; clarsimp)
qed

lemmas urge_bisim = Bisimulation_Invariant_axioms
```

end

context *Simple_Network_Impl_Defs*
begin

lemma *dom_bounds*: *dom bounds = fst ‘ set bounds′*
unfolding *bounds_def* **by** (*simp add*: *dom_map_of_conv_image_fst*)

lemma *mem_trans_N_iff*:
t ∈ Simple_Network_Language.trans (N i) ⟷ t ∈ set (fst (snd (snd (automata ! i))))
if *i < n_ps*
unfolding *N_eq*[*OF that*] **by** (*auto split*: *prod.splits simp*: *automaton_of_def trans_def*)

lemma *length_automata_eq_n_ps*:
length automata = n_ps
unfolding *n_ps_def* **by** *simp*

lemma *N_p_trans_eq*:
Simple_Network_Language.trans (N p) = set (fst (snd (snd (automata ! p)))) **if** *p < n_ps*
using *mem_trans_N_iff*[*OF that*] **by** *auto*

lemma *loc_set_compute*:
loc_set = ⋃ ((λ(_, _, *t*, _). ⋃ ((λ(*l*, _, _, _, _, _, *l′*). {*l*, *l′*}) ‘ *set t*)) ‘ *set automata*)
unfolding *loc_set_def setcompr_eq_image*
apply (*safe*; *clarsimp simp*: *N_p_trans_eq n_ps_def*)
apply (*drule nth_mem*, *erule bexI*[*rotated*], *force*)+
apply (*drule mem_nth*, *force*)+
done

lemma *var_set_compute*:
var_set =
(⋃*S* ∈ (λ*t*. (*fst* ∘ *snd*) ‘ *set t*) ‘ ((λ(_, _, *t*, _). *t*) ‘ *set automata*). ⋃*b* ∈ *S*. *vars_of_bexp b*)
∪
(⋃*S* ∈ (λ*t*. (*fst* ∘ *snd o snd* ∘ *snd* ∘ *snd*) ‘ *set t*) ‘ ((λ(_, _, *t*, _). *t*) ‘ *set automata*).
⋃*f* ∈ *S*. ⋃ (*x*, *e*) ∈ *set f*. {*x*} ∪ *vars_of_exp e*)
unfolding *var_set_def setcompr_eq_image*
by (*rule arg_cong2*[**where** *f* = (∪)]; *auto simp*: *N_p_trans_eq n_ps_def*,
(*drule nth_mem*, *erule bexI*[*rotated*],
metis (*no_types*, *lifting*) *insert_iff mem_case_prodI prod.collapse*)+,
(*drule mem_nth*, *force*)+)

lemma *states_loc_setD*:
set L ⊆ loc_set **if** *L ∈ states*
using *states_loc_set that* **by** *auto*

end

context *Simple_Network_Impl*
begin

lemma *sem_bounds_eq*: *sem.bounds = bounds*

unfolding *sem.bounds_def bounds_def* **unfolding** *sem_def* **by** *simp*

lemma *n_ps_eq*[*simp*]:
sem.n_ps = *n_ps*
unfolding *n_ps_def sem.n_ps_def* **unfolding** *sem_def* **by** *auto*

lemma *sem_loc_set_eq*:
sem.loc_set = *loc_set*
unfolding *sem.loc_set_def loc_set_def n_ps_eq setcompr_eq_image*
apply (*simp add*: *sem_N_eq N_eq*)
apply (*rule arg_cong2*[**where** *f* = (∪)])
apply (*safe*; *clarsimp*;
force split: *prod.splits simp*: *conv_automaton_def trans_def automaton_of_def n_ps_def*)+
done

lemma *sem_states_eq*:
sem.states = *states*
unfolding *sem.states_def states_def n_ps_eq setcompr_eq_image*
by (*clarsimp simp*: *sem_N_eq N_eq*;
force simp: *conv_automaton_def trans_def automaton_of_def n_ps_def split*: *prod.splits*)+

end

locale *Simple_Network_Rename_Defs* =
Simple_Network_Impl_Defs automata broadcast bounds' **for** *automata* ::
('*s list* × '*s list* × (('*a* :: *countable*) *act*, '*s*, '*c*, '*t*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, '*t*) *cconstraint*) *list*) *list*
and *broadcast bounds'* +
fixes *renum_acts* :: '*a* ⇒ *nat*
and *renum_vars* :: '*x* ⇒ *nat*
and *renum_clocks* :: '*c* ⇒ *nat*
and *renum_states* :: *nat* ⇒ '*s* ⇒ *nat*
begin

definition
map_cconstraint f g xs = *map* (*map_acconstraint f g*) *xs*

definition
renum_cconstraint = *map_cconstraint renum_clocks id*

definition
renum_act = *map_act renum_acts*

definition
renum_bexp = *map_bexp renum_vars*

definition
renum_exp = *map_exp renum_vars*

definition
renum_upd = (λ(*x*, *upd*). (*renum_vars x*, *renum_exp upd*))

abbreviation

renum_upds ≡ *map renum_upd*

definition
renum_reset = *map renum_clocks*

definition *renum_automaton* **where**
renum_automaton i ≡ λ(*committed*, *urgent*, *trans*, *inv*). *let*
committed′ = *map* (*renum_states i*) *committed*;
urgent′ = *map* (*renum_states i*) *urgent*;
trans′ = *map* (λ(*l*, *b*, *g*, *a*, *upd*, *r*, *l′*).
(*renum_states i l*, *renum_bexp b*, *renum_cconstraint g*, *renum_act a*, *renum_upds upd*,
renum_reset r, *renum_states i l′*)
) *trans*;
inv′ = *map* (λ(*l*, *g*). (*renum_states i l*, *renum_cconstraint g*)) *inv*
in (*committed′*, *urgent′*, *trans′*, *inv′*)

definition
vars_inv ≡ *the_inv renum_vars*

definition
map_st ≡ λ(*L*, *s*). (*map_index renum_states L*, *s o vars_inv*)

definition
clocks_inv ≡ *the_inv renum_clocks*

definition
map_u u = *u o clocks_inv*

lemma *map_u_add*[*simp*]:
map_u (*u* ⊕ *d*) = *map_u u* ⊕ *d*
by (*auto simp*: *map_u_def cval_add_def*)

definition *renum_label* **where**
renum_label = *map_label renum_acts*

sublocale *renum*: *Simple_Network_Impl_Defs*
map_index renum_automaton automata
map renum_acts broadcast
map (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds′*
.

lemma *renum_n_ps_simp*[*simp*]:
renum.n_ps = *n_ps*
unfolding *n_ps_def renum.n_ps_def* **by** *simp*

end

locale *Simple_Network_Rename_Defs_int* =
Simple_Network_Rename_Defs automata +
Simple_Network_Impl automata
for *automata* ::
('*s list* × '*s list* × (('*a* :: *countable*) *act*, '*s*, '*c*, *int*, '*x* :: *countable*, *int*) *transition list*

```
    × (('s :: countable) × ('c :: countable, int) cconstraint) list) list
begin

sublocale renum: Simple_Network_Impl
  map_index renum_automaton automata
  map renum_acts broadcast
  map (λ(a,p). (renum_vars a, p)) bounds'
  .

end

lemma
  fixes f :: 'b :: countable ⇒ nat
  assumes inj_on f S finite S infinite (UNIV :: 'b set)
  shows extend_bij_inj: inj (extend_bij f S) and extend_bij_surj: surj (extend_bij f S)
    and extend_bij_extends: ∀ x ∈ S. extend_bij f S x = f x
proof −
  have infinite (ℕ :: nat set)
    unfolding Nats_def by simp
  from assms this have bij (extend_bij f S)
    by (intro extend_bij_bij) auto
  then show inj (extend_bij f S) and surj (extend_bij f S)
    unfolding bij_def by fast+
  from assms ‹infinite ℕ› show ∀ x ∈ S. extend_bij f S x = f x
    by (intro extend_bij_extends) auto
qed

lemma default_map_of_map:
  default_map_of y (map (λ(a, b). (f a, g b)) xs) (f a) = g (default_map_of x xs a)
  if inj f y = g x
  using that unfolding default_map_of_alt_def
 by (auto 4 4 dest: injD[OF that(1)] map_of_SomeD simp: map_of_eq_None_iff map_of_mapk_SomeI)

lemma default_map_of_map_2:
  default_map_of y (map (λ(a, b). (a, g b)) xs) a = g (default_map_of x xs a) if y = g x
  unfolding default_map_of_alt_def using that by (auto simp: map_of_map)

lemma map_of_map':
  map_of (map (λ(k, v). (k, f k v)) xs)
  = (λk. case map_of xs k of Some v ⇒ Some (f k v) | _ ⇒ None)
  by (induct xs) (auto simp: fun_eq_iff)

lemma default_map_of_map_3:
  default_map_of y (map (λ(a, b). (a, g a b)) xs) a = g a (default_map_of x xs a)
  if ⋀k. y = g k x
  unfolding default_map_of_alt_def using that by (auto simp: map_of_map')

lemma dom_map_of_map:
  dom (map_of (map (λ (a, b). (f a, g b)) xs)) = f ‘ fst ‘ set xs
  unfolding dom_map_of_conv_image_fst by (auto 4 3)

lemma inj_image_eqI:
  S = T if inj f f ‘ S = f ‘ T
  using that by (meson inj_image_eq_iff)
```

lemmas [*finite_intros*] = *finite_set*

lemma *map_of_NoneI*:
map_of xs x = *None* **if** *x* ∉ *fst* ‘ *set xs*
by (*simp add*: *map_of_eq_None_iff that*)

lemma *bij_f_the_inv_f*:
f (*the_inv f x*) = *x* **if** *bij f*
using *that f_the_inv_f* **unfolding** *bij_def* **by** (*simp add*: *f_the_inv_f*)

fun *map_sexp* ::
(*nat* ⇒ *'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'* .

abbreviation *sem* ≡ (*set broadcast*, *map automaton_of automata*, *map_of bounds'*)

sublocale *Prod_TA_sem* (*set broadcast*, *map automaton_of automata*, *map_of bounds'*) .

end

context *Simple_Network_Impl*
begin

lemma *sem_state_guard_eq*:
(*fst* ∘ *snd*) ‘ *trans* (*sem.N p*) = (*fst* ∘ *snd*) ‘ *trans* (*N p*) **if** *p* < *n_ps*
unfolding *sem_N_eq*[*OF* ‹*p* < *n_ps*›] *N_eq*[*OF* ‹*p* < *n_ps*›]
unfolding *automaton_of_def conv_automaton_def trans_def*
by (*force split*: *prod.splits*)

lemma *sem_state_update_eq*:
(*fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *trans* (*sem.N p*) = (*fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *trans* (*N p*)
if *p* < *n_ps*
unfolding *sem_N_eq*[*OF* ‹*p* < *n_ps*›] *N_eq*[*OF* ‹*p* < *n_ps*›]
unfolding *automaton_of_def conv_automaton_def trans_def*
by (*force split*: *prod.splits*)

lemma *sem_var_set_eq*:
sem.var_set = *var_set*
unfolding *sem.var_set_def var_set_def n_ps_eq* **using** *sem_state_guard_eq sem_state_update_eq*
by (*simp cong*: *image_cong_simp add*: *setcompr_eq_image*)

end

locale *Simple_Network_Rename_Defs_real* =
Simple_Network_Rename_Defs automata +
Simple_Network_Impl_real automata
for *automata* ::
(*'s list* × *'s list*
× ((*'a* :: *countable*) *act*, *'s*, *'c*, *real*, *'x* :: *countable*, *int*) *transition list*
× ((*'s* :: *countable*) × (*'c* :: *countable*, *real*) *cconstraint*) *list*) *list*
begin

sublocale *renum*: *Simple_Network_Impl_real*
map_index renum_automaton automata
map renum_acts broadcast
map (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds'*
.

end

locale *Simple_Network_Rename'* =
Simple_Network_Rename_Defs **where** *automata* = *automata* **for** *automata* ::
(*'s list* × *'s list*
× ((*'a* :: *countable*) *act*, *'s*, *'c*, *'t*, *'x* :: *countable*, *int*) *transition list*
× ((*'s* :: *countable*) × (*'c* :: *countable*, *'t*) *cconstraint*) *list*) *list* +
assumes *bij_renum_clocks*: *bij renum_clocks*
and *renum_states_inj*: ∀ *p*<*n_ps*. *inj* (*renum_states p*)
and *bij_renum_vars*: *bij renum_vars*
and *bounds'_var_set*: *fst* ‘ *set bounds'* = *var_set*
and *inj_renum_acts*: *inj renum_acts*

locale *Simple_Network_Rename_real* =
Simple_Network_Rename_Defs_real **where** *automata* = *automata* +

```
  Simple_Network_Rename' where automata = automata
  for automata ::
    ('s list × 's list
      × (('a :: countable) act, 's, 'c, real, 'x :: countable, int) transition list
      × (('s :: countable) × ('c :: countable, real) cconstraint) list) list +
  assumes urgency_removed: ∀ 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)))

  by auto

lemma clocks_inv_inv:
  clocks_inv (renum_clocks a) = a
  unfolding clocks_inv_def by (subst the_inv_f_f; rule bij_renum_clocks[THEN bij_is_inj]
HOL.refl)

lemma map_u_renum_cconstraint_clock_valI:
  map_u u ⊢ renum_cconstraint cc if u ⊢ cc
  using that
  unfolding clock_val_def list_all_length
  unfolding renum_cconstraint_def map_cconstraint_def
  unfolding map_u_def
  apply clarsimp
  subgoal for n
    by (cases cc ! n) (auto 4 4 simp: clocks_inv_inv split: acconstraint.split)
  done

lemma map_u_renum_cconstraint_clock_valD:
  u ⊢ cc if map_u u ⊢ renum_cconstraint cc
  using that
  unfolding clock_val_def list_all_length
  unfolding renum_cconstraint_def map_cconstraint_def
  unfolding map_u_def
  apply clarsimp
  subgoal for n
    by (cases cc ! n) (auto 4 4 simp: clocks_inv_inv split: acconstraint.split)
  done

lemma inj_renum_states: inj (renum_states p) if p < n_ps
  using renum_states_inj ‹p < n_ps› by blast

lemma inv_renum_sem_I:
  assumes
    u ⊢ Simple_Network_Language.inv (N p) l p < n_ps l ∈ loc_set
  shows
    map_u u ⊢ Simple_Network_Language.inv (renum.N p) (renum_states p l)
  using assms
  unfolding inv_def
  apply −
```

apply (*subst* (*asm*) *N_eq*, *assumption*)
apply (*subst renum.N_eq*, *subst renum_n_ps_simp*, *assumption*)
apply (*subst nth_map_index*, *simp add*: *n_ps_def*)
unfolding *conv_automaton_def automaton_of_def*
apply (*clarsimp split*: *prod.split_asm simp*: *renum_automaton_def comp_def*)
unfolding *aux_1*
apply (*subst default_map_of_map*[**where** *x* = []])
subgoal
by (*intro inj_renum_states* ‹*p* < *n_ps*›)
apply (*simp add*: *renum_cconstraint_def map_cconstraint_def*; *fail*)
apply (*erule map_u_renum_cconstraint_clock_valI*)
done

lemma *inv_renum_sem_D*:
assumes
map_u u ⊢ *Simple_Network_Language.inv* (*renum.N p*) (*renum_states p l*) *p* < *n_ps l* ∈ *loc_set*
shows
u ⊢ *Simple_Network_Language.inv* (*N p*) *l*
using *assms*
unfolding *inv_def*
apply −
apply (*subst N_eq*, *assumption*)
apply (*subst* (*asm*) *renum.N_eq*, *subst renum_n_ps_simp*, *assumption*)
apply (*subst* (*asm*) *nth_map_index*, *simp add*: *n_ps_def*)
unfolding *conv_automaton_def automaton_of_def*
apply (*clarsimp split*: *prod.split simp*: *renum_automaton_def comp_def*)
unfolding *aux_1*
apply (*subst* (*asm*) *default_map_of_map*[**where** *x* = []])
subgoal
by (*intro inj_renum_states* ‹*p* < *n_ps*›)
apply (*simp add*: *renum_cconstraint_def map_cconstraint_def*; *fail*)
apply (*erule map_u_renum_cconstraint_clock_valD*)
done

lemma *dom_the_inv_comp*:
dom (*m o the_inv f*) = *f* ‘ *dom m* **if** *inj f range f* = *UNIV*
unfolding *dom_def*
apply (*clarsimp*, *safe*)
subgoal for *x y*
apply (*subgoal_tac f* (*the_inv f x*) = *x*)
apply *force*
using *that* **by** (*auto intro*: *f_the_inv_f*)
subgoal for *x y*
using *that* **by** (*auto simp*: *the_inv_f_f*)
done

lemma *inj_renum_vars*:
inj renum_vars
using *bij_renum_vars* **unfolding** *bij_def* **..**

lemma *surj_renum_vars*:
surj renum_vars
using *bij_renum_vars* **unfolding** *bij_def* **..**

```
lemma map_of_inv_map:
  map_of xs (the_inv f x) = map_of (map (λ (a, b). (f a, b)) xs) x
  if inj f surj f the_inv f x ∈ dom (map_of xs)
  apply (subgoal_tac x = f (the_inv f x))
  subgoal premises prems
    using domD[OF that(3)] ‹inj f›
    apply (subst (2) prems)
    apply safe
    apply (subst map_of_mapk_SomeI; assumption)
    done
  subgoal
    apply (rule sym, rule f_the_inv_f)
    using that by auto
  done

lemma dom_vars_invD:
  assumes x ∈ dom (s ∘ vars_inv)
  shows x ∈ renum_vars ‘ dom s (is ?A) and the_inv renum_vars x ∈ dom s (is ?B)
proof −
  show ?A
   using assms unfolding vars_inv_def dom_the_inv_comp[OF inj_renum_vars surj_renum_vars]
.
  then show ?B
    apply (elim imageE)
    apply simp
    apply (subst the_inv_f_f, rule inj_renum_vars, assumption)
    done
qed

lemma bounded_renumI:
  assumes bounded (map_of bounds′) s
  shows bounded (map_of (map (λ(a, y). (renum_vars a, y)) bounds′)) (s o vars_inv)
  using assms unfolding bounded_def
  apply elims
  apply intros
  subgoal
    unfolding dom_map_of_conv_image_fst vars_inv_def
   by (auto intro!: image_cong simp: dom_the_inv_comp[OF inj_renum_vars surj_renum_vars]
image_comp)
  subgoal for x
    apply (frule dom_vars_invD)
    apply (frule dom_vars_invD(2))
    apply (drule bspec, assumption)
    apply (auto simp: map_of_inv_map[OF inj_renum_vars surj_renum_vars] vars_inv_def)
    done
  subgoal for x
    apply (frule dom_vars_invD)
    apply (frule dom_vars_invD(2))
    apply (drule bspec, assumption)
    apply (auto simp: map_of_inv_map[OF inj_renum_vars surj_renum_vars] vars_inv_def)
    done
  done
```

lemma *map_of_renum_vars_simp*:
assumes
dom (*s o vars_inv*)
= *dom* (*map_of* (*map* (λ(*a*, *y*). (*renum_vars a*, *y*)) *bounds'*))
x ∈ *dom s dom s* ⊆ *var_set*
shows *map_of* (*map* (λ(*a*, *y*). (*renum_vars a*, *y*)) *bounds'*) (*renum_vars x*) = *map_of bounds' x*
proof −
have ∗:
map (λ(*a*, *y*). (*renum_vars a*, *y*)) *bounds'* = *map* (λ(*a*, *y*). (*renum_vars a*, *y*)) *bounds'*
by *auto*
have *x* ∈ *dom* (*map_of bounds'*)
unfolding *dom_map_of_conv_image_fst*
using *assms*
unfolding *vars_inv_def*
apply −
apply (*subst* (*asm*) *dom_the_inv_comp*, *rule inj_renum_vars*, *rule surj_renum_vars*)
apply (*subst* (*asm*) *dom_map_of_map*)
apply *clarsimp*
apply (*subst* (*asm*) *inj_on_image_eq_iff*[*OF inj_renum_vars*])
by *auto*
then obtain *y* **where** *map_of bounds' x* = *Some y*
by *auto*
then show *?thesis*
by (*subst map_of_mapk_SomeI*) (*auto intro*: *inj_renum_vars*)
qed

lemma *bounded_renumD*:
assumes
Simple_Network_Language.bounded
(*map_of* (*map* (λ(*a*, *y*). (*renum_vars a*, *y*)) *bounds'*)) (*s o vars_inv*)
and *dom s* ⊆ *var_set*
shows *Simple_Network_Language.bounded* (*map_of bounds'*) *s*
proof −
show *?thesis*
using *assms*(*1*)
unfolding *bounded_def*
apply *elims*
apply *intros*
subgoal
unfolding *vars_inv_def*
apply (*subst* (*asm*) *dom_the_inv_comp*[*OF inj_renum_vars surj_renum_vars*])
apply (*subst* (*asm*) *dom_map_of_map*)
apply (*rule inj_image_eqI*[*OF inj_renum_vars*], *simp add*: *dom_map_of_conv_image_fst*)
done
subgoal for *x*
using *assms*(*2*) **unfolding** *vars_inv_def*
apply (*subst* (*asm*) (*2*) *dom_the_inv_comp*[*OF inj_renum_vars surj_renum_vars*])
apply (*drule bspec*, *erule imageI*)
apply (*simp add*: *the_inv_f_f*[*OF inj_renum_vars*] *map_of_renum_vars_simp*[*unfolded vars_inv_def*])
done
subgoal for *x*
using *assms*(*2*) **unfolding** *vars_inv_def*

```
    apply (subst (asm) (2) dom_the_inv_comp[OF inj_renum_vars surj_renum_vars])
    apply (drule bspec, erule imageI)
     apply (simp add: the_inv_f_f[OF inj_renum_vars] map_of_renum_vars_simp[unfolded
vars_inv_def])
    done
  done
qed

lemma dom_bounds_var_set: dom bounds = var_set
  unfolding dom_bounds using bounds'_var_set .

