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

Introduction

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

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

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

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

Usage Theory `Simple_Network_Language_Export_Code` (page 261) describes how to obtain executable code for and run MUNTA from within Isabelle.

This entry also provides a build setup for the MUNTA executable using the MLton SML compiler. Build and run instructions are given in theory `Munta_Compile_MLton` (on page 261). A number of example model files can be found in the `benchmarks` directory.

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

OCaml Code Generation While the default backend targets SML/MLton, MUNTA in principle also supports code generation to OCaml (c.f. theory `Simple_Network_Language_Export_Code` on page 261). Among its use cases is the browser-based version of the GUI, which is compiled from the OCaml code. For instructions on compiling with the `dune` build system, see the `ocaml` branch: https://github.com/wimmers/munta/tree/ocaml.

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

unbundle *no_library_syntax*

1 Networks of Timed Automata

1.1 Syntax and Operational Semantics

Input, output and internal transitions

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

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

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

inductive *step_n* ::
('a, 'b, 'c, 't, 's) nta ⇒ '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** $N \vdash_N \langle l', Z'\rangle \rightarrow* \langle l'', Z''\rangle$ $N \vdash_N \langle l, Z\rangle \rightarrow_b \langle l', Z'\rangle$
⟨*proof*⟩

lemma *step_n_t_I*:
$[\![\forall\ p \in \{..<length\ N\}.\ u \vdash inv\_of\ (N!p)\ (L!p)$; $\forall\ p \in \{..<length\ N\}.\ u \oplus d \vdash inv\_of\ (N!p)$ $(L!p)$;
$d \geq 0$
$]\!] \Longrightarrow N \vdash_N \langle L, u\rangle \rightarrow_{Del} \langle L, u \oplus d\rangle$
⟨*proof*⟩

1.2 Product Automaton

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

lemma *guard_concat*:
assumes $\forall\ g \in set\ xs.\ u \vdash g$
shows $u \vdash concat\ xs$
⟨*proof*⟩

lemma *guard_append*:
assumes $u \vdash g1$ $u \vdash g2$
shows $u \vdash g1\ @\ g2$
⟨*proof*⟩

lemma *concat_guard*:
assumes $u \vdash concat\ xs$ $g \in set\ xs$
shows $u \vdash g$
⟨*proof*⟩

lemma *guard_consE*:
assumes $u \vdash ac\ \#\ cc$
obtains $u \vdash_a ac$ $u \vdash cc$
⟨*proof*⟩

lemma *guard_consI*:
assumes $u \vdash_a ac$ $u \vdash cc$
shows $u \vdash ac\ \#\ cc$
⟨*proof*⟩

lemma *collect_clock_pairs_append_cases*:
assumes $x \in collect\_clock\_pairs\ (g1\ @\ g2)$
shows $x \in collect\_clock\_pairs\ g1 \vee x \in collect\_clock\_pairs\ g2$
⟨*proof*⟩

lemma *list_update_nth_split*:
assumes $j < length\ xs$
shows $P\ (xs[i := x]\ !\ j) = ((i = j \longrightarrow P\ x) \wedge (i \neq j \longrightarrow P\ (xs\ !\ j)))$
⟨*proof*⟩

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 $l\ !\ q \in state\_set\ (T\ !\ q)$
⟨*proof*⟩

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*:
$Timed\_Automata.collect\_clki\ product\_invariant = \bigcup\ (Timed\_Automata.collect\_clki\ `\ set\ I)$
⟨*proof*⟩

lemma *states_length*:
assumes $L \in states$
shows $length\ L = length\ N$
⟨*proof*⟩

lemma *collect_clkt_product_trans_subs*:
$Timed\_Automata.collect\_clkt\ product\_trans \subseteq \bigcup\ (Timed\_Automata.collect\_clkt\ `\ set\ T)$
⟨*proof*⟩

lemma *collect_clkvt_product_trans_subs*:
$collect\_clkvt\ product\_trans \subseteq \bigcup\ (collect\_clkvt\ `\ set\ T)$
⟨*proof*⟩

lemma *statesI_aux*:
fixes L
assumes L: $L \in states$
assumes

$p < length\ T$
$(l, g, a, r, l') \in T\ !\ p$
shows $(L[p := l]) \in states$
⟨*proof*⟩

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

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

lemma *collect_clkt_product_trans_sups*:
$Timed\_Automata.collect\_clkt\ product\_trans \supseteq \bigcup\ (Timed\_Automata.collect\_clkt\ `\ set\ T)$
⟨*proof*⟩

lemma *collect_clkvt_product_trans_sups*:
$collect\_clkvt\ product\_trans \supseteq \bigcup\ (collect\_clkvt\ `\ set\ T)$
⟨*proof*⟩

lemma *collect_clkt_product_trans*:
$Timed\_Automata.collect\_clkt\ product\_trans = \bigcup\ (Timed\_Automata.collect\_clkt\ `\ set\ T)$
⟨*proof*⟩

lemma *collect_clkvt_product_trans*:
$collect\_clkvt\ product\_trans = \bigcup\ (collect\_clkvt\ `\ set\ T)$
⟨*proof*⟩

end

context
assumes *finite_trans*: $\forall\ A \in set\ N.\ finite\ (trans\_of\ A)$
begin

lemma *finite_states*:
finite states
⟨*proof*⟩

lemma *finite_product_trans_i*:
finite product_trans_i
⟨*proof*⟩

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*})
⟨*proof*⟩

lemma *finite_product_trans_s*:
finite product_trans_s
⟨*proof*⟩

lemma *finite_trans_of_product*:
shows *finite* (*trans_of product_ta*)
⟨*proof*⟩

end

lemma *inv_of_product*[*simp*]:
inv_of product_ta = *product_invariant*
⟨*proof*⟩

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*])
⟨*proof*⟩

lemma *finite_invariant_of_product*[*folded inv_of_product*]:
assumes ∀ *A* ∈ *set N*. *finite* (*range* (*inv_of A*))
shows *finite* (*range product_invariant*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *states_step*:
L' ∈ *states* **if** *N* ⊢$_N$ ⟨*L*, *u*⟩ →$_a$ ⟨*L'*, *u'*⟩ *L* ∈ *states*
⟨*proof*⟩

lemma *states_steps*:
L' ∈ *states* **if** *N* ⊢$_N$ ⟨*L*, *u*⟩ →* ⟨*L'*, *u'*⟩ *L* ∈ *states*
⟨*proof*⟩

end

lemma *steps_altI*:

$A \vdash \langle l,\ Z\rangle \rightarrow* \langle l'',\ Z''\rangle$ **if**
$A \vdash \langle l,\ Z\rangle \rightarrow* \langle l',\ Z'\rangle$ $A \vdash \langle l',\ Z'\rangle \rightarrow \langle l'',\ Z''\rangle$
⟨*proof*⟩

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

lemma
len[*simp*]: *length* L = *length* N
⟨*proof*⟩

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

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

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

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

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

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

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

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

⟨*proof*⟩

lemma *states_product_step*[*intro*]:
$L' \in$ *states* **if** *product_ta* $\vdash \langle L, u\rangle \rightarrow \langle L', u'\rangle$
⟨*proof*⟩

lemma *product_steps_sound*:
$N \vdash_N \langle L, u\rangle \rightarrow* \langle L', u'\rangle$ **if** *product_ta* $\vdash \langle L, u\rangle \rightarrow* \langle L', u'\rangle$
⟨*proof*⟩

lemma *product_steps_complete*:
product_ta $\vdash \langle L, u\rangle \rightarrow* \langle L', u'\rangle$ **if** $N \vdash_N \langle L, u\rangle \rightarrow* \langle L', u'\rangle$
⟨*proof*⟩

lemma *product_correct*:
product_ta $\vdash \langle L, u\rangle \rightarrow* \langle L', u'\rangle \longleftrightarrow N \vdash_N \langle L, u\rangle \rightarrow* \langle L', u'\rangle$
⟨*proof*⟩

end

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

unbundle *no_library_syntax*

2 Networks of Timed Automata With Discrete State

2.1 Syntax and Operational Semantics

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

type_synonym
('*a*, '*c*, '*time*, '*s*, '*st*) *transition* =
'*s* ∗ ('*st* ⇒ ('*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*) ⊢ ⟨*L*, *s*, *u*⟩ →*Act a* ⟨*L*′, *s*′, *u*′⟩
if (*l*, *g*, (*Sil a*, *c*, *m*), *f*, *l*′) ∈ *fst* (*N*!*p*)
u ⊢ *g s* ∀ *p* ∈ {..<*length N*}. *u*′ ⊢ *snd* (*N*!*p*) (*L*′!*p*)
r = *f s*
L!*p* = *l* *p* < *length L* *L*′ = *L*[*p* := *l*′] *u*′ = [*r*→*0*]*u*
c s ∀ *p* < *length I*. (*I* ! *p*) (*L*′ ! *p*) *s*′ *Some s*′ = *m s*
length N = *length I* |
step_sn_s:
(*N*, *I*) ⊢ ⟨*L*, *s*, *u*⟩ →*Syn a* ⟨*L*′, *s*′, *u*′⟩
if (*l1*, *g1*, (*In a*, *ci*, *mi*), *f1*, *l1*′) ∈ *fst* (*N*!*p*)
(*l2*, *g2*, (*Out a*, *co*, *mo*), *f2*, *l2*′) ∈ *fst* (*N*!*q*) *u* ⊢ *g1 s* *u* ⊢ *g2 s*
∀ *p* ∈ {..<*length N*}. *u*′ ⊢ *snd* (*N*!*p*) (*L*′!*p*)
r1 = *f1 s* *r2* = *f2 s*
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*
ci s *co s* ∀ *p* < *length I*. (*I* ! *p*) (*L*′ ! *p*) *s*′
Some so = *mo s* *Some s*′ = *mi so* *length N* = *length I*

inductive_cases[*elim*!]: *N* ⊢ ⟨*L*, *s*, *u*⟩ →*Syn a* ⟨*L*′, *s*′, *u*′⟩

inductive *steps_sn* ::
(′*a*, ′*c*, ′*t*, ′*s*, ′*st*) *snta* ⇒ ′*s list* ⇒ ′*st* ⇒ (′*c*, (′*t*::*time*)) *cval*
⇒ ′*s list* ⇒ ′*st* ⇒ (′*c*, ′*t*) *cval* ⇒ *bool*
(_ ⊢ ⟨_, _, _⟩ →* ⟨_, _, _⟩ [*61*, *61*, *61*,*61*,*61*] *61*)
where
refl: *N* ⊢ ⟨*L*, *s*, *u*⟩ →* ⟨*L*, *s*, *u*⟩ |
step: *N* ⊢ ⟨*L*, *s*, *u*⟩ →* ⟨*L*′, *s*′, *u*′⟩ ⟹ *N* ⊢ ⟨*L*′, *s*′, *u*′⟩ →*l* ⟨*L*′′, *s*′′, *u*′′⟩
⟹ *N* ⊢ ⟨*L*, *s*, *u*⟩ →* ⟨*L*′′, *s*′′, *u*′′⟩

declare *steps_sn.intros*[*intro*]

lemma *stepI2*:
N ⊢ ⟨*l*, *s*, *u*⟩ →* ⟨*l*′′, *s*′′, *u*′′⟩ **if**
N ⊢ ⟨*l*′, *s*′, *u*′⟩ →* ⟨*l*′′, *s*′′, *u*′′⟩ *N* ⊢ ⟨*l*, *s*, *u*⟩ →*a* ⟨*l*′, *s*′, *u*′⟩
⟨*proof*⟩

abbreviation *state_set* :: (′*a*, ′*c*, ′*t*, ′*s*, ′*st*) *transition set* ⇒ ′*s set* **where**
state_set T ≡ *fst* ‘ *T* ∪ (*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
$T\_s\ p\ s = \{(l,\ g\ s,\ a,\ f\ s,\ l')\ |\ l\ g\ a\ f\ l'.\ (l,\ g,\ a,\ f,\ l') \in fst\ (fst\ A\ !\ p)\}$

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

abbreviation $P \equiv snd\ A$

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

abbreviation $product\ s \equiv Product\_TA\_Defs.product\_ta\ (N\_s\ s)$

abbreviation $T'\ s \equiv trans\_of\ (product\ s)$
abbreviation $I'\ s \equiv inv\_of\ (product\ s)$

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

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

definition
$prod\_trans \equiv prod\_trans\_i \cup prod\_trans\_s$

definition
$prod\_invariant \equiv \lambda\ (L,\ s).\ I'\ s\ L$

definition $prod\_ta :: ('a,\ 'c,\ 't,\ 's\ list \times\ 'st)\ ta$ **where**
$prod\_ta \equiv (prod\_trans,\ prod\_invariant)$

lemma *prod_ta_cases*:
assumes $prod\_ta \vdash L \longrightarrow^{g,a,r} L'$
shows $(L,\ g,\ a,\ r,\ L') \in prod\_trans\_i \vee (L,\ g,\ a,\ r,\ L') \in prod\_trans\_s$
⟨*proof*⟩

lemma *inv_of_simp*:
$inv\_of\ prod\_ta\ (L,\ s) = I'\ s\ L$
⟨*proof*⟩

lemma *I'_simp*:
$I'\ s\ L = I'\ s'\ L$
⟨*proof*⟩

lemma *collect_clki_prod_invariant*:
Timed_Automata.collect_clki prod_invariant = *Timed_Automata.collect_clki* (*I'* *s*)
⟨*proof*⟩

lemma *collect_clki_prod_invariant'*:
Timed_Automata.collect_clki prod_invariant
⊆ ⋃ {*Timed_Automata.collect_clki* (*snd* (*fst A* ! *p*)) | *p*. *p* < *length* (*fst A*)}
⟨*proof*⟩

lemma *collect_clkt_prod_trans_subs*:
Timed_Automata.collect_clkt prod_trans ⊆ *Timed_Automata.collect_clkt* (⋃ (*T'* ‘ *UNIV*))
⟨*proof*⟩

lemma *collect_clkvt_prod_trans_subs*:
collect_clkvt prod_trans ⊆ *collect_clkvt* (⋃ (*T'* ‘ *UNIV*))
⟨*proof*⟩

lemma *T_simp*:
T ! *q* = *trans_of* (*N* ! *q*) **if** *q* < *length N*
⟨*proof*⟩

abbreviation *N* ≡ *fst A*

context
fixes *Q*
assumes *finite_state*:
∀ *l*. ∀ *q* < *p*. (*P* ! *q*) *l s* ⟶ *Q s*
finite {*s*. *Q s*}
and *finite_trans*: ∀ *A* ∈ *set N*. *finite* (*fst A*)
and *p_gt_0*: *p* > *0*
begin

lemma *finite_state'*:
finite {*s*. ∀ *q*<*p*. (*P* ! *q*) (*L* ! *q*) *s*} (**is** *finite ?S*)
⟨*proof*⟩

lemma *finite_trans'*:
∀ *A*∈*set* (*N_s s*). *finite* (*trans_of A*)
⟨*proof*⟩

lemma *finite_states*:
finite (*Product_TA_Defs.states* (*N_s s*))
⟨*proof*⟩

lemma
finite (*T'* *s*)
⟨*proof*⟩

lemma *finite_product_1*:
finite (*T'* *s*)

⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *Some_finite*:
finite {*x*. *Some x* = *y*}
⟨*proof*⟩

lemma *finite_prod_trans*:
finite prod_trans **if** *p* > *0*
⟨*proof*⟩

end

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

lemma *N_s_length*:
length (*N_s s*) = *p*
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*⟩
⟨*proof*⟩