lemma sem_states_loc_setD: L ! p ∈ loc_set if p < length automata L ∈ states for L p
  using that states_loc_set by (force simp: n_ps_def)

lemma trans_N_renumD:
  assumes (l, b, g, a, f, r, l') ∈ Simple_Network_Language.trans (N p) p < n_ps
  shows (renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f,
renum_reset r, renum_states p l')
  ∈ Simple_Network_Language.trans (renum.N p)
  using assms
 unfolding mem_trans_N_iff[OF assms(2)] renum.mem_trans_N_iff[unfolded renum_n_ps_simp,
OF assms(2)]
  by (force split: prod.split simp: renum_automaton_def n_ps_def)

lemma match_assumption2:
  assumes P a1 b1 a1 = a b1 = b shows P a b
  using assms by auto

lemma inj_pair:
  assumes inj f inj g
  shows inj (λ(a, b). (f a, g b))
  using assms unfolding inj_on_def by auto

lemma injective_functions:
  inj renum_reset inj renum_upd inj renum_act inj renum_cconstraint inj renum_bexp
  ⋀p. p < length automata ⟹ inj (renum_states p)
  subgoal
    unfolding renum_reset_def using bij_renum_clocks[THEN bij_is_inj] by simp
  subgoal
    unfolding renum_upd_def renum_exp_def using bij_renum_vars[THEN bij_is_inj]
    by (intro inj_pair exp.inj_map inj_mapI)
  subgoal
    unfolding renum_act_def using inj_renum_acts by (rule act.inj_map)
  subgoal
    unfolding renum_cconstraint_def map_cconstraint_def
    by (intro inj_mapI acconstraint.inj_map bij_renum_clocks bij_is_inj bij_id)
  subgoal
    unfolding renum_bexp_def by (intro bexp.inj_map inj_renum_vars)
  subgoal
    by (rule inj_renum_states, simp add: n_ps_def)
  done

lemma trans_N_renumI:
  assumes (
```

```
   renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f, renum_reset
r,
    renum_states p l')
  ∈ trans (renum.N p)  p < n_ps
  shows (l, b, g, a, f, r, l') ∈ trans (N p)
  using assms
  unfolding mem_trans_N_iff[OF assms(2)] renum.mem_trans_N_iff[unfolded renum_n_ps_simp,
OF assms(2)]
  apply (clarsimp split: prod.split_asm simp: renum_automaton_def n_ps_def)
  apply (fo_rule match_assumption2, assumption)
   apply (auto elim!: injD[rotated, THEN sym] intro: injective_functions)
  done

lemma renum_acconstraint_eq_convD:
  assumes map_acconstraint renum_clocks id g = conv_ac g'
  obtains g1 where g = conv_ac g1 g' = map_acconstraint renum_clocks id g1
  subgoal premises
    using assms
    apply (cases g'; cases g; clarsimp)
    subgoal for m c
      by (rule that[of acconstraint.LT c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.LE c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.EQ c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.GT c m]; simp)
    subgoal for m c
      by (rule that[of acconstraint.GE c m]; simp)
    done
  done

lemma renum_cconstraint_eq_convD:
  assumes renum_cconstraint g = conv_cc g'
  obtains g1 where g = conv_cc g1 g' = renum_cconstraint g1
proof −
  let ?f = λ(ac, ac'). SOME ac1. ac = conv_ac ac1 ∧ ac' = map_acconstraint renum_clocks id
ac1
  let ?g = map ?f (zip g g')
  from assms have length g = length g'
    unfolding renum_cconstraint_def map_cconstraint_def by −(drule arg_cong[where f =
length], simp)
  then have g = conv_cc ?g ∧ g' = renum_cconstraint ?g
    using assms
    by (simp add: comp_def renum_cconstraint_def map_cconstraint_def)
      (induction rule: list_induct2; simp; elim conjE renum_acconstraint_eq_convD; smt someI)
  then show ?thesis
    by (blast intro: that)
qed

lemma trans_sem_N_renumI:
  assumes (
  renum_states p l, renum_bexp b, renum_cconstraint g, renum_act a, renum_upds f, renum_reset
r,
```

renum_states p l′)
∈ *trans* (*renum.N p*) *p* < *n_ps*
shows (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *trans* (*N p*)
using *assms*(*1*) ‹*p* < *n_ps*› **by** (*simp add*: *trans_N_renumI*)

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 −
obtain *b′ g′ a′ f′ r′ l1* **where** *b* = *renum_bexp b′ g* = *renum_cconstraint g′ f* = *renum_upds f′*
a = *renum_act a′ r* = *renum_reset r′ l′* = *renum_states p l1*
using *assms*
unfolding *N_eq*[*OF assms*(*2*)] *renum.N_eq*[*unfolded renum_n_ps_simp*, *OF assms*(*2*)]
apply (*subst* (*asm*) *nth_map_index*)
subgoal
by (*simp add*: *n_ps_def*)
unfolding *renum_automaton_def automaton_of_def trans_def* **by** (*auto split*: *prod.split_asm*)
with *assms* **show** *?thesis*
by *untag* (*fastforce dest*!: *trans_sem_N_renumI*)
qed

lemma *committed_renum_eq*:
committed (*renum.N p*) = *renum_states p* ‘ *committed* (*N p*) **if** *p* < *n_ps*
unfolding
committed_def N_eq[*OF* ‹*p* < *n_ps*›] *renum.N_eq*[*unfolded renum_n_ps_simp*, *OF* ‹*p* < *n_ps*›]
apply (*subst nth_map_index*)
subgoal
using ‹*p* < *n_ps*› **by** (*simp add*: *n_ps_def*)
unfolding *automaton_of_def conv_automaton_def renum_automaton_def* **by** (*simp split*: *prod.split*)

lemma *urgent_renum_eq*:
urgent (*renum.N p*) = *renum_states p* ‘ *urgent* (*N p*) **if** *p* < *n_ps*
unfolding
urgent_def N_eq[*OF* ‹*p* < *n_ps*›] *renum.N_eq*[*unfolded renum_n_ps_simp*, *OF* ‹*p* < *n_ps*›]
apply (*subst nth_map_index*)
subgoal
using ‹*p* < *n_ps*› **by** (*simp add*: *n_ps_def*)
unfolding *automaton_of_def conv_automaton_def renum_automaton_def* **by** (*simp split*: *prod.split*)

lemma *renum_urgency_removed*:
∀ *i* < *n_ps*. *urgent* (*renum.N i*) = {}
using *urgency_removed*
apply *intros*
apply (*simp only*: *urgent_renum_eq*)
apply *simp*
done

```
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
   apply (induction rule: check_bexp_is_val.inducts)
   apply (solves ‹
    auto
      intro: check_bexp_is_val.intros
      simp: renum_bexp_def renum_exp_def vars_inv_def the_inv_f_f[OF inj_renum_vars]
  ›)+
  subgoal
    apply (clarsimp simp: renum_bexp_def renum_exp_def vars_inv_def, safe)
    using is_val.simps apply fastforce+
    done
  apply (auto intro: check_bexp_is_val.intros simp: renum_exp_def)
  done

method solve_case =
  auto
    intro: check_bexp_is_val.intros
    simp: renum_bexp_def renum_exp_def vars_inv_def the_inv_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 ⟹ is_val s e v
  apply (induction
      s o vars_inv renum_bexp b bv and s o vars_inv renum_exp e _
      arbitrary: b and e rule: check_bexp_is_val.inducts)
  subgoal for b
    by (cases b; solve_case)
  subgoal for e bv b
    by (cases b; solve_case)
  subgoal for e1 a e2 bv b
    by (cases b; solve_case)
  subgoal for e1 a e2 bv b
    by (cases b; solve_case)
  subgoal for e1 a e2 bv b
    by (cases b; solve_case)
  subgoal for a v bv x b
    by (cases b; solve_case)
  subgoal for a v bv x b
    by (cases b; solve_case)
  subgoal for a v bv x b
    by (cases b; solve_case)
  subgoal for a v bv x b
    by (cases b; solve_case)
  subgoal for a v bv x b
    by (cases b; solve_case)
  subgoal for c e
    by (cases e; solve_case)
```

```
  subgoal for x v e
    by (cases e; solve_case)
  subgoal for e1 v1 e2 v2 b bv e
    by (clarsimp simp: renum_bexp_def renum_exp_def; safe; cases e; solve_case)
  subgoal for e1 v1 e2 v2 f e
    by (clarsimp simp: renum_bexp_def renum_exp_def; cases e; solve_case)
  subgoal for e1 v f e
    by (cases e; solve_case)
  done

lemma renum_reset_map_u:
  [renum_reset r→0]map_u u = map_u ([r→0]u)
  unfolding map_u_def
  apply (rule ext)
  subgoal for x
    apply (cases clocks_inv x ∈ set r; simp add: clocks_inv_def)
    subgoal
      using bij_renum_clocks[THEN bij_is_surj]
      by (auto
          simp: renum_reset_def f_the_inv_f[OF bij_is_inj, OF bij_renum_clocks]
          dest: imageI[where f = renum_clocks]
        )
    subgoal
      by (subst clock_set_id)
        (auto simp: renum_reset_def the_inv_f_f[OF bij_is_inj, OF bij_renum_clocks])
    done
  done

lemma bounded_renumI':
  assumes bounded bounds s'
  shows bounded renum.bounds (s' o vars_inv)
  using assms unfolding renum.bounds_def bounds_def by (simp add: bounded_renumI)

lemma bounded_renumD':
  assumes bounded renum.bounds (s' o vars_inv) dom s' ⊆ var_set
  shows bounded bounds s'
  using assms unfolding renum.bounds_def bounds_def by (simp add: bounded_renumD)

lemma is_upd_renumD:
  assumes is_upd s f s'
  shows is_upd (s o vars_inv) (renum_upd f) (s' o vars_inv)
  using assms
  unfolding is_upd_def renum_upd_def
  by (force dest!: is_val_renumD
                simp: bij_f_the_inv_f[OF bij_renum_vars] the_inv_f_f[OF inj_renum_vars]
vars_inv_def)

lemma is_upds_renumD:
  assumes is_upds s1 ps s'
  shows is_upds (s1 o vars_inv) (renum_upds ps) (s' o vars_inv)
  using assms by induction (auto intro: is_upds.intros simp: comp_def dest!: is_upd_renumD)

lemma is_upds_concat_renumD:
  assumes is_upds s1 (concat ps) s'
```

shows *is_upds (s1 o vars_inv) (concat_map renum_upds ps) (s' o vars_inv)*
using *assms* **by** (*induction ps arbitrary*: *s1*) (*auto simp*: *comp_def map_concat dest*!: *is_upds_renumD*)

lemma *Ball_mono*:
assumes $\forall x \in S.\ P\ x$ $\bigwedge x.\ x \in S \Longrightarrow P\ x \Longrightarrow Q\ x$
shows $\forall x \in S.\ Q\ x$
using *assms* **by** *blast*

lemma *atLeastLessThan_upperD*:
assumes $S \subseteq \{l..<u\}$ $x \in S$
shows $x < u$
using *assms* **by** *auto*

lemma *imp_mono_rule*:
assumes *P1* $\longrightarrow$ *P2*
and *Q1* $\Longrightarrow$ *P1*
and *Q1* $\Longrightarrow$ *P2* $\Longrightarrow$ *Q2*
shows *Q1* $\longrightarrow$ *Q2*
using *assms* **by** *blast*

lemma *inj_id*:
inj id
by *auto*

lemma *step_single_renumD*:
assumes *step_u sem L s u a L' s' u' L* $\in$ *states dom s* $\subseteq$ *var_set*
shows *step_u renum.sem*
(*map_index renum_states L*) (*s o vars_inv*) (*map_u u*)
(*renum_label a*)
(*map_index renum_states L'*) (*s' o vars_inv*) (*map_u u'*)
using *assms*(*1−3*)
apply (*cases a*)

supply [*simp del*] = *map_map_index set_map*
— To keep the simplifier from breaking through locale abstractions

— Delay
subgoal
supply [*simp*] = *length_automata_eq_n_ps L_len*
apply (*subst* (*asm*) *A_split*)
apply (*subst renum.A_split*)
apply (*simp only*: *renum_label_def label.map renum_n_ps_simp*)
apply (*erule step_u_elims'*)
apply *simp*
apply (*rule step_u.intros*)
preferT ‹*''target invariant''*› **subgoal**
by *tag* (*auto 4 3 simp*: *dest*: *inv_renum_sem_I*[*OF _ _ sem_states_loc_setD*])
preferT ‹*''nonnegative''*› **subgoal**
by *assumption*
preferT ‹*''urgent''*› **subgoal**
using *renum_urgency_removed* **by** − (*tag*, *auto*)
preferT ‹*''bounded''*› **subgoal**
by *tag* (*rule bounded_renumI'*)
done

— Internal
subgoal for *a′*
supply [*simp*] = *length_automata_eq_n_ps L_len*
apply (*subst* (*asm*) *A_split*)
apply (*subst renum.A_split*)
apply (*simp only*: *renum_label_def label.map renum_n_ps_simp*)
apply (*erule step_u_elims′*)
unfolding *TAG_def*[*of ″range″*]
apply *simp*
apply (*rule step_u.intros*)
preferT ‹*TRANS ″silent″*›
apply (*solves* ‹*tag*, *drule* (*1*) *trans_N_renumD*, *subst nth_map*, (*simp add*: *renum_act_def*)+›)
preferT ‹*″committed″*› **subgoal**
by *tag* (*auto simp*: *committed_renum_eq dest*!: *inj_renum_states*[*THEN injD*, *rotated*])
preferT ‹*″bexp″*› **subgoal**
by *tag* (*erule check_bexp_renumD*)
preferT ‹*″guard″*› **subgoal**
by *tag* (*rule map_u_renum_cconstraint_clock_valI*)
preferT ‹*″target invariant″*› **subgoal**
apply (*tag− ″new loc″ TRANS ″silent″*)
apply *clarsimp*
apply (*rule inv_renum_sem_I*[*OF _ _ sem_states_loc_setD*[*OF _ state_preservation_updI*]])
apply *fastforce*+
done
preferT ‹*″loc″*›
apply (*tag*, *simp*; *fail*)
preferT ‹*″range″*›
apply (*tag*, *simp*; *fail*)
preferT ‹*″new loc″*›
apply (*tag*, *simp only*: *map_index_update*; *fail*)
preferT ‹*″new valuation″*›
apply (*tag*, *simp only*: *renum_reset_map_u*; *fail*)
preferT ‹*″is_upd″*› **subgoal**
by (*tag*, *erule is_upds_renumD*)
preferT ‹*″bounded″*›
apply (*tag*, *erule bounded_renumI′*; *fail*)
done

— Binary
subgoal for *a′*
supply [*simp*] = *length_automata_eq_n_ps L_len*
apply (*subst* (*asm*) *A_split*)
apply (*subst renum.A_split*)
apply (*simp only*: *renum_label_def label.map renum_n_ps_simp*)
apply (*erule step_u_elims′*)
unfolding *TAG_def*[*of RECV ″range″*] *TAG_def*[*of SEND ″range″*]
apply *simp*
apply (*rule step_u.intros*)
preferT ‹*″not broadcast″*› **subgoal**
apply *tag*
unfolding *renum.broadcast_def*
apply (*clarsimp simp*: *set_map*)
apply (*drule injD*[*OF inj_renum_acts*])

```
    apply (subst (asm) broadcast_def, simp)
    done
  preferT ‹TRANS ''in''›
  apply (solves ‹tag, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+›)
  preferT ‹TRANS ''out''›
  apply (solves ‹tag, drule (1) trans_N_renumD, subst nth_map, (simp add: renum_act_def)+›)
  preferT ‹''committed''› subgoal
    by tag (auto simp: committed_renum_eq dest!: inj_renum_states[THEN injD, rotated])

  preferT ‹''bexp''› subgoal
    by tag (erule check_bexp_renumD)

  preferT ‹''bexp''› subgoal
    by tag (erule check_bexp_renumD)

  preferT ‹''guard''› subgoal
    by tag (rule map_u_renum_cconstraint_clock_valI)

  preferT ‹''guard''› subgoal
    by tag (rule map_u_renum_cconstraint_clock_valI)

  preferT ‹''target invariant''› subgoal
    apply (tag− ''new loc'' TRANS _)
    apply clarsimp
  apply (rule inv_renum_sem_I[OF _ _ sem_states_loc_setD[OF _ state_preservation_updI]])
      apply (fastforce intro!: state_preservation_updI)+
    done
  preferT ‹''new loc''›
    apply (tag, simp add: map_index_update; fail)
  preferT ‹''new valuation''›
    apply (tag, simp add: renum_reset_map_u[symmetric] renum_reset_def; fail)
  preferT ‹''upd''›
    apply (tag, erule is_upds_renumD; fail)
  preferT ‹''upd''›
    apply (tag, erule is_upds_renumD; fail)
  preferT ‹''bounded''›
    apply (tag, erule bounded_renumI'; fail)
  apply (tag, simp; fail)+
  done

— Broadcast
subgoal for a'
  supply [simp] = length_automata_eq_n_ps L_len
  apply (subst (asm) A_split)
  apply (subst renum.A_split)
  apply (simp only: renum_label_def label.map renum_n_ps_simp)
  apply (erule step_u_elims')
  apply simp
  unfolding TAG_def[of SEND ''range'']

  apply (rule step_u.intros)

  preferT ‹''broadcast''› subgoal
    unfolding renum.broadcast_def by (tag, subst (asm) broadcast_def, simp add: set_map)
```

preferT ‹*TRANS ''out''*›
apply (*solves*‹*tag*, *simp*, *drule* (*1*) *trans_N_renumD*, *subst nth_map*, (*simp add*: *renum_act_def*)+›)

preferT ‹*TRANS ''in''*›
  apply (*tag*− *SEL* _)
 apply (*solves* ‹*auto dest*!: *trans_N_renumD simp add*: *renum_act_def atLeastLessThan_upperD*›)

preferT ‹*''committed''*› **subgoal**
  apply (*tag*− *SEL* _)
  apply (*simp add*: *committed_renum_eq*)
  apply (*erule disj_mono*[*rule_format*, *rotated 2*], (*simp*; *fail*))
  apply (*erule disj_mono*[*rule_format*, *rotated 2*])
  subgoal
    apply *clarify*
    apply (*rule rev_bexI*, *assumption*)
    apply (*auto simp*: *committed_renum_eq*)
    done
 apply (*auto simp*: *committed_renum_eq dest*!: *inj_renum_states*[*THEN injD*, *rotated*]; *fail*)
  done

preferT ‹*''maximal''*› **subgoal**
  apply *tag*
  apply *simp*
  apply (*erule all_mono*[*THEN mp*, *OF impI*, *rotated*])
  apply (*erule* (*1*) *imp_mono_rule*)
  apply (*erule* (*1*) *imp_mono_rule*)
  apply *clarify*
  apply (*drule trans_sem_N_renumI'*, *assumption*)
  apply (*untag*, *clarsimp simp*: *renum_act_def*)
  apply (*drule check_bexp_renumI*)
  apply (*drule InD2*[*OF inj_renum_acts*])
  apply (*fastforce dest*!: *map_u_renum_cconstraint_clock_valD*)
  done

preferT ‹*''target invariant''*› **subgoal for** *l b g f r l' p ps bs gs fs rs ls' s'*
  apply (*tag*− *SEL* _ *''new loc''* *TRANS* _)
  apply (*subgoal_tac fold* (λ*p L*. *L*[*p* := *ls' p*]) *ps L* ∈ *states*)
  subgoal
    apply *simp*
    apply (*erule all_mono*[*THEN mp*, *OF impI*, *rotated*], *erule* (*1*) *imp_mono_rule*,
          drule (*1*) *inv_renum_sem_I*)
    subgoal
      apply (*rule sem_states_loc_setD*, *simp*)
      apply (*rule state_preservation_updI*)
      subgoal
        by *blast*
      .
    subgoal
      apply (*fo_rule match_assumption2*[**where** *P* = *clock_val*], *assumption*, *rule HOL.refl*)
      apply (*drule states_lengthD*, *simp*)
      done
    done
  subgoal

```
      apply (rule state_preservation_fold_updI)
       apply (erule Ball_mono)
       apply (simp add: atLeastLessThan_upperD; blast)
      by simp
    done

  preferT ‹''bexp''›
    apply (tag, erule check_bexp_renumD; fail)
  preferT ‹''bexp''›
    apply (solves ‹tag, auto elim: check_bexp_renumD›)
  preferT ‹''guard''›
    apply (tag, erule map_u_renum_cconstraint_clock_valI; fail)
  preferT ‹''guard''›
    apply (solves ‹tag, auto elim: map_u_renum_cconstraint_clock_valI›)
  preferT ‹''new loc''›
    apply (tag, simp add: map_trans_broad_aux1[symmetric] map_index_update; fail)
  preferT ‹''new valuation''›
      apply (tag, simp add: renum_reset_map_u[symmetric] renum_reset_def map_concat
comp_def; fail)
  preferT ‹''upd''›
    apply (tag, erule is_upds_renumD; fail)
  preferT ‹''upds''›
    apply (tag, drule is_upds_concat_renumD, simp add: comp_def; fail)
  preferT ‹''bounded''›
    apply (tag, erule bounded_renumI'; fail)
  apply (tag, simp; fail)+
  done
done

lemma inj_the_inv:
  inj (the_inv f) if bij f
  by (auto simp: bij_f_the_inv_f[OF that] dest: arg_cong[where f = f] intro!: injI)

lemma inj_vars_inv:
  inj vars_inv
  using bij_renum_vars unfolding vars_inv_def by (rule inj_the_inv)

lemma comp_vars_inv_upd_commute:
  (s o vars_inv)(x ↦ y) = s(vars_inv x ↦ y) o vars_inv
  by (intro ext) (auto dest: injD[OF inj_vars_inv])

lemma is_upd_renumI:
  assumes is_upd (s o vars_inv) (renum_upd f) s'
  shows is_upd s f (s' o renum_vars)
  using assms
  unfolding is_upd_def renum_upd_def
  by (clarsimp dest!: is_val_renumI split: prod.split_asm simp: comp_vars_inv_upd_commute)
    (force simp: the_inv_f_f[OF inj_renum_vars] vars_inv_def)

lemma is_upd_renumI':
  assumes is_upd (s o vars_inv) (renum_upd f) s'
  obtains s1 where is_upd s f s1 s1 = s' o renum_vars
  by (simp add: assms is_upd_renumI)
```

lemma *is_upd_renumI''*:
assumes *is_upd s* (*renum_upd f*) *s'*
shows *is_upd* (*s o renum_vars*) *f* (*s' o renum_vars*)
proof −
have *s* = (*s o renum_vars*) *o vars_inv*
by (*intro ext*) (*auto simp*: *bij_f_the_inv_f*[*OF bij_renum_vars*] *vars_inv_def*)
with *assms* **show** *?thesis*
by (*intro is_upd_renumI*) *simp*
qed

lemma *is_upds_renumI*:
assumes *is_upds* (*s o vars_inv*) (*renum_upds upds*) *s'*
shows ∃ *s1*. *is_upds s upds s1* ∧ *s1* = *s' o renum_vars*
using *assms* **apply** (*induction s o vars_inv renum_upds upds s' arbitrary*: *upds s*)
subgoal for *upds s*
apply *simp*
apply (*subgoal_tac s o vars_inv o renum_vars* = *s*)
apply (*solves* ‹*auto intro*: *is_upds.intros simp*: *comp_def*›)
apply (*rule ext*)
apply (*simp add*: *vars_inv_def the_inv_f_f*[*OF inj_renum_vars*])
done

apply *clarsimp*
apply (*erule is_upd_renumI'*)
apply (*auto simp*: *vars_inv_def bij_f_the_inv_f*[*OF bij_renum_vars*] *comp_def intro*!: *is_upds.intros*)
done

lemma *is_upds_renumI'*:
assumes *is_upds* (*s o vars_inv*) (*renum_upds f*) *s'*
shows *is_upds s f* (*s' o renum_vars*)
using *is_upds_renumI*[*OF assms*] **by** *simp*

lemma *is_upds_renumI''*:
assumes *is_upds s* (*renum_upds ps*) *s'*
shows *is_upds* (*s o renum_vars*) *ps* (*s' o renum_vars*)
using *assms*
by (*induction renum_upds ps s' arbitrary*: *ps*) (*auto 4 3 intro*: *is_upd_renumI'' is_upds.intros*)

lemma *is_upds_concat_renumI''*:
assumes *is_upds s* (*concat_map renum_upds ps*) *s'*
shows *is_upds* (*s o renum_vars*) (*concat ps*) (*s' o renum_vars*)
apply (*rule is_upds_renumI''*)
using *assms* **by** (*simp add*: *map_concat*)

lemma *bounded_renumD1*:
assumes *bounded renum.bounds s' dom* (*s'* ∘ *renum_vars*) ⊆ *var_set*
shows *bounded bounds* (*s' o renum_vars*)
using *assms*
by (*intro bounded_renumD'*) (*auto simp*: *vars_inv_def bij_f_the_inv_f*[*OF bij_renum_vars*]
comp_def)

lemma *renum_reset_append*:
renum_reset xs @ *renum_reset ys* = *renum_reset* (*xs* @ *ys*)
unfolding *renum_reset_def* **by** *simp*

lemma *if_eq_distrib*:
(*if i = j then f i a else f j b*) = (*f j* (*if i = j then a else b*))
by *auto*

lemma *dom_comp_eq_vimage*:
dom (*s o f*) = *f −‘ dom s*
unfolding *dom_def* **by** *auto*

lemma *dom_comp_vars_inv_eqD*:
assumes *dom s′* = *dom* (*s o vars_inv*)
shows *dom* (*s′ o renum_vars*) = *dom s*
using *assms inj_renum_vars surj_renum_vars* **unfolding** *vars_inv_def*
by (*subst* (*asm*) *dom_the_inv_comp*) (*auto simp*: *dom_comp_eq_vimage dest*: *injD*)

lemma *sem_trans_upd_domD*:
assumes (*L* ! *p*, *b*, *g′*, *a*, *f*, *r*, *l1*) ∈ *trans* (*N p*) *p* < *n_ps*
shows *fst ‘ set f* ⊆ *var_set*
proof −
from *assms* **have** *fst ‘ set f* ⊆ *var_set*
unfolding *var_set_def*
apply −
apply (*rule semilattice_sup_class.sup.coboundedI2*)
apply *clarsimp*
apply (*inst_existentials* (*fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*) *‘ trans* (*N p*) *p*)
apply *force+*
done
then show *?thesis* **.**
qed

lemma *SilD*:
fixes *map_action*
assumes *Sil a* = *map_act map_action a1*
obtains *a′* **where** *a1* = *Sil a′ a* = *map_action a′*
using *assms* **by** (*cases a1*) *auto*

lemma *InD*:
fixes *map_action*
assumes *In a* = *map_act map_action a1*
obtains *a′* **where** *a1* = *In a′ a* = *map_action a′*
using *assms* **by** (*cases a1*) *auto*

lemma *OutD*:
fixes *map_action*
assumes *Out a* = *map_act map_action a1*
obtains *a′* **where** *a1* = *Out a′ a* = *map_action a′*
using *assms* **by** (*cases a1*) *auto*

lemma *In_OutD*:
assumes *In a* = *renum_act a1 Out a* = *renum_act a2*
obtains *a′* **where** *a* = *renum_acts a′ a1* = *In a′ a2* = *Out a′*
using *assms* **unfolding** *renum_act_def* **by** (*elim InD OutD*) (*auto simp*: *injD*[*OF inj_renum_acts*])

lemma *renum_sem_broadcast_eq*:

```
  renum.broadcast = renum_acts ‘ set broadcast
  unfolding renum.broadcast_def by simp

lemma inj_eq_iff:
  f x = f y ⟷ x = y if inj f
  using that unfolding inj_def by auto

lemma trans_sem_N_renum_broadcastI:
  assumes
    ∀ p∈set ps. (renum_states p (L ! p), bs p, gs p, In a, fs p, rs p, ls p) ∈ trans (renum.N p)
    set ps ⊆ {0..<n_ps}
  obtains bs' gs' fs' rs' ls' a' where
    ∀ p∈set ps. (L ! p, bs' p, gs' p, In a', fs' p, rs' p, ls' p) ∈ trans (N p)
    ∀ p∈set ps. bs p = renum_bexp (bs' p)
    ∀ p∈set ps. gs p = renum_cconstraint (gs' p)
    ps ≠ [] ⟶ a = renum_acts a'
    ∀ p∈set ps. fs p = renum_upds (fs' p)
    ∀ p∈set ps. rs p = renum_reset (rs' p)
    ∀ p∈set ps. ls p = renum_states p (ls' p)
proof −
  define t where
    t p ≡ SOME (b', g', a', f', r', l1). (L ! p, b', g', a', f', r', l1) ∈ trans (N p)
    ∧ bs p = renum_bexp b' ∧ gs p = renum_cconstraint g' ∧ In a = renum_act a' ∧ fs p =
renum_upds f'
    ∧ rs p = renum_reset r' ∧ ls p = renum_states p l1 for p
  define bs' where bs' p ≡ case t p of (b', g', a', f', r', l1) ⇒ b' for p
  define gs' where gs' p ≡ case t p of (b', g', a', f', r', l1) ⇒ g' for p
  define as' where as' p ≡ case t p of (b', g', a', f', r', l1) ⇒ a' for p
  define fs' where fs' p ≡ case t p of (b', g', a', f', r', l1) ⇒ f' for p
  define rs' where rs' p ≡ case t p of (b', g', a', f', r', l1) ⇒ r' for p
  define ls' where ls' p ≡ case t p of (b', g', a', f', r', l1) ⇒ l1 for p

  have ∗: case t p of (b', g', a', f', r', l1) ⇒
    (L ! p, b', g', a', f', r', l1) ∈ trans (N p)
    ∧ bs p = renum_bexp b' ∧ gs p = renum_cconstraint g' ∧ In a = renum_act a' ∧ fs p =
renum_upds f'
    ∧ rs p = renum_reset r' ∧ ls p = renum_states p l1 if p ∈ set ps for p
    using assms(1)
    apply −
    apply (drule bspec[OF _ that])
    apply (drule trans_sem_N_renumI')
    subgoal
      using assms(2) that by auto
    unfolding t_def by (untag, elims, fo_rule someI, auto)

  show ?thesis
  proof (cases ps)
    case Nil
    then show ?thesis
      by (auto intro: that)
  next
    case (Cons p ps')
    then have p ∈ set ps
      by auto
```

```
    define a' where a' = as' p
    have 1: as' q = a' and In a = renum_act a' if q ∈ set ps for q
      unfolding as'_def a'_def using *[OF that] *[OF ‹p ∈ set ps›]
      by (auto dest: injD[OF injective_functions(3)])
    then obtain a1 where a' = In a1 a = renum_acts a1
      using ‹p ∈ set ps› unfolding renum_act_def by (auto elim!: InD)
    then show ?thesis
      apply −
      apply (rule that[of bs' gs' a1 fs' rs' ls'])
      unfolding bs'_def gs'_def fs'_def rs'_def ls'_def
      by (solves ‹(intros, frule *)?, auto simp: as'_def dest: 1›)+
  qed
qed

lemmas all_mono_rule = all_mono[THEN mp, OF impI, rotated]

method prop_monos =
  erule all_mono_rule
  | erule (1) imp_mono_rule
  | erule disj_mono[rule_format, rotated 2]

lemma inv_sem_N_renum_broadcastI:
  assumes
∀ pa<n_ps.
  [renum_reset r @ concat (map rs ps)→0]map_u u
    ⊢ inv (renum.N pa)
        ((fold (λp L. L[p := ls p]) ps (map_index renum_states L)) [p := renum_states p l1] ! pa)
∀ p∈set ps. rs p = renum_reset (rs' p)
∀ p∈set ps. ls p = renum_states p (ls' p)
∀ p∈set ps. ls' p ∈ (⋃(l, b, g, a, r, u, l') ∈ trans (N p). {l, l'})
l1 ∈ (⋃(l, b, g, a, r, u, l') ∈ trans (N p). {l, l'})
L ∈ states
shows
∀ pa<n_ps.
  [r @ concat (map rs' ps)→0]u ⊢ inv (N pa) ((fold (λp L. L[p := ls' p]) ps L) [p := l1] ! pa)
proof −
  have [simp]: renum_reset r @ concat (map rs ps) = renum_reset (r @ concat (map rs' ps))
    using assms(2) by (simp cong: map_cong add: map_concat renum_reset_def)
  have [simp]: length L = n_ps
    using ‹L ∈ states› by auto
  have [simp]:
    ((fold (λp L. L[p := ls p]) ps (map_index renum_states L)) [p := renum_states p l1] ! q)
    = renum_states q ((fold (λp L. L[p := ls' p]) ps L) [p := l1] ! q)
    if q < n_ps for q
    using assms(4−) that
    apply (cases p = q)
     apply (simp add: fold_upds_aux_length)
    apply (simp
        add: assms(3) map_trans_broad_aux1[symmetric] fold_upds_aux_length
        cong: fold_cong)
    done
  have (fold (λp L. L[p := ls' p]) ps L)[p := l1] ! q ∈ loc_set if q < n_ps for q
    using assms(4−)
    apply (intro sem_states_loc_setD)
```

```
    subgoal
      using that by (simp add: n_ps_def)
    apply (erule state_preservation_updI)
    by (rule state_preservation_fold_updI)
  then show ?thesis
    using assms
     by (auto dest: inv_renum_sem_D simp: renum_reset_map_u simp del: map_map_index
set_map)
qed

method solve_triv =
  assumption
  | erule (1) bounded_renumD'; fail
  | rule inv_renum_sem_D[OF _ _ sem_states_loc_setD]; simp; fail
  | rule check_bexp_renumI; simp; fail
  | rule map_u_renum_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 ! p, b', g', a, f', r', l1) ∈ trans (N p) dom s = var_set p < n_ps
  shows ∀(x, _)∈set (renum_upds f'). x ∈ dom (s o vars_inv)
  unfolding renum_upd_def using assms
  by (auto simp: dom_comp_eq_vimage vars_inv_def the_inv_f_f[OF inj_renum_vars]
          dest!: sem_trans_upd_domD)

lemma step_single_renumI:
  assumes
    step_u renum.sem
      (map_index renum_states L) (s o vars_inv) (map_u u) a L' s' u'
    L ∈ states dom s ⊆ var_set dom s = var_set
  shows ∃ a1 L1 s1 u1. step_u sem L s u a1 L1 s1 u1 ∧ renum_label a1 = a ∧
    L' = map_index renum_states L1 ∧ s' = s1 o vars_inv ∧ u' = map_u u1
  using assms(1−3)
  supply [simp] = length_automata_eq_n_ps L_len
  supply [simp del] = map_map_index set_map
  apply (subst A_split)
  apply (subst (asm) renum.A_split)
  apply (simp only: renum_label_def label.map renum_n_ps_simp)

  using [[goals_limit=2]]

  apply (cases a)

  — Delay
  subgoal
    apply (simp only:)
```

```
  apply (erule step_u_elims')
  apply intros
  apply (rule step_u.intros)
  preferT ‹''nonnegative''›
    apply assumption
  preferT ‹''urgent''› subgoal
    using urgency_removed by − (tag, auto)
  preferT ‹''target invariant''›
    apply (solves ‹tag, simp, prop_monos+, solve_triv+›)
  apply (tag, solve_triv)+
  done

— Internal
subgoal for a'
  apply (simp only:)
  apply (erule step_u_elims')
  subgoal premises prems[tagged] for l b g f r l' p
  using prems apply tag
  using sym[OF [[get_tagged ‹''loc''›]]]
  usingT TRANS _ ‹''range''›
  apply simp
  apply (drule (1) trans_sem_N_renumI', untag)
  apply elims
  unfolding renum_act_def
  apply (rule SilD, assumption)

  apply intros
  apply (rule step_u.intros(2))

  preferT ‹TRANS ''silent''› apply (tag, solve_triv)

  preferT ‹''committed''›
 apply (solves ‹tag, prop_monos; auto simp: committed_renum_eq dest: injD[OF inj_renum_states]›)

  preferT ‹''bexp''› apply (tag, solve_triv)
  preferT ‹''guard''› apply (tag, solve_triv)

  preferT ‹''target invariant''›
    apply (solves ‹tag ''new valuation'' ''new loc'',
      simp add: map_index_update[symmetric] renum_reset_map_u, prop_monos+,
      rule inv_renum_sem_D[OF _ _ sem_states_loc_setD],
      solve_triv+›)

  preferT ‹''is_upd''›
    apply (tag, solve_triv)

  preferT ‹''bounded''› subgoal
    apply (tag ‹''is_upd''›, erule bounded_renumD1)
    apply (drule is_upds_dom)
    subgoal
      apply (simp add: dom_comp_eq_vimage)
      unfolding renum_upd_def
      apply (clarsimp simp: vars_inv_def the_inv_f_f[OF inj_renum_vars] set_map)
      apply (drule (1) sem_trans_upd_domD)
```

```
      using assms(4)
      apply auto
      done
    apply (drule dom_comp_vars_inv_eqD, simp)
    done

  apply (tag, solve_triv)+
  usingT ‹''new loc''› apply solve_triv


  apply (rule ext)
  usingT ‹''new valuation''› apply solve_triv+
  done
done

— Binary
subgoal for a'
  apply (simp only:)
  apply (erule step_u_elims')
  subgoal premises prems[tagged] for l1 b1 g1 f1 r1 l1' p l2 b2 g2 f2 r2 l2' q s'a
  using prems apply tag
  apply (tag RECV ''loc'' SEND ''loc'')
  apply (drule sym[of map_index renum_states L ! _])+
  apply (tag TRANS _ RECV ''range'' SEND ''range'')
  apply simp
  apply (drule (1) trans_sem_N_renumI')
  apply (drule (1) trans_sem_N_renumI', untag)
  apply elims
  apply (rule In_OutD, assumption, assumption)

  apply intros
     apply (rule step_u.intros(3))
                  preferT ‹''not broadcast''›
                  apply (tag, simp add: renum_sem_broadcast_eq broadcast_def)
  using inj_renum_acts
  apply auto[1]
  preferT ‹TRANS ''in''› apply (tag, solve_triv)
  preferT ‹TRANS ''out''› apply (tag, solve_triv)
  preferT ‹''committed''› subgoal
    by (tag, auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states])
  preferT ‹''target invariant''›
    apply (
      solves ‹tag ''new valuation'' ''new loc'',
      simp add: map_index_update[symmetric] renum_reset_map_u renum_reset_append,
      prop_monos+,
      rule inv_renum_sem_D[OF _ _ sem_states_loc_setD],
      solve_triv+›)

  preferT ‹''upd''› apply (tag, solve_triv)
  preferT ‹''upd''› apply (tag, solve_triv)

  preferT ‹''bounded''› subgoal
    usingT− ‹''upd''›
    apply −
```

```
    apply (tag, erule bounded_renumD1)
    apply (drule is_upds_dom)
    subgoal
      using assms(4) by − (drule trans_sem_upd_domI; simp split: prod.splits)
    apply (drule is_upds_dom)
    subgoal
      using assms(4) by − (drule trans_sem_upd_domI; simp split: prod.splits)
    apply (simp, drule dom_comp_vars_inv_eqD, simp)
    done
  apply (tag, solve_triv)+
  apply (tag ''new loc'', solve_triv)
  apply (rule ext; solve_triv)
  apply (tag ''new valuation'', solve_triv)
  done
done

— Broadcast
subgoal for a'
  apply (simp only:)
  apply (erule step_u_elims')
  subgoal premises prems[tagged] for l b g f r l' p ps bs gs fs rs ls' s'a
  using prems apply (tag SEND ''range'' TRANS ''out'')
  using sym[OF [[get_tagged ‹SEND ''loc''›]]]
  apply simp
  apply (frule (1) trans_sem_N_renumI')
  apply elims
  subgoal premises prems[tagged] for b' g' a'a f' r' l1
    using prems(1−6) usingT− TRANS ''in'' SEL _ ‹''renum action''›
    unfolding renum_act_def
    apply −
    apply (rule OutD, assumption)
    apply (simp add: atLeastLessThan_upperD)
    apply (erule(1) trans_sem_N_renum_broadcastI)
    apply (subgoal_tac map fs ps = map renum_upds (map fs' ps))
     defer
     apply (simp; fail)

    subgoal for a1 bs' gs' fs' rs' ls1 a2

      apply intros
         apply (rule step_u.intros(4)[where ps = ps])

      preferT ‹TRANS ''out''› apply (tag ‹TRANS ''orig''›, solve_triv)

      preferT ‹''broadcast''›
       apply (tag,
       clarsimp simp: renum_sem_broadcast_eq inj_eq_iff[OF inj_renum_acts] broadcast_def;
fail)

      preferT ‹TRANS ''in''›
      apply (tag,
        cases ps = []; simp add: atLeastLessThan_upperD inj_eq_iff[OF inj_renum_acts]; fail)

      preferT ‹''committed''› subgoal
```

```
    apply (tag, prop_monos)
    apply (solves ‹auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states]›)
    apply prop_monos
    subgoal
        by (auto simp: committed_renum_eq inj_eq_iff[OF inj_renum_states] atLeast-
LessThan_upperD)
    subgoal premises prems
      using ‹L ∈ states› prems(21)
      by (auto simp: inj_eq_iff[OF inj_renum_states] committed_renum_eq)
    done

  preferT ‹''bexp''› apply (tag ‹''renum bexp''›, solve_triv)

  preferT ‹''bexp''› apply (tag, erule Ball_mono, solve_triv; fail)

  preferT ‹''guard''› apply (tag ‹''renum guard''›, solve_triv)

  preferT ‹''guard''› apply (tag, erule Ball_mono, solve_triv)

  preferT ‹''maximal''› subgoal
    apply tag
    apply simp
    apply prop_monos+
    apply clarify
    apply (drule (1) trans_N_renumD)+
    apply (simp add: renum_act_def)
    apply (drule check_bexp_renumD)
    apply (drule map_u_renum_cconstraint_clock_valI)
    apply blast
    done

  preferT ‹''target invariant''›
    apply (tag ''new loc'' ''new valuation'' ''renum reset'' ''renum loc'' TRANS _)
    apply (simp add: atLeastLessThan_upperD)
    apply (solves ‹erule (2) inv_sem_N_renum_broadcastI, blast, fast, solve_triv›)

  preferT ‹''upds''›
    apply (tag ''renum upds'', simp only:, solve_triv)

  preferT ‹''bounded''› subgoal
    apply (tag ''upd'' ''upds'' ''renum upds'', simp only:)
    apply (erule bounded_renumD1)
    apply (drule is_upds_dom)
    subgoal
      using assms(4) usingT ‹TRANS ''orig''› by (intro trans_sem_upd_domI)

    apply (drule is_upds_dom)
    subgoal
      using assms(4) usingT ‹''upd''› ‹''upds''›
      apply (simp add: set_map)
      apply (erule Ball_mono, drule trans_sem_upd_domI, assumption)
       apply (simp add: atLeastLessThan_upperD set_map; fail)+
      done
    apply (simp, drule dom_comp_vars_inv_eqD, simp)
```

```
      done

    preferT ‹''upd''› apply (tag ''renum upds'', solve_triv)

    apply (tag, solve_triv)+

    apply (tag ''new loc'' ''renum loc'',
      simp add: map_trans_broad_aux1 map_index_update cong: fold_cong; fail)

    apply (solves ‹auto simp: vars_inv_def bij_f_the_inv_f[OF bij_renum_vars]›)
    apply (tag ''new valuation'' ''renum reset'',
        simp add: renum_reset_map_u[symmetric] renum_reset_def map_concat comp_def
cong: map_cong;
      fail)
    done
   done
  done
  done
 done

lemma step_u_var_set_invariant:
 assumes step_u sem L s u a L' s' u' dom s = var_set
 shows dom s' = var_set
 using assms dom_bounds_var_set by (auto 4 4 dest!: bounded_inv simp: bounded_def)

lemmas step_u_invariants = states_inv step_u_var_set_invariant

interpretation single: Bisimulation_Invariant
 λ(L, s, u) (L', s', u'). ∃ a. step_u sem L s u a L' s' u'
 λ(L, s, u) (L', s', u'). ∃ a. step_u renum.sem L s u a L' s' u'
 λ(L, s, u) (L', s', u'). L' = map_index renum_states L ∧ s' = s o vars_inv ∧ u' = map_u u
 λ (L, s, u). L ∈ states ∧ dom s = var_set λ_. True
 apply standard
 supply [simp del] = map_map_index set_map
   apply clarsimp
 subgoal for L s u L' s' u' a
   by (drule (1) step_single_renumD, auto)
 subgoal
   by clarsimp (drule step_single_renumI[rotated]; blast)
 subgoal
   by clarsimp (intro conjI; elim step_u_invariants)
 subgoal
   .
 done

interpretation Bisimulation_Invariant
 λ(L, s, u) (L', s', u'). step_u' sem L s u L' s' u'
 λ(L, s, u) (L', s', u'). step_u' renum.sem L s u L' s' u'
 λ(L, s, u) (L', s', u'). L' = map_index renum_states L ∧ s' = s o vars_inv ∧ u' = map_u u
 λ (L, s, u). L ∈ states ∧ dom s = var_set λ_. True
proof −
 define rsem where rsem = renum.sem
 note step_single_renumD = step_single_renumD[folded rsem_def]
  have rsem ⊢ ⟨map_index renum_states L, (s ∘∘ Simple_Network_Rename_Defs.vars_inv)
```

renum_vars, *map_u u*⟩ → ⟨*map_index renum_states L'*, (*s'* ∘∘ *Simple_Network_Rename_Defs.vars_inv*) *renum_vars*, *map_u u'*⟩
if *sem* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩
and *L* ∈ *states*
and *dom s* = *var_set*
for *L* :: *'s list*
and *s* :: *'x* ⇒ *int option*
and *u* :: *'c* ⇒ *real*
and *L'* :: *'s list*
and *s'* :: *'x* ⇒ *int option*
and *u'* :: *'c* ⇒ *real*
proof −
from *that*(*1*) **obtain** *L1 s1 u1 a* **where**
sem ⊢ ⟨*L*, *s*, *u*⟩ →*Del* ⟨*L1*, *s1*, *u1*⟩ *a* ≠ *Del sem* ⊢ ⟨*L1*, *s1*, *u1*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩
by (*elim step_u'_elims*)
with *that*(*2*−) **show** *?thesis*
apply −
apply (*rule step_u'.intros*[**where** *a* = *renum_label a*])
apply (*erule* (*1*) *step_single_renumD*[**where** *a* = *Del*, *unfolded renum_label_def*, *simplified*], *blast*)
apply (*cases a*; (*fast* | *simp add*: *renum_label_def*))
apply (*erule step_single_renumD*)
apply (*blast dest*: *step_u_invariants*)+
done
qed
moreover have ∃ *a aa b*. *sem* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*a*, *aa*, *b*⟩ ∧ *L'* = *map_index renum_states a* ∧ *s'* = (*aa* ∘∘ *Simple_Network_Rename_Defs.vars_inv*) *renum_vars* ∧ *u'* = *map_u b*
if *L* ∈ *states*
and *dom s* = *var_set*
and *rsem* ⊢ ⟨*map_index renum_states L*, (*s* ∘∘ *Simple_Network_Rename_Defs.vars_inv*) *renum_vars*, *map_u u*⟩ → ⟨*L'*, *s'*, *u'*⟩
for *L* :: *'s list*
and *s* :: *'x* ⇒ *int option*
and *u* :: *'c* ⇒ *real*
and *L'* :: *nat list*
and *s'* :: *nat* ⇒ *int option*
and *u'* :: *nat* ⇒ *real*
proof −
from *that*(*3*) **obtain** *L1 s1 u1 a* **where**
rsem ⊢ ⟨*map_index renum_states L*, (*s o vars_inv*), *map_u u*⟩ →*Del* ⟨*L1*, *s1*, *u1*⟩
a ≠ *Del rsem* ⊢ ⟨*L1*, *s1*, *u1*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩
by (*elim step_u'_elims*)
with *that*(*1*,*2*) **show** *?thesis*
apply −
apply (*drule* (*1*) *step_single_renumI*[*folded rsem_def*], *blast*)
apply *safe*
subgoal premises *prems* **for** *a1 L1a s1a u1a*
proof −
{ **fix** *a1a L1b s1b u1b*
have
L ∈ *states* ⟹
dom s = *var_set* ⟹
renum_label a1a ≠
Simple_Network_Language.label.Del ⟹