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'*⟩
⟨*proof*⟩

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'*⟩
⟨*proof*⟩

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'*⟩

⟨*proof*⟩

lemma *trans_of_N_s_1*:
(*fst* ‘ *trans_of* (*N_s s* ! *q*)) = *fst* ‘ *fst* (*N* ! *q*) **if** *q* < *p*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma
fst ‘ *trans_of* (*N_s s* ! *q*) = *fst* ‘ *trans_of* (*N_s s'* ! *q*) **if** *q* < *p*
⟨*proof*⟩

lemma *states'_simp*:
states' s = *states' s'*
⟨*proof*⟩

lemma *states_step*:
L' ∈ *states' s* **if** *A* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L'*, *s'*, *u'*⟩ *L* ∈ *states' s*
⟨*proof*⟩

lemma *states_steps*:
L' ∈ *states' s'* **if** *A* ⊢ ⟨*L*, *s*, *u*⟩ →∗ ⟨*L'*, *s'*, *u'*⟩ *L* ∈ *states' s*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *inv_prod_simp*:
inv_of prod_ta (*l*, *s'*) = *Product_TA_Defs.product_invariant* (*N_s s'*) *l* **if** *length l* = *p*
⟨*proof*⟩

lemma *inv_of_N_simp*:
$map\ inv\_of\ (N\_s\ s')\ !\ q = I\ !\ q$ **if** $q < p$
⟨*proof*⟩

lemma *product_inv_prod_simp*:
$inv\_of\ prod\_ta\ (l,\ s') = I'\ s\ l$ **if** $length\ l = p$
⟨*proof*⟩

lemma *product_inv_prod*[*intro*]:
$u \vdash inv\_of\ prod\_ta\ (l,\ s')$ **if** $u \vdash inv\_of\ product\_ta\ l\ length\ l = p$
⟨*proof*⟩

lemma *A_simp*[*simp*]:
$N' = N\ P' = P$ **if** $A = (N',\ P')$
⟨*proof*⟩

lemma *length_L*[*intro*]:
$length\ L = p$
⟨*proof*⟩

lemma *prod_complete_delay*:
assumes *step*: $A \vdash \langle L,\ s,\ u\rangle \rightarrow_{Del} \langle L',\ s',\ u'\rangle$
obtains d **where** $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow^{d} \langle(L',\ s'),\ u'\rangle$
⟨*proof*⟩

lemma *prod_complete_silent*:
assumes *step*: $A \vdash \langle L,\ s,\ u\rangle \rightarrow_{Act\ a} \langle L',\ s',\ u'\rangle$
obtains a **where** $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow_{a} \langle(L',\ s'),\ u'\rangle$
⟨*proof*⟩

lemma *prod_complete_sync*:
assumes *step*: $A \vdash \langle L,\ s,\ u\rangle \rightarrow_{Syn\ a} \langle L',\ s',\ u'\rangle$
obtains a **where** $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow_{a} \langle(L',\ s'),\ u'\rangle$
⟨*proof*⟩

lemma *prod_complete*:
assumes *step*: $A \vdash \langle L,\ s,\ u\rangle \rightarrow_{a} \langle L',\ s',\ u'\rangle$
shows $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow \langle(L',\ s'),\ u'\rangle$
⟨*proof*⟩

lemma *A_unfold*:
$A = (N,\ P)$
⟨*proof*⟩

lemma *prod_sound'_delay*:
assumes *step*: $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow^{d} \langle(L',\ s'),\ u'\rangle$
shows $A \vdash \langle L,\ s,\ u\rangle \rightarrow_{Del} \langle L',\ s',\ u'\rangle \wedge product\_ta \vdash \langle L,\ u\rangle \rightarrow \langle L',\ u'\rangle$
$\wedge\ (\forall\, p{<}length\ P.\ (P\ !\ p)\ (L'\ !\ p)\ s')$
⟨*proof*⟩

lemma *prod_sound'_action*:
assumes *step*: $prod\_ta \vdash \langle(L,\ s),\ u\rangle \rightarrow_{a} \langle(L',\ s'),\ u'\rangle$
obtains a **where** $(A \vdash \langle L,\ s,\ u\rangle \rightarrow_{a} \langle L',\ s',\ u'\rangle \wedge a \neq Del) \wedge product\_ta \vdash \langle L,\ u\rangle \rightarrow \langle L',\ u'\rangle$
$\wedge\ (\forall\, p{<}length\ P.\ (P\ !\ p)\ (L'\ !\ p)\ s')$

⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*⟩
⟨*proof*⟩

lemma *inv_prod_step*[*intro*]:
∀ *p*<*length P*. (*P* ! *p*) (*L'* ! *p*) *s'* **if** *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩
⟨*proof*⟩

lemma *prod_steps_sound*:
assumes *step*: *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L'*, *s'*), *u'*⟩
shows *A* ⊢ ⟨*L*, *s*, *u*⟩ →∗ ⟨*L'*, *s'*, *u'*⟩
⟨*proof*⟩

lemma *prod_steps_complete*:
prod_ta ⊢ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L'*, *s'*), *u'*⟩ **if** *A* ⊢ ⟨*L*, *s*, *u*⟩ →∗ ⟨*L'*, *s'*, *u'*⟩
⟨*proof*⟩

lemma *prod_correct*:
prod_ta ⊢ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L'*, *s'*), *u'*⟩ ⟷ *A* ⊢ ⟨*L*, *s*, *u*⟩ →∗ ⟨*L'*, *s'*, *u'*⟩
⟨*proof*⟩

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
⟨proof⟩

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

lemma steps_exec:
  assumes steps prog n s (pc, st, m, f, rs) prog pc = Some HALT
  shows ∃ pcs'. exec prog n s pcs = Some ((pc, st, m, f, rs), pcs')
⟨proof⟩

lemma visited_halt:
  assumes visited prog n (pc, st, m, f, rs) s pcs prog pc = Some HALT
  shows s = (pc, st, m, f, rs)
  ⟨proof⟩

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

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

lemma visited_exec:
  assumes visited prog n s (pc, st, m, f, rs) pcs prog pc = Some HALT
```

shows *exec prog n s pcs′ = Some ((pc, st, m, f, rs), pc # pcs @ pcs′)*
⟨*proof*⟩

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)*
⟨*proof*⟩

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*⟩

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*

$\Rightarrow$ *'s list* $\Rightarrow$ *int list* $\Rightarrow$ (*nat*, *'t*) *cval* $\Rightarrow$ *bool*
(_ $\vdash_n$ ⟨_, _, _⟩ $\rightarrow*$ ⟨_, _, _⟩ [*61*,*61*,*61*,*61*,*61*,*61*] *61*)
where
refl: $A \vdash_n \langle L, s, u\rangle \rightarrow* \langle L, s, u\rangle$ |
step: $A \vdash_n \langle L, s, u\rangle \rightarrow* \langle L', s', u'\rangle \Longrightarrow A \vdash_n \langle L', s', u'\rangle \rightarrow_a \langle L'', s'', u''\rangle$
$\Longrightarrow A \vdash_n \langle L, s, u\rangle \rightarrow* \langle L'', s'', u''\rangle$

declare *steps_un.intros*[*intro*]

lemma *stepI2*:
$A \vdash_n \langle L, s, u\rangle \rightarrow* \langle L'', s'', u''\rangle$ **if**
$A \vdash_n \langle L', s', u'\rangle \rightarrow* \langle L'', s'', u''\rangle$ $A \vdash_n \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$
⟨*proof*⟩

3.2 Equivalent State Network Automaton

definition *stripp* $p \equiv$ *map_option strip o p*
definition *stripfp* $p \equiv$ *map_option stripf o p*
definition *striptp* $p \equiv$ *map_option stript o p*

locale *Equiv_TA_Defs* =
fixes A :: (*'a*, *'t*, *'s*) *unta*
and n :: *nat* — Fuel
begin

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

definition *make_f pc_u* $\equiv \lambda$ *s*.
case (*exec P' n* (*pc_u*, [], *s*, *True*, []) []) *of*
None $\Rightarrow$ []
| *Some* ((_, _, _, _, *r*), _) $\Rightarrow$ *r*

definition *make_mt pc_u* $\equiv \lambda$ *s*.
case (*exec PT n* (*pc_u*, [], *s*, *True*, []) []) *of*
None $\Rightarrow$ *None*
| *Some* ((_, _, *s'*, _, *r*), _) $\Rightarrow$ *Some s'*

definition *make_mf pc_u* $\equiv \lambda$ *s*.
case (*exec PF n* (*pc_u*, [], *s*, *True*, []) []) *of*
None $\Rightarrow$ *None*
| *Some* ((_, _, *s'*, _, *r*), _) $\Rightarrow$ *Some s'*

definition *make_c pc_g* $\equiv \lambda$ *s*.
case (*exec PT n* (*pc_g*, [], *s*, *True*, []) []) *of*
None $\Rightarrow$ *False*
| *Some* ((_, _, _, *f*, _), _) $\Rightarrow$ *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* ⟨*proof*⟩

lemma *bounded_finite*:
finite {*s*. *bounded B s*} (**is** *finite ?S*)
⟨*proof*⟩

lemma *finite_state*:
∀ *q* < *p*. ∀ *l*. *finite* {*s*. (*defs.P* ! *q*) *l s*}
⟨*proof*⟩

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*⟩

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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *stepsc_visitedc*:
∃ *cc. visitedc P n u start end cc* **if** *stepsc P n u start end*
⟨*proof*⟩

lemma *visitedc_stepsc*:
stepsc P n u start end **if** *visitedc P n u start end cc*
⟨*proof*⟩

lemma *steps_visited*:
∃ *cc. visited P n start end cc* **if** *steps P n start end*
⟨*proof*⟩

lemma *visited_steps*:
steps P n start end **if** *visited P n start end cc*
⟨*proof*⟩

context
fixes *P n u start*

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

lemma *stepsc_t_sound'*:
assumes
$steps\ (striptp\ P)\ n\ start\ end$
shows
$stepsc\ P\ n\ u\ start\ end$
⟨*proof*⟩

lemma *stepsc_t_complete'*:
assumes
$stepsc\ P\ n\ u\ start\ end$
shows
$steps\ (striptp\ P)\ n\ start\ end$
⟨*proof*⟩

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

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

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

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

end

context
fixes $P\ n\ start$
assumes *time_indep*: $time\_indep\ P\ n\ start$
begin

lemma *time_indep'*:
⋀ *pc' st s' f' rs cmd*.
stepsc P n u start (*pc'*, *st*, *s'*, *f'*, *rs*) ⟹ *P pc'* = *Some cmd*
⟹ *is_instr cmd*
⟨*proof*⟩

lemma *stepsc_f_complete'*:
assumes
stepsc P n u start end
shows
steps (*stripfp P*) *n start end*
⟨*proof*⟩

lemma *stepsc_f_sound'*:
assumes
steps (*stripfp P*) *n start end*
shows
stepsc P n u start end
⟨*proof*⟩

lemma *stepsc_f_equiv*:
steps (*stripfp P*) *n start end* ⟷ *stepsc P n u start end*
⟨*proof*⟩

lemma *stepst_f_equiv*:
(∃ *pcs'*. *exec* (*stripfp P*) *n start pcs* = *Some* ((*pc*, *st*, *m*, *f*, *rs*), *pcs'*))
⟷ *stepst P n u start* (*pc*, *st*, *m*, *f*, *rs*)
⟨*proof*⟩

end

lemma *exec_acc*:
assumes *exec P n s pcs* = *Some* (*s'*, *pcs'*)
shows ∃ *pcs''*. *pcs'* = *pcs''* @ *pcs*
⟨*proof*⟩

lemma *exec_acc'*:
assumes *Some* (*s'*, *pcs'*) = *exec P n s pcs*
shows ∃ *pcs''*. *pcs'* = *pcs''* @ *pcs*
⟨*proof*⟩

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*)
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *stepsc_mono*:

assumes *stepsc P n u start end* $n' \geq n$
shows *stepsc P* n' *u start end*
⟨*proof*⟩

lemma *stepst_mono*:
assumes *stepst P n u start end* $n' \geq n$
shows *stepst P* n' *u start end*
⟨*proof*⟩

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

lemma *exec_len*:
n ≥ (*length pcs'* − *length pcs*) **if** *exec P n s pcs* = *Some* (*s'*, *pcs'*)
⟨*proof*⟩

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''*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *length_defs_P*[*simp*]:
length defs.P = *p*
⟨*proof*⟩

lemma *inv'*:
∀ *p*<*length defs.P*. (*defs.P* ! *p*) (*L* ! *p*) *s*
⟨*proof*⟩

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*⟩

lemma *A_simp*[*simp*]:
PP = *P N'* = *N I'* = *I B'* = *B* **if** *A* = (*PP*, *N'*, *I'*, *B'*)
⟨*proof*⟩

lemma *A_unfold*:
A = (*P*, *N*, *I*, *B*)
⟨*proof*⟩

sublocale *prod*: *Prod_TA state_ta* ⟨*proof*⟩

lemma *inv_simp*:
snd (*defs.N* ! *q*) (*L'* ! *q*) = *snd* (*N* ! *q*) (*L'* ! *q*) **if** *q* < *p* **for** *L'*
⟨*proof*⟩

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

lemma *ball_lessThan*[*simp*]:
(∀ *x* ∈ {..<*m*}. *Q x*) ⟷ (∀ *x* < *m*. *Q x*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*, [])

⟨*proof*⟩

lemma *P_steps_upd*:
assumes
Some $s'' = make\_mf\ pc\_u\ s'\ (l,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (N\ !\ q)\ q < p$
shows
$\exists\ pc\ st\ f.\ stepst\ P\ n\ u'\ (pc\_u,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ f,\ make\_f\ pc\_u\ s')$
⟨*proof*⟩

lemma *P_steps_reset*:
assumes
$q < p\ (l,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (N\ !\ q)$
shows
$(\exists\, pc\ st\ s''\ f.\ stepst\ P\ n\ u\ (pc\_u,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ f,\ make\_f\ pc\_u\ s')) \vee$
$(\nexists\, pc\ st\ s''\ f\ r'.\ stepst\ P\ n\ u\ (pc\_u,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ f,\ r')) \wedge$
$make\_f\ pc\_u\ s' = []$
⟨*proof*⟩

lemma *steps_P_reset*:
assumes
$(\exists\, pc\ st\ s''\ f.\ stepst\ P\ n\ u\ (pc\_u,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ f,\ r)) \vee$
$(\nexists\, pc\ st\ s''\ f\ r'.\ stepst\ P\ n\ u\ (pc\_u,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ f,\ r')) \wedge r = []$
$q < p\ (l,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (N\ !\ q)$
shows $make\_f\ pc\_u\ s' = r$
⟨*proof*⟩

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

lemma *steps_P_guard*:
assumes
$stepst\ P\ n\ u'\ (pc\_g,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ True,\ rs)$
$q < p\ (l,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (N\ !\ q)$
shows
$make\_c\ pc\_g\ s'\ (\mathbf{is}\ ?A)\ u' \vdash make\_g\ pc\_g\ s'\ (\mathbf{is}\ ?B)$
⟨*proof*⟩

lemma *P_steps_guard*:
assumes
$make\_c\ pc\_g\ s'\ u' \vdash make\_g\ pc\_g\ s'$
$q < p\ (l,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (N\ !\ q)$
shows
$\exists\ pc\ s''\ st\ rs.\ stepst\ P\ n\ u'\ (pc\_g,\ [],\ s',\ True,\ [])\ (pc,\ st,\ s'',\ True,\ rs)$
⟨*proof*⟩

lemma *P_bounded*:
assumes
$(defs.P\ !\ q)\ (L'\ !\ q)\ s'\ q < p$

shows *bounded B s′*
⟨*proof*⟩

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*⟩

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*⟩

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′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *states′_updI*:
assumes (*L* ! *q*, *g*, (*a*, *c*, *m*), *f*, *l′*) ∈ *fst* (*defs.N* ! *q*)
shows *L*[*q* := *l′*] ∈ *defs.states′ s′*
⟨*proof*⟩

lemma *states′_updI″*:
assumes
(*L′* ! *q*, *g*, (*a*, *c*, *m*), *f*, *l′*) ∈ *fst* (*defs.N* ! *q*)
(*L′* ! *q′*, *g′*, (*a′*, *c′*, *m′*), *f′*, *l″*) ∈ *fst* (*defs.N* ! *q′*)
L′ ∈ *defs.states′ s″* *q* ≠ *q′*
shows *L′*[*q* := *l′*, *q′* := *l″*] ∈ *defs.states′ s′*
⟨*proof*⟩

lemma *equiv_sound*:
assumes *step*: *state_ta* ⊢ ⟨*L*, *s*, *u*⟩ →*a* ⟨*L′*, *s′*, *u′*⟩
shows *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →*a* ⟨*L′*, *s′*, *u′*⟩
⟨*proof*⟩

lemma *state_ta_unfold*:
state_ta = (*defs.N*, *defs.P*)
⟨*proof*⟩

lemma *equiv_complete*:

assumes *step*: $A \vdash_n \langle L, s, u\rangle \to_a \langle L', s', u'\rangle$
shows *state_ta* $\vdash \langle L, s, u\rangle \to_a \langle L', s', u'\rangle$
⟨*proof*⟩

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

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

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

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

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

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

lemma *equiv_correct*:
state_ta $\vdash \langle L, s, u\rangle \to* \langle L', s', u'\rangle \longleftrightarrow A \vdash_n \langle L, s, u\rangle \to* \langle L', s', u'\rangle$ **if** $p > 0$
⟨*proof*⟩

lemma *prod_correct*:
defs.prod_ta $\vdash \langle (L, s), u\rangle \to* \langle (L', s'), u'\rangle \longleftrightarrow A \vdash_n \langle L, s, u\rangle \to* \langle L', s', u'\rangle$ **if** $p > 0$
⟨*proof*⟩

end

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

4 Implementation of UPPAAL Style Networks

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

lemma *step_resets*:
$\forall\ c \in set\ r'.\ \exists\ x\ pc.\ Some\ (INSTR\ (STOREC\ c\ x)) = P\ pc$
if *stepc cmd u* $(pc,\ st,\ s,\ f,\ r) = Some\ (pc',\ st',\ s',\ f',\ r')$
$\forall\ c \in set\ r.\ \exists\ x\ pc.\ Some\ (INSTR\ (STOREC\ c\ x)) = P\ pc\ P\ pc = Some\ cmd$
⟨*proof*⟩

lemma *step_resets′*:
$\forall\ c \in set\ r'.\ \exists\ x\ pc.\ Some\ (INSTR\ (STOREC\ c\ x)) = P\ pc$
if *step instr* $(pc,\ st,\ s,\ f,\ r) = Some\ (pc',\ st',\ s',\ f',\ r')$
$\forall\ c \in set\ r.\ \exists\ x\ pc.\ Some\ (INSTR\ (STOREC\ c\ x)) = P\ pc\ P\ pc = Some\ (INSTR\ instr)$
⟨*proof*⟩

lemma *step_resets″*:
$\forall\ c \in set\ r'.\ \exists\ x\ pc.\ Some\ (STOREC\ c\ x) = P\ pc$
if *step instr* $(pc,\ st,\ s,\ f,\ r) = Some\ (pc',\ st',\ s',\ f',\ r')$
$\forall\ c \in set\ r.\ \exists\ x\ pc.\ Some\ (STOREC\ c\ x) = P\ pc\ P\ pc = Some\ instr$
⟨*proof*⟩

lemma *steps_reset*:
$\forall\ c \in set\ r'.\ \exists\ x\ pc.\ Some\ (STOREC\ c\ x) = P\ pc$
if *steps P n* $(pc,\ st,\ s,\ f,\ r)\ (pc',\ st',\ s',\ f',\ r')\ \forall\ c \in set\ r.\ \exists\ x\ pc.\ Some\ (STOREC\ c\ x) = P\ pc$
⟨*proof*⟩

lemma *exec_reset*:
$\forall\ c \in set\ r'.\ \exists\ x\ pc.\ Some\ (STOREC\ c\ x) = P\ pc$
if $Some\ ((pc',\ st',\ s',\ f',\ r'),\ pcs') = exec\ P\ n\ (pc,\ st,\ s,\ f,\ [])\ pcs$
⟨*proof*⟩

lemma *exec_pointers*:
$\forall\ pc \in set\ pcs'.\ \exists\ pc\ instr.\ Some\ instr = P\ pc$
if $Some\ ((pc',\ st',\ s',\ f',\ r'),\ pcs') = exec\ P\ n\ (pc,\ st,\ s,\ f,\ r)\ pcs$
$\forall\ pc \in set\ pcs.\ \exists\ pc\ instr.\ Some\ instr = P\ pc$
⟨*proof*⟩

lemma *exec_pointers′*:
$\forall\ pc \in set\ pcs'.\ \exists\ pc\ instr.\ Some\ instr = P\ pc$
if $Some\ ((pc',\ st',\ s',\ f',\ r'),\ pcs') = exec\ P\ n\ (pc,\ st,\ s,\ f,\ r)\ []$
⟨*proof*⟩

context *Prod_TA_Defs*
begin

lemma *finite_range_I′*:
assumes $\forall A{\in}\{0..{<}p\}.\ finite\ (range\ (snd\ (N\ !\ A)))$

shows *finite* (*range* (*I'* *s*))
⟨*proof*⟩

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

lemma *finite_rangeI*:
assumes ∀ *A*∈{*0*..<*p*}. *finite* (*range* (*snd* (*N* ! *A*)))
shows *finite* (*range prod_invariant*)
⟨*proof*⟩

end

context *Equiv_TA_Defs*
begin

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

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

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

lemma *P_Storec_iff*:
(*Some* (*INSTR* (*STOREC x xa*)) = *P pc*) ⟷ (*Some* (*STOREC x xa*) = *PF pc*)
⟨*proof*⟩

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*}
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *product_trans_resets*:
collect_clkvt (⋃*s*. *defs.T'* *s*) ⊆ {*c*. ∃ *x pc*. *Some* (*INSTR* (*STOREC c x*)) = *P pc*}
⟨*proof*⟩

lemma *product_trans_guards*:
Timed_Automata.collect_clkt (⋃*s*. *defs.T'* *s*)
⊆ {*constraint_pair ac* | *ac*. ∃ *pc*. *Some* (*CEXP ac*) = *P pc*}
⟨*proof*⟩

end

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

fun *check_bexp* :: *bexp* ⇒ *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* ⟨*proof*⟩

abbreviation *EA* ≡ *equiv.state_ta*

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

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

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

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

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

end

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

locale *UPPAAL_Reachability_Problem_precompiled* =
UPPAAL_Reachability_Problem_precompiled_defs +
assumes *process_length*: *length inv* = *p length trans* = *p length pred* = *p*
and *lengths*:
∀ *i* < *p*. *length* (*pred* ! *i*) = *length* (*trans* ! *i*) ∧ *length* (*inv* ! *i*) = *length* (*trans* ! *i*)
and *state_set*: ∀ *T* ∈ *set trans*. ∀ *xs* ∈ *set T*. ∀ (_, _, _, *l*) ∈ *set xs*. *l* < *length T*
assumes *consts_nats*: *snd* ‘ *clkp_set′* ⊆ ℕ

assumes *clock_set*: $clk\_set' = \{1..m\}$
and *p_gt_0*: $p > 0$
and *m_gt_0*: $m > 0$
and *processes_have_trans*: $\forall\ i < p.\ trans\ !\ i \neq []$ — Necessary for refinement
and *start_has_trans*: $\forall\ q < p.\ trans\ !\ q\ !\ 0 \neq []$ — Necessary for refinement

assumes *resets_zero*: $\forall\ x\ c.\ Some\ (INSTR\ (STOREC\ c\ x)) \in set\ prog \longrightarrow x = 0$

begin

lemma *consts_nats'*:
$\forall\ I \in set\ inv.\ \forall\ cc \in set\ I.\ \forall\ (c, d) \in collect\_clock\_pairs\ cc.\ d \in \mathbb{N}$

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

lemma *clk_pairs_N_inv*:
$\bigcup\ (collect\_clock\_pairs\ `\ range\ (snd\ x)) \subseteq \bigcup\ (collect\_clock\_pairs\ `\ set\ (concat\ inv))$
if $x \in set\ equiv.defs.N$ **for** x
⟨*proof*⟩

lemma *clkp_set_simp_1*:
$\bigcup\ (collect\_clock\_pairs\ `\ set\ (concat\ inv)) \supseteq Timed\_Automata.collect\_clki\ (snd\ A)$
⟨*proof*⟩

lemma *clk_set_simp_2*:
$\{c.\ \exists\ x.\ Some\ (INSTR\ (STOREC\ c\ x)) \in set\ prog\} \supseteq collect\_clkvt\ (trans\_of\ A)$
⟨*proof*⟩

lemma *clkp_set_simp_3*:
$\{constraint\_pair\ ac \mid ac.\ Some\ (CEXP\ ac) \in set\ prog\} \supseteq Timed\_Automata.collect\_clkt\ (trans\_of\ A)$
⟨*proof*⟩

lemma *clkp_set'_subs*:
$Timed\_Automata.clkp\_set\ A \subseteq clkp\_set'$
⟨*proof*⟩

lemma *clk_set'_subs*:
$clk\_set\ A \subseteq clk\_set'$
⟨*proof*⟩

lemma *clk_set*:
$clk\_set\ A \subseteq \{1..m\}$
⟨*proof*⟩

lemma
$\forall (\_, d) \in Timed\_Automata.clkp\_set\ A.\ d \in \mathbb{Z}$
⟨*proof*⟩

lemma *clkp_set'_consts_nat*:
$\forall (\_, d) \in clkp\_set'.\ d \in \mathbb{N}$
⟨*proof*⟩

lemma *clkp_set_consts_nat*:
$\forall (\_,\ d) \in Timed\_Automata.clkp\_set\ A.\ d \in \mathbb{N}$
⟨*proof*⟩

lemma *finite_clkp_set'*:
finite clkp_set'
⟨*proof*⟩

lemma *finite_clkp_set_A*[*intro*, *simp*]:
finite (*Timed_Automata.clkp_set A*)
⟨*proof*⟩

lemma *clkp_set'_bounds*:
$a \in \{Suc\ 0..m\}$ **if** $(a,\ b) \in clkp\_set'$
⟨*proof*⟩

lemma *finite_range_inv_of_A*[*intro*, *simp*]:
finite (*range* (*inv_of A*))
⟨*proof*⟩

lemma *Collect_fold_pair*:
$\{f\ a\ b \mid a\ b.\ P\ a\ b\} = (\lambda\ (a,\ b).\ f\ a\ b)\ `\ \{(a,\ b).\ P\ a\ b\}$ **for** *P*
⟨*proof*⟩

lemma *finite_T*[*intro*, *simp*]:
finite (*trans_of A*)
⟨*proof*⟩

sublocale *TA_Start_No_Ceiling A* $(init,\ s_0)$ *m*
⟨*proof*⟩

lemma *P_p*[*simp*]:
length P = *p*
⟨*proof*⟩

lemma *length_I*[*simp*]:
length equiv.I = *p*
⟨*proof*⟩

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

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*
  ⟨*proof*⟩

context
  fixes *prog* :: *int instrc option list*
    and *strip* :: *real instrc* ⇒ *instr*
  assumes *instr_id*[*simp*]:
    strip (*INSTR cmd*) = *cmd*
    strip (*CEXP ac*) ∉ {*CALL*, *RETURN*, *HALT*} ∪ (*JMPZ* ‘ *UNIV*)
begin

private definition [*simp*]:
  P′ ≡ *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*
  ⟨*proof*⟩

lemmas *steps_out_of_range* = *steps_out_of_range′*[*unfolded striptp_def P′_def*]

lemma *steps_steps_approx′*:
  assumes *steps P′ n* (*pc*, *st*, *s*, *f*, *rs*) (*pc′*, *st′*, *s′*, *f′*, *rs′*) *pc′* < *length prog*
  shows *pc′* ∈ *steps_approx n prog pc*
⟨*proof*⟩

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

end

context
  fixes *prog* :: *int instrc option list*
begin

private abbreviation *P i* ≡ *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*
  ⟨*proof*⟩

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*⟩

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*⟩

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*

⟨*proof*⟩

lemma *visitedc_resets_mono*:
set r ⊆ *set r'* **if** *visitedc P n u* (*pc*, *st*, *s*, *f*, *r*) (*pc'*, *st'*, *s'*, *f'*, *r'*) *pcs*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *visited_fuel_mono*:
visited P n' s s' pcs **if** *visited P n s s' pcs n'* ≥ *n*
⟨*proof*⟩

lemma *visitedc_fuel_mono*:
visitedc P n' u s s' pcs **if** *visitedc P n u s s' pcs n'* ≥ *n*
⟨*proof*⟩

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'*])
⟨*proof*⟩

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'*)
⟨*proof*⟩

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'*])
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma *steps_fuel_mono*:
steps P n' s s' **if** *steps P n s s' n'* ≥ *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *steps_approx_out_of_range*:
steps_approx n prog pc = {} **if** *pc* ≥ *length prog*
⟨*proof*⟩

lemma *steps_resets_mono*:
set r ⊆ *set r'* **if** *steps P n* (*pc*, *st*, *s*, *f*, *r*) (*pc'*, *st'*, *s'*, *f'*, *r'*)
⟨*proof*⟩

lemma *resets_start*:
assumes
∀ *pc* ∈ {*pc*..*pc'*}. ∃ *c x*. *prog* ! *pc* = *Some* (*INSTR* (*STOREC c x*))
steps
(*map_option stripf o* (λ*pc*. *if pc* < *size prog then prog* ! *pc else None*))
n (*pc*, *st*, *s*, *f*, *r*) (*pc_t*, *st'*, *s'*, *f'*, *r'*)
prog ! *pc_t* = *Some* (*INSTR HALT*)
shows {*c*. ∃ *x*. ∃ *pc* ∈ {*pc* .. *pc'*}. *prog* ! *pc* = *Some* (*INSTR* (*STOREC c x*))} ⊆ *set r'*
⟨*proof*⟩

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
)

⟨*proof*⟩

termination
⟨*proof*⟩

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

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)
⟨*proof*⟩

lemma *bexp_atLeastLessThan_iff*:
(∀ pc ∈ {pc_s..<pc_t}. P pc) ⟷ (∀ pc. pc_s ≤ pc ∧ pc < pc_t ⟶ P pc)
⟨*proof*⟩

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′)
⟨*proof*⟩

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
)

⟨*proof*⟩

termination
⟨*proof*⟩

lemma *find_next_halt_finds_halt*:
prog ! *pc'* = *Some* (*INSTR HALT*) ∧ *pc* ≤ *pc'* ∧ *pc'* < *length prog*
if *find_next_halt prog pc* = *Some pc'*
⟨*proof*⟩

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*⟩

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*⟩

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)
⟨*proof*⟩

lemma *clkp_set'_alt_def*:
clkp_set' =
⋃ (*collect_clock_pairs* ‘ *set* (*concat inv*)) ∪ (*constraint_pair* ‘ *collect_cexp*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *clk_set'_alt_def*: *clk_set'* = (*fst* ‘ *clkp_set'* ∪ *fst* ‘ *collect_store*)
⟨*proof*⟩

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
)

⟨*proof*⟩

termination
⟨*proof*⟩

lemma *is_conj_block'_len_prog*:
pc' < length prog **if** *is_conj_block' prog pc pc'*
⟨*proof*⟩

lemma *check_conj_block'*:
check_conj_block' prog pc = Some pc' ⟹ is_conj_block' prog pc pc'
⟨*proof*⟩

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')
⟨*proof*⟩

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*⟩

lemma
pc = *pc′* − *1* **if**
stepc (*CEXP cc*) *u* (*pc*, *st*, *m*, *f*, *rs*) = *Some* (*pc′*, *st′*, *m′*, *f′*, *rs′*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stepsn_stepsc*:
assumes *stepsn P n′ u s s′ n′* < *n*
shows *stepsc P n u s s′*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *stepsc_halt*:
s′ = (*pc*, *s*) **if** *stepsc P n u* (*pc*, *s*) *s′ P pc* = *Some* (*INSTR HALT*)
⟨*proof*⟩

lemma *stepsn_halt*:
s′ = (*pc*, *s*) **if** *stepsn P n u* (*pc*, *s*) *s′ P pc* = *Some* (*INSTR HALT*)
⟨*proof*⟩

lemma *is_conj_block′_pc_mono*:
pc ≤ *pc′* **if** *is_conj_block′ prog pc pc′*
⟨*proof*⟩

lemma *is_conj_block'_halt*:
prog ! *pc'* = *Some* (*INSTR HALT*) **if** *is_conj_block' prog pc pc'*
⟨*proof*⟩

lemma *numeral_4_eq_4*:
4 = *Suc* (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *is_conj_block'_is_conj*:
assumes *is_conj_block' P pc pc'*
and *stepsn* (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'*
⟨*proof*⟩

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'*
⟨*proof*⟩

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*⟩

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*⟩

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'*

⟨*proof*⟩

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′*