```
        sem ⊢ ⟨L, s, u⟩ →a1 ⟨L1a, s1a, u1a⟩ ⟹
        renum_label a1 =
        Simple_Network_Language.label.Del ⟹
        L1 = map_index renum_states L1a ⟹
        u1 = map_u u1a ⟹
        s1 =
        (s1a ∘∘ Simple_Network_Rename_Defs.vars_inv)
         renum_vars ⟹
        sem ⊢ ⟨L1a, s1a, u1a⟩ →a1a ⟨L1b, s1b, u1b⟩ ⟹
        a = renum_label a1a ⟹
        L' = map_index renum_states L1b ⟹
        s' =
        (s1b ∘∘ Simple_Network_Rename_Defs.vars_inv)
         renum_vars ⟹
        u' = map_u u1b ⟹
        a1 = Simple_Network_Language.label.Del
         by (cases a1; simp add: renum_label_def)
      } note ∗ = this
      from prems show ?thesis
        apply −
      apply (drule step_single_renumI[folded rsem_def])
        apply (blast dest: step_u_invariants)+
      apply (subgoal_tac a1 = Del)
       apply clarsimp
       apply intros
          apply (erule step_u'.intros)
      apply (auto intro: ∗)
      done
  qed
  done
qed
moreover have x1b ∈ states ∧ dom x1c = var_set
  if x1 ∈ states
    and dom x1a = var_set
    and sem ⊢ ⟨x1, x1a, x2a⟩ → ⟨x1b, x1c, x2c⟩
  for x1 :: 's list
    and x1a :: 'x ⇒ int option
    and x2a :: 'c ⇒ real
    and x1b :: 's list
    and x1c :: 'x ⇒ int option
    and x2c :: 'c ⇒ real
  using that by (elim step_u'_elims) (auto 4 4 dest: step_u_invariants)
moreover note ∗ = calculation
show Bisimulation_Invariant
   (λ(L, s, u) (L', s', u'). sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λ(L, s, u) (L', s', u'). renum.sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
   (λ(L, s, u) (L', s', u').
       L' = map_index renum_states L ∧
       s' = (s o vars_inv) ∧
       u' = map_u u)
   (λ(L, s, u). L ∈ states ∧ dom s = var_set) (λ_. True)
  unfolding rsem_def[symmetric]
  apply (standard; clarsimp split: prod.splits)
  by (rule ∗; assumption)+
```

qed

interpretation *Bisimulation_Invariant*
λ(L, s, u) (L′, s′, u′). step_u′ sem L s u L′ s′ u′
λ(L, s, u) (L′, s′, u′). step_u′ renum.sem L s u L′ s′ u′
λ(L, s, u) (L′, s′, u′). L′ = map_index renum_states L ∧ s′ = s o vars_inv ∧ u′ = map_u u
λ (L, s, u). L ∈ states ∧ dom s = var_set λ_. True
proof −
define *rsem* **where** *rsem = renum.sem*
note *step_single_renumD = step_single_renumD[folded rsem_def]*
have *rsem ⊢ ⟨map_index renum_states L, (s ∘∘ Simple_Network_Rename_Defs.vars_inv) renum_vars, map_u u⟩ → ⟨map_index renum_states L′, (s′ ∘∘ Simple_Network_Rename_Defs.vars_inv) renum_vars, map_u u′⟩*
if *sem ⊢ ⟨L, s, u⟩ → ⟨L′, s′, u′⟩*
and *L ∈ states*
and *dom s = var_set*
for *L :: 's list*
and *s :: 'x ⇒ int option*
and *u :: 'c ⇒ real*
and *L′ :: 's list*
and *s′ :: 'x ⇒ int option*
and *u′ :: 'c ⇒ real*
proof −
from *that(1)* **obtain** *L1 s1 u1 a* **where**
sem ⊢ ⟨L, s, u⟩ →Del ⟨L1, s1, u1⟩ a ≠ Del sem ⊢ ⟨L1, s1, u1⟩ →a ⟨L′, s′, u′⟩
by (*elim step_u′_elims*)
with *that(2−)* **show** *?thesis*
apply −
apply (*rule step_u′.intros*[**where** *a = renum_label a*])
apply (*erule (1) step_single_renumD*[**where** *a = Del*, *unfolded renum_label_def*, *simplified*], *blast*)
apply (*cases a*; (*fast* | *simp add*: *renum_label_def*))
apply (*erule step_single_renumD*)
apply (*blast dest*: *step_u_invariants*)+
done
qed
moreover have *∃ a aa b. sem ⊢ ⟨L, s, u⟩ → ⟨a, aa, b⟩ ∧ L′ = map_index renum_states a ∧ s′ = (aa ∘∘ Simple_Network_Rename_Defs.vars_inv) renum_vars ∧ u′ = map_u b*
if *L ∈ states*
and *dom s = var_set*
and *rsem ⊢ ⟨map_index renum_states L, (s ∘∘ Simple_Network_Rename_Defs.vars_inv) renum_vars, map_u u⟩ → ⟨L′, s′, u′⟩*
for *L :: 's list*
and *s :: 'x ⇒ int option*
and *u :: 'c ⇒ real*
and *L′ :: nat list*
and *s′ :: nat ⇒ int option*
and *u′ :: nat ⇒ real*
proof −
from *that(3)* **obtain** *L1 s1 u1 a* **where**
rsem ⊢ ⟨map_index renum_states L, (s o vars_inv), map_u u⟩ →Del ⟨L1, s1, u1⟩
a ≠ Del rsem ⊢ ⟨L1, s1, u1⟩ →a ⟨L′, s′, u′⟩
by (*elim step_u′_elims*)
with *that(1,2)* **show** *?thesis*

```
    apply −
    apply (drule (1) step_single_renumI[folded rsem_def], blast)
    apply safe
    subgoal premises prems for a1 L1a s1a u1a
    proof −
      {fix a1a L1b s1b u1b
       have
        L ∈ states ⟹
        dom s = var_set ⟹
        renum_label a1a ≠
        Simple_Network_Language.label.Del ⟹
        sem ⊢ ⟨L, s, u⟩ →a1 ⟨L1a, s1a, u1a⟩ ⟹
        renum_label a1 =
        Simple_Network_Language.label.Del ⟹
        L1 = map_index renum_states L1a ⟹
        u1 = map_u u1a ⟹
        s1 =
        (s1a ∘∘ Simple_Network_Rename_Defs.vars_inv)
         renum_vars ⟹
        sem ⊢ ⟨L1a, s1a, u1a⟩ →a1a ⟨L1b, s1b, u1b⟩ ⟹
        a = renum_label a1a ⟹
        L′ = map_index renum_states L1b ⟹
        s′ =
        (s1b ∘∘ Simple_Network_Rename_Defs.vars_inv)
         renum_vars ⟹
        u′ = map_u u1b ⟹
        a1 = Simple_Network_Language.label.Del
        by (cases a1; simp add: renum_label_def)
      } note ∗∗ = this
      from prems show ?thesis
        apply −
        apply (drule step_single_renumI[folded rsem_def])
          apply (blast dest: step_u_invariants)+
        apply (subgoal_tac a1 = Del)
         apply clarsimp
         apply intros
           apply (erule step_u′.intros)
        apply (auto intro: ∗∗)
        done
  qed
  done
qed
moreover have x1b ∈ states ∧ dom x1c = var_set
  if x1 ∈ states
    and dom x1a = var_set
    and sem ⊢ ⟨x1, x1a, x2a⟩ → ⟨x1b, x1c, x2c⟩
  for x1 :: 's list
    and x1a :: 'x ⇒ int option
    and x2a :: 'c ⇒ real
    and x1b :: 's list
    and x1c :: 'x ⇒ int option
    and x2c :: 'c ⇒ real
  using that by (elim step_u′_elims) (auto 4 4 dest: step_u_invariants)
moreover note ∗ = calculation
```

show *Bisimulation_Invariant*
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *sem* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*). *renum.sem* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(λ(*L*, *s*, *u*) (*L'*, *s'*, *u'*).
L' = *map_index renum_states L* ∧
s' = (*s o vars_inv*) ∧
u' = *map_u u*)
(λ(*L*, *s*, *u*). *L* ∈ *states* ∧ *dom s* = *var_set*) (λ_. *True*)
unfolding *rsem_def*[*symmetric*]
apply (*standard*; *clarsimp split*: *prod.splits*)
by (*rule* ∗; *assumption*)+
qed

lemmas *renum_bisim* = *Bisimulation_Invariant_axioms*

end

locale *Simple_Network_Impl_Formula_real* =
Simple_Network_Rename_real **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* +
fixes Φ :: (*nat*, '*s*, '*x*, *int*) *formula*
and s_0 :: ('*x* × *int*) *list*
and L_0 :: '*s list*
begin

definition Φ' **where**
Φ' = *map_formula renum_states renum_vars id* Φ

definition a_0 **where**
a_0 = (L_0, *map_of* s_0, λ_. *0*)

definition a_0' **where**
a_0' = (*map_index renum_states* L_0, *map_of* (*map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0), λ_. *0*)

context
assumes L_0*_states*: L_0 ∈ *states*
assumes s_0*_dom*: *fst* ‘ *set* s_0 = *var_set* **and** s_0*_distinct*: *distinct* (*map fst* s_0)
begin

lemma *state_eq_aux*:
assumes *x* ∉ *renum_vars* ‘ *var_set*
shows *vars_inv x* ∉ *var_set*
unfolding *vars_inv_def*
proof *safe*
assume *the_inv renum_vars x* ∈ *var_set*
then have *renum_vars* (*the_inv renum_vars x*) = *x*
by (*intro f_the_inv_f inj_renum_vars*) (*simp add*: *surj_renum_vars*)
with *assms* ‹_ ∈ *var_set*› **show** *False*
by *force*
qed

lemma *state_eq*:
assumes *fst ‘ set* s_0 = *var_set distinct* (*map fst* s_0)
shows *map_of* (*map* (λ(*x*, *y*). (*renum_vars x*, *y*)) s_0) = (*map_of* s_0 ∘∘∘ *the_inv_into*) *UNIV* *renum_vars*
(**is** *?l* = *?r*)
proof (*rule ext*)
fix *x*
show *?l x* = *?r x*
proof (*cases x* ∈ *renum_vars ‘ fst ‘ set* s_0)
case *True*
then show *?thesis*
apply *clarsimp*
apply (*subst map_of_mapk_SomeI′*)
subgoal
using *inj_renum_vars* **by** (*auto intro*: *inj_on_subset*)
apply (*rule map_of_is_SomeI*, *rule assms*, *assumption*)
apply (*simp add*: *the_inv_f_f*[*OF inj_renum_vars*] s_0*_distinct*)
done
next
case *False*
then have *vars_inv x* ∉ *fst ‘ set* s_0
using *state_eq_aux assms*(*1*) **unfolding** *vars_inv_def* **by** *auto*
with *False* **show** *?thesis*
apply −
apply (*frule map_of_NoneI*)
apply (*simp add*: *vars_inv_def*)
apply (*auto simp*: *map_of_eq_None_iff*)
done
qed
qed

interpretation *Bisimulation_Invariant*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ renum.sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *L′* = *map_index renum_states L* ∧ *s′* = *s o vars_inv* ∧ *u′* = *map_u u*
λ (*L*, *s*, *u*). *L* ∈ *states* ∧ *dom s* = *var_set* λ_. *True*
by (*rule renum_bisim*)

lemma *start_equiv*:
A_B.equiv′ a_0 $a_0{}'$
unfolding *A_B.equiv′_def* a_0*_def* $a_0{}'$*_def*
apply (*clarsimp simp*: *vars_inv_def*, *intro conjI*)
subgoal
by (*intro state_eq* s_0*_dom* s_0*_distinct*)
subgoal
unfolding *map_u_def* **by** *auto*
subgoal
by (*rule* L_0*_states*)
subgoal
using s_0*_dom dom_map_of_conv_image_fst*[*of* s_0] **by** *fastforce*
done

lemma *check_sexp_equiv*:

assumes *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'*)
using *assms* **unfolding** *A_B.equiv'_def*
by (*induction e*)
(*simp add*: *inj_eq states_lengthD renum_states_inj vars_inv_def the_inv_f_f*[*OF inj_renum_vars*])+

lemma *models_iff*:
sem,a_0 ⊨ Φ = *renum.sem*,a_0' ⊨ Φ' **if** *locs_of_formula* Φ ⊆ {*0*..<*n_ps*}
proof −
have *rel_ctl_formula compatible* (*ctl_of* Φ) (*ctl_of* Φ')
using *that* **unfolding** Φ'*_def*
by (*cases* Φ; *auto simp*: *check_sexp_equiv prop_of_def rel_fun_def*)
with *start_equiv* **show** *?thesis*
by (*simp add*: *models_ctl_iff CTL_compatible*[*THEN rel_funD*, *symmetric*])
qed

lemma *has_deadlock_iff*:
has_deadlock sem a_0 ⟷ *has_deadlock renum.sem* a_0'
unfolding *has_deadlock_def* **using** *start_equiv* **by** (*intro deadlock_iff*, *unfold A_B.equiv'_def*)
auto

end

end

lemma *Bisimulation_Invariants_Bisimulation_Invariant*:
assumes *Bisimulation_Invariants A B sim PA PA PB PB*
shows *Bisimulation_Invariant A B sim PA PB*
proof −
interpret *Bisimulation_Invariants A B sim PA PA PB PB*
by (*rule assms*)
show *?thesis*
by (*standard*; *blast intro*: *A_B_step B_A_step*)
qed

lemma *Bisimulation_Invariants_Bisimulation_Invariant_iff*:
Bisimulation_Invariants A B sim PA PA PB PB ⟷ *Bisimulation_Invariant A B sim PA PB*
using
Bisimulation_Invariants_Bisimulation_Invariant Bisimulation_Invariant_Bisimulation_Invariants
by *blast*

lemmas *Bisimulation_Invariant_composition* =
Bisimulation_Invariant_Invariants_composition[
THEN Bisimulation_Invariants_Bisimulation_Invariant,
OF _ Bisimulation_Invariant_Bisimulation_Invariants]

lemma *Bisimulation_Invariant_refl*:
Bisimulation_Invariant A A (=) *P P* **if** ⋀*a b*. *P a* ⟹ *A a b* ⟹ *P b*
by (*rule Bisimulation_Invariant.intro*) (*auto intro*: *that*)

locale *Simple_Network_Rename_int* =
Simple_Network_Rename_Defs_int **where** *automata* = *automata* +
Simple_Network_Rename' **where** *automata* = *automata*
for *automata* ::
('*s list* × '*s list*
× (('*a* :: *countable*) *act*, '*s*, '*c*, *int*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, *int*) *cconstraint*) *list*) *list* +
assumes *urgency_removed*: ∀ (_, *U*, _, _) ∈ *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*
unfolding *n_ps_def Prod_TA_Defs.n_ps_def* **by** *simp*

lemma *var_set_eq1*:
Prod_TA_Defs.var_set
(*set broadcast*, *map automaton_of* (*map conv_automaton automata*),
map_of bounds') = *var_set*
unfolding *sem_def sem_var_set_eq*[*symmetric*] **by** *simp*

lemma *urgency_removed'*:
∀ *i*<*n_ps*. *urgent*
(*Prod_TA_Defs.N* (*set broadcast*, *map automaton_of* (*map conv_automaton automata*), *map_of bounds'*) *i*)
= {}
unfolding *urgent_def n_ps_def Prod_TA_Defs.n_ps_def Prod_TA_Defs.N_def*
unfolding *conv_automaton_def automaton_of_def*
using *urgency_removed* **by** (*fastforce dest*: *nth_mem split*: *prod.split*)

sublocale *real*: *Simple_Network_Rename_real* **where** *automata* = *map conv_automaton automata*
apply *standard*
unfolding *n_ps_eq1 var_set_eq1*
by (*rule bij_renum_clocks renum_states_inj bij_renum_vars bounds'_var_set inj_renum_acts urgency_removed'*)+

lemma *sem_unfold1*:
real.sem = *sem*
unfolding *sem_def* **by** *simp*

lemma *var_set_eq*:
real.var_set = *sem.var_set*
unfolding *sem_unfold1*[*symmetric*] **..**

lemma *map_acconstraint_conv_ac_commute*:
map_acconstraint renum_clocks id (*conv_ac ac*) = *conv_ac* (*map_acconstraint renum_clocks id ac*)
by (*cases ac*; *simp*)

lemma *map_ccconstraint_conv_cc_commute*:
renum_cconstraint (*conv_cc g*) = *conv_cc* (*renum_cconstraint g*)
unfolding *renum_cconstraint_def map_cconstraint_def* **by** (*simp add*: *map_acconstraint_conv_ac_commute*)

lemma *rename_conv_automaton_commute*:
real.renum_automaton n (conv_automaton x) = conv_automaton (real.renum_automaton n x)
unfolding *real.renum_automaton_def conv_automaton_def*
by (*clarsimp split*: *prod.split simp*: *map_ccconstraint_conv_cc_commute*)

lemma *sem_unfold2*:
real.renum.sem = renum.sem
by (*simp add*: *Simple_Network_Impl.sem_def rename_conv_automaton_commute*)

sublocale *renum_bisim*: *Bisimulation_Invariant*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ renum.sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *L′ = map_index renum_states L ∧ s′ = s o vars_inv ∧ u′ = map_u u*
λ (*L*, *s*, *u*). *L ∈ sem.states ∧ dom s = var_set λ_. True*
apply (*rule Bisimulation_Invariant_sim_replace*)
apply (*rule Bisimulation_Invariant_composition*)
apply (*rule real.renum_bisim*[*unfolded sem_unfold1 sem_unfold2 sem_var_set_eq*])
apply (*rule Bisimulation_Invariant_refl*)
apply *auto*
done

lemmas *renum_bisim = renum_bisim.Bisimulation_Invariant_axioms*

end

locale *Simple_Network_Rename_Start′ =*
Simple_Network_Rename′ **where** *automata = automata*
for *automata* ::
(*′s list × ′s list*
× ((*′a* :: *countable*) *act*, *′s*, *′c*, *′t*, *′x* :: *countable*, *int*) *transition list*
× ((*′s* :: *countable*) *×* (*′c* :: *countable*, *′t*) *cconstraint*) *list*) *list +*
fixes *s*$_0$ :: (*′x × int*) *list*
and *L*$_0$ :: *′s list*
assumes *L*$_0$*_states*: *L*$_0$ *∈ states*
assumes *s*$_0$*_dom*: *fst ‘ set s*$_0$ *= var_set* **and** *s*$_0$*_distinct*: *distinct* (*map fst s*$_0$)
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 *a*$_0$ **where**
a$_0$ *= (L*$_0$, *map_of s*$_0$, *λ_. 0*)

definition *a*$_0$*′* **where**
a$_0$*′ = (map_index renum_states L*$_0$, *map_of* (*map* (*λ*(*x*, *v*). (*renum_vars x*, *v*)) *s*$_0$), *λ_. 0*)

lemma *state_eq_aux*:
assumes $x \notin$ *renum_vars* ‘ *var_set*
shows *vars_inv x* $\notin$ *var_set*
unfolding *vars_inv_def*
proof *safe*
assume *the_inv renum_vars x* $\in$ *var_set*
then have *renum_vars* (*the_inv renum_vars x*) = *x*
by (*intro f_the_inv_f real.inj_renum_vars*) (*simp add*: *real.surj_renum_vars*)
with *assms* ‹_ $\in$ *var_set*› **show** *False*
by *force*
qed

lemma *state_eq*:
assumes *fst* ‘ *set* s_0 = *var_set distinct* (*map fst* s_0)
shows *map_of* (*map* (λ(*x*, *y*). (*renum_vars x*, *y*)) s_0) = (*map_of* s_0 ∘∘∘ *the_inv_into*) *UNIV* *renum_vars*
(**is** *?l* = *?r*)
proof (*rule ext*)
fix *x*
show *?l x* = *?r x*
proof (*cases x* $\in$ *renum_vars* ‘ *fst* ‘ *set* s_0)
case *True*
then show *?thesis*
apply *clarsimp*
apply (*subst map_of_mapk_SomeI′*)
subgoal
using *real.inj_renum_vars* **by** (*auto intro*: *inj_on_subset*)
apply (*rule map_of_is_SomeI*, *rule assms*, *assumption*)
apply (*simp add*: *the_inv_f_f*[*OF real.inj_renum_vars*] s_0*_distinct*)
done
next
case *False*
then have *vars_inv x* $\notin$ *fst* ‘ *set* s_0
using *state_eq_aux assms*(*1*) **unfolding** *vars_inv_def* **by** *auto*
with *False* **show** *?thesis*
apply −
apply (*frule map_of_NoneI*)
apply (*simp add*: *vars_inv_def*)
apply (*auto simp*: *map_of_eq_None_iff*)
done
qed
qed

lemma *start_equiv*:
renum_bisim.A_B.equiv′ a_0 $a_0{}'$
unfolding *renum_bisim.A_B.equiv′_def* a_0*_def* $a_0{}'$*_def*
apply (*clarsimp simp*: *vars_inv_def*, *intro conjI*)
subgoal
by (*intro state_eq* s_0*_dom* s_0*_distinct*)
subgoal
unfolding *map_u_def* **by** *auto*
subgoal