⟨*proof*⟩

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″*
⟨*proof*⟩

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″*
⟨*proof*⟩

lemma *steps_approx_finite*[*intro*,*simp*]:
finite (*steps_approx n P pc_s*)
⟨*proof*⟩

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*⟩

lemma *conv_P_conj_block′*[*intro*]:
is_conj_block′ (*conv_P P*) *pc pc′* **if** *is_conj_block′ P pc pc′*
⟨*proof*⟩

lemma *conv_P_conj_block*[*intro*]:
is_conj_block (*conv_P P*) *pc pc′* **if** *is_conj_block P pc pc′*
⟨*proof*⟩

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′* ⟨*proof*⟩

lemma *is_conj*:

fixes $u :: nat \Rightarrow real$
defines $S \equiv steps\_approx\ n\ P\ pc\_s$
defines $pc\_c \equiv Min\ \{pc.\ \exists\ ac.\ pc \in S \wedge P\ !\ pc = Some\ (CEXP\ ac)\}$
assumes $is\_conj\_block\ P\ pc\_c\ (Max\ S)$
and $stepst\ (conv\_prog\ prog)\ n\ u\ (pc\_s, st, s, f, rs)\ (pc\_t, st\_t, s\_t, True, rs\_t)$
and $stepsc\ (conv\_prog\ prog)\ n\ u\ (pc\_s, st, s, f, rs)\ (pc', st', s', f', rs')$
and $P\ !\ pc' = Some\ (CEXP\ ac)\ pc' < length\ P$
shows $(u \vdash_a conv\_ac\ ac) \wedge pc\_t = Max\ S$
⟨*proof*⟩

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*
⟨*proof*⟩

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*:
$check\_conj\_block\ pc\ pc' \Longrightarrow is\_conj\_block\ P\ pc\ pc'$
⟨*proof*⟩

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*⟩

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 $u \vdash_a conv\_ac\ ac$
⟨*proof*⟩

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'*
⟨*proof*⟩

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
  ⟨proof⟩

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 $(g,\ a,\ r,\ L') \in set\ (trans\_i\_fun\ L)$
shows $(g,\ a,\ r,\ L') \in set\ (trans\_fun\ L)$
⟨*proof*⟩

lemma *trans_s_fun_trans_fun*:
assumes $(g,\ a,\ r,\ L') \in set\ (trans\_s\_fun\ L)$
shows $(g,\ a,\ r,\ L') \in set\ (trans\_fun\ L)$
⟨*proof*⟩

end

context *Prod_TA_Defs*
begin

lemma *prod_trans_i_alt_def*:
$prod\_trans\_i =$
$\{((L,\ s),\ g,\ a,\ r,\ (L',\ s')) \mid L\ s\ g\ c\ a\ r\ m\ L'\ s'.$
$(L,\ g,\ (a,\ Networks.label.Act\ (c,\ m)),\ r,\ L') \in Product\_TA\_Defs.product\_trans\_i\ (N\_s\ s)$
$\wedge$
$(\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s')$
$\wedge\ c\ s \wedge Some\ s' = m\ s\}$
⟨*proof*⟩

lemma *prod_trans_s_alt_def*:
$prod\_trans\_s =$
$\{((L,\ s),\ g,\ a,\ r,\ (L',\ s')) \mid L\ s\ g\ ci\ co\ a\ r\ mi\ mo\ L'\ s1\ s'.$
$ci\ s \wedge co\ s$
$\wedge (\forall\ q < p.\ (P\ !\ q)\ (L\ !\ q)\ s) \wedge (\forall\ q < p.\ (P\ !\ q)\ (L'\ !\ q)\ s')$
$\wedge (L,\ g,\ (a,\ Networks.label.Syn\ (ci,\ mi)\ (co,\ mo)),\ r,\ L')$
$\in Product\_TA\_Defs.product\_trans\_s\ (N\_s\ s)$
$\wedge\ Some\ s' = mi\ s1 \wedge Some\ s1 = mo\ s$
$\}$
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled_defs*
begin

lemma *T_s_unfold_1*:
$fst\ `\ equiv.defs.T\_s\ q\ s = fst\ `\ fst\ (equiv.N\ !\ q)$ **if** $q < p$
⟨*proof*⟩

lemma *T_s_unfold_2*:
$(snd\ o\ snd\ o\ snd\ o\ snd)\ `\ equiv.defs.T\_s\ q\ s = (snd\ o\ snd\ o\ snd\ o\ snd)\ `\ fst\ (equiv.N\ !\ q)$
if $q < p$
⟨*proof*⟩

end

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

context
Prod_TA_Defs
begin

lemma *states'_alt_def*:
states' s =
{*L*. *length L* = *p* ∧
(∀ *q* < *p*. (*L* ! *q*) ∈ *fst* ‘ (*fst* (*fst A* ! *q*)) ∪ (*snd o snd o snd o snd*) ‘ (*fst* (*fst A* ! *q*)))}
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled*
begin

lemma *PF_unfold*:
equiv.PF = *stripfp* (*conv_prog PROG*)
⟨*proof*⟩

lemma *PT_unfold*:
equiv.PT = *striptp* (*conv_prog PROG*)
⟨*proof*⟩

lemma *states'I*:
l ∈ *equiv.defs.states' s* **if** *A* ⊢ (*l*, *s*) ⟶^{g,a,r} (*l'*, *s'*)
⟨*proof*⟩

lemma *A_lengthD*:
length l = *p* **if** *A* ⊢ (*l*, *s*) ⟶^{g,a,r} (*l'*, *s'*)
⟨*proof*⟩

lemma *N_s_state_trans*:
assumes *equiv.defs.N_s s* ! *q* ⊢ *l* ! *q* ⟶$^{g,(a,\ c,\ m'),r}$ *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*
⟨*proof*⟩

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*⟩

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*⟩

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 $x \in clkp\_set''\ q\ l$
⟨*proof*⟩

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 $A \vdash (l, s) \longrightarrow^{g,a,r} (l', s')$
⟨*proof*⟩

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 ⟨*proof*⟩

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*))}
⟨*proof*⟩

lemma *init_states*:
init ∈ *equiv.defs.states'* s_0
⟨*proof*⟩

lemma *p_p*[*simp*]:
defs'.p = *p*
⟨*proof*⟩

lemma *T_s_unfold_1'*:
fst ‘ *defs'.defs.T_s q s* = *fst* ‘ *fst* (*defs'.N* ! *q*) **if** *q* < *p*
⟨*proof*⟩

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*
⟨*proof*⟩

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*))}
⟨*proof*⟩

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

lemma *init_states_defs′*[*intro*]:
init ∈ *defs′.defs.states′* s_0
⟨*proof*⟩

lemma
defs′.I = *equiv.I*
⟨*proof*⟩

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

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

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

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

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

end

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

$\forall\ i < p.\ \forall\ l < length\ (trans\ !\ i).\ \forall\ (g,\ a,\ r,\ l') \in set\ (trans\ !\ i\ !\ l).$
$guaranteed\_execution\_cond\ prog\ r\ max\_steps$

begin

definition $k\_fun\ l\ c \equiv if\ c > 0 \wedge c \leq m\ then\ Max\ \{k\ !\ i\ !\ (fst\ l\ !\ i)\ !\ c \mid i\ .\ i < p\}\ else\ 0$

lemma p_p':
$equiv.p = p$
⟨*proof*⟩

lemma *clkp_set_clk_set_subs*:
$fst\ `\ clkp\_set\ A\ (l,\ s) \subseteq clk\_set\ A$
⟨*proof*⟩

lemma *k_ceiling_1*:
$\forall\ l.\ \forall (x,m) \in clkp\_set\ A\ l.\ m \leq k\_fun\ l\ x$

⟨*proof*⟩

lemma *k_ceiling_2*:
$\forall\ l\ g\ a\ r\ l'\ c.\ A \vdash l \longrightarrow^{g,a,r} l' \wedge c \notin set\ r \longrightarrow k\_fun\ l'\ c \leq k\_fun\ l\ c$
⟨*proof*⟩

lemma
shows *k_ceiling'*:
$\forall\ l.\ \forall (x,m) \in clkp\_set\ A\ l.\ m \leq k\_fun\ l\ x$
$\forall\ l\ g\ a\ r\ l'\ c.\ A \vdash l \longrightarrow^{g,a,r} l' \wedge c \notin set\ r \longrightarrow k\_fun\ l'\ c \leq k\_fun\ l\ c$
and *k_bound'*:
$\forall\ l.\ \forall\ i > m.\ k\_fun\ l\ i = 0$
and *k_0'*:
$\forall\ l.\ k\_fun\ l\ 0 = 0$
⟨*proof*⟩

sublocale *Reachability_Problem A* $(init,\ s_0)$ *m k_fun PR_CONST* $(\lambda\ (l,\ s).\ F\ l\ s)$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *equiv_P_simp*:
equiv.P = *PROG*
⟨*proof*⟩

lemma *time_indep_P*[*intro*]:
time_indep (*conv_prog equiv.P*) *max_steps* (*pred* ! *q* ! (*L* ! *q*), [], *s*, *True*, [])
if $q < p\ L \in$ *equiv.defs.states'* *s*
⟨*proof*⟩

lemma *time_indep_PROG*[*intro*]:
time_indep (*conv_prog PROG*) *max_steps* (*pc_u*, [], *s*, *True*, [])
if $q < p\ (l, pc\_g, a, pc\_u, l') \in T\ q$
⟨*proof*⟩

lemma *facts*[*intro*]:
$u \vdash_a ac$ **if**
$q <$ *defs'.p*
$(l, pc\_g, a, pc\_u, l') \in$ *fst* (*defs'.N* ! *q*)
stepst defs'.P max_steps u (*pc_g*, [], *s*, *True*, []) (*pc_t*, *st_t*, *s_t*, *True*, *rs_t*)
stepsc defs'.P max_steps u (*pc_g*, [], *s*, *True*, []) (*pc'*, *st*, *s'*, *f'*, *rs*)
defs'.P pc' = *Some* (*CEXP ac*)
⟨*proof*⟩

sublocale *product'*:
Equiv_TA conv N max_steps init s_0
⟨*proof*⟩

lemma *fst_S*[*simp*]:
fst ‘ ($\lambda(l, g, a, r, l')$. $(l,$ *map conv_ac g*, *a*, *r*, $l')$) ‘ *S* = *fst* ‘ *S*
⟨*proof*⟩

lemma *snd_S*[*simp*]:
(*snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ ($\lambda(l, g, a, r, l')$. $(l,$ *map conv_ac g*, *a*, *r*, $l')$) ‘ *S*
= (*snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *S*
⟨*proof*⟩

end

locale *UPPAAL_Reachability_Problem_precompiled'* =
UPPAAL_Reachability_Problem_precompiled_start_state +
UPPAAL_Reachability_Problem_precompiled_defs' +
UPPAAL_Reachability_Problem_precompiled_ceiling +
assumes *action_set*:
$\forall\ T \in set\ trans.\ \forall\ xs \in set\ T.\ \forall\ (\_, a, \_) \in set\ xs.\ pred\_act\ (\lambda\ a.\ a < na)\ a$
begin

sublocale *Reachability_Problem_Impl_Defs* _ _ *A* (*init*, s_0) *m k_fun PR_CONST* (λ (*l*, *s*). *F l s*) ⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *in_actions_by_state_preserv*:
assumes
(*q*, *g1*, *a*, *r1*) ∈ *set* (*acc* ! *a*) *a* < *length acc*
shows
(*q*, *g1*, *a*, *r1*) ∈ *set* (*actions_by_state y xs acc* ! *a*)
∧ *a* < *length* (*actions_by_state y xs acc*)
⟨*proof*⟩

lemma *length_actions_by_state_preserv*[*simp*]:
shows *length* (*actions_by_state y xs acc*) = *length acc*
⟨*proof*⟩

lemma *in_all_actions_by_stateI*:
assumes
a < *na q* < *p* (*g1*, *a*, *r1*) ∈ *set* (*M* !! *q* !! (*L* ! *q*))
shows
(*q*, *g1*, *a*, *r1*) ∈ *set* (*all_actions_by_state M L* ! *a*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *in_actions_by_stateD*:
assumes
(*q*, *g*, *a*, *t*) ∈ *set* (*actions_by_state i xs acc* ! *j*) (*q*, *g*, *a*, *t*) ∉ *set* (*acc* ! *j*)
j < *length acc*
shows
(*g*, *a*, *t*) ∈ *set xs* ∧ *j* = *a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *length_all_actions_by_state_preserv*:
length (*all_actions_by_state M L*) = *na*
⟨*proof*⟩

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*
⟨*proof*⟩

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 $(pa,\ g1,\ a,\ r1) \in set\ (all\_actions\_by\_state\ (IArray\ (map\ IArray\ trans\_in\_map))\ L\ !\ a)$
⟨*proof*⟩

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 $(pa,\ g1,\ a,\ r1) \in set\ (all\_actions\_by\_state\ (IArray\ (map\ IArray\ trans\_out\_map))\ L\ !\ a)$
⟨*proof*⟩

lemma *in_pairs_by_actionD2*:
assumes
$(g,\ a,\ r,\ L',\ s') \in set\ (pairs\_by\_action\ (L,\ s)\ xs\ ys)$
$\forall\ (q,\ g,\ a'',\ m,\ l) \in set\ xs.\ a'' = a'$
$\forall\ (q,\ g,\ a'',\ m,\ l) \in set\ ys.\ a'' = a'$
shows *check_pred* $L'\ s'$
⟨*proof*⟩

lemma *in_pairs_by_actionD1*:
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 g1 g2 r1 r2 pc_u1 pc_u2 l1' l2' s1
x1 x2 x3 x4 x5 y1 y2 y3 y4
where
$pa \neq q$
$(pa,\ pc\_g1,\ a,\ pc\_u1,\ l1') \in set\ ys$
$(q,\ pc\_g2,\ a,\ pc\_u2,\ l2') \in set\ xs$
$L' = L[pa := l1',\ q := l2']$
$runf\ pc\_u1\ s1 = Some\ ((x1,\ x2,\ s',\ x3,\ x4),\ x5)$
$runf\ pc\_u2\ s = Some\ ((y1,\ y2,\ s1,\ y3,\ r2),\ y4)$
$check\_g\ pc\_g1\ s = Some\ g1\ check\_g\ pc\_g2\ s = Some\ g2$
$r1 = make\_reset\ pc\_u1\ s$
$g = g1\ @\ g2\ r = r1\ @\ r2$
⟨*proof*⟩

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
$pa \neq q$
$(pa,\ pc\_g1,\ a,\ pc\_u1,\ l1') \in set\ ys$

$(q, pc\_g2, a, pc\_u2, l2') \in set\ xs$
$L' = L[pa := l1', q := l2']$

$runf\ pc\_u1\ s1 = Some\ ((x1, x2, s', x3, x4), x5)$
$runf\ pc\_u2\ s = Some\ ((y1, y2, s1, y3, r2), y4)$
$check\_g\ pc\_g1\ s = Some\ g1\ check\_g\ pc\_g2\ s = Some\ g2$
$r1 = make\_reset\ pc\_u1\ s$
$g = g1\ @\ g2\ r = r1\ @\ r2$

$check\_pred\ L'\ s'$

⟨*proof*⟩

lemma *in_trans_funD*:
assumes $y \in set\ (trans\_fun\ L)$
shows $y \in set\ (trans\_s\_fun\ L) \vee y \in set\ (trans\_i\_fun\ L)$
⟨*proof*⟩

lemma *states'_states'*[*intro*]:
$L \in equiv.defs.states'\ s$ **if** $(L, s) \in states'$
⟨*proof*⟩

lemma *bounded'_bounded*:
$bounded'\ s \longleftrightarrow bounded\ bounds\ s$ **if** $length\ s = length\ bounds$
⟨*proof*⟩

lemma *bounded_bounded'*:
$bounded\ bounds\ s \Longrightarrow bounded'\ s$
⟨*proof*⟩

lemma *P_unfold*:
$(\forall\, q{<}p.\ (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \longleftrightarrow (check\_pred\ L\ s)$ **if** $length\ s = length\ bounds$
⟨*proof*⟩

lemma *P_unfold_1*:
$(\forall\, q{<}p.\ (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \Longrightarrow (check\_pred\ L\ s)$
⟨*proof*⟩

lemma *equiv_PT*[*simp*]:
$equiv.PT = PT$
⟨*proof*⟩

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 $(L\ !\ q, pc\_g, a, pc\_u, l') \in fst\ (equiv.N\ !\ q)$
⟨*proof*⟩

lemma *trans_ND*:
assumes

$(L\ !\ q,\ pc\_g,\ a,\ pc\_u,\ l') \in fst\ (equiv.N\ !\ q)$
$q < p$
shows
$equiv.defs.N\_s\ s\ !\ q \vdash L\ !\ q \longrightarrow^{equiv.make\_g\ pc\_g\ s,(a,\ equiv.make\_c\ pc\_g,\ equiv.make\_mf\ pc\_u),equiv.make\_f\ pc\_u\ s}\ l'$
⟨*proof*⟩

lemma *make_f_unfold*:
$equiv.make\_f\ pc\ s = make\_reset\ pc\ s$
⟨*proof*⟩

lemma *make_g_simp*:
assumes $check\_g\ pc\_g\ s = Some\ g1$
shows $g1 = equiv.make\_g\ pc\_g\ s$
⟨*proof*⟩

lemma *make_c_simp*:
assumes $check\_g\ pc\_g\ s = Some\ g1$
shows $equiv.make\_c\ pc\_g\ s$
⟨*proof*⟩

lemma *make_reset_simp*:
assumes $runf\ pc\_u\ s = Some\ ((y1,\ y2,\ s1,\ y3,\ r2),\ y4)$
shows $make\_reset\ pc\_u\ s = r2$
⟨*proof*⟩

lemma *make_mf_simp*:
assumes $runf\ pc\_u\ s = Some\ ((y1,\ y2,\ s1,\ y3,\ r2),\ y4)$
shows $equiv.make\_mf\ pc\_u\ s = Some\ s1$
⟨*proof*⟩

lemma *trans_fun_trans_of'*:
$(trans\_fun,\ trans\_of\ A) \in transition\_rel\ states'$
⟨*proof*⟩

lemma *transition_rel_mono*:
$(a,\ b) \in transition\_rel\ B$ **if** $(a,\ b) \in transition\_rel\ C\ B \subseteq C$
⟨*proof*⟩

end

context
Equiv_TA_Defs
begin

lemma *state_set_subs*:
$state\_set\ (trans\_of\ (defs.product\ s''))$
$\subseteq \{L.\ length\ L = defs.p \wedge (\forall\ q{<}defs.p.\ L\ !\ q \in State\_Networks.state\_set\ (fst\ (defs.N\ !\ q)))\}$
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled_defs*

begin

lemma *N_s_state_indep*:
assumes $(L\ !\ q,\ g,\ a,\ r,\ l') \in map\ trans\_of\ (equiv.defs.N\_s\ s)\ !\ q\ q < p$
obtains $g\ r$ **where** $(L\ !\ q,\ g,\ a,\ r,\ l') \in map\ trans\_of\ (equiv.defs.N\_s\ s')\ !\ q$
⟨*proof*⟩

lemma *fst_product_state_indep*:
$fst\ `\ fst\ (equiv.defs.product\ s) = fst\ `\ fst\ (equiv.defs.product\ s')$
⟨*proof*⟩

lemma *last_product_state_indep*:
$(snd\ o\ snd\ o\ snd\ o\ snd)\ `\ fst\ (equiv.defs.product\ s)$
$= (snd\ o\ snd\ o\ snd\ o\ snd)\ `\ fst\ (equiv.defs.product\ s')$
⟨*proof*⟩

lemma *state_set_T'*:
$state\_set\ (equiv.defs.T'\ s'') \supseteq fst\ `\ state\_set\ (trans\_of\ A)$
⟨*proof*⟩

lemma *state_set_T'2*:
$length\ L = equiv.defs.p$
$\forall\ q<equiv.defs.p.\ L\ !\ q \in State\_Networks.state\_set\ (fst\ (equiv.defs.N\ !\ q))$
if $(L,\ s) \in state\_set\ (trans\_of\ A)$
⟨*proof*⟩

lemma *state_set_states'*:
$L \in equiv.defs.states'\ s$ **if** $(L,\ s) \in state\_set\ (trans\_of\ A)$
⟨*proof*⟩

lemma *state_set_pred*:
$\forall\ q<p.\ (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s$ **if** $(L,\ s) \in state\_set\ (trans\_of\ A)$
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *bounded_bounded''*:
$bounded\ bounds\ s \Longrightarrow length\ s = length\ bounds$
⟨*proof*⟩

lemma *P_bounded*:
$(\forall\ q<p.\ (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \Longrightarrow bounded\ bounds\ s$
⟨*proof*⟩

lemma *P_state_length*:
$(\forall\ q<p.\ (equiv.defs.P\ !\ q)\ (L\ !\ q)\ s) \Longrightarrow length\ s = length\ bounds$
⟨*proof*⟩

lemma *state_set_state_length*:
$length\ s = length\ bounds$ **if** $(L,\ s) \in state\_set\ (trans\_of\ A)$

⟨*proof*⟩

lemma *state_set_states*:
state_set (*trans_of A*) ⊆ *states′*
⟨*proof*⟩

lemma *p_p_2*[*simp*]:
defs′.defs.p = *p*
⟨*proof*⟩

lemma *len_product′_N*[*simp*]:
length defs′.defs.N = *p*
⟨*proof*⟩

lemma *len_equiv_N*:
length equiv.defs.N = *p*
⟨*proof*⟩

lemma
defs′.p = *p*
⟨*proof*⟩

lemma *equiv_p_p*: *equiv.p* = *p*
⟨*proof*⟩

lemma *init_states*:
init ∈ *equiv.defs.states′ s*$_0$
⟨*proof*⟩

lemma *start_pred′*:
check_pred init s$_0$
⟨*proof*⟩

lemma *start_states′*:
(*init*, *s*$_0$) ∈ *states′*
⟨*proof*⟩

lemma *trans_fun_trans_of*[*intro*, *simp*]:
(*trans_fun*, *trans_of A*) ∈ *transition_rel states*
⟨*proof*⟩

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

lemma *states_states′*:
states ⊆ *states′*
⟨*proof*⟩

lemma *states′_length*[*simp*]:
length L = *p* **if** (*L*, *s*) ∈ *states′*
⟨*proof*⟩

lemma *inv_simp*:

$I\ q\ (L\ !\ q) = inv\ !\ q\ !\ (L\ !\ q)$ **if** $q < p\ (L, s) \in states'$
⟨*proof*⟩

lemma *inv_fun_inv_of'*:
$(inv\_fun, inv\_of\ A) \in inv\_rel\ Id\ states'$
⟨*proof*⟩

lemma *inv_rel_mono*:
$(a, b) \in inv\_rel\ Id\ B$ **if** $(a, b) \in inv\_rel\ Id\ C\ B \subseteq C$
⟨*proof*⟩

lemma *inv_fun_inv_of*[*intro*, *simp*]:
$(inv\_fun, inv\_of\ A) \in inv\_rel\ Id\ states$
⟨*proof*⟩

definition $final\_fun \equiv \lambda\ (L, s).\ hd\_of\_formula\ formula\ L\ s$

lemma *final_fun_final'*:
$(final\_fun, (\lambda\ (l, s).\ F\ l\ s)) \in inv\_rel\ Id\ states'$
⟨*proof*⟩

lemma *final_fun_final*[*intro*, *simp*]:
$(final\_fun, (\lambda\ (l, s).\ F\ l\ s)) \in inv\_rel\ Id\ states$
⟨*proof*⟩

lemma *fst_clkp_setD*:
assumes $(c, d) \in clkp\_set\ A\ l$
shows $c > 0\ c \leq m\ d \in range\ int$
⟨*proof*⟩

lemma *init_has_trans*:
$(init, s_0) \in fst\ `\ (trans\_of\ A) \longleftrightarrow trans\_fun\ (init, s_0) \neq []$
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

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

definition
$k\_impl \equiv \lambda\ (l, \_).\ IArray\ (map\ (\lambda\ c.\ Max\ \{k\_i\ !!\ i\ !!\ (l\ !\ i)\ !!\ c \mid i.\ i < p\})\ [0..<m+1])$

lemma *k_impl_alt_def*:
$k\_impl =$
$(\lambda\ (l, \_).\ IArray\ (map\ (\lambda\ c.\ Max\ ((\lambda\ i.\ k\_i\ !!\ i\ !!\ (l\ !\ i)\ !!\ c)\ `\ \{0..<p\}))\ [0..<m+1]))$
⟨*proof*⟩

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

lemma *Max_int_commute*:

int (*Max S*) = *Max* (*int* ‘ *S*) **if** *finite S S* ≠ {}
⟨*proof*⟩

lemma *k_impl_k'*[*intro*]:
k_impl (*l*, *s*) = *IArray* (*k'* (*l*, *s*)) **if** (*l*, *s*) ∈ *states'*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *k_impl_k'_0*[*intro*]:
k_impl (*init*, s_0) = *IArray* (*k'* (*init*, s_0))
⟨*proof*⟩

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 l_0 = (*init*, s_0)
and $l_0 i$ = (*init*, s_0)
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*
⟨*proof*⟩

lemma *F_reachable_correct'*:
impl.op.F_reachable
⟷ (∃ *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'*
) **if** *formula* = *formula.EX* φ
⟨*proof*⟩

lemma *PT_PT*:
defs'.PT = *equiv.PT*
⟨*proof*⟩

lemma *P_P*[*simp*]:
defs'.defs.P = *equiv.defs.P*
⟨*proof*⟩

lemma *map_map_filter*:
map f (*List.map_filter g xs*) = *List.map_filter* (*map_option f* ∘ *g*) *xs*
⟨*proof*⟩

lemma *make_g_conv*:
defs'.make_g = conv_cc oo equiv.make_g
⟨*proof*⟩

lemma *make_c_conv*:
defs'.make_c = equiv.make_c
⟨*proof*⟩

lemma *make_f_conv*:
defs'.make_f = equiv.make_f
⟨*proof*⟩

lemma *make_mf_conv*:
defs'.make_mf = equiv.make_mf
⟨*proof*⟩

lemmas *make_convs = make_g_conv make_c_conv make_f_conv make_mf_conv*

lemma *state_trans_conv*:
Equiv_TA_Defs.state_trans_t (conv N) max_steps q
= (λ (a, b, c). (a, λ x. conv_cc (b x), c)) ‘ equiv.state_trans q **if** ‹*q < p*›
⟨*proof*⟩

lemma *map_conv_t*:
map trans_of (defs'.defs.N_s s) ! q = conv_t ‘ (map trans_of (equiv.defs.N_s s) ! q)
if ‹*q < p*›
⟨*proof*⟩

lemma *product_trans_t_conv*:
Product_TA_Defs.product_trans_s (defs'.defs.N_s s)
= conv_t ‘ Product_TA_Defs.product_trans_s (equiv.defs.N_s s)
⟨*proof*⟩

lemma *product_trans_t_conv'*:
Product_TA_Defs.product_trans_i (defs'.defs.N_s s)
= conv_t ‘ Product_TA_Defs.product_trans_i (equiv.defs.N_s s)
⟨*proof*⟩

lemma *prod_trans_s_conv*:
defs'.defs.prod_trans_s = conv_t ‘ equiv.defs.prod_trans_s
⟨*proof*⟩

lemma *prod_trans_i_conv*:
defs'.defs.prod_trans_i = conv_t ‘ equiv.defs.prod_trans_i
⟨*proof*⟩

lemma *prod_trans_conv*:
defs'.defs.prod_trans = conv_t ‘ equiv.defs.prod_trans
⟨*proof*⟩

lemma *prod_invariant_conv*:
defs'.defs.prod_invariant = (map conv_ac ∘∘ Prod_TA_Defs.prod_invariant) EA
⟨*proof*⟩

lemma *prod_conv*: *defs'.defs.prod_ta* = *conv_A A*
⟨*proof*⟩

lemma *F_reachable_correct*:
impl.op.F_reachable
⟷ (∃ *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*
⟨*proof*⟩

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 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*
⟨*proof*⟩

corollary *reachability_checker'_hoare*:
<*emp*> *reachability_checker'*
<λ *r*. ↑(*r* = (∃ *L' s' u u'*.
conv_A 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*
⟨*proof*⟩

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 ⊢_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
    )
  ))
  ∈ unit_assn^k →_a id_assn if formula = formula.EX φ
  ⟨proof⟩

corollary reachability_checker_hoare:
  <emp> reachability_checker
  <λ r. ↑(r =
      (
        if start_inv_check
        then
          if
            (
            ∃ L' s' u u'.
              conv N ⊢_m ax_steps ⟨init, 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
 ⟨proof⟩

lemma list_all_concat:
  list_all Q (concat xxs) ⟷ (∀ xs ∈ set xxs. list_all Q xs)
  ⟨proof⟩

lemma inv_of_init_unfold:
  u ⊢ inv_of (conv_A A) (init, s_0) ⟷ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0))
⟨proof⟩

corollary reachability_checker_hoare':
  <emp> reachability_checker
  <λ r. ↑(r =
      (
```

```
        if (∀ u. (∀ c∈{1..m}. u c = 0) ⟶ (∀ i < p. u ⊢ conv_cc (inv ! i ! 0)))
        then
          if
            (
            ∃ L' s' u u'.
              conv N ⊢_m ax_steps ⟨init, 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
  ⟨proof⟩

Post-processing  schematic_goal succs_impl_alt_def:
  impl.succs_impl ≡ ?impl
  ⟨proof⟩

 lemma reachability_checker'_alt_def':
  reachability_checker' ≡
    let
      key = return ∘ fst;
      sub = impl.subsumes_impl;
      copy = impl.state_copy_impl;
      start = impl.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
  ⟨proof⟩

 schematic_goal reachability_checker_alt_def:
  reachability_checker ≡ ?impl
  ⟨proof⟩

end

lemmas [code] = UPPAAL_Reachability_Problem_precompiled'.k_impl_def

Some post refinements  code_thms fw_upd'
code_thms fw_impl'
code_thms fw_impl

abbreviation plus_int :: int ⇒ int ⇒ int where
  plus_int a b ≡ a + b

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

```
  dbm_add_int (Le a) (Le b) = (Le (plus_int a b)) |
  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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
    ⟨proof⟩

  lemma runt_impl:
    runt = run_impl PT
    ⟨proof⟩

  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
  ⟨proof⟩
```

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'
  ⟨proof⟩
```

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
  ⟨proof⟩
```

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)
  ⟨proof⟩
```

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)
⟨*proof*⟩

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

lemma *all_actions_by_state_impl*:
all_actions_by_state = all_actions_by_state_impl [0..<p] (repeat [] na)
⟨*proof*⟩

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

lemma *trans_i_from_impl*:
trans_i_from = trans_i_from_impl PF PT PROG' (IArray bounds) (IArray (map IArray trans_i_map))
⟨*proof*⟩

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *PF_alt_def*:
PF = (λ pc. if pc < length prog then (IArray (map (map_option stripf) prog)) !! pc else None)
⟨*proof*⟩

lemma *PT_alt_def*:
PT = (λ pc. if pc < length prog then (IArray (map (map_option stript) prog)) !! pc else None)
⟨*proof*⟩

schematic_goal *reachability_checker_alt_def_refined*:

reachability_checker ≡ *?impl*
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

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′
⟨*proof*⟩

lemma *check_pre*:
UPPAAL_Reachability_Problem_precompiled p m inv pred trans prog ⟷ *check_pre*
⟨*proof*⟩

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*
⟨*proof*⟩

end

end

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

export_code *UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp* **in** *SML* **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* ⟨*proof*⟩

concrete_definition *reachability_checker_impl*
uses *UPPAAL_Reachability_Problem_precompiled'.reachability_checker_alt_def_refined*

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

lemmas [*code*] =

UPPAAL_Reachability_Problem_precompiled_defs.P_def

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

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

definition
init_pred_check ≡ 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
  )
  ⟨proof⟩

export_code UPPAAL_Reachability_Problem_precompiled_start_state_axioms

context UPPAAL_Reachability_Problem_precompiled_defs
begin

  lemma collect_store''_alt_def:
    collect_store'' pc ≡
    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'}
        )
    ⟨proof⟩

  lemma collect_cexp'_alt_def:
    collect_cexp' pc ≡
      ⋃ ((λ cmd. case cmd of Some (CEXP ac) ⇒ {ac} | _ ⇒ {}) ‘
          ((!) prog) ‘ steps_approx max_steps prog pc
      )
    ⟨proof⟩

end

lemmas [code] =
  UPPAAL_Reachability_Problem_precompiled_defs'.PROG'_def
  UPPAAL_Reachability_Problem_precompiled_start_state_def
  UPPAAL_Reachability_Problem_precompiled_ceiling_axioms_def
  UPPAAL_Reachability_Problem_precompiled_defs.N_def
  UPPAAL_Reachability_Problem_precompiled_defs.collect_store''_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs.clkp_set''_def
  UPPAAL_Reachability_Problem_precompiled_defs.collect_cexp'_alt_def
  UPPAAL_Reachability_Problem_precompiled_defs'.pairs_by_action_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.make_reset_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_g_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.run_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.make_cconstr_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.check_pred_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.all_actions_by_state_impl_def
  UPPAAL_Reachability_Problem_precompiled_defs'.trans_i_from_impl_def

lemmas [code] =
  Equiv_TA_Defs.state_ta_def Prod_TA_Defs.N_s_def Product_TA_Defs.states_def

export_code UPPAAL_Reachability_Problem_precompiled'_axioms in SML module_name
Test