```
    unfolding sem_states_eq by (rule L₀_states)
  subgoal
    using s₀_dom dom_map_of_conv_image_fst[of s₀] by fastforce
  done

lemma check_sexp_equiv:
  assumes renum_bisim.A_B.equiv' (L, s, u) (L', s', u') locs_of_sexp e ⊆ {0..<n_ps}
  shows
  check_sexp e L (the ∘ s) ⟷
   check_sexp (map_sexp renum_states renum_vars id e) L' (the ∘ s')
  using assms unfolding renum_bisim.A_B.equiv'_def
  by (induction e)
     (simp add:
     inj_eq sem.states_lengthD renum_states_inj vars_inv_def the_inv_f_f[OF real.inj_renum_vars])+

lemma check_sexp_compatible:
  assumes locs_of_sexp e ⊆ {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'))
  using check_sexp_equiv[OF _ assms] by auto

lemma has_deadlock_iff:
  has_deadlock sem a₀ ⟷ has_deadlock renum.sem a₀'
  unfolding has_deadlock_def using start_equiv
  by (intro renum_bisim.deadlock_iff, unfold renum_bisim.A_B.equiv'_def) auto

lemma state_formula_compatible:
  (⋃x ∈ set_state_formula φ. locs_of_sexp x) ⊆ {0..<n_ps} ⟹
  rel_state_formula renum_bisim.compatible
    (map_state_formula (λP (L, s, _). check_sexp P L (the o s)) φ)
    (map_state_formula (λP (L, s, _).
      check_sexp (map_sexp (λp. renum_states p) renum_vars id P) L (the o s))
     φ) and path_formula_compatible:
  (⋃x ∈ set_path_formula ψ. locs_of_sexp x) ⊆ {0..<n_ps} ⟹
  rel_path_formula renum_bisim.compatible
    (map_path_formula (λP (L, s, _). check_sexp P L (the o s)) ψ)
    (map_path_formula (λP (L, s, _).
      check_sexp (map_sexp (λp. renum_states p) renum_vars id P) L (the o s))
     ψ)
   by (induction φ and ψ) (auto simp: check_sexp_equiv prop_of_def rel_fun_def)

lemmas models_state_compatible = renum_bisim.models_state_compatible[OF state_formula_compatible]

end

locale Simple_Network_Rename_Formula_int =
  Simple_Network_Rename_Start_int where automata = automata
  for automata ::
    ('s list × '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
begin
```

definition Φ' **where**
$\Phi' = map_formula\ renum_states\ renum_vars\ id\ \Phi$

lemma *models_iff*:
$sem,a_0 \models \Phi = renum.sem,a_0' \models \Phi'$ **if** $locs_of_formula\ \Phi \subseteq \{0..<n_ps\}$
proof −
have *rel_ctl_formula renum_bisim.compatible* (*ctl_of* Φ) (*ctl_of* Φ')
using *that* **unfolding** Φ'*_def*
by (*cases* Φ; *auto simp*: *check_sexp_equiv prop_of_def rel_fun_def*)
with *start_equiv* **show** *?thesis*
by (*simp add*: *models_ctl_iff renum_bisim.CTL_compatible*[*THEN rel_funD*, *symmetric*])
qed

end

lemma *vars_of_bexp_finite*[*finite_intros*]:
finite (*vars_of_bexp* (*b*::('*a*, '*b*) *bexp*))
and *vars_of_exp_finite*[*finite_intros*]:
finite (*vars_of_exp* (*e*::('*a*, '*b*) *exp*))
by (*induction b* **and** *e*) *auto*

lemma *set_bexp_vars_of_bexp*:
set_bexp (*b*::('*a*, '*b*) *bexp*) = *vars_of_bexp b*
and *set_exp_vars_of_exp*:
set_exp (*e*::('*a*, '*b*) *exp*) = *vars_of_exp e*
by (*induction b* **and** *e*) *auto*

definition (**in** *Prod_TA_Defs*)
$act_set \equiv (\bigcup\ p \in \{0..<n_ps\}.\ \bigcup\ (l, e, g, a, _) \in trans\ (N\ p).\ act.set_act\ a) \cup broadcast$

lemma (**in** *Simple_Network_Impl*) *act_set_compute*:
act_set =
$\bigcup\ ((\lambda(_, _, t, _).\ \bigcup\ ((\lambda(l, e, g, a, _).\ act.set_act\ a)\ `\ set\ t))\ `\ set\ automata) \cup set\ broadcast$
unfolding *act_set_def*
apply (*fo_rule arg_cong2*)
apply (*clarsimp simp*: *N_p_trans_eq n_ps_def act_set_def broadcast_def*)
apply *safe*
apply (*clarsimp simp*: *N_p_trans_eq n_ps_def act_set_def broadcast_def*)
apply (*drule nth_mem*, *erule bexI*[*rotated*], *force*)
apply (*drule mem_nth*, *force*)
unfolding *broadcast_def*
apply *simp*+
done

lemma *set1_acconstraint_elim*:
assumes $c \in set1_acconstraint\ ac$
obtains x **where** $(c, x) = constraint_pair\ ac$
using *assms* **by** (*cases ac*) *auto*

lemma (**in** *Simple_Network_Impl*)
assumes $(x1, x2, T, I) \in set\ automata\ (l, b, g, a, f, r, l') \in set\ T$

shows *clk_set'I1*[*intro*]: $c \in set\ r \Longrightarrow c \in clk_set'$
and *clk_set'I2*[*intro*]: $ac \in set\ g \Longrightarrow c \in set1_acconstraint\ ac \Longrightarrow c \in clk_set'$
and *loc_setI1*[*intro*]: $l \in loc_set$ **and** *loc_setI2*[*intro*]: $l' \in loc_set$
and *act_setI*[*intro*]: $a' \in set_act\ a \Longrightarrow a' \in act_set$
and *var_setI1*[*intro*]: $v \in set_bexp\ b \Longrightarrow v \in var_set$
and *var_setI2*[*intro*]: $(x, e) \in set\ f \Longrightarrow x \in var_set$
and *var_setI3*[*intro*]: $(x, e) \in set\ f \Longrightarrow v \in set_exp\ e \Longrightarrow v \in var_set$
using *assms* **unfolding**
loc_set_compute act_set_compute var_set_compute set_bexp_vars_of_bexp set_exp_vars_of_exp
clk_set'_def
apply −
apply *force*
apply (*fastforce elim*: *set1_acconstraint_elim simp*: *clkp_set'_def collect_clock_pairs_def*)
apply (*simp*; *blast*)+
done

lemma (**in** *Simple_Network_Impl*) *clk_set'I3*[*intro*]:
assumes $(x1, x2, T, I) \in set\ automata$
shows $(l, g') \in set\ I \Longrightarrow ac \in set\ g' \Longrightarrow c \in set1_acconstraint\ ac \Longrightarrow c \in clk_set'$
using *assms* **unfolding** *clk_set'_def clkp_set'_def collect_clock_pairs_def*
by (*force elim*!: *set1_acconstraint_elim*)

definition
$find_remove\ P = map_option\ (\lambda(xs, x, ys).\ (x, xs\ @\ ys))\ o\ List.extract\ P$

fun *merge_pairs* :: $('a \times 'b\ list)\ list \Rightarrow ('a \times 'b\ list)\ list \Rightarrow ('a \times 'b\ list)\ list$ **where**
$merge_pairs\ []\ ys = ys$ |
$merge_pairs\ ((k, v)\ \#\ xs)\ ys = (case\ find_remove\ (\lambda(k', _).\ k' = k)\ ys\ of$
$None \Rightarrow (k, v)\ \#\ merge_pairs\ xs\ ys$
| $Some\ ((_, v'), ys) \Rightarrow (k, v\ @\ v')\ \#\ merge_pairs\ xs\ ys$
)

definition
$conv_urge\ urge \equiv \lambda(committed, urgent, trans, inv).$
$(committed,$
$[],$
$map\ (\lambda(l, b, g, a, f, r, l').\ (l, b, g, a, f, urge\ \#\ r, l'))\ trans,$
$merge_pairs\ (map\ (\lambda l.\ (l, [acconstraint.LE\ urge\ 0]))\ urgent)\ inv)$

lemma *map_of_distinct_upd2*:
assumes $x \notin set\ (map\ fst\ xs)$
shows $map_of\ (xs\ @\ (x,y)\ \#\ ys) = (map_of\ (xs\ @\ ys))(x \mapsto y)$
using *assms* **by** (*induction xs*) *auto*

lemma *map_of_distinct_lookup*:
assumes $x \notin set\ (map\ fst\ xs)$
shows $map_of\ (xs\ @\ (x,y)\ \#\ ys)\ x = Some\ y$
using *map_of_distinct_upd2*[*OF assms*] **by** *auto*

```
lemma find_remove_SomeD:
  assumes find_remove P xs = Some (x, ys)
  shows ∃ as bs. xs = as @ x # bs ∧ ys = as @ bs ∧ (∀ x ∈ set as. ¬ P x) ∧ P x
  using assms unfolding find_remove_def by (auto dest: extract_SomeE)

lemma find_map:
  find Q (map f xs) = map_option f (find P xs) if ∀ x ∈ set xs. Q (f x) ⟷ P x
  using that by (induction xs) auto

lemma find_remove_map:
  find_remove Q (map f xs) = map_option (λ(x, xs). (f x, map f xs)) (find_remove P xs)
  if ∀ x ∈ set xs. Q (f x) ⟷ P x
  using that
  by (induction xs)
     (auto simp: find_remove_def extract_Nil_code extract_Cons_code split: option.split)

lemma map_merge_pairs2:
  map (λ(k, v). (g k, map f v)) (merge_pairs xs ys)
  = merge_pairs (map (λ(k, v). (g k, map f v)) xs) (map (λ(k, v). (g k, map f v)) ys)
  if inj: inj_on g (fst ‘ (set xs ∪ set ys))
proof −
  have ∗:
    find_remove (λ(k', _). k' = g k) (map (λ(k, v). (g k, map f v)) ys)
   = map_option
      (λ((k, v), ys). ((g k, map f v), map (λ(k, v). (g k, map f v)) ys))
      (find_remove (λ(k', _). k' = k) ys)
    if inj_on g (fst ‘ (set xs ∪ set ys)) k ∈ fst ‘ set xs for k xs ys
    using that
    by (subst find_remove_map[where P = (λ(k', _). k' = k)])
      (auto elim: inj_onD[rotated, where A = fst ‘ (set xs ∪ set ys)]
        intro!: arg_cong2[where f = map_option])
  show ?thesis
    using inj
  proof (induction xs arbitrary: ys)
    case Nil
    then show ?case
      by simp
  next
    case (Cons x xs)
    then show ?case
      apply (clarsimp split: prod.split option.split simp: ∗[OF Cons(2)])
      apply (subst Cons.IH)
       apply (drule find_remove_SomeD)
       apply auto
      done
  qed
qed

lemma map_merge_pairs:
  map (λ(k, v). (k, map f v)) (merge_pairs xs ys)
  = merge_pairs (map (λ(k, v). (k, map f v)) xs) (map (λ(k, v). (k, map f v)) ys)
  using map_merge_pairs2[where g = id] by auto

lemma map_map_commute:
```

map f (map g xs) = map g (map f xs) **if** $\forall x \in$ *set xs. f (g x) = g (f x)*
using *that* **by** *simp*

lemma *conv_automaton_conv_urge_commute*:
conv_automaton (conv_urge c A) = conv_urge c (conv_automaton A)
unfolding *comp_def conv_automaton_def conv_urge_def*
by (*simp split*: *prod.split add*: *map_merge_pairs comp_def*)

lemma *conv_automaton_conv_urge_commute'*:
conv_automaton o conv_urge c = conv_urge c o conv_automaton
unfolding *comp_def conv_automaton_conv_urge_commute* **..**

locale *Simple_Network_Rename_Defs_int_urge* =
Simple_Network_Rename_Defs_int **where** *automata = automata* **for** *automata* ::
('*s list* × '*s list*
× (('*a* :: *countable*) *act*, '*s*, '*c*, *int*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, *int*) *cconstraint*) *list*) *list*
+ **fixes** *urge* :: '*c*

lemma (**in** *Simple_Network_Rename_Defs*) *conv_automaton_of*:
Simple_Network_Language.conv_A (automaton_of A) = automaton_of (conv_automaton A)
unfolding *automaton_of_def conv_automaton_def*
Simple_Network_Language.conv_A_def
by (*force*
simp: *default_map_of_alt_def map_of_map Simple_Network_Language.conv_t_def split*: *prod.splits*
)

locale *Simple_Network_Rename* =
Simple_Network_Rename_Defs_int_urge **where** *automata = automata* **for** *automata* ::
('*s list* × '*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*:
$\forall i{<}n_ps.\ \forall x{\in}loc_set.\ \forall y{\in}loc_set.$ *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 *infinite_types*:
infinite (*UNIV* :: '*c set*) *infinite* (*UNIV* :: '*x set*) *infinite* (*UNIV* :: '*s set*)
infinite (*UNIV* :: '*a set*)
and *bounds'_var_set*: *fst* ‘ *set bounds'* = *var_set*
and *loc_set_invs*: $\bigcup$ ((λ*g. fst* ‘ *set g*) ‘ *set* (*map* (*snd o snd o snd*) *automata*)) ⊆ *loc_set*
and *loc_set_committed*: $\bigcup$ ((*set o fst*) ‘ *set automata*) ⊆ *loc_set*
and *loc_set_urgent*: $\bigcup$ ((*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'
unfolding *clk_set'_def* **unfolding** *clkp_set'_def* **by** (*intro finite_intros*) *auto*

lemmas [*finite_intros*] = *trans_N_finite*

lemma *set_act_finite*[*finite_intros*]:
finite (*set_act a*)
by (*cases a*) *auto*

lemma *loc_set_finite*:
finite loc_set
unfolding *loc_set_def* **by** (*auto intro*!: *finite_intros*)

lemma *var_set_finite*[*finite_intros*]:
finite var_set
unfolding *var_set_def* **by** (*auto intro*!: *finite_intros*)

lemma *act_set_finite*:
finite act_set
unfolding *act_set_def broadcast_def* **by** (*auto intro*!: *finite_intros*)

lemma *bij_extend_bij_renum_clocks*:
bij (*extend_bij renum_clocks* (*insert urge clk_set'*))
by (*intro extend_bij_bij renum_clocks_inj clk_set'_finite infinite_types finite.insertI*) *simp*

lemma *renum_vars_bij_extends*[*simp*]:
extend_bij renum_vars var_set x = *renum_vars x* **if** *x* ∈ *var_set*
by (*intro extend_bij_extends*[*rule_format*] *renum_vars_inj var_set_finite infinite_types that*)

lemma *bij_extend_bij_renum_states*:
bij (*extend_bij renum_vars var_set*)
by (*intro extend_bij_bij renum_vars_inj var_set_finite infinite_types*) *simp*

lemma *inj_extend_bij_renum_states*: *inj* (*extend_bij* (*renum_states p*) *loc_set*) **if** *p* < *n_ps*
using *renum_states_inj infinite_types loc_set_finite* ‹*p* < *n_ps*›
by (*intro extend_bij_inj*) (*auto intro*!: *inj_onI*)

lemma *renum_states_extend*:
extend_bij (*renum_states p*) *loc_set l* = *renum_states p l* **if** *l* ∈ *loc_set p* < *n_ps*
using *renum_states_inj infinite_types loc_set_finite* ‹*p* < *n_ps*› ‹*l* ∈ *loc_set*›
by (*intro extend_bij_extends*[*rule_format*]) (*auto intro*!: *inj_onI*)

lemma *renum_acts_bij_extends*[*simp*]:
extend_bij renum_acts act_set x = *renum_acts x* **if** *x* ∈ *act_set*
by (*intro extend_bij_extends*[*rule_format*] *renum_acts_inj act_set_finite infinite_types that*)

lemma *inj_extend_bij_renum_acts*: *inj* (*extend_bij renum_acts act_set*)
using *renum_acts_inj infinite_types act_set_finite* **by** (*intro extend_bij_inj*) *auto*

lemma *constraint_clk_conv_ac*:
constraint_clk (*conv_ac ac*) = *constraint_clk ac*
by (*cases ac*) *auto*

interpretation *urge*: *Prod_TA_sem_urge*
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
urge
apply (*standard*; *clarsimp*)
subgoal

```
    using urge_not_in_clk_set
    unfolding conv_automaton_of
    unfolding automaton_of_def conv_automaton_def
    apply (clarsimp split: prod.split_asm)
    apply (drule default_map_of_in_listD)
    unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
    apply clarsimp
    subgoal premises prems
      using prems(1,7,8,10)
      unfolding constraint_clk_conv_ac unfolding constraint_clk_constraint_pair
      by force
    done
  subgoal
    using urge_not_in_clk_set
    unfolding conv_automaton_of
    unfolding automaton_of_def conv_automaton_def
    apply (clarsimp split: prod.split_asm)
    unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
    unfolding constraint_clk_conv_ac unfolding constraint_clk_constraint_pair
    apply force
    done

  subgoal
    using urge_not_in_clk_set
    unfolding conv_automaton_of
    unfolding automaton_of_def conv_automaton_def
    apply (clarsimp split: prod.split_asm)
    unfolding clk_set'_def clkp_set'_def collect_clock_pairs_def
    apply force
    done
  done


sublocale rename: Simple_Network_Rename_Defs_int
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set
  extend_bij renum_clocks (insert urge clk_set')
  λp. extend_bij (renum_states p) loc_set
  map (conv_urge urge) automata
  .

sublocale rename: Simple_Network_Rename_int
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set
  extend_bij renum_clocks (insert urge clk_set')
  λp. extend_bij (renum_states p) loc_set
  map (conv_urge urge) automata
  apply (standard;
      (intro allI impI bij_extend_bij_renum_clocks inj_extend_bij_renum_states
        inj_extend_bij_renum_acts bij_extend_bij_renum_states bounds'_var_set)?)
    apply (simp add: Prod_TA_Defs.n_ps_def; fail)
  subgoal
```

```
  unfolding bounds'_var_set rename.var_set_compute var_set_compute unfolding conv_urge_def
   by (fo_rule arg_cong2; fastforce)
 subgoal
   unfolding conv_urge_def by auto
 done

definition
 renum_upd' = map (λ(x, upd). (renum_vars x, map_exp renum_vars upd))

definition
 renum_reset' = map renum_clocks

definition renum_automaton' where
 renum_automaton' i ≡ λ(committed, trans, inv). let
   committed' = map (renum_states i) committed;
   trans' = map (λ(l, g, a, upd, r, l').
     (renum_states i l,
     map_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 −
 have renum_clocks: extend_bij renum_clocks (insert urge clk_set') c = renum_clocks c
   if c ∈ clk_set' for c
   apply (rule extend_bij_extends[rule_format])
     apply (rule renum_clocks_inj clk_set'_finite finite.insertI)+
   using that infinite_types by auto
 have renum_cconstraint: rename.renum_cconstraint g = renum_cconstraint g
   if ⋃ (set1_acconstraint ‘ (set g)) ⊆ clk_set' for g
   unfolding rename.renum_cconstraint_def renum_cconstraint_def map_cconstraint_def
   apply (rule map_cong, rule HOL.refl)
   apply (rule acconstraint.map_cong_pred, rule HOL.refl)
   using that
   apply (auto intro: renum_clocks[simplified] simp: pred_acconstraint_def)
   done
 show ?thesis
   using that[folded length_automata_eq_n_ps]
   unfolding rename.renum_automaton_def renum_automaton_def
   apply (clarsimp split: prod.split)
   apply safe
   subgoal committed
     using loc_set_committed
     by (subst renum_states_extend) (auto 4 3 simp: n_ps_def dest!: nth_mem)
   subgoal urgent
```

```
    using loc_set_urgent
    by (subst renum_states_extend) (auto 4 3 simp: n_ps_def dest!: nth_mem)
  subgoal start_locs
    by (subst renum_states_extend) (auto simp: n_ps_def dest!: nth_mem)
  subgoal state
    unfolding rename.renum_bexp_def renum_bexp_def
      by (auto dest!: nth_mem intro!: renum_vars_bij_extends bexp.map_cong_pred simp:
pred_bexp_def)
  subgoal guards
    by (auto dest!: nth_mem intro!: renum_cconstraint)
  subgoal actions
    unfolding rename.renum_act_def renum_act_def
   by (auto simp: pred_act_def dest!: nth_mem intro!: renum_acts_bij_extends act.map_cong_pred)
  subgoal upds
   unfolding renum_upd_def rename.renum_upd_def rename.renum_exp_def renum_exp_def
   by (auto dest!: nth_mem intro!: renum_vars_bij_extends exp.map_cong_pred simp: pred_exp_def)
  subgoal clock_resets
     unfolding rename.renum_reset_def renum_reset_def by (auto dest!: nth_mem intro!:
renum_clocks)
  subgoal dest_locs
    by (subst renum_states_extend) (auto simp: n_ps_def dest!: nth_mem)
  subgoal inv_locs
    using loc_set_invs
    by (subst renum_states_extend) (auto 4 4 simp: n_ps_def dest!: nth_mem)
  subgoal renum_clocks
    by (auto dest!: nth_mem intro!: renum_cconstraint)
  done
qed

lemma renum_urge_automaton_eq1:
  renum_automaton p (conv_urge urge (automata ! p))
  = conv_urge (renum_clocks urge) (renum_automaton p (automata ! p)) if p < n_ps
  unfolding renum_automaton_def conv_urge_def
  apply (simp split: prod.split)
  apply safe
  subgoal
    unfolding renum_reset_def by simp
  unfolding renum_cconstraint_def map_cconstraint_def
  apply (subst map_merge_pairs2)
  subgoal
    using loc_set_urgent loc_set_invs ‹p < n_ps› unfolding inj_on_def n_ps_def
    by (auto; force
        elim!: renum_states_inj[unfolded n_ps_def, simplified, rule_format, rotated −1]
        intro: nth_mem)
  apply (fo_rule arg_cong2; simp)
  done

lemma renum_urge_automaton_eq2:
  rename.real.renum_automaton p (conv_urge urge (automata ! p))
  = conv_urge
      (extend_bij renum_clocks (insert urge clk_set') urge)
      (rename.renum_automaton p (automata ! p))
  if p < n_ps
  unfolding rename.renum_automaton_def conv_urge_def
```

apply (*simp split*: *prod.split*)
apply *safe*
subgoal
unfolding *rename.renum_reset_def* **by** *simp*
unfolding *rename.renum_cconstraint_def map_cconstraint_def*
apply (*subst map_merge_pairs2*)
subgoal
by (*rule inj_extend_bij_renum_states that subset_UNIV inj_on_subset*)+
apply (*fo_rule arg_cong2*)
apply *simp*+
done

lemma *renum_urge_automaton_eq*:
rename.real.renum_automaton p (*conv_urge urge* (*automata* ! *p*)) =
renum_automaton p (*conv_urge urge* (*automata* ! *p*)) **if** *p* < *n_ps*
using *that*
unfolding *renum_urge_automaton_eq1*[*OF that*] *renum_urge_automaton_eq2*[*OF that*]
apply (*fo_rule arg_cong2*)
subgoal
by (*intro*
extend_bij_extends[*rule_format*] *renum_clocks_inj clk_set'_finite finite.insertI infinite_types*
) *auto*
by (*rule renum_automaton_eq*)

lemma *rename_N_eq_sem*:
rename.renum.sem =
Simple_Network_Language_Model_Checking.N
(*map renum_acts broadcast*)
(*map_index renum_automaton* (*map* (*conv_urge urge*) *automata*))
(*map* (λ(*a*,*p*). (*renum_vars a*,*p*)) *bounds'*)

unfolding *rename.renum.sem_def*
unfolding *Simple_Network_Language.conv_def*
apply *simp*
apply (*intro conjI*)
subgoal
by (*rule image_cong*; *intro HOL.refl renum_acts_bij_extends*; *simp add*: *act_set_def broadcast_def*)
subgoal
apply (*rule map_index_cong*)
unfolding *conv_automaton_of*
apply *safe*
apply (*fo_rule arg_cong*)+
apply (*erule renum_urge_automaton_eq*[*folded length_automata_eq_n_ps*])
done
subgoal
using *bounds'_var_set* **by** − (*fo_rule arg_cong*, *auto intro*: *renum_vars_bij_extends*)
done

lemma *map_of_merge_pairs*:
map_of (*merge_pairs xs ys*) = (λ*x*.
(*if x* ∈ *fst* ‘ *set xs* ∧ *x* ∈ *fst* ‘ *set ys then Some* (*the* (*map_of xs x*) @ *the* (*map_of ys x*))
else if x ∈ *fst* ‘ *set xs then map_of xs x*
else map_of ys x))