export_code UPPAAL_Reachability_Problem_precompiled' in SML module_name Test
```

hide_const *check_and_verify*

definition [*code*]:
check_and_verify p m k max_steps I T prog final bounds P 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 s_0 *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*, s_0, *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*⟩

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'*⟩
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *equiv_correct*:
state_ta ⊢ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩ = *A* ⊢$_n$ ⟨*L*, *s*, *u*⟩ →$_a$ ⟨*L'*, *s'*, *u'*⟩
⟨*proof*⟩

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*)
⟨*proof*⟩

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

lemma *prod_correct*:

defs.prod_ta $\vdash \langle (L,\ s),\ u\rangle \to \langle (L',\ s'),\ u'\rangle =$
$(\exists\ a.\ A \vdash_n \langle L,\ s,\ u\rangle \to_a \langle L',\ s',\ u'\rangle)$
⟨*proof*⟩

context
assumes $0 < p$
begin

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

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

lemma *step_u_inv*:
all_prop $L'\ s'$ **if** $A \vdash_n \langle L,\ s,\ u\rangle \to_a \langle L',\ s',\ u'\rangle$
⟨*proof*⟩

lemma *step_inv*:
all_prop $L'\ s'$ **if** *state_ta* $\vdash \langle L,\ s,\ u\rangle \to_a \langle L',\ s',\ u'\rangle$
⟨*proof*⟩

lemma *Equiv_TA_I*:
Equiv_TA $A\ n\ L'\ s'$ **if** $*$[*unfolded all_prop_def*]: *all_prop* $L'\ s'$
⟨*proof*⟩

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

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

lemma *prod_correct_step'*:
defs.prod_ta $\vdash' \langle (L,\ s),\ u\rangle \to \langle (L',\ s'),\ u'\rangle =$
$A \vdash^n \langle L,\ s,\ u\rangle \to \langle L',\ s',\ u'\rangle$
⟨*proof*⟩

lemma *all_prop_start*:
all_prop $L\ s$
⟨*proof*⟩

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

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′′*⟩
⟨*proof*⟩

lemma *steps_un′_complete*:
defs.prod_ta ⊢′ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L′′*, *s′′*), *u′′*⟩
if *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →∗ ⟨*L′′*, *s′′*, *u′′*⟩
⟨*proof*⟩

lemma *steps′_sound*:
A ⊢*n* ⟨*L*, *s*, *u*⟩ →∗ ⟨*L′′*, *s′′*, *u′′*⟩
if *defs.prod_ta* ⊢′ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L′′*, *s′′*), *u′′*⟩
⟨*proof*⟩

lemma *prod_reachable_correct*:
defs.prod_ta ⊢′ ⟨(*L*, *s*), *u*⟩ →∗ ⟨(*L′*, *s′*), *u′*⟩ ⟷ *A* ⊢*n* ⟨*L*, *s*, *u*⟩ →∗ ⟨*L′*, *s′*, *u′*⟩
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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)
  ) a₀


definition
  has_deadlock A n a₀ ≡
    Graph_Defs.deadlock (λ (L, s, u) (L', s', u'). A ⊢^n ⟨L, s, u⟩ → ⟨L', s', u'⟩) a₀

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

context Reachability_Problem
begin

lemma reaches_steps':
  reaches (l, u) (l', u') ⟷ conv_A A ⊢' ⟨l, u⟩ →* ⟨l', u'⟩
  ⟨proof⟩

lemma clocks_I:
  (∀ c. c ∈ clk_set (conv_A A) ⟶ u c = u' c) if ∀ c ∈ {1..n}. u c = u' c
  ⟨proof⟩

lemma init_dbm_reaches_iff:
  (∃ u ∈ [curry init_dbm]_{v,n}. ∃ u'. conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩)
  ⟷ ([curry (init_dbm :: real DBM')]_{v,n} ≠ {} ∧
    (∀ u ∈ [curry init_dbm]_{v,n}. ∃ u'. conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩))

⟨proof⟩

theorem reachable_decides_emptiness_new:
  (∃ D'. E** a₀ (l', D') ∧ [curry (conv_M D')]_{v,n} ≠ {})
  ⟷ [curry (init_dbm :: real DBM')]_{v,n} ≠ {} ∧
    (∀ u ∈ [curry init_dbm]_{v,n}. ∃ u'. conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩)
  ⟨proof⟩

lemma reachable_decides_emptiness'_new:
  (∃ D'. E** a₀ (l', D') ∧ [curry (conv_M D')]_{v,n} ≠ {})
  ⟷ (∀ u. (∀ c ∈ {1..n}. u c = 0) ⟶ (∃ u'. conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩))
  ⟨proof⟩

lemma reachability_check_new_aux:
  (∃ D'. E** a₀ (l', D') ∧ [curry (conv_M D')]_{v,n} ≠ {} ∧ F l')
  ⟷ (∀ u. (∀ c ∈ {1..n}. u c = 0) ⟶ (∃ u'. conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩ ∧ F l'))
  ⟨proof⟩

theorem reachability_check_new:
    (∃ D'. E** a₀ (l', D') ∧ F_rel (l', D'))
```

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

lemma *init_state_in_state_set*:
$l_0 \in state\_set\ (trans\_of\ A)$ **if** $\neg\ deadlock\ (l_0, u_0)$
⟨*proof*⟩

lemma *init_state_in_state_set'*:
$l_0 \in state\_set\ (trans\_of\ A)$ **if** ($\forall\, u_0$. ($\forall\, c{\in}\{1..n\}$. $u_0\ c = 0$) $\longrightarrow \neg\ deadlock\ (l_0, u_0)$)
⟨*proof*⟩

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* ($\lambda\ r. \neg\ r \longleftrightarrow$
($\nexists\, x$. ($\lambda a\ b.\ E\_op''.E\_from\_op\ a\ b \wedge \neg\ check\_diag\ n\ (snd\ b))^{**}$ $(l_0, init\_dbm)\ x \wedge$
$F\ (fst\ x) \wedge Q\ (fst\ x) \wedge$
($\exists\, a$. ($\lambda a\ b.\ E\_op''.E\_from\_op\ a\ b \wedge \neg\ check\_diag\ n\ (snd\ b) \wedge Q\ (fst\ b))^{**}\ x\ a \wedge$
($\lambda a\ b.\ E\_op''.E\_from\_op\ a\ b \wedge \neg\ check\_diag\ n\ (snd\ b) \wedge Q\ (fst\ b))^{++}\ a\ a$)
))
⟨*proof*⟩

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. ($\forall\, u_0$. ($\forall\, c{\in}\{1..n\}$. $u_0\ c = 0$) $\longrightarrow \neg\ deadlock\ (l_0, u_0)$) $\longrightarrow$
($\neg\ r$) =
($\forall\, u_0$. ($\forall\, c{\in}\{1..n\}$. $u_0\ c = 0$) $\longrightarrow$
leadsto ($\lambda(l, u).\ F\ l$) ($\lambda(l, u).\ \neg\ Q\ l$) (l_0, u_0))))))
$\in unit\_assn^k \rightarrow_a bool\_assn$
⟨*proof*⟩

end

end

context *UPPAAL_Reachability_Problem_precompiled'*
begin

lemma *F_reachable_correct'_new*:
impl.op.F_reachable

$\longleftrightarrow$ ($\exists$ L' s'. $\forall$ u. ($\forall$ $c \in \{1..m\}$. $u\ c = 0$) $\longrightarrow$ ($\exists$ u'.
$conv\_A\ A \vdash' \langle(init, s_0), u\rangle \rightarrow* \langle(L', s'), u'\rangle$
$\wedge$ $check\_bexp\ \varphi\ L'\ s'$)
) **if** $formula = formula.EX\ \varphi$
⟨*proof*⟩

lemma *F_reachable_correct'_new'*:
impl.op.F_reachable
$\longleftrightarrow$ ($\exists$ L' s'. $\forall$ u. ($\forall$ $c \in \{1..m\}$. $u\ c = 0$) $\longrightarrow$ ($\exists$ u'.
$conv\_A\ A \vdash' \langle(init, s_0), u\rangle \rightarrow* \langle(L', s'), u'\rangle$
$\wedge$ $\neg$ $check\_bexp\ \varphi\ L'\ s'$)
) **if** $formula = formula.AG\ \varphi$
⟨*proof*⟩

lemma *F_reachable_correct_new*:
impl.op.F_reachable
$\longleftrightarrow$ ($\exists$ L' s'. $\forall$ u. ($\forall$ $c \in \{1..m\}$. $u\ c = 0$) $\longrightarrow$ ($\exists$ u'.
$conv\ N \vdash^m ax\_steps\ \langle init, s_0, u\rangle \rightarrow* \langle L', s', u'\rangle$
$\wedge$ $check\_bexp\ \varphi\ L'\ s'$)
) **if** $formula = formula.EX\ \varphi$
⟨*proof*⟩

lemma *F_reachable_correct_new'*:
impl.op.F_reachable
$\longleftrightarrow$ ($\exists$ L' s'. $\forall$ u. ($\forall$ $c \in \{1..m\}$. $u\ c = 0$) $\longrightarrow$ ($\exists$ u'.
$conv\ N \vdash^m ax\_steps\ \langle init, s_0, u\rangle \rightarrow* \langle L', s', u'\rangle$
$\wedge$ $\neg$ $check\_bexp\ \varphi\ L'\ s'$)
) **if** $formula = formula.AG\ \varphi$
⟨*proof*⟩

definition
Alw_ev_checker = *dfs_map_impl'*
(*impl.succs_P_impl' final_fun*) *impl.a*$_0$*_impl impl.subsumes_impl* (*return* $\circ$ *fst*)
impl.state_copy_impl

definition
leadsto_checker ψ = *do* {
$r \leftarrow$ *leadsto_impl*
impl.state_copy_impl (*impl.succs_P_impl'* (λ (L, s). $\neg$ $check\_bexp\ \psi\ L\ s$))

impl.a$_0$*_impl impl.subsumes_impl* (*return* $\circ$ *fst*)
impl.succs_impl' impl.emptiness_check_impl impl.F_impl
(*impl.Q_impl* (λ (L, s). $\neg$ $check\_bexp\ \psi\ L\ s$))
impl.tracei;
return ($\neg$ r)
}

definition
model_checker = (
case formula of
formula.EX _ $\Rightarrow$ *reachability_checker'* |
formula.AG _ $\Rightarrow$ *do* {
$r \leftarrow$ *reachability_checker'*;
return ($\neg$ 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
  ⟨proof⟩

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)
  ⟨proof⟩

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)
  ⟨proof⟩

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⟩

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⟩

lemma final_fun_final':
  ((λ (L, s). P L s), (λ (L, s). P L s)) ∈ inv_rel Id states' for P
  ⟨proof⟩

lemma final_fun_final[intro, simp]:
  ((λ (L, s). P L s), (λ (L, s). P L s)) ∈ inv_rel Id states for P
  ⟨proof⟩

abbreviation u₀ ≡ (λ _. 0 :: real)

lemma deadlock_start_iff:
  Bisim_A.B.deadlock (init, s₀, λ_. 0) ⟷ deadlock ((init, s₀), u₀)
  ⟨proof⟩

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, s₀, u₀) ⟶
        r = (conv N,(init, s₀, u₀) ⊨_m ax_steps formula)
      )
    )
   )
  ∈ unit_assn^k →_a bool_assn
⟨proof⟩

theorem model_check'_hoare:
  <emp>
    model_checker
  <λr. ↑ ((¬ Bisim_A.B.deadlock (init, s₀, λ_. 0)) ⟶ r = (
    conv N,(init, s₀, u₀) ⊨_m ax_steps formula
  ))>_t
  ⟨proof⟩

lemma Alw_ev_checker_alt_def':
  Alw_ev_checker ≡
    do {
      x ← let
        key = return ∘ fst;
```

```
      sub = impl.subsumes_impl;
      copy = impl.state_copy_impl;
      start = impl.a₀_impl;
      succs =  impl.succs_P_impl' final_fun
    in dfs_map_impl' succs start sub key copy;
    _ ← return ();
    return x
  }
⟨proof⟩

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

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


lemmas succs_P'_impl_alt_def =
  impl.succs_P_impl'_def[OF final_fun_final, unfolded succs_P_impl_alt_def]

lemmas succs_impl'_alt_def =
  impl.succs_impl'_def[unfolded succs_impl_alt_def]

lemma reachability_checker'_alt_def':
  reachability_checker' ≡
    do {
      x ← do {
        let key = return ∘ fst;
        let sub = impl.subsumes_impl;
        let copy = impl.state_copy_impl;
        let start = impl.a₀_impl;
        let final = impl.F_impl;
        let succs =  impl.succs_impl;
        let empty = impl.emptiness_check_impl;
        let tracei = impl.tracei;
        pw_impl key copy tracei sub start final succs empty
```

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

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

schematic_goal Alw_ev_checker_alt_def:
 Alw_ev_checker ≡ ?impl
 ⟨proof⟩

schematic_goal leadsto_checker_alt_def:
 leadsto_checker ≡ ?impl
 ⟨proof⟩

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

schematic_goal Alw_ev_checker_alt_def_refined:
 Alw_ev_checker ≡ ?impl
 ⟨proof⟩

schematic_goal leadsto_checker_alt_def_refined:
 leadsto_checker ≡ ?impl
 ⟨proof⟩

end

concrete_definition reachability_checker'
 uses UPPAAL_Reachability_Problem_precompiled'.reachability_checker'_alt_def_refined

concrete_definition Alw_ev_checker
 uses UPPAAL_Reachability_Problem_precompiled'.Alw_ev_checker_alt_def_refined

concrete_definition leadsto_checker
 uses UPPAAL_Reachability_Problem_precompiled'.leadsto_checker_alt_def_refined

context UPPAAL_Reachability_Problem_precompiled'
begin

lemmas model_checker_def_refined = model_checker_def[unfolded
   reachability_checker'.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
   Alw_ev_checker.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
   leadsto_checker.refine[OF UPPAAL_Reachability_Problem_precompiled'_axioms]
 ]

end

concrete_definition model_checker uses
 UPPAAL_Reachability_Problem_precompiled'.model_checker_def_refined
```

definition [*code*]:
precond_mc p m k max_steps I T prog final bounds P 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*⟩

theorem *model_check_alt*:
<*emp*> *precond_mc p m k max_steps I T prog formula bounds P* s_0 *na*
<λ *r*. ↑ (
if UPPAAL_Reachability_Problem_precompiled' p m max_steps I T prog bounds P s_0 *na k*
then r ≠ *None* ∧
(¬ *has_deadlock* (*conv* (*N p I P T prog bounds*)) *max_steps* (*repeat 0 p*, s_0, λ_ . *0*) ⟶
r = *Some* (
conv (*N p I P T prog bounds*),(*repeat 0 p*, s_0, λ_ . *0*) ⊨$_m$*ax_steps formula*
))
else r = *None*
)>$_t$
⟨*proof*⟩

prepare_code_thms *dfs_map_impl'_def leadsto_impl_def*

lemmas [*code*] =
reachability_checker'_def
Alw_ev_checker_def
leadsto_checker_def
model_checker_def[*unfolded UPPAAL_Reachability_Problem_precompiled_defs.F_def PR_CONST_def*]

export_code
precond_mc Pure.type init_pred_check time_indep_check1 time_indep_check1 conjunction_check2
checking *SML*

end
theory *Simple_Expressions*
imports *Main*
begin

datatype ('*a*, '*b*) *bexp* =
true |

not ('*a*, '*b*) *bexp* |
and ('*a*, '*b*) *bexp* ('*a*, '*b*) *bexp* |
or ('*a*, '*b*) *bexp* ('*a*, '*b*) *bexp* |
imply ('*a*, '*b*) *bexp* ('*a*, '*b*) *bexp* | — Boolean connectives
eq ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* |
le ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* |
lt ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* |
ge ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* |
gt ('*a*, '*b*) *exp* ('*a*, '*b*) *exp*
and ('*a*, '*b*) *exp* =
const '*b* | *var* '*a* | *if_then_else* ('*a*, '*b*) *bexp* ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* |
binop '*b* ⇒ '*b* ⇒ '*b* ('*a*, '*b*) *exp* ('*a*, '*b*) *exp* | *unop* '*b* ⇒ '*b* ('*a*, '*b*) *exp*

inductive *check_bexp* :: ('*a* ⇀ '*b*) ⇒ ('*a*, '*b* :: *linorder*) *bexp* ⇒ *bool* ⇒ *bool* **and** *is_val*
where
check_bexp s true True |
check_bexp s (*not e*) (¬ *b*) **if** *check_bexp s e b* |
check_bexp s (*and e1 e2*) (*a* ∧ *b*) **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*or e1 e2*) (*a* ∨ *b*) **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*imply e1 e2*) (*a* ⟶ *b*) **if** *check_bexp s e1 a check_bexp s e2 b* |
check_bexp s (*eq a b*) (*x* = *v*) **if** *is_val s a v is_val s b x* |
check_bexp s (*le a b*) (*v* ≤ *x*) **if** *is_val s a v is_val s b x* |
check_bexp s (*lt a b*) (*v* < *x*) **if** *is_val s a v is_val s b x* |
check_bexp s (*ge a b*) (*v* ≥ *x*) **if** *is_val s a v is_val s b x* |
check_bexp s (*gt a b*) (*v* > *x*) **if** *is_val s a v is_val s b x* |
is_val s (*const c*) *c* |
is_val s (*var x*) *v* **if** *s x* = *Some v* |
is_val s (*if_then_else b e1 e2*) (*if bv then v1 else v2*)
if *is_val s e1 v1 is_val s e2 v2 check_bexp s b bv* |
is_val s (*binop f e1 e2*) (*f v1 v2*) **if** *is_val s e1 v1 is_val s e2 v2* |
is_val s (*unop f e*) (*f v*) **if** *is_val s e v*

inductive_simps *check_bexp_simps*:
check_bexp s bexp.true bv
check_bexp s (*bexp.not b*) *bv*
check_bexp s (*bexp.and b1 b2*) *bv*
check_bexp s (*bexp.or b1 b2*) *bv*
check_bexp s (*bexp.imply b1 b2*) *bv*
check_bexp s (*le i x*) *bv*
check_bexp s (*lt i x*) *bv*
check_bexp s (*ge i x*) *bv*
check_bexp s (*eq i x*) *bv*
check_bexp s (*gt i x*) *bv*

inductive_simps *is_val_simps*:
is_val s (*const c*) *d*
is_val s (*var x*) *v*
is_val s (*if_then_else b e1 e2*) *v*
is_val s (*binop f e1 e2*) *v*
is_val s (*unop f e*) *v*

inductive_cases *check_bexp_elims*:
check_bexp s bexp.true bv
check_bexp s (*bexp.not b*) *bv*

check_bexp s (bexp.and b1 b2) bv
check_bexp s (bexp.or b1 b2) bv
check_bexp s (bexp.imply b1 b2) bv
check_bexp s (le i x) bv
check_bexp s (lt i x) bv
check_bexp s (ge i x) bv
check_bexp s (eq i x) bv
check_bexp s (gt i x) bv

inductive_cases *is_val_elims*:
is_val s (const c) d
is_val s (var x) v
is_val s (if_then_else b e1 e2) v
is_val s (binop f e1 e2) v
is_val s (unop f e) v

type_synonym
*('a, 'b) upd = ('a * ('a, 'b) exp) list*

definition *is_upd* **where**
is_upd s upd s' ≡ ∃ 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''*
⟨*proof*⟩

lemma *is_upds_appendE*:
assumes *is_upds s (upds1 @ upds2) s''*
obtains *s'* **where** *is_upds s upds1 s' is_upds s' upds2 s''*
⟨*proof*⟩

lemma *is_upds_NilD*:
s' = s **if** *is_upds s [] s'*
⟨*proof*⟩

lemma *is_upds_all_NilD*:
assumes *is_upds s (concat upds) s' (∀ xs∈set upds. xs = [])*
shows *s' = s*
⟨*proof*⟩

lemma *is_upd_const_simp*:
is_upd s (*x*, *const c*) *s'* ⟷ *s'* = *s*(*x* := *Some c*)
⟨*proof*⟩

end
theory *Simple_Network_Language*
imports *Main*
Munta_Model_Checker.State_Networks
Munta_Model_Checker.UPPAAL_State_Networks_Impl
FinFun.FinFun
Simple_Expressions
Munta_Tagging
begin

6 Simple Networks of Automata With Broadcast Channels and Committed Locations

no_notation *top_assn* (*true*)

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

Comments:

- Should there be an error transition + state if states run 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*)›
⟨*proof*⟩

lemma *clock_val_append_iff*:
‹*u* ⊢ *g1* @ *g2* ⟷ *u* ⊢ *g1* ∧ *u* ⊢ *g2*›
⟨*proof*⟩

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* ∧ *p* < *length L* ∧ *L*′ = *L*[*p* := *l*′] ∧ *is_upds s f s*′ ∧ *check_bexp s b True* ∧
L ∈ *states* ∧ *bounded bounds s* ∧ *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
$trans\_broad =$
$\{((L,\ s),\ g\ @\ concat\ (map\ gs\ ps),\ Broad\ a,\ r\ @\ concat\ (map\ rs\ ps),\ (L',\ s'')) \mid$
$L\ s\ L'\ s'\ s''\ a\ p\ l\ b\ g\ f\ r\ l'\ bs\ gs\ fs\ rs\ ls'\ ps.$
$a \in broadcast\ \wedge$
$(l,\ b,\ g,\ Out\ a,\ f,\ r,\ l') \in trans\ (N\ p) \wedge$
$(\forall p \in set\ ps.\ (L\ !\ p,\ bs\ p,\ gs\ p,\ In\ a,\ fs\ p,\ rs\ p,\ ls'\ p) \in trans\ (N\ p)) \wedge$
$(l \in committed\ (N\ p) \vee (\exists p \in set\ ps.\ L\ !\ p \in committed\ (N\ p))$
$\vee (\forall p < n\_ps.\ L\ !\ p \notin committed\ (N\ p))) \wedge$
$(\forall q < n\_ps.\ q \notin set\ ps \wedge p \neq q \longrightarrow$
$\neg\ (\exists b\ g\ f\ r\ l'.\ (L!q,\ b,\ g,\ In\ a,\ f,\ r,\ l') \in trans\ (N\ q) \wedge check\_bexp\ s\ b\ True)) \wedge$
$L!p = l \wedge$
$p < length\ L \wedge set\ ps \subseteq \{0..<n\_ps\} \wedge p \notin set\ ps \wedge distinct\ ps \wedge sorted\ ps \wedge$
$check\_bexp\ s\ b\ True \wedge (\forall p \in set\ ps.\ check\_bexp\ s\ (bs\ p)\ True) \wedge$
$L' = (fold\ (\lambda p\ L\ .\ L[p := ls'\ p])\ ps\ L)[p := l'] \wedge$
$is\_upds\ s\ f\ s' \wedge is\_upds\ s'\ (concat\_map\ fs\ ps)\ s'' \wedge$
$L \in states \wedge bounded\ bounds\ s \wedge bounded\ bounds\ s''$
$\}$

definition
$trans\_prod = trans\_int \cup trans\_bin \cup trans\_broad$

definition
$prod\_ta = (trans\_prod,\ prod\_inv :: ('s\ list \times ('x \Rightarrow 'v\ option) \Rightarrow \_))$

lemma *inv_of_prod*[*simp*]:
$inv\_of\ prod\_ta = prod\_inv$
⟨*proof*⟩

lemma *trans_of_prod*[*simp*]:
$trans\_of\ prod\_ta = trans\_prod$
⟨*proof*⟩

lemma *A_split*:
$‹A = (broadcast,\ map\ N\ [0..<n\_ps],\ bounds)›$
⟨*proof*⟩

lemma *map_N_n_ps_simp*:
$map\ N\ [0..<n\_ps]\ !\ p = N\ p$
⟨*proof*⟩

lemma *N_split_simp*[*simp*]:
assumes $A = (broadcast',\ N',\ B)$
shows $N'\ !\ i = N\ i$

⟨*proof*⟩

lemma *state_preservation_updI*:
assumes $l' \in (\bigcup (l, b, g, a, r, u, l') \in trans\ (N\ p).\ \{l, l'\})\ L \in states$
shows $L[p := l'] \in states$
⟨*proof*⟩

lemma *state_preservation_fold_updI*:
assumes $\forall p \in set\ ps.\ ls'\ p \in (\bigcup (l, b, g, a, r, u, l') \in trans\ (N\ p).\ \{l, l'\})\ L \in states$
shows $fold\ (\lambda p\ L.\ L[p := ls'\ p])\ ps\ L \in states$
⟨*proof*⟩

lemma *state_set_states*:
Simulation_Graphs_TA.state_set prod_ta $\subseteq \{(l, s).\ l \in states\}$
⟨*proof*⟩

lemma *trans_prod_bounded_inv*:
‹*bounded bounds s'*› **if** ‹$((L, s), g, a, r, (L', s')) \in trans\_prod$›
⟨*proof*⟩

lemma *trans_prod_states_inv*:
‹$L' \in states$› **if** ‹$((L, s), g, a, r, (L', s')) \in trans\_prod$› ‹$L \in states$›
⟨*proof*⟩

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*]:
‹$u \vdash prod\_inv\ (L, s)$› **if** ‹$\forall p{<}n\_ps.\ u \vdash Simple\_Network\_Language.inv\ (N\ p)\ (L\ !\ p)$›
⟨*proof*⟩

lemma *prod_invD*[*dest*]:
‹$\forall p{<}n\_ps.\ u \vdash Simple\_Network\_Language.inv\ (N\ p)\ (L\ !\ p)$› **if** ‹$u \vdash prod\_inv\ (L, s)$› ‹$L \in states$›
⟨*proof*⟩

lemma *delay_sound*:
assumes $prod\_ta \vdash \langle(L, s), u\rangle \rightarrow^d \langle(L', s'), u'\rangle\ L \in states\ bounded\ bounds\ s$
$\forall N \in set\ (fst\ (snd\ A)).\ urgent\ N = \{\}$
shows $A \vdash \langle L, s, u\rangle \rightarrow_{Del} \langle L', s', u'\rangle$
⟨*proof*⟩

lemma *action_sound*:
$A \vdash \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$ **if** $prod\_ta \vdash \langle(L, s), u\rangle \rightarrow_a \langle(L', s'), u'\rangle$
⟨*proof*⟩

lemma *bounded_inv*:
‹*bounded bounds s'*› **if** ‹$A \vdash \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$›
⟨*proof*⟩

lemma *states_inv*:
‹$L' \in states$› **if** ‹$A \vdash \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$› ‹$L \in states$›
⟨*proof*⟩

end

locale *Prod_TA* =
Prod_TA_sem A **for** $A :: ('a, 's, 'c, 't :: time, 'x, 'v :: linorder)\ nta$ +
assumes *broadcast_receivers_unguarded*:
$\forall p < n\_ps.\ \forall l\ b\ g\ a\ f\ r\ l'.\ (l, b, g, In\ a, f, r, l') \in trans\ (N\ p) \wedge a \in broadcast \longrightarrow g = []$
begin

lemma *action_complete*:
prod_ta $\vdash \langle(L, s), u\rangle \rightarrow_a \langle(L', s'), u'\rangle$
if $A \vdash \langle L, s, u\rangle \rightarrow_a \langle L', s', u'\rangle$ $a \neq Del\ L \in states\ bounded\ bounds\ s$
⟨*proof*⟩

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

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* $L \in states\ \forall N \in set\ (fst\ (snd\ A)).\ urgent\ N = \{\}$
⟨*proof*⟩

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*:
$x \in set\ xs \Longrightarrow \exists\ i < length\ xs.\ xs\ !\ i = x$
⟨*proof*⟩

lemma *union_subsetI*:
$A \cup B \subseteq C \cup D$ **if** $A \subseteq C\ B \subseteq D$
⟨*proof*⟩

lemma *map_eq_imageD*:
$f\ `\ set\ xs = set\ ys$ **if** $map\ f\ xs = ys$
⟨*proof*⟩

lemma *if_contract*:
$(if\ a\ then\ x\ else\ if\ b\ then\ x\ else\ y) = (if\ a \vee b\ then\ x\ else\ y)$ **for** $a\ x\ b\ y$
⟨*proof*⟩

More intros **named_theorems** *more_intros*
named_theorems *more_elims*
lemmas [*more_intros*] =
image_eqI[*rotated*] *CollectI subsetI*

lemmas [*more_elims*] =
CollectE

Finiteness **lemma** *finite_prodI*:
finite {(*a*,*b*). *P a* ∧ *Q b*} **if** *finite* {*a*. *P a*} *finite* {*a*. *Q a*}
⟨*proof*⟩

lemma *finite_prodI3*:
finite {(*a*,*b*,*c*). *P a* ∧ *Q b* ∧ *Q1 c*}
if *finite* {*a*. *P a*} *finite* {*a*. *Q a*} *finite* {*a*. *Q1 a*}
⟨*proof*⟩

lemma *finite_prodI4*:
finite {(*a*,*b*,*c*,*d*). *P a* ∧ *Q b* ∧ *Q1 c* ∧ *Q2 d*}
if *finite* {*a*. *P a*} *finite* {*a*. *Q a*} *finite* {*a*. *Q1 a*} *finite* {*a*. *Q2 a*}
⟨*proof*⟩

lemma *finite_prodI5*:
finite {(*a*,*b*,*c*,*d*,*e*). *P a* ∧ *Q b* ∧ *Q1 c* ∧ *Q2 d* ∧ *Q3 e*}
if *finite* {*a*. *P a*} *finite* {*a*. *Q a*} *finite* {*a*. *Q1 a*} *finite* {*a*. *Q2 a*} *finite* {*a*. *Q3 a*}
⟨*proof*⟩

named_theorems *finite_intros*
named_theorems *more_finite_intros*

lemmas [*finite_intros*] =
finite_UnI finite_Union finite_imageI
finite_lists_length_eq finite_lists_length_le
finite_subset_distinct distinct_finite_set

lemmas [*more_finite_intros*] =
finite_prodI finite_prodI3 finite_prodI4 finite_prodI5

Lists **lemma** *fold_evD2*:
assumes
P y (*fold f xs acc*) ¬ *P y acc*
⋀ *acc x*. ¬ *P y acc* ⟹ *Q acc* ⟹ *P y* (*f x acc*) ⟹ *x* ∈ *set xs* ⟹ *x* = *y*
Q acc ⋀ *acc x*. *Q acc* ⟹ *Q* (*f x acc*)
⋀ *acc x*. ¬ *P y acc* ⟹ *Q acc* ⟹ *P y* (*f x acc*) ⟹ *R y*
shows ∃ *ys zs*. *xs* = *ys* @ *y* # *zs* ∧ ¬ *P y* (*fold f ys acc*) ∧ *P y* (*f y* (*fold f ys acc*)) ∧ *R y*
⟨*proof*⟩

lemmas *fold_evD2′* = *fold_evD2*[**where** *R* = λ _. *True*, *simplified*]

lemma *filter_map_map_filter*:
filter P (*map f xs*) = *List.map_filter* (λ *x*. *let y* = *f x in if P y then Some y else None*) *xs*
⟨*proof*⟩

lemma *distinct_map_filterI*:
distinct (*List.map_filter f xs*)
if $\forall x \in set\ xs.\ \forall y \in set\ xs.\ \forall a.\ f\ x = Some\ a \wedge f\ y = Some\ a \longrightarrow x = y$ *distinct xs*
⟨*proof*⟩

lemma *filter_eqI*:
assumes
subseq ys xs $\forall x \in set\ ys.\ P\ x$
$\forall zs.\ subseq\ zs\ xs \wedge length\ zs > length\ ys \longrightarrow (\exists\ x \in set\ zs.\ \neg\ P\ x)$
shows *filter P xs* = *ys*
⟨*proof*⟩

lemma *filter_greatest_subseqD*:
$\exists\ x \in set\ zs.\ \neg\ P\ x$ **if** *subseq zs xs length zs* > *length* (*filter P xs*)
⟨*proof*⟩

lemma *filter_eq_iff_greatest_subseq*:
filter P xs = *ys* $\longleftrightarrow$
subseq ys xs $\wedge$ $(\forall x \in set\ ys.\ P\ x) \wedge$
$(\forall zs.\ subseq\ zs\ xs \wedge length\ zs > length\ ys \longrightarrow (\exists\ x \in set\ zs.\ \neg\ P\ x))$
⟨*proof*⟩

lemma *subseq_subsetD*:
set xs $\subseteq$ *set ys* **if** *subseq xs ys*
⟨*proof*⟩

lemma *subseq_distinct*:
distinct xs **if** *distinct ys subseq xs ys*
⟨*proof*⟩

lemma *finite_subseqs*:
finite {*xs. subseq xs ys*} (**is** *finite ?S*)
⟨*proof*⟩

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*⟩

lemma *subseq_sorted_wrt*:
sorted_wrt R xs **if** *sorted_wrt R ys subseq xs ys*
⟨*proof*⟩

lemma *subseq_sorted*:
sorted xs **if** *sorted ys subseq xs ys*
⟨*proof*⟩

lemma *sorted_distinct_subset_subseqI*:
assumes *sorted xs distinct xs sorted ys set xs* $\subseteq$ *set ys*
shows *subseq xs ys*
⟨*proof*⟩

lemma *sorted_distinct_subseq_iff*:

assumes *sorted ys distinct ys*
shows *subseq xs ys* $\longleftrightarrow$ (*sorted xs* $\wedge$ *distinct xs* $\wedge$ *set xs* $\subseteq$ *set ys*)
⟨*proof*⟩

lemma *subseq_mapE*:
assumes *subseq xs* (*map f ys*)
obtains *xs'* **where** *subseq xs' ys map f xs'* = *xs*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list_all2_fst_aux*:
map fst xs = *ys* **if** *list_all2* (λ*x y*. *fst x* = *y*) *xs ys*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *list_all2_map_eq_iff*:
list_all2 (λ*x y*. *f x* = *g y*) *xs ys* $\longleftrightarrow$ *map f xs* = *map g ys*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*: $state\_set\ (trans\_of\ A) \subseteq S$
assumes *invs*: $\forall\ l \in S.\ inv\_of\ A\ l = inv\_of\ B\ l$
assumes *trans_eq*: $trans\_of\ A = trans\_of\ B$
begin

lemma *delay1*:
$A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle$ **if** $l \in S\ B \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle$
⟨*proof*⟩

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*⟩

lemma *delay2*:
$B \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle$ **if** $l \in S\ A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l',u'\rangle$
⟨*proof*⟩