```
proof −
  have 1: False if find_remove (λ(k', _). k' = k) ys = None (k, x) ∈ set ys for k x ys
    using that unfolding find_remove_def by (auto simp: extract_None_iff)
  have 2: map_of xs k = Some x if
    find_remove (λ(k', _). k' = k) xs = Some ((k', x), ys) for k k' x xs ys
    using that
   by (auto 4 4 split: option.split dest: map_of_SomeD find_remove_SomeD simp: map_add_def)
  show ?thesis
    apply (rule ext)
    apply (induction xs arbitrary: ys)
     apply (simp; fail)
    apply (clarsimp split: if_split_asm option.split)
    apply (auto 4 3 split: option.split simp: map_add_def dest: find_remove_SomeD 2 1)
    done
qed

lemma default_map_of_merge_pairs:
  default_map_of [] (merge_pairs xs ys) = (λx.
  (if x ∈ fst ‘ set xs then the (map_of xs x) @ default_map_of [] ys x
   else default_map_of [] ys x))
  unfolding default_map_of_alt_def map_of_merge_pairs
  by (rule ext) (auto 4 3 dest: map_of_SomeD weak_map_of_SomeI split: if_split_asm)

lemma automaton_of_conv_urge_commute:
  automaton_of (conv_urge urge A) = urge.add_reset (urge.add_inv (automaton_of A))
  unfolding conv_urge_def urge.add_reset_def urge.add_inv_def automaton_of_def
  apply (clarsimp split: prod.splits)
  apply (rule ext)
  unfolding default_map_of_merge_pairs
  apply (clarsimp, safe)
  apply (clarsimp)
  subgoal premises prems for _ urgent _ _ l
  proof −
    have ∗: map_of (map (λx. (x, [acconstraint.LE urge 0])) urgent)
      = (λl. if l ∈ set urgent then Some [acconstraint.LE urge 0] else None)
      using map_of_map_keys[where
          m = λl. if l ∈ set urgent then Some [acconstraint.LE urge 0] else None]
      by (force cong: map_cong simp: dom_def)
    from ‹l ∈ set urgent› show ?thesis
      by (subst ∗) auto
  qed
  by auto

lemma urge_commute:
  rename.sem = urge.A_urge
  unfolding rename.sem_def urge.A_urge_def
  apply simp
 unfolding conv_automaton_conv_urge_commute conv_automaton_of automaton_of_conv_urge_commute
  by fast

lemma conv_automaton_of':
  automaton_of ∘ conv_automaton = Simple_Network_Language.conv_A o automaton_of
  unfolding comp_def conv_automaton_of ..
```

sublocale *urge_bisim*: *Bisimulation_Invariant*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *step_u′ rename.sem L s u L′ s′ u′*
λ(*L*, *s*, *u*) (*L′*, *s′*, *u′*). *L′* = *L* ∧ *s′* = *s* ∧ *u′* = *u*(*urge* := *0*)
λ (*L*, *s*, *u*). *L* ∈ *states λ*(*L*, *s*, *u*). *True*
unfolding *urge_commute sem_def conv_automaton_of′* **by** (*rule urge.urge_bisim*[*unfolded conv_states*])

lemma *check_sexp_equiv*:
assumes *urge_bisim.A_B.equiv′* (*L*, *s*, *u*) (*L′*, *s′*, *u′*)
shows *check_sexp e L* (*the o s*) = *check_sexp e L′* (*the o s′*)
using *assms* **unfolding** *urge_bisim.A_B.equiv′_def* **by** *simp*

context
fixes a_0 :: *′s list* × (*′x* ⇒ *int option*) × (*′c* ⇒ *real*)
assumes *start*: *fst* a_0 ∈ *states snd* (*snd* a_0) *urge* = *0*
begin

lemma *start_equiv*: *urge_bisim.A_B.equiv′* a_0 a_0
using *start* **unfolding** *urge_bisim.A_B.equiv′_def* **by** *auto*

lemma *urge_models_iff*:
sem,a_0 ⊨ Φ ⟷ *rename.sem*,a_0 ⊨ Φ
proof −
have *rel_ctl_formula urge_bisim.compatible* (*ctl_of* Φ) (*ctl_of* Φ)
by (*cases* Φ) (*auto simp*: *prop_of_def rel_fun_def check_sexp_equiv*)
with *start_equiv* **show** *?thesis*
by (*simp only*: *models_ctl_iff urge_bisim.CTL_compatible*[*THEN rel_funD*, *symmetric*])
qed

lemma *urge_has_deadlock_iff*:
has_deadlock sem a_0 ⟷ *has_deadlock rename.sem* a_0
unfolding *has_deadlock_def* **using** *start_equiv*
by (*intro urge_bisim.deadlock_iff*, *unfold urge_bisim.A_B.equiv′_def*) *auto*

lemma *state_formula_compatible*:
(⋃ *x* ∈ *set_state_formula φ*. *locs_of_sexp x*) ⊆ {*0*..<*n_ps*} ⟹
rel_state_formula urge_bisim.compatible
(*map_state_formula* (*λP* (*L*, *s*, _). *check_sexp P L* (*the o s*)) *φ*)
(*map_state_formula* (*λP* (*L*, *s*, _). *check_sexp P L* (*the o s*)) *φ*) **and** *path_formula_compatible*:
(⋃ *x* ∈ *set_path_formula ψ*. *locs_of_sexp x*) ⊆ {*0*..<*n_ps*} ⟹
rel_path_formula urge_bisim.compatible
(*map_path_formula* (*λP* (*L*, *s*, _). *check_sexp P L* (*the o s*)) *ψ*)
(*map_path_formula* (*λP* (*L*, *s*, _). *check_sexp P L* (*the o s*)) *ψ*)
by (*induction φ* **and** *ψ*) (*auto simp*: *check_sexp_equiv prop_of_def rel_fun_def*)

lemmas *urge_models_state_compatible* =
urge_bisim.models_state_compatible[*OF state_formula_compatible*]

end

end

locale *Simple_Network_Rename_Start* =
  Simple_Network_Rename **where** *automata* = *automata*
  for *automata* ::
    ('*s list* × '*s list*
      × (('*a* :: *countable*) *act*, '*s*, '*c*, *int*, '*x* :: *countable*, *int*) *transition list*
      × (('*s* :: *countable*) × ('*c* :: *countable*, *int*) *cconstraint*) *list*) *list* +
  fixes s_0 :: ('*x* × *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*
  unfolding *rename.length_automata_eq_n_ps*[*symmetric*] *n_ps_def* **by** *simp*

lemma *rename_states_eq*:
  rename.states = *states*
  unfolding *rename.states_def states_def rename_n_ps_eq*
  by (*simp add*: *rename.N_eq*[*unfolded rename_n_ps_eq*] *N_eq n_ps_def del*: *map_map*)
    (*auto simp*: *automaton_of_def conv_urge_def trans_def split*: *prod.splits*)

lemma *state_eq*:
  map_of (*map* (λ(*x*, *y*). (*extend_bij renum_vars var_set x*, *y*)) s_0) =
    map_of (*map* (λ(*x*, *y*). (*renum_vars x*, *y*)) s_0)
  using s_0*_dom* **by** − (*rule arg_cong*, *auto intro*: *renum_vars_bij_extends*)

lemma *sexp_eq*:
  assumes
    ‹*vars_of_sexp e* ⊆ *var_set*›
    ‹*set2_sexp e* ⊆ *loc_set*›
    ‹*locs_of_sexp e* ⊆ {*0*..<*n_ps*}›
  shows ‹*map_sexp* (λ*p*. *extend_bij* (*renum_states p*) *loc_set*) (*extend_bij renum_vars var_set*)
id e =
      map_sexp renum_states renum_vars id e›
  using *assms* **by** (*induction e*; *clarsimp simp*: *renum_states_extend*)

lemma L_0*_dom*:
  length L_0 = *n_ps set* L_0 ⊆ *loc_set*
  using L_0*_states* **by** (*auto intro*!: *states_loc_setD*)

definition
  a_0 = (L_0, *map_of* s_0, λ_. *0*)

definition
  a_0' = (
    map_index (λ*p*. *renum_states p*) L_0,
    map_of (*map* (λ(*x*, *y*). (*renum_vars x*, *y*)) s_0),
    λ_. *0*)

sublocale *rename*: *Simple_Network_Rename_Start_int*
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set

extend_bij renum_clocks (insert urge clk_set')
λp. extend_bij (renum_states p) loc_set
s_0 L_0
map (conv_urge urge) automata
apply (*standard*; (*rule L_0_states[folded rename_states_eq] s_0_distinct*)*?*)
unfolding *s_0_dom rename.bounds'_var_set[symmetric] bounds'_var_set* **..**

lemma *rename_a_0_eq*:
rename.a_0 = a_0
unfolding *rename.a_0_def a_0_def* **..**

lemma *rename_a_0'_eq*:
rename.a_0' = a_0'
unfolding *a_0'_def rename.a_0'_def*
apply (*clarsimp*, *rule conjI*)
subgoal
using *L_0_dom* **by** (*auto intro*!: *map_index_cong renum_states_extend*)
subgoal
unfolding *vars_inv_def* **using** *s_0_dom s_0_distinct*
by (*auto simp*: *state_eq Simple_Network_Rename_Defs.vars_inv_def*)
done

lemma *has_deadlock_iff*:
has_deadlock rename.renum.sem a_0' ⟷ has_deadlock sem a_0
proof −
have *has_deadlock sem a_0 ⟷ has_deadlock rename.sem a_0*
by (*rule urge_has_deadlock_iff*) (*auto intro*: *L_0_states simp*: *a_0_def*)
also have *... ⟷ has_deadlock rename.renum.sem a_0'*
by (*rule rename.has_deadlock_iff*[*unfolded rename_a_0_eq rename_a_0'_eq*])
finally show *?thesis* **..**
qed

lemma *N_eq_sem*:
Simple_Network_Language_Model_Checking.N broadcast automata bounds' = sem
unfolding *conv_alt_def sem_def*
by *safe* (*rule nth_equalityI*; *simp add*: *conv_N_eq N_eq sem_N_eq conv_automaton_of n_ps_def*)

lemma *rename_N_eq_sem'*:
Simple_Network_Language_Model_Checking.N
(*map renum_acts broadcast*)
(*map_index renum_automaton automata*)
(*map* (λ(*a*,*p*). (*renum_vars a*,*p*)) *bounds'*)
= renum.sem
unfolding *renum.sem_def*
unfolding *renum.conv_alt_def*
by *safe* (*rule nth_equalityI*; *simp add*: *conv_N_eq N_eq sem_N_eq conv_automaton_of n_ps_def*)

lemmas *has_deadlock_iff' =*
has_deadlock_iff[*unfolded rename_N_eq_sem*, *folded N_eq_sem*, *unfolded a_0_def a_0'_def*]

lemmas *start_equiv = start_equiv*[*of a_0*, *unfolded a_0_def*, *simplified*, *OF L_0_states*]

lemmas *urge_models_state_compatible =*
urge_models_state_compatible[*THEN rel_funD*, *OF _ _ _ start_equiv*,

of a_0, *unfolded* a_0*_def*, *simplified*, *OF* L_0*_states*]

lemmas *rename_models_state_compatible* =
rename.models_state_compatible[*THEN rel_funD*, *OF _ rename.start_equiv*,
*unfolded rename_*a_0*_eq* a_0*_def rename_n_ps_eq rename_*a_0*'_eq* a_0*'_def*]

lemmas *models_state_compatible* =
transp_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'*)
λ*p*. *extend_bij* (*renum_states p*) *loc_set*
s_0 L_0
map (*conv_urge urge*) *automata*
apply (*standard*; (*rule* L_0*_states*[*folded rename_states_eq*] s_0*_distinct*)*?*)
unfolding s_0*_dom rename.bounds'_var_set*[*symmetric*] *bounds'_var_set* **.**

definition
Φ' = *map_formula* (λ*p*. *renum_states p*) *renum_vars id* Φ

lemma *rename_*Φ*'_eq*:
*rename.*Φ' = Φ'
using *formula_dom* **unfolding** *rename.*Φ*'_def* Φ*'_def* **by** (*induction* Φ; *clarsimp simp*: *sexp_eq*)

lemma *models_iff1*:
rename.renum.sem,a_0' ⊨ Φ' ⟷ *rename.sem*,a_0 ⊨ Φ
by (*intro rename.models_iff*[*unfolded rename_*a_0*_eq rename_*a_0*'_eq rename_*Φ*'_eq rename_n_ps_eq*]
formula_dom sym)

lemma *models_iff2*:
rename.sem,a_0 ⊨ Φ ⟷ *sem*,a_0 ⊨ Φ
by (*rule sym*, *intro urge_models_iff formula_dom*) (*auto intro*: *formula_dom* L_0*_states simp*: a_0*_def*)

lemma *models_iff*:
rename.renum.sem,$a_0{}'$ $\models \Phi' \longleftrightarrow$ *sem*,a_0 $\models \Phi$
unfolding *models_iff1 models_iff2* **..**

lemmas *models_iff'* =
models_iff[*unfolded rename_N_eq_sem*, *folded N_eq_sem*, *unfolded* a_0*_def* $a_0{}'$*_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 φ
assumes *formula_is_false*: *formula* = *formula.EX* (*not sexp.true*)
begin

lemma *F_is_FalseI*:
shows *PR_CONST F* = (λ_. *False*)
by (*subst formula_is_false*) *auto*

lemma *final_fun_is_False*:
Fi a = *False*
by (*subst formula_is_false*) *auto*

lemmas *deadlock_checker_hoare* = *impl.deadlock_checker_hoare*[
OF final_fun_is_False F_is_FalseI, *folded deadlock_start_iff has_deadlock_def*]

end

schematic_goal *deadlock_checker_alt_def*:
impl.deadlock_checker ≡ *?impl*
unfolding *impl.deadlock_checker_def*
unfolding *impl.succs_impl_def*
unfolding *impl.check_deadlock_neg_impl_def impl.check_deadlock_impl_def*
apply (*abstract_let impl.is_start_in_states_impl is_start*)
unfolding *impl.is_start_in_states_impl_def*
apply (*abstract_let impl.E_op''_impl E_op''_impl*)
unfolding *impl.E_op''_impl_def fw_impl'_int*
apply (*abstract_let trans_impl trans_impl*)
unfolding *trans_impl_def*
apply (*abstract_let int_trans_impl int_trans_impl*)
apply (*abstract_let bin_trans_from_impl bin_trans_impl*)
apply (*abstract_let broad_trans_from_impl broad_trans_impl*)
unfolding *int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def*
apply (*abstract_let trans_in_broad_grouped trans_in_broad_grouped*)
apply (*abstract_let trans_out_broad_grouped trans_out_broad_grouped*)
apply (*abstract_let trans_in_map trans_in_map*)

apply (*abstract_let trans_out_map trans_out_map*)
apply (*abstract_let int_trans_from_all_impl int_trans_from_all_impl*)
unfolding *int_trans_from_all_impl_def*
apply (*abstract_let int_trans_from_vec_impl int_trans_from_vec_impl*)
unfolding *int_trans_from_vec_impl_def*
apply (*abstract_let int_trans_from_loc_impl int_trans_from_loc_impl*)
unfolding *int_trans_from_loc_impl_def*
apply (*abstract_let trans_i_map trans_i_map*)
unfolding *trans_out_broad_grouped_def trans_out_broad_map_def*
unfolding *trans_in_broad_grouped_def trans_in_broad_map_def*
unfolding *trans_in_map_def trans_out_map_def*
unfolding *trans_i_map_def*
apply (*abstract_let trans_map trans_map*)
apply (*abstract_let inv_fun :: nat list × int list ⇒ _ inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
unfolding *k_impl_alt_def*
apply (*abstract_let k_i k_i*)
apply (*abstract_let n_ps n_ps*)

unfolding *k_i_def impl.state_copy_impl_def impl.a$_0$_impl_def impl.abstr_repair_impl_def*
impl.subsumes_impl_def impl.emptiness_check_impl_def impl.abstra_repair_impl_def
impl.init_dbm_impl_def
by (*rule Pure.reflexive*)

end

concrete_definition *deadlock_checker* **uses**
Simple_Network_Impl_nat_ceiling_start_state.deadlock_checker_alt_def

definition *precond_dc* **where**
precond_dc
show_clock show_state broadcast bounds′ automata m num_states num_actions k L$_0$ s$_0$ formula
≡
if Simple_Network_Impl_nat_ceiling_start_state
broadcast bounds′ automata m num_states num_actions k L$_0$ s$_0$ formula
then
deadlock_checker
broadcast bounds′ automata m num_states num_actions k L$_0$ s$_0$ show_clock show_state
≫= (λ x. return (Some x))
else return None

theorem *deadlock_check*:
<emp> precond_dc
show_clock show_state broadcast bounds automata m num_states num_actions k L$_0$ s$_0$
(formula.EX (not sexp.true))
<λ Some r ⇒ ↑(
Simple_Network_Impl_nat_ceiling_start_state
broadcast bounds automata m num_states num_actions k L$_0$ s$_0$ (formula.EX (not sexp.true)) ∧
r = has_deadlock (N broadcast automata bounds) (L$_0$, map_of s$_0$, λ_ . 0)
)
| None ⇒ ↑(¬ Simple_Network_Impl_nat_ceiling_start_state

```
    broadcast bounds automata m num_states num_actions k L₀ s₀ (formula.EX (not sexp.true)))
  >ₜ
proof −
  define A where A ≡ N broadcast automata bounds
  define check where check ≡ A,(L₀, map_of s₀, λ_ . 0) ⊨ (formula.EX (not sexp.true))
  note [sep_heap_rules] = Simple_Network_Impl_nat_ceiling_start_state.deadlock_checker_hoare[
    OF _ HOL.refl,
    of broadcast bounds automata m num_states num_actions k L₀ s₀ _,
    unfolded UPPAAL_Reachability_Problem_precompiled_defs.init_def,
    folded A_def
    ]
  show ?thesis
    unfolding A_def[symmetric] precond_dc_def
    by (sep_auto simp: deadlock_checker.refine[symmetric] pure_def has_deadlock_def)
qed

end
theory Shortest_SCC_Paths
  imports Gabow_SCC.Gabow_SCC_Code
begin

definition compute_SCC_tr ::
  ('a :: hashable ⇒ bool, 'a ⇒ 'a list, 'a list, 'b) gen_g_impl_scheme ⇒ _ where
  compute_SCC_tr =
   Gabow_SCC_Code.compute_SCC_tr (=) bounded_hashcode_nat (def_hashmap_size TYPE('a))

— Example usage:
term compute_SCC_tr (| gi_V = (λ x. True), gi_E = (λ x. [3]), gi_V0 = [1], ... = 3 |)
```

Efficiently calculate shortest paths in a graph with non-negative edge weights, and where the only cycles are 0-cycles.

```
definition
calc_shortest_scc_paths G n ≡
let
  sccs = compute_SCC_tr G;
  d = replicate n None @ [Some 0];
  d = (
    fold
      (λ vs d.
        fold
          (λ u d.
            fold
              (λ v d.
                case d ! u of
                  None ⇒ d
                | Some du ⇒ (
                  case d ! v of
                    None ⇒ d[v := Some (du + more G u v)]
                  | Some dv ⇒
                      if du + more G u v < dv
                      then d[v := Some (du + more G u v)]
                      else d
                  )
              )
```

```
            (gi_E G u)
            d
          )
          vs
          d
      )
      sccs
      d
  );
  d = (
    fold
      (λ vs d.
        let
          dscc =
            fold
            (λ v dscc.
              case dscc of
                None ⇒ d ! v
              | Some d' ⇒ (
                  case d ! v of
                    None ⇒ dscc
                  | Some dv ⇒ Some (min dv d')
                )
            )
            vs
            None
        in
          fold (λ v d. d[v:=dscc]) 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* ≡ *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* × *nat list* ×
(*String.literal act*, *nat*, *String.literal*, *int*, *String.literal*, *int*) *transition list*
× (*nat* × (*String.literal*, *int*) *cconstraint*) *list*) *list*
begin

definition *check_renaming* **where** *check_renaming urge* Φ L_0 s_0 ≡ *combine* [
assert (∀ *i*<*n_ps*. ∀ *x*∈*loc_set*. ∀ *y*∈*loc_set*. *renum_states i x* = *renum_states i y* ⟶ *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 (⋃ ((λ*g*. *fst* ‘ *set g*) ‘ *set* (*map* (*snd o snd o snd*) *automata*)) ⊆ *loc_set*)
STR ''*Invariant locations are contained in the location set*'',
assert (⋃ ((*set o fst*) ‘ *set automata*) ⊆ *loc_set*)
STR ''*Broadcast locations are containted in the location set*'',
assert ((⋃*x*∈*set automata*. *set* (*fst* (*snd x*))) ⊆ *loc_set*)
STR ''*Urgent locations are containted in the location set*'',
assert (*urge* ∉ *clk_set*′)
STR ''*Urge not in clock set*'',
assert (L_0 ∈ *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* Φ ⊆ *loc_set*)
STR ''*Formula locations are contained in the location set*'',
assert (*locs_of_formula* Φ ⊆ {*0*..<*n_ps*})
STR ''*Formula automata are contained in the automata set*'',
assert (*vars_of_formula* Φ ⊆ *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* × *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*: $\bigcup$ ((λ*g*. *fst ‘ set g*) ‘ *set* (*map* (*snd o snd o snd*) *automata*)) ⊆ *loc_set*
and *loc_set_broadcast*: $\bigcup$ ((*set o fst*) ‘ *set automata*) ⊆ *loc_set*
and *loc_set_urgent*: $\bigcup$ ((*set o* (*fst o snd*)) ‘ *set automata*) ⊆ *loc_set*
and *urge_not_in_clk_set*: *urge* ∉ *clk_set'*
assumes L_0*_states*: $L_0 \in states$
and s_0*_dom*: *fst ‘ set* s_0 = *var_set* **and** s_0*_distinct*: *distinct* (*map fst* s_0)
assumes *formula_dom*:
set2_formula $\Phi \subseteq loc_set$
locs_of_formula $\Phi \subseteq \{0..{<}n_ps\}$
vars_of_formula $\Phi \subseteq var_set$
begin

interpretation *Simple_Network_Rename_Formula*
by (*standard*;
rule renum_states_inj renum_clocks_inj renum_vars_inj bounds'_var_set renum_acts_inj
loc_set_invs loc_set_broadcast loc_set_urgent urge_not_in_clk_set
infinite_literal infinite_UNIV_nat L_0*_states* s_0*_dom* s_0*_distinct formula_dom*
)+ — slow: 40s

lemmas *Simple_Network_Rename_intro* = *Simple_Network_Rename_Formula_axioms*

end

lemma *is_result_assert_iff*:
is_result (*assert b m*) ⟷ *b*
unfolding *assert_def* **by** *auto*

lemma *is_result_combine_Cons_iff*:
is_result (*combine* (*x # xs*)) ⟷ *is_result x* ∧ *is_result* (*combine xs*)
by (*cases x*; *cases combine xs*) *auto*

lemma *is_result_combine_iff*:
is_result (*a* <|> *b*) ⟷ *is_result a* ∧ *is_result b*
by (*cases a*; *cases b*) (*auto simp*: *combine2_def*)

context *Simple_Network_Rename_Formula_String_Defs*
begin

lemma *check_renaming*:
Simple_Network_Rename_Formula_String
broadcast bounds'
renum_acts renum_vars renum_clocks renum_states
automata urge Φ s_0 L_0 $\longleftrightarrow$
is_result (*check_renaming urge* Φ L_0 s_0)

unfolding *check_renaming_def Simple_Network_Rename_Formula_String_def*
by (*simp add*: *is_result_combine_Cons_iff is_result_assert_iff del*: *combine.simps*(*2*))

end

context *Simple_Network_Impl_nat_defs*
begin

definition *check_precond1* **where**
check_precond1 =
combine [
assert ($m > 0$)
(*STR ''At least one clock''*),
assert ($0 <$ *length automata*)
(*STR ''At least one automaton''*),
assert ($\forall i{<}$*n_ps. let* (_, _, *trans*, _) = (*automata* ! *i*) *in* $\forall$ (l, _, _, _, _, _, l') $\in$ *set trans.*
$l <$ *num_states i* $\wedge$ $l' <$ *num_states i*)
(*STR ''Number of states is correct* (*transitions*)*''*),
assert ($\forall i <$ *n_ps. let* (_, _, _, *inv*) = (*automata* ! *i*) *in* $\forall$ (x, _) $\in$ *set inv.* $x <$ *num_states i*)
(*STR ''Number of states is correct* (*invariants*)*''*),
assert ($\forall$ (_, _, *trans*, _) $\in$ *set automata.* $\forall$ (_, _, _, _, f, _, _) $\in$ *set trans.*
$\forall$ (x, *upd*) $\in$ *set f.* $x <$ *n_vs* $\wedge$ ($\forall i \in$ *vars_of_exp upd.* $i <$ *n_vs*))
(*STR ''Variable set bounded* (*updates*)*''*),
assert ($\forall$ (_, _, *trans*, _) $\in$ *set automata.* $\forall$ (_, b, _, _, _, _, _) $\in$ *set trans.*
$\forall i \in$ *vars_of_bexp b.* $i <$ *n_vs*)
(*STR ''Variable set bounded* (*guards*)*''*),
assert ($\forall$ $i <$ *n_vs. fst* (*bounds'* ! *i*) = *i*)
(*STR ''Bounds first index''*),
assert ($\forall a \in$ *set broadcast.* $a <$ *num_actions*)
(*STR ''Broadcast actions bounded''*),
assert ($\forall$ (_, _, *trans*, _) $\in$ *set automata.* $\forall$ (_, _, _, a, _, _, _) $\in$ *set trans.*
pred_act ($\lambda a.$ $a <$ *num_actions*) *a*)
(*STR ''Actions bounded* (*transitions*)*''*),
assert ($\forall$ (_, _, *trans*, _) $\in$ *set automata.* $\forall$ (_, _, g, _, _, r, _) $\in$ *set trans.*
($\forall c \in$ *set r.* $0 < c \wedge c \leq m$) $\wedge$
($\forall$ (c, x) $\in$ *collect_clock_pairs g.* $0 < c \wedge c \leq m \wedge x \in \mathbb{N}$))
(*STR ''Clock set bounded* (*transitions*)*''*),
assert ($\forall$ (_, _, _, *inv*) $\in$ *set automata.* $\forall$ (l, g) $\in$ *set inv.*
($\forall$ (c, x) $\in$ *collect_clock_pairs g.* $0 < c \wedge c \leq m \wedge x \in \mathbb{N}$))
(*STR ''Clock set bounded* (*invariants*)*''*),
assert ($\forall$ (_, _, *trans*, _) $\in$ *set automata.* $\forall$ (_, _, g, a, _, _, _) $\in$ *set trans.*
case a of In a $\Rightarrow$ $a \in$ *set broadcast* $\longrightarrow$ g = [] | _ $\Rightarrow$ *True*)
(*STR ''Broadcast receivers are unguarded''*),
assert ($\forall$ (_, U, _, _)$\in$*set automata.* U = [])
STR ''Urgency was removed''

]

lemma *check_precond1*:
is_result check_precond1
⟷ *Simple_Network_Impl_nat_urge broadcast bounds' automata m num_states num_actions*
unfolding *check_precond1_def Simple_Network_Impl_nat_def Simple_Network_Impl_nat_urge_def*
Simple_Network_Impl_nat_urge_axioms_def
by (*simp add*: *is_result_combine_Cons_iff is_result_assert_iff del*: *combine.simps*(2))

context
fixes *k* :: *nat list list list*
and L_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*)).
∀ (*x*, *m*) ∈ *collect_clock_pairs g*. *m* ≤ *int* (*k* ! *i* ! *l* ! *x*))
(*STR* ''*Ceiling invariants*''),
assert (∀ *i* < *n_ps*. ∀ (*l*, _, *g*, _) ∈ *set* ((*fst o snd o snd*) (*automata* ! *i*)).
(∀ (*x*, *m*) ∈ *collect_clock_pairs g*. *m* ≤ *int* (*k* ! *i* ! *l* ! *x*)))
(*STR* ''*Ceiling transitions*''),
assert (∀ *i* < *n_ps*. ∀ (*l*, *b*, *g*, *a*, *upd*, *r*, *l'*) ∈ *set* ((*fst o snd o snd*) (*automata* ! *i*)).
∀ *c* ∈ {0..<*m*+1} − *set r*. *k* ! *i* ! *l'* ! *c* ≤ *k* ! *i* ! *l* ! *c*)
(*STR* ''*Ceiling resets*''),
assert (*length k* = *n_ps*)
(*STR* ''*Ceiling length*''),
assert (∀ *i* < *n_ps*. *length* (*k* ! *i*) = *num_states i*)
(*STR* ''*Ceiling length automata*)''),
assert (∀ *xs* ∈ *set k*. ∀ *xxs* ∈ *set xs*. *length xxs* = *m* + 1)
(*STR* ''*Ceiling length clocks*''),
assert (∀ *i* < *n_ps*. ∀ *l* < *num_states i*. *k* ! *i* ! *l* ! 0 = 0)
(*STR* ''*Ceiling zero clock*''),
assert (∀ (_, _, _, *inv*) ∈ *set automata*. *distinct* (*map fst inv*))
(*STR* ''*Unambiguous invariants*''),
assert (*bounded bounds* (*map_of* s_0))
(*STR* ''*Initial state bounded*''),
assert (*length* L_0 = *n_ps*)
(*STR* ''*Length of initial state*''),
assert (∀ *i* < *n_ps*. L_0 ! *i* ∈ *fst* ‘ *set* ((*fst o snd o snd*) (*automata* ! *i*)))
(*STR* ''*Initial state has outgoing transitions*''),
assert (*vars_of_formula formula* ⊆ {0..<*n_vs*})
(*STR* ''*Variable set of formula*'')
]

lemma *check_precond2*:
is_result check_precond2 ⟷
Simple_Network_Impl_nat_ceiling_start_state_axioms broadcast bounds' automata m num_states
k L_0 s_0 *formula*
unfolding *check_precond2_def Simple_Network_Impl_nat_ceiling_start_state_axioms_def*

by (*simp add*: *is_result_combine_Cons_iff is_result_assert_iff del*: *combine.simps*(*2*))

end

definition
check_precond k L$_0$ *s*$_0$ *formula* ≡ *check_precond1* <|> *check_precond2 k L*$_0$ *s*$_0$ *formula*

lemma *check_precond*:
Simple_Network_Impl_nat_ceiling_start_state broadcast bounds' automata m num_states
num_actions k L$_0$ *s*$_0$ *formula* ⟷ *is_result* (*check_precond k L*$_0$ *s*$_0$ *formula*)
unfolding *check_precond_def is_result_combine_iff check_precond1 check_precond2*
Simple_Network_Impl_nat_ceiling_start_state_def
..

end

derive *show acconstraint act sexp formula*

fun *shows_exp* **and** *shows_bexp* **where**
shows_exp (*const c*) = *show c* |
shows_exp (*var v*) = *show v* |
shows_exp (*if_then_else b e1 e2*) =
shows_bexp b @ '' *?* '' @ *shows_exp e1* @ '' : '' @ *shows_exp e2* |
shows_exp (*binop _ e1 e2*) = ''*binop* '' @ *shows_exp e1* @ '' '' @ *shows_exp e2* |
shows_exp (*unop _ e*) = ''*unop* '' @ *shows_exp e* |
shows_bexp (*bexp.lt a b*) = *shows_exp a* @ '' < '' @ *shows_exp b* |
shows_bexp (*bexp.le a b*) = *shows_exp a* @ '' <= '' @ *shows_exp b* |
shows_bexp (*bexp.eq a b*) = *shows_exp a* @ '' = '' @ *shows_exp b* |
shows_bexp (*bexp.ge a b*) = *shows_exp a* @ '' >= '' @ *shows_exp b* |
shows_bexp (*bexp.gt a b*) = *shows_exp a* @ '' > '' @ *shows_exp b* |
shows_bexp bexp.true = ''*true*'' |
shows_bexp (*bexp.not b*) = ''! '' @ *shows_bexp b* |
shows_bexp (*bexp.and a b*) = *shows_bexp a* @ '' && '' @ *shows_bexp b* |
shows_bexp (*bexp.or a b*) = *shows_bexp a* @ '' || '' @ *shows_bexp b* |
shows_bexp (*bexp.imply a b*) = *shows_bexp a* @ '' −> '' @ *shows_bexp b*

instantiation *bexp* :: (*show*, *show*) *show*
begin

definition *shows_prec p* (*e* :: (_, _) *bexp*) *rest* = *shows_bexp e* @ *rest* **for** *p*

definition *shows_list* (*es*) *s* =
map shows_bexp es |> *intersperse* '', '' |> (λ*xs*. ''['' @ *concat xs* @ '']'' @ *s*)
instance
by *standard* (*simp_all add*: *shows_prec_bexp_def shows_list_bexp_def show_law_simps*)

end

instantiation *exp* :: (*show*, *show*) *show*
begin

definition *shows_prec p* (*e* :: (_, _) *exp*) *rest* = *shows_exp e* @ *rest* **for** *p*

definition *shows_list* (*es*) *s* =

```
  map shows_exp es |> intersperse '', '' |> (λxs. ''['' @ concat xs @ '']'' @ s)
instance
  by standard (simp_all add: shows_prec_exp_def shows_list_exp_def show_law_simps)

end


definition
  pad m s = replicate m CHR '' '' @ s

definition shows_rat r ≡ case r of fract.Rat s f b ⇒
  (if s then '''' else ''−'') @ show f @ (if b ≠ 0 then ''.'' @ show b else '''')

fun shows_json :: nat ⇒ JSON ⇒ string where
  shows_json n (Nat m) = show m |> pad n
| shows_json n (Rat r) = shows_rat r |> pad n
| shows_json n (JSON.Int r) = show r |> pad n
| shows_json n (Boolean b) = (if b then ''true'' else ''false'') |> pad n
| shows_json n Null = ''null'' |> pad n
| shows_json n (String s) = pad n ('''' @ s @ '''')
| shows_json n (JSON.Array xs) = (
  if xs = Nil then
    pad n ''[]''
  else
    pad n ''[↩''
    @ concat (map (shows_json (n + 2)) xs |> intersperse '',↩'')
    @ ''↩''
    @ pad n '']''
  )
| shows_json n (JSON.Object xs) = (
  if xs = Nil then
    pad n ''{}''
  else
    pad n ''{↩''
    @ concat (
      map (λ(k, v). pad (n + 2) ('''' @ k @ '''') @ '':↩'' @ shows_json (n + 4) v) xs
      |> intersperse '',↩''
    )
    @ ''↩''
    @ pad n ''}''
  )

instantiation JSON :: show
begin

definition shows_prec p (x :: JSON) rest = shows_json 0 x @ rest for p

definition shows_list jsons s =
  map (shows_json 0) jsons |> intersperse '', '' |> (λxs. ''['' @ concat xs @ '']'' @ s)
instance
  by standard (simp_all add: shows_prec_JSON_def shows_list_JSON_def show_law_simps)

end
```

definition *rename_network* **where**
rename_network broadcast bounds′ automata renum_acts renum_vars renum_clocks renum_states
≡
let
automata = map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states) automata;
broadcast = map renum_acts broadcast;
bounds′ = map (λ(a,b,c). (renum_vars a, b, c)) bounds′
in
(broadcast, automata, bounds′)

definition
show_clock inv_renum_clocks = show o inv_renum_clocks

definition
show_locs inv_renum_states = show o map_index inv_renum_states

definition
show_vars inv_renum_vars = show o map_index (λi v. show (inv_renum_vars i) @ ''='' @ show v)

definition
show_state inv_renum_states inv_renum_vars ≡ λ(L, vs).
let L = show_locs inv_renum_states L; vs = show_vars inv_renum_vars vs in
''<'' @ L @ ''>, <'' @ vs @ ''>''

definition *do_rename_mc* **where**
do_rename_mc f dc broadcast bounds′ automata k urge L₀ s₀ formula
m num_states num_actions renum_acts renum_vars renum_clocks renum_states
inv_renum_states inv_renum_vars inv_renum_clocks
≡
let
_ = println (STR ''Checking renaming'');
formula = (if dc then formula.EX (not sexp.true) else formula);
renaming_valid = Simple_Network_Rename_Formula_String_Defs.check_renaming
broadcast bounds′ renum_acts renum_vars renum_clocks renum_states automata urge formula 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 formula);
_ = 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 satisified:''
       in
         let _ = map println es in println STR '''');
    let _ = println (STR ''Running precond_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 precond_dc else precond_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 −
 have ∗:
   Simple_Network_Rename_Formula_String
        broadcast bounds renum_acts renum_vars renum_clocks renum_states automata urge
formula s₀ L₀
 = Simple_Network_Rename_Formula
     broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s₀ L₀ automata
formula

   unfolding
     Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
     Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
     Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
   using infinite_literal by auto
 define A where A ≡ N broadcast automata bounds
 define check where check ≡ A,(L₀, map_of s₀, λ_ . 0) ⊨ formula
 define A′ where A′ ≡ N
   (map renum_acts broadcast)
   (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
     (map (conv_urge urge) automata))
   (map (λ(a,p). (renum_vars a, p)) bounds)
 define check′ where check′ ≡
   A′,(map_index renum_states L₀, map_of (map (λ(x, v). (renum_vars x, v)) s₀), λ_ . 0) ⊨
   map_formula renum_states renum_vars id formula
 define preconds_sat where preconds_sat ≡
   Simple_Network_Impl_nat_ceiling_start_state
     (map renum_acts broadcast)
     (map (λ(a,p). (renum_vars a, p)) bounds)
     (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
       (map (conv_urge urge) automata))
     m num_states num_actions k
     (map_index renum_states L₀) (map (λ(x, v). (renum_vars x, v)) s₀)
     (map_formula renum_states renum_vars id formula)
 define renaming_valid where renaming_valid ≡
   Simple_Network_Rename_Formula
     broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s₀ L₀ automata
formula
```

```
  have [simp]: check ⟷ check'
    if renaming_valid
    using that unfolding check_def check'_def A_def A'_def renaming_valid_def
    by (rule Simple_Network_Rename_Formula.models_iff'[symmetric])
  have test[symmetric, simp]:
    Simple_Network_Language_Model_Checking.has_deadlock A (L₀, map_of s₀, λ_. 0)
  ⟷Simple_Network_Language_Model_Checking.has_deadlock A'
     (map_index renum_states L₀, map_of (map (λ(x, y). (renum_vars x, y)) s₀), λ_. 0)
   if renaming_valid
    using that unfolding check_def check'_def A_def A'_def renaming_valid_def
    unfolding Simple_Network_Rename_Formula_def
    by (elim conjE) (rule Simple_Network_Rename_Start.has_deadlock_iff'[symmetric])
  note [sep_heap_rules] =
    model_check[
    of _ _
    map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata)
    m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x, v))
s₀
    map_formula renum_states renum_vars id formula,
    folded A'_def preconds_sat_def renaming_valid_def, folded check'_def, simplified
    ]
  show ?thesis
    unfolding rename_mc_def do_rename_mc_def rename_network_def
    unfolding if_False
     unfolding Simple_Network_Rename_Formula_String_Defs.check_renaming[symmetric] *
Let_def
    unfolding
      A_def[symmetric] check_def[symmetric]
      preconds_sat_def[symmetric] renaming_valid_def[symmetric]
    by (sep_auto simp: model_checker.refine[symmetric] split: bool.splits)
qed

theorem deadlock_check_rename:
  <emp> rename_mc True broadcast bounds automata k urge L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
    <λ Sat   ⇒ ↑(  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 ⇒ ↑(¬ Simple_Network_Rename_Formula
       broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s₀ L₀ automata
        (formula.EX (not sexp.true)))
     | Preconds_Unsat ⇒ ↑(¬ Simple_Network_Impl_nat_ceiling_start_state
        (map renum_acts broadcast)
        (map (λ(a,p). (renum_vars a, p)) bounds)
        (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
          (map (conv_urge urge) automata))
        m num_states num_actions k
        (map_index renum_states L₀) (map (λ(x, v). (renum_vars x, v)) s₀)
        (formula.EX (not sexp.true)))
    >ₜ
proof −
  have ∗:
```

```
    Simple_Network_Rename_Formula_String
        broadcast bounds renum_acts renum_vars renum_clocks renum_states automata urge
        (formula.EX (not sexp.true)) s₀ L₀
  = Simple_Network_Rename_Formula
       broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s₀ L₀ automata
        (formula.EX (not sexp.true))

    unfolding
      Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
      Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
      Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
    using infinite_literal by auto
  define A where A ≡ N broadcast automata bounds
  define A' where A' ≡ N
    (map renum_acts broadcast)
    (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata))
    (map (λ(a,p). (renum_vars a, p)) bounds)
  define preconds_sat where preconds_sat ≡
    Simple_Network_Impl_nat_ceiling_start_state
      (map renum_acts broadcast)
      (map (λ(a,p). (renum_vars a, p)) bounds)
      (map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
        (map (conv_urge urge) automata))
      m num_states num_actions k
      (map_index renum_states L₀) (map (λ(x, v). (renum_vars x, v)) s₀)
      (formula.EX (not sexp.true))
  define renaming_valid where renaming_valid ≡
    Simple_Network_Rename_Formula
      broadcast bounds renum_acts renum_vars renum_clocks renum_states urge s₀ L₀ automata
      (formula.EX (not sexp.true))
  have test[symmetric, simp]:
    Simple_Network_Language_Model_Checking.has_deadlock A (L₀, map_of s₀, λ_. 0)
  ⟷Simple_Network_Language_Model_Checking.has_deadlock A'
     (map_index renum_states L₀, map_of (map (λ(x, y). (renum_vars x, y)) s₀), λ_. 0)
   if renaming_valid
   using that unfolding check_def A_def A'_def renaming_valid_def Simple_Network_Rename_Formula_def
    by (elim conjE) (rule Simple_Network_Rename_Start.has_deadlock_iff'[symmetric])
  note [sep_heap_rules] =
    deadlock_check[
    of _ _
    map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
      (map (conv_urge urge) automata)
    m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x, v))
s₀,
    folded preconds_sat_def A'_def renaming_valid_def,
    simplified
    ]
  show ?thesis
    unfolding rename_mc_def do_rename_mc_def  rename_network_def
    unfolding if_True
     unfolding Simple_Network_Rename_Formula_String_Defs.check_renaming[symmetric] *
Let_def
```

unfolding *A_def*[*symmetric*] *preconds_sat_def*[*symmetric*] *renaming_valid_def*[*symmetric*]
by (*sep_auto simp*: *deadlock_checker.refine*[*symmetric*] *split*: *bool.splits*)
qed

Code Setup for the Model Checker **lemmas** [*code*] =
reachability_checker_def
Alw_ev_checker_def
leadsto_checker_def
model_checker_def[*unfolded PR_CONST_def*]

lemmas [*code*] =
Prod_TA_Defs.n_ps_def
Simple_Network_Impl_Defs.n_vs_def
automaton_of_def
Simple_Network_Impl_nat_defs.pairs_by_action_impl_def
Simple_Network_Impl_nat_defs.all_actions_from_vec_def
Simple_Network_Impl_nat_defs.all_actions_by_state_def
Simple_Network_Impl_nat_defs.compute_upds_impl_def
Simple_Network_Impl_nat_defs.actions_by_state_def
Simple_Network_Impl_nat_defs.check_boundedi_def
Simple_Network_Impl_nat_defs.get_committed_def
Simple_Network_Impl_nat_defs.make_combs_def
Simple_Network_Impl_nat_defs.trans_map_def
Simple_Network_Impl_nat_defs.actions_by_state'_def
Simple_Network_Impl_nat_defs.bounds_map_def
Simple_Network_Impl_nat_defs.bin_actions_def
mk_updsi_def

lemma (**in** *Simple_Network_Impl_nat_defs*) *bounded_s_0_iff*:
bounded bounds (*map_of* s_0) $\longleftrightarrow$ *bounded* (*map_of bounds'*) (*map_of* s_0)
unfolding *bounds_def snd_conv* **..**

lemma *int_Nat_range_iff*:
(n :: *int*) $\in \mathbb{N} \longleftrightarrow n \geq 0$ **for** n
using *zero_le_imp_eq_int* **unfolding** *Nats_def* **by** *auto*

lemmas [*code*] =
Simple_Network_Impl_nat_ceiling_start_state_def
Simple_Network_Impl_nat_ceiling_start_state_axioms_def[
unfolded Simple_Network_Impl_nat_defs.bounded_s_0_iff]
Simple_Network_Impl_nat_def[*unfolded int_Nat_range_iff*]
Simple_Network_Impl_nat_urge_def
Simple_Network_Impl_nat_urge_axioms_def

lemmas [*code_unfold*] = *bounded_def dom_map_of_conv_image_fst*

export_code *Simple_Network_Impl_nat_ceiling_start_state_axioms*

export_code *precond_mc* **in** *SML* **module_name** *Test*

export_code *precond_dc* **checking** *SML*

Code Setup for Renaming **lemmas** [*code*] =

Simple_Network_Rename_Formula_String_def
Simple_Network_Impl.clk_set'_def
Simple_Network_Impl.clkp_set'_def

lemmas [*code*] =
Simple_Network_Rename_Defs.renum_automaton_def
Simple_Network_Rename_Defs.renum_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_s_0_iff]

lemma (**in** *Prod_TA_Defs*) *states_mem_iff*:
$L \in states \longleftrightarrow length\ L = n_ps \wedge (\forall\ i.\ i < n_ps \longrightarrow$
$(\exists\ (l, b, g, a, r, u, l') \in fst\ (snd\ (snd\ (fst\ (snd\ A)\ !\ i)))).\ L\ !\ i = l \vee L\ !\ i = l')$
unfolding *states_def trans_def N_def* **by** (*auto split*: *prod.split*)

lemmas [*code_unfold*] =
Prod_TA_Defs.states_mem_iff
Simple_Network_Impl.act_set_compute
Simple_Network_Impl_Defs.var_set_compute
Simple_Network_Impl_Defs.loc_set_compute
setcompr_eq_image
Simple_Network_Impl.length_automata_eq_n_ps[*symmetric*]

export_code *rename_mc* **in** *SML* **module_name** *Test*

Calculating the Clock Ceiling **context** *Simple_Network_Impl_nat_defs*
begin

definition *clkp_inv i l* ≡
$\bigcup g \in set\ (filter\ (\lambda(a, b).\ a = l)\ (snd\ (snd\ (snd\ (automata\ !\ i))))).\ collect_clock_pairs\ (snd\ g)$

definition *clkp_set'' i l* ≡
clkp_inv i l ∪ ($\bigcup$ $(l', b, g, _) \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$.
if $l' = l$ *then collect_clock_pairs g 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
$bound_g\ l \equiv$
$Max\ (\{0\} \cup \bigcup\ ((\lambda\ (x,\ d).\ if\ x = c\ then\ \{d\}\ else\ \{\})\ `\ clkp_set''\ q\ l))$

definition
$bound_inv\ l \equiv$
$Max\ (\{0\} \cup \bigcup\ ((\lambda\ (x,\ d).\ if\ x = c\ then\ \{d\}\ else\ \{\})\ `\ clkp_inv\ q\ l))$

definition
$bound\ l \equiv max\ (bound_g\ l)\ (bound_inv\ l)$

definition
$resets\ l \equiv$
$fold$
$(\lambda\ (l1,\ b,\ g,\ a,\ f,\ r,\ l')\ xs.\ if\ l1 \neq l \vee l' \in set\ xs \vee c \in set\ r\ then\ xs\ else\ (l' \#\ xs))$
$(fst\ (snd\ (snd\ (automata\ !\ q))))$
$[]$

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
$E\ l \equiv if\ l = n\ then\ [0..<n]\ else\ filter\ (\lambda\ l'.\ l \in set\ (E'\ l'))\ [0..<n]$

Weights already turned around.

definition
$W\ l\ l' \equiv if\ l = n\ then\ -\ bound\ l'\ else\ 0$

definition G **where**
$G \equiv (\!|\ gi_V = V,\ gi_E = E,\ gi_V0 = [n],\ \ldots = W\ |\!)$

definition
$local_ceiling_single \equiv$
let
$w = calc_shortest_scc_paths\ G\ n$
in
$map\ (\lambda\ x.\ case\ x\ of\ None \Rightarrow 0 \mid Some\ x \Rightarrow nat(-x))\ w$

end

```
definition
  local_ceiling ≡
    rev $
    fold
      (λ q xs.
        (λ x. rev x # xs) $
        fold
          (λ l xs.
            (λ x. (0 # rev x) # xs) $
            fold
              (λ c xs. local_ceiling_single q c ! l # xs)
              [1..<Suc m]
              []
          )
          [0..<n q]
          []
      )
      [0..<n_ps]
      []


end


lemmas [code] =
  Simple_Network_Impl_nat_defs.local_ceiling_def
  Simple_Network_Impl_nat_defs.local_ceiling_single_def
  Simple_Network_Impl_nat_defs.n_def
  Simple_Network_Impl_nat_defs.G_def
  Simple_Network_Impl_nat_defs.W_def
  Simple_Network_Impl_nat_defs.V_def
  Simple_Network_Impl_nat_defs.E'_def
  Simple_Network_Impl_nat_defs.E_def
  Simple_Network_Impl_nat_defs.resets_def
  Simple_Network_Impl_nat_defs.bound_def
  Simple_Network_Impl_nat_defs.bound_inv_def
  Simple_Network_Impl_nat_defs.bound_g_def
  Simple_Network_Impl_nat_defs.collect_resets_def
  Simple_Network_Impl_nat_defs.clkp_set''_def
  Simple_Network_Impl_nat_defs.clkp_inv_def

export_code Simple_Network_Impl_nat_defs.local_ceiling checking SML_imp

Calculating the Renaming  definition mem_assoc x = list_ex (λ(y, _). x = y)

definition mk_renaming str xs ≡
do {
  mapping ← fold_error
    (λx m.
      if mem_assoc x m then Error [STR ''Duplicate name: '' + str x] else (x,length m) # m |>
Result
    ) xs [];
```

```
  Result (let
    m = map_of mapping;
    f = (λx.
      case m x of
        None ⇒ let _ = println (STR ''Key error: '' + str x) in undefined
      | Some v ⇒ v);
    m = map_of (map prod.swap mapping);
    f_inv = (λx.
      case m x of
        None ⇒ let _ = println (STR ''Key error: '' + String.implode (show x)) in undefined
      | Some v ⇒ v)
  in (f, f_inv)
  )
}

definition
  extend_domain m d n ≡
    let
      (i, xs) = fold
        (λx (i, xs). if x ∈ set d then (i + 1, (x, i + 1) # xs) else (i, xs)) d (n, []);
      m' = map_of xs
    in
      (λx. if x ∈ set d then the (m' x) else m x)


lemma is_result_make_err:
  is_result (make_err m e) ⟷ False
  by (cases e) auto


lemma is_result_assert_msg_iff:
  is_result (assert_msg b m r) ⟷ is_result r ∧ b
  unfolding assert_msg_def by (simp add: is_result_make_err)

context Simple_Network_Impl
begin

definition action_set where
  action_set ≡
    (⋃(_, _, trans, _) ∈ set automata. ⋃(_, _, _, a, _, _, _) ∈ set trans. set_act a)
    ∪ set broadcast

definition loc_set' where
  loc_set' p = (⋃(l, _, _, _, _, _, l')∈set (fst (snd (snd (automata ! p))))). {l, l'}) for p

end


definition
  concat_str = String.implode o concat o map String.explode

Unsafe Glue Code  definition list_of_set' :: 'a set ⇒ 'a list where
  list_of_set' xs = undefined
```

```
definition list_of_set :: 'a set ⇒ 'a list where
  list_of_set xs = list_of_set' xs |> remdups

code_printing
  constant list_of_set' ⇀ (SML) (fn Set xs => xs) _
      and               (OCaml) (fun x -> match x with Set xs -> xs) _



definition
  mk_renaming' xs ≡ mk_renaming (String.implode o show) xs

definition make_renaming ≡ 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)) ←
      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' ← 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
      (λi 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 ⊆ 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 ≡ λdc 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 ⇒ 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) ⇒ 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 = (λi. ids_to_names i o inv_renum_states i);
    r ← 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 ≡ λdc ids_to_names (broadcast, automata, bounds) L₀ s₀ formula. do {
  r ← preproc_mc dc ids_to_names (broadcast, automata, bounds) L₀ s₀ formula;
  return (case r of
    Error es ⇒
      intersperse (STR ''↩'') es
      |> concat_str
      |> (λe. STR ''Error during preprocessing:↩'' + e)
      |> err
  | Result Renaming_Failed ⇒ STR ''Renaming failed'' |> err
  | Result Preconds_Unsat ⇒ STR ''Input invalid'' |> err
  | Result Unsat ⇒
      (if dc then STR ''Model has no deadlock!'' else STR ''Property is not satisfied!'') |> Result
  | Result Sat ⇒
      (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 ≡ case parse_all lx_ws parser s of
    Inl e ⇒ Error [e () ''Parser: '' |> String.implode]
  | Inr r ⇒ Result r

definition get_nat :: string ⇒ JSON ⇒ nat Error_List_Monad.result where
  get_nat s json ≡ case json of
    Object as ⇒ Error []
  | _ ⇒ Error [STR ''JSON Get: expected object'']
 for json

definition of_object :: JSON ⇒ (string ⇀ JSON) Error_List_Monad.result where
  of_object json ≡ case json of
```

```
    Object as ⇒ map_of as |> Result
  | _ ⇒ Error [STR ''json_to_map: expected object'']
 for json

definition get where
  get m x ≡ case m x of
    None ⇒ Error [STR ''(Get) key not found: '' + String.implode (show x)]
  | Some a ⇒ Result a

definition
  get_default def m x ≡ case m x of None ⇒ def | Some a ⇒ a

definition default where
  default def x ≡ case x of Result s ⇒ s | Error e ⇒ def

definition of_array where
  of_array json ≡ case json of
    JSON.Array s ⇒ Result s
  | _ ⇒ Error [STR ''of_array: expected sequence'']
 for json

definition of_string where
  of_string json ≡ case json of
    JSON.String s ⇒ Result (String.implode s)
  | _ ⇒ Error [STR ''of_array: expected sequence'']
 for json

definition of_nat where
  of_nat json ≡ case json of
    JSON.Nat n ⇒ Result n
  | _ ⇒ Error [STR ''of_nat: expected natural number'']
 for json

definition of_int where
  of_int json ≡ case json of
    JSON.Int n ⇒ Result n
  | _ ⇒ Error [STR ''of_int: expected integral number'']
 for json

definition [consuming]:
  lx_underscore = exactly ''_'' with (λ_. CHR ''_'')

definition [consuming]:
  lx_hyphen = exactly ''-'' with (λ_. CHR ''-'')

definition [consuming]:
  ta_var_ident ≡
    (lx_alphanum ‖ lx_underscore)
    -- Parser_Combinator.repeat (lx_alphanum ‖ lx_underscore ‖ lx_hyphen)
    with uncurry (#)


definition [consuming]:
  parse_bound ≡ ta_var_ident --
```

```
  exactly ''['' *-- lx_int -- exactly '':'' *-- lx_int --* exactly '']''

definition parse_bounds ≡ parse_list' (lx_ws *-- parse_bound with (λ(s,p). (String.implode s, p)))

lemma
  parse parse_bounds (STR ''id[-1:2], id[-1:0]'')
 = Result [(STR ''id'', -1, 2), (STR ''id'', -1, 0)]
  by eval

lemma parse parse_bounds (STR '''') = Result []
  by eval

definition [consuming]:
  scan_var = ta_var_ident

abbreviation seq_ignore_left_ws (infixr **-- 60)
  where p **-- q ≡ 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 ≡ scan_acconstraint || scan_loc

abbreviation scan_parens' ≡ scan_parens ''('' '')''

definition [consuming]: scan_infix_pair a b s ≡ a --- exactly s **-- token b

lemma [fundef_cong]:
  assumes ⋀l2. ll_fuel l2 ≤ ll_fuel l' ⟹ A l2 = A' l2
```

assumes $\bigwedge l2.\ ll_fuel\ l2 + (if\ length\ s > 0\ then\ 1\ else\ 0) \leq ll_fuel\ l' \Longrightarrow B\ l2 = B'\ l2$
assumes $s = s'\ l{=}l'$
shows *scan_infix_pair A B s l = scan_infix_pair A' B' s' l'*
using *assms* **unfolding** *scan_infix_pair_def gen_token_def*
by (*cases s*; *intro Parser_Combinator.bind_cong repeat_cong assms*) *auto*

lemma [*fundef_cong*]:
assumes $\bigwedge l2.\ ll_fuel\ l2 < ll_fuel\ l \Longrightarrow A\ l2 = A'\ l2\ l = l'$
shows *scan_parens' A l = scan_parens' A' l'*
using *assms* **unfolding** *scan_parens_def gen_token_def*
by (*intro Parser_Combinator.bind_cong repeat_cong assms*) *auto*

lemma *token_cong*[*fundef_cong*]:
assumes $\bigwedge l2.\ ll_fuel\ l2 \leq ll_fuel\ l \Longrightarrow A\ l2 = A'\ l2\ l = l'$
shows *token A l = token A' l'*
using *assms* **unfolding** *scan_parens_def gen_token_def*
by (*intro Parser_Combinator.bind_cong repeat_cong assms*) *auto*

lemma *is_cparser_scan_parens'*[*parser_rules*]:
is_cparser (*scan_parens' a*)
unfolding *scan_parens_def* **by** *simp*

fun *aexp* **and** *mexp* **and** *scan_exp* **and** *scan_7* **and** *scan_6* **and** *scan_0* **where** — slow: 90s
aexp ::=
token lx_int with exp.const ‖ *token scan_var with exp.var o String.implode* ‖
scan_parens' (*scan_exp −−− exactly ''?'' **−− scan_7 −−− exactly '':'' **−− scan_exp*)
with (λ (*e1*, *b*, *e2*). *exp.if_then_else b e1 e2*) ‖
*tk_lparen **−− scan_exp −−** tk_rparen*
| *mexp* ::= *chainL1 aexp* (*multiplicative_op with* (λ*f a b*. *exp.binop f a b*))
| *scan_exp* ::= *chainL1 mexp* (*additive_op with* (λ*f a b*. *exp.binop f a b*))
| *scan_7* ::=
scan_infix_pair scan_6 scan_7 ''−>'' with uncurry bexp.imply ‖
scan_infix_pair scan_6 scan_7 ''||'' with uncurry bexp.or ‖
scan_6 |
scan_6 ::=
scan_infix_pair scan_0 scan_6 ''&&'' with uncurry bexp.and ‖
scan_0 |
scan_0 ::=
(*exactly ''~'' ‖ exactly ''!''*) **−− scan_parens' scan_7 with bexp.not* ‖
token (*exactly ''true''*) *with* (λ_. *bexp.true*) ‖
scan_infix_pair aexp aexp ''<='' with uncurry bexp.le ‖
scan_infix_pair aexp aexp ''<'' with uncurry bexp.lt ‖
scan_infix_pair aexp aexp ''=='' with uncurry bexp.eq ‖
scan_infix_pair aexp aexp ''>'' with uncurry bexp.gt ‖
scan_infix_pair aexp aexp ''>='' with uncurry bexp.ge ‖
scan_parens' scan_7

context
fixes *elem* :: (*char*, *'bexp*) *parser*
and *Imply Or And* :: *'bexp* ⇒ *'bexp* ⇒ *'bexp*
and *Not* :: *'bexp* ⇒ *'bexp*
begin

fun *scan_7'* **and** *scan_6'* **and** *scan_0'* **where**

```
  scan_7' ::=
    scan_infix_pair scan_6' scan_7' ''->'' with uncurry Imply ||
    scan_infix_pair scan_6' scan_7' ''||'' with uncurry Or ||
    scan_6' |
  scan_6' ::=
    scan_infix_pair scan_0' scan_6' ''&&'' with uncurry And ||
    scan_0' |
  scan_0' ::=
    (exactly ''~'' || exactly ''!'') **-- scan_parens' scan_7' with Not ||
    elem ||
    scan_parens' scan_7'

context
  assumes [parser_rules]: is_cparser elem
begin

lemma [parser_rules]:
  is_cparser scan_0'
  by (simp add: scan_0'.simps[abs_def])

lemma [parser_rules]:
  is_cparser scan_6'
  by (subst scan_6'.simps[abs_def]) simp

end
end

abbreviation scan_bexp ≡ scan_7' scan_bexp_elem sexp.imply sexp.or sexp.and sexp.not

lemma [parser_rules]:
  is_cparser scan_bexp
  by (subst scan_7'.simps[abs_def]) simp

lemma parse scan_bexp (STR ''a < 3 && b>=2 || ~ (c <= 4)'')
= Result (sexp.or (and (lt STR ''a'' 3) (ge STR ''b'' 2)) (not (sexp.le STR ''c'' 4)))
  by eval

definition [consuming]: scan_prefix p head = exactly head **-- p for p

definition [consuming]: scan_formula ≡
  scan_prefix scan_bexp ''E<>'' with formula.EX ||
  scan_prefix scan_bexp ''E[]'' with EG ||
  scan_prefix scan_bexp ''A<>'' with AX ||
  scan_prefix scan_bexp ''A[]'' with AG ||
  scan_infix_pair scan_bexp scan_bexp ''-->'' with uncurry Leadsto


lemma is_cparser_token[parser_rules]:
  is_cparser (token a) if is_cparser a
  using that unfolding gen_token_def by simp

definition [consuming]: scan_action ≡
  (scan_var --* token (exactly ''?'')) with In o String.implode ||
  (scan_var --* token (exactly ''!'')) with Out o String.implode ||
```

```
  scan_var with Sil o String.implode

abbreviation orelse (infix orelse 58) where
  a orelse b ≡ 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_bexp :: (String.literal, String.literal, String.literal, int) sexp ⇒ _ where
  sexp_to_bexp (lt a (b :: int)) = bexp.lt (exp.var a) (exp.const b) |> Result |
  sexp_to_bexp (le a b) = bexp.le (exp.var a) (exp.const b) |> Result |
  sexp_to_bexp (eq a b) = bexp.eq (exp.var a) (exp.const b) |> Result |
  sexp_to_bexp (ge a b) = bexp.ge (exp.var a) (exp.const b) |> Result |
  sexp_to_bexp (gt a b) = bexp.gt (exp.var a) (exp.const b) |> Result |
  sexp_to_bexp (and a b) =
    do {a ← sexp_to_bexp a; b ← sexp_to_bexp b; bexp.and a b |> Result} |
  sexp_to_bexp (sexp.or a b) =
    do {a ← sexp_to_bexp a; b ← sexp_to_bexp b; bexp.or a b |> Result} |
  sexp_to_bexp (imply a b) =
    do {a ← sexp_to_bexp a; b ← sexp_to_bexp b; bexp.imply a b |> Result} |
  sexp_to_bexp 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 φ) = rename_locs_sexp f φ ≫= Result o formula.EX |
  rename_locs_formula f (EG φ) = rename_locs_sexp f φ ≫= Result o EG |
  rename_locs_formula f (AX φ) = rename_locs_sexp f φ ≫= Result o AX |
  rename_locs_formula f (AG φ) = rename_locs_sexp f φ ≫= Result o AG |
  rename_locs_formula f (Leadsto φ ψ) =
    do {φ ← rename_locs_sexp f φ; ψ ← rename_locs_sexp f ψ; Leadsto φ ψ |> Result}

definition convert :: JSON ⇒
  ((nat ⇒ nat ⇒ String.literal) × (String.literal ⇒ nat) × String.literal list ×
    (nat list × nat list ×
     (String.literal act, nat, String.literal, int, String.literal, int) transition list
      × (nat × (String.literal, int) cconstraint) list) list ×
   (String.literal × int × int) list ×
   (nat, nat, String.literal, int) formula × nat list × (String.literal × int) list
  ) Error_List_Monad.result where
  convert json ≡ 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
      (λ_. 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 (lx_ws *−− 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 (λa. get a ''name'' ≫= of_string) automata;
    assert (distinct process_names) (STR ''Process names are ambiguous'');
     assert (locs_of_formula formula ⊆ set process_names) (STR ''Unknown process name in
formula'');
    let process_names_to_index = List_Index.index process_names;
    init_locs ← combine_map
      (λa. 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 ‹
  fun assert comp exp =
    if comp = exp then () else error (Assertion failed! expected: ^ exp ^ but got: ^ comp)
  fun test dc file =
  let
    val dir = Path.append @{master_dir} @{path ../benchmarks/}
    val file = Path.explode file |> Path.append dir
    val s = File.read file
  in
    @{code parse_convert_run_test} dc s end
›
```

Now we can run Munta on a few examples:

```
ML_val ‹assert (test false HDDI_02.muntax ()) Property is not satisfied!›
ML_val ‹assert (test true HDDI_02.muntax ()) Model has no deadlock!›

ML_val ‹assert (test false simple.muntax ()) Property is satisfied!›
ML_val ‹assert (test true simple.muntax ()) Model has no deadlock!›

ML_val ‹assert (test false light_switch.muntax ()) Property is satisfied!›
ML_val ‹assert (test true light_switch.muntax ()) Model has no deadlock!›

ML_val ‹assert (test false PM_test.muntax ()) Property is not satisfied!›
ML_val ‹assert (test true PM_test.muntax ()) Model has a deadlock!›

ML_val ‹assert (test false bridge.muntax ()) Property is satisfied!›
ML_val ‹assert (test true bridge.muntax ()) Model has no deadlock!›

ML_val ‹assert (test false fischer.muntax ()) Property is satisfied!›
ML_val ‹assert (test true fischer.muntax ()) Model has no deadlock!›
```

ML_val ‹*assert* (*test false PM_all_4.muntax* ()) *Property is not satisfied*!›
ML_val ‹*assert* (*test true PM_all_4.muntax* ()) *Model has no deadlock*!›

end

13 Build and Test Exported Program With MLton

theory *Munta_Compile_MLton*
imports *Munta_Model_Checker.Simple_Network_Language_Export_Code*
begin

— Produces a command for checking a single benchmark, e.g.: `./check_benchmark.sh 568 "Property is not satisfied" ./munta -m benchmarks/PM_all_5.muntax`
ML ‹
fun mk_benchmark_check_gen munta mk_prop name num_states satisfied =
let
val prop = mk_prop satisfied
val benchmark = name @ .muntax
in
— this line checks that number of states and property satisfaction are correct
./check_benchmark.sh @ string_of_int num_states @ @ prop
— this line runs Munta on the actual benchmark
@ @ munta @ @ benchmark
end
val mk_benchmark_check =
mk_benchmark_check_gen ./munta −m
(fn satisfied =>
if satisfied then
verbatim ‹*Property is satisfied*!›
else
verbatim ‹*Property is not satisfied*!›)
val mk_benchmark_check_dc =
mk_benchmark_check_gen ./munta −dc −m
(fn satisfied =>
if satisfied then
verbatim ‹*Model has a deadlock*!›
else
verbatim ‹*Model has no deadlock*!›)
val it = mk_benchmark_check PM_all_5 568 false
val it1 = mk_benchmark_check_dc hddi_08 466 false
›

Here is how to compile Munta with MLton and then run some benchmarks:

compile_generated_files *code/Simple_Model_Checker.ML* (**in** *Simple_Network_Language_Export_Code*)
external_files
‹*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›
  ‹csma_05.muntax›
  ‹csma_06.muntax›
  ‹fischer.muntax›
  ‹fischer_05.muntax›
  ‹hddi_08.muntax›
  ‹light_switch.muntax› (in ../benchmarks)
export_files ‹munta› (exe)
where ‹fn dir =>
  let
    val exec = Generated_Files.execute dir
    val _ =
      exec ‹Preparation›
        mv code/Simple_Model_Checker.ML Simple_Model_Checker.ML
    val _ =
      exec ‹Compilation›
        (////verbatim//$ISABELLE_MLTON//ISABELLE_MLTON_OPTIONS///
         verbatim ‹$ISABELLE_MLTON › ^
          — these additional settings have been copied from the AFP entry PAC_Checker
          — they bring down benchmarks/PM_all_6.muntax from 1000 s to 180 s on an M1 Mac
           −const 'MLton.safe false' −verbose 1 −inline 700 −cc−opt −O3 ^
          — this one from PAC_Checker does not work on ARM64 though
          //codegen//native///
          — 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//1624//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.