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*⟩

lemma *step1*:
$A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$ **if** $l \in S\ B \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$
⟨*proof*⟩

lemma *step2*:
$B \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$ **if** $l \in S\ A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$
⟨*proof*⟩

lemma *S_inv*:
$l' \in S$ **if** $A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle\ l \in S$
⟨*proof*⟩

interpretation *interp*: *Bisimulation_Invariant*
$\lambda\ (l,\ u)\ (l',\ u').\ A \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$
$\lambda\ (l,\ u)\ (l',\ u').\ B \vdash \langle l,\ u\rangle \rightarrow \langle l',u'\rangle$
$\lambda\ lu\ lu'.\ lu' = lu\ \lambda\ (l,\ u).\ l \in S\ \lambda\ (l,\ u).\ l \in S$
⟨*proof*⟩

end

end
theory *Simple_Network_Language_Impl*
imports
Simple_Network_Language
Munta_Base.Normalized_Zone_Semantics_Impl_Refine
HOL−Library.IArray HOL−Library.AList
Munta_Base.More_Methods
Munta_Base.Bijective_Embedding
TA_Impl_Misc2
TA_More2

TA_Equivalences
HOL−Eisbach.Eisbach_Tools
Munta_Base.TA_Syntax_Bundles
begin

unbundle *no_library_syntax*

Default maps **definition** *default_map_of* :: $'b \Rightarrow ('a \times 'b)\ list \Rightarrow 'a \Rightarrow 'b$ **where**
default_map_of a xs ≡ *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*)
⟨*proof*⟩

lemma *range_default_map_of*:
range (*default_map_of x xs*) ⊆ *snd* ‘ *set xs* ∪ {*x*}
⟨*proof*⟩

lemma *finite_range_default_map_of*:
finite (*range* (*default_map_of x m*))
⟨*proof*⟩

lemma *map_index'_inj*:
L = *L'*
if *length L* = *length L' map_index' n f L* = *map_index' n g L' set L* ⊆ *S set L'* ⊆ *T*
∀ *i* < *length L* + *n*. ∀ *x* ∈ *S*. ∀ *y* ∈ *T*. *f i x* = *g i y* ⟶ *x* = *y*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *map_index_update*:
map_index f (*xs*[*i* := *a*]) = (*map_index f xs*)[*i* := *f i a*]
⟨*proof*⟩

lemma *map_trans_broad_aux1*:
map_index map_loc (*fold* (λ*p L*. *L*[*p* := *ls' p*]) *ps L*) =
fold (λ*p L*. *L*[*p* := *map_loc p* (*ls' p*)]) *ps* (*map_index map_loc L*)
⟨*proof*⟩

lemma *InD2*:
fixes *map_action*
assumes *inj map_action In* (*map_action a*) = *map_act map_action a'*
shows *a'* = *In a*

⟨*proof*⟩

lemma *OutD2*:
fixes *map_action*
assumes *inj map_action Out* (*map_action a*) = *map_act map_action a′*
shows *a′* = *Out a*
⟨*proof*⟩

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* (λ*p 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* = [])
}
⟨*proof*⟩

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′) ⟨proof⟩

lemma *L_len[intro, dest]*:
length L = n_ps **if** *L ∈ states*
⟨proof⟩

lemma *N_eq*:
‹N i = automaton_of (automata ! i)› **if** *‹i < n_ps›*
⟨proof⟩

end

locale *Simple_Network_Impl* =
fixes *automata* ::
('s list × 's list × ('a act, 's, 'c, int, 'x, int) transition list
× ('s × ('c, int) cconstraint) list) list
and *broadcast* :: *'a list*
and *bounds′* :: *('x × (int × int)) list*
begin

sublocale *Simple_Network_Impl_Defs automata broadcast bounds′ ⟨proof⟩*

end

Mapping through the product construction **lemma** *f_the_inv_f*:
f (the_inv f x) = x **if** *inj f x ∈ range f*
⟨proof⟩

method *fprem* =
(match **premises in** *R*: *_* ⇒ *‹rule R[elim_format]›, assumption)*

context *Simple_Network_Impl*
begin

Conversion from integers to reals commutes with product construction. **sublocale** *conv*: *Prod_TA_Defs*
(set broadcast, map (Simple_Network_Language.conv_A o automaton_of) automata, map_of bounds′) ⟨proof⟩

lemma (**in** −) *conv_ac_inj*:
ac = ac′ **if** *conv_ac ac = conv_ac ac′*
⟨proof⟩

lemma (**in** −) *conv_cc_inj*:
cc = cc′ **if** *conv_cc cc = conv_cc cc′*
⟨proof⟩

context
begin

lemma *conv_alt_def*:
conv (*set broadcast*, *map automaton_of automata*, *map_of bounds'*) =
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
⟨*proof*⟩ **lemma** *2*:
Simple_Network_Language.conv_A o automaton_of = (λ(*committed*, *urgent*, *trans*, *inv*).
(*set committed*,
set urgent,
set (*map Simple_Network_Language.conv_t trans*),
default_map_of [] (*map* (λ (*l*, *g*). (*l*, *conv_cc g*)) *inv*)))
⟨*proof*⟩

lemma *conv_n_ps_eq*:
conv.n_ps = *n_ps*
⟨*proof*⟩

lemma *conv_N_eq*:
conv.N i = *Simple_Network_Language.conv_A* (*N i*) **if** *i* < *n_ps*
⟨*proof*⟩ **lemma** *5*:
inv (*conv.N i*) = *conv_cc o* (*inv* (*N i*)) **if** *i* < *n_ps*
⟨*proof*⟩

lemma *trans_conv_N_eq*:
trans (*conv.N i*) = *Simple_Network_Language.conv_t* ‘ (*trans* (*N i*)) **if** *i* < *n_ps*
⟨*proof*⟩ **lemma** *71*:
(*l*, *b*, *conv_cc g*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*conv.N i*)
if (*l*, *b*, *g*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*N i*) *i* < *n_ps*
⟨*proof*⟩ **lemma** *72*:
(*l*, *b*, *conv_cc g*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*conv.N i*)
⟷ (*l*, *b*, *g*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*N i*) **if** *i* < *n_ps*
⟨*proof*⟩ **lemma** *73*:
∃ *g'*. *g* = *conv_cc g'* ∧ (*l*, *b*, *g'*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*N i*)
if (*l*, *b*, *g*, *a*, *r*, *u*, *l'*)∈*Simple_Network_Language.trans* (*conv.N i*) *i* < *n_ps*
⟨*proof*⟩

lemma *conv_bounds*[*simp*]:
conv.bounds = *bounds*
⟨*proof*⟩

lemma *conv_states*[*simp*]:
conv.states = *states*
⟨*proof*⟩ **lemma** *9*:
committed (*conv.N p*) = *committed* (*N p*) **if** ‹*p* < *n_ps*›
⟨*proof*⟩ **lemma** *10*:
conv.broadcast = *set broadcast*
⟨*proof*⟩

lemma *conv_trans_int*:
conv.trans_int = (λ(*l*, *g*, *a*, *r*, *l'*). (*l*, *map conv_ac g*, *a*, *r*, *l'*)) ‘ *trans_int*
⟨*proof*⟩

lemma *conv_trans_bin*:
conv.trans_bin = (λ(*l*, *g*, *a*, *r*, *l'*). (*l*, *map conv_ac g*, *a*, *r*, *l'*)) ‘ *trans_bin*
⟨*proof*⟩

lemma *n_ps_rangeD*:
$p < n\_ps$ **if** $p \in set\ ps$ $set\ ps \subseteq \{0..<n\_ps\}$
⟨*proof*⟩

lemma *maximalD*:
$(\forall\, g\ f\ r\ l'.$
$(L\ !\ q,\ b,\ g,\ In\ a',\ f,\ r,\ l')$
$\notin (\lambda(l,\ b,\ g,\ a,\ f,\ r,\ l').$
$(l,\ b,\ map\ conv\_ac\ g,\ a,\ f,\ r,\ l'))\ `\ trans\ (N\ q)) \vee \neg\ check\_bexp\ s\ b\ True$ **if**
$\forall\, q{<}n\_ps.\ q \notin set\ ps \wedge p \neq q \longrightarrow (\forall\, b.\ (\forall\, g\ f\ r\ l'.$
$(L\ !\ q,\ b,\ g,\ In\ a',\ f,\ r,\ l') \notin trans\ (N\ q)) \vee \neg\ check\_bexp\ s\ b\ True)$
$q{<}n\_ps\ q \notin set\ ps\ p \neq q$
for *b ps p q L a' s* ⟨*proof*⟩

lemma *conv_trans_broad*:
$conv.trans\_broad = (\lambda(l,\ g,\ a,\ r,\ l').\ (l,\ map\ conv\_ac\ g,\ a,\ r,\ l'))\ `\ trans\_broad$
⟨*proof*⟩

lemma *conv_prod_ta*:
$conv.prod\_ta = Normalized\_Zone\_Semantics\_Impl.conv\_A\ prod\_ta$
⟨*proof*⟩

end

Fundamentals **definition** *clkp_set'* ≡
$(\bigcup\ A \in set\ automata.\ \bigcup\ g \in set\ (snd\ (snd\ (snd\ A))).\ collect\_clock\_pairs\ (snd\ g))$
$\cup\ (\bigcup\ A \in set\ automata.\ \bigcup\ (l,\ b,\ g,\ \_) \in set\ (fst\ (snd\ (snd\ A))).\ collect\_clock\_pairs\ g)$

definition *clk_set'* **where**
‹*clk_set'* =
fst ` *clkp_set'* ∪
$(\bigcup\ A \in set\ automata.\ \bigcup\ (\_,\ \_,\ \_,\ \_,\ \_,\ r,\ \_) \in set\ (fst\ (snd\ (snd\ A))).\ set\ r)$›

lemma (**in** −) *default_map_of_in_listD*:
$x \in \bigcup\ (set\ `\ snd\ `\ set\ invs)$ **if** $x \in set\ (default\_map\_of\ []\ invs\ l)$
⟨*proof*⟩

lemma *collect_clock_pairs_invsI*:
$(a,\ b) \in \bigcup\ ((collect\_clock\_pairs\ o\ snd)\ `\ set\ invs)$
if $(a,\ b) \in collect\_clock\_pairs\ (default\_map\_of\ []\ invs\ l)$
⟨*proof*⟩

lemma *mem_trans_N_iff*:
$t \in Simple\_Network\_Language.trans\ (N\ i) \longleftrightarrow t \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$
if $i < n\_ps$
⟨*proof*⟩

lemma *length_automata_eq_n_ps*:
$length\ automata = n\_ps$
⟨*proof*⟩

lemma *clkp_set'_subs*:

Timed_Automata.clkp_set prod_ta ⊆ *clkp_set'*
⟨*proof*⟩

lemma *collect_clkvt_subs*:
collect_clkvt (*trans_of prod_ta*) ⊆
(⋃ *A* ∈ *set automata*. ⋃ (_, _, _, _, _, *r*, _) ∈ *set* (*fst* (*snd* (*snd A*))). *set r*)
⟨*proof*⟩

lemma *clk_set'_subs*: *clk_set prod_ta* ⊆ *clk_set'*
⟨*proof*⟩

end

lemma (**in** *Prod_TA_Defs*) *finite_range_invI*:
finite (*range prod_inv*) **if** *assms*: ∀ *i* < *n_ps*. *finite* (*range* (*inv* (*N i*)))
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma (**in** *Prod_TA_Defs*) *finite_states*:
assumes *finite_trans*: ∀ *p* < *n_ps*. *finite* (*Simple_Network_Language.trans* (*N p*))
shows *finite states*
⟨*proof*⟩

context *Simple_Network_Impl*
begin

lemma *trans_N_finite*:
assumes *p* < *n_ps*
shows *finite* (*Simple_Network_Language.trans* (*N p*))
⟨*proof*⟩

lemma *states_finite*:
finite states
⟨*proof*⟩

lemma *bounded_finite*:
finite {*s*. *bounded bounds s*}
⟨*proof*⟩

lemma *trans_prod_finite*:
finite trans_prod
⟨*proof*⟩

lemma *prod_inv_finite*:
finite (*range prod_inv*)

⟨*proof*⟩

end

Collecting variables from expressions. **fun** *vars_of_bexp* **and** *vars_of_exp* **where**
vars_of_bexp (*not e*) = *vars_of_bexp e*
| *vars_of_bexp* (*and e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*bexp.or e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*imply e1 e2*) = (*vars_of_bexp e1* ∪ *vars_of_bexp e2*)
| *vars_of_bexp* (*eq i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*le i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*lt i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*ge i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp* (*gt i x*) = *vars_of_exp i* ∪ *vars_of_exp x*
| *vars_of_bexp bexp.true* = {}
| *vars_of_exp* (*const c*) = {}
| *vars_of_exp* (*var x*) = {*x*}
| *vars_of_exp* (*if_then_else b e1 e2*) = *vars_of_bexp b* ∪ *vars_of_exp e1* ∪ *vars_of_exp e2*
| *vars_of_exp* (*binop _ e1 e2*) = *vars_of_exp e1* ∪ *vars_of_exp e2*
| *vars_of_exp* (*unop _ e*) = *vars_of_exp e*

definition (**in** *Prod_TA_Defs*)
var_set =
(⋃ *S* ∈ {(*fst* ∘ *snd*) ‘ *trans* (*N p*) | *p*. *p* < *n_ps*}. ⋃ *b* ∈ *S*. *vars_of_bexp b*) ∪
(⋃ *S* ∈ {(*fst* ∘ *snd* ∘ *snd* ∘ *snd* ∘ *snd*) ‘ *trans* (*N p*) | *p*. *p* < *n_ps*}.
⋃ *f* ∈ *S*. ⋃ (*x*, *e*) ∈ *set f*. {*x*} ∪ *vars_of_exp e*)

locale *Simple_Network_Impl_nat_defs* =
Simple_Network_Impl automata
for *automata* ::
(*nat list* × *nat list* × (*nat act*, *nat*, *nat*, *int*, *nat*, *int*) *transition list*
× (*nat* × (*nat*, *int*) *cconstraint*) *list*) *list* +
fixes *m* :: *nat* **and** *num_states* :: *nat* ⇒ *nat* **and** *num_actions* :: *nat*

locale *Simple_Network_Impl_nat* =
Simple_Network_Impl_nat_defs +
assumes *has_clock*: *m* > *0*
assumes *non_empty*: *0* < *length automata*

assumes *trans_num_states*:
∀ *i* < *n_ps*. *let* (_, _, *trans*, _) = (*automata* ! *i*) *in* ∀ (*l*, _, _, _, _, _, *l′*) ∈ *set trans*.
l < *num_states i* ∧ *l′* < *num_states i*
and *inv_num_states*:
∀ *i* < *n_ps*. *let* (_, _, _, *inv*) = (*automata* ! *i*) *in* ∀ (*x*, _) ∈ *set inv*. *x* < *num_states i*
assumes *var_set*:
∀ (_, _, *trans*, _) ∈ *set automata*. ∀ (_, _, _, _, *f*, _, _) ∈ *set trans*.
∀ (*x*, *upd*) ∈ *set f*. *x* < *n_vs* ∧ (∀ *i* ∈ *vars_of_exp upd*. *i* < *n_vs*)
∀ (_, _, *trans*, _) ∈ *set automata*. ∀ (_, *b*, _, _, _, _, _) ∈ *set trans*.
∀ *i* ∈ *vars_of_bexp b*. *i* < *n_vs*
assumes *bounds*:
∀ *i* < *n_vs*. *fst* (*bounds′* ! *i*) = *i*
assumes *action_set*:
∀ *a* ∈ *set broadcast*. *a* < *num_actions*

$\forall (\_, \_, trans, \_) \in set\ automata.\ \forall (\_, \_, \_, a, \_, \_, \_) \in set\ trans.$
$pred\_act\ (\lambda a.\ a < num\_actions)\ a$
assumes *clock_set*:
$\forall (\_, \_, trans, \_) \in set\ automata.\ \forall (\_, \_, g, \_, \_, r, \_) \in set\ trans.$
$(\forall\, c \in set\ r.\ 0 < c \wedge c \leq m) \wedge$
$(\forall\ (c, x) \in collect\_clock\_pairs\ g.\ 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$

$\forall (\_, \_, \_, inv) \in set\ automata.\ \forall (l, g) \in set\ inv.$
$(\forall\ (c, x) \in collect\_clock\_pairs\ g.\ 0 < c \wedge c \leq m \wedge x \in \mathbb{N})$

assumes *broadcast_receivers*:
$\forall (\_, \_, trans, \_) \in set\ automata.\ \forall (\_, \_, g, a, \_, \_, \_) \in set\ trans.$
$case\ a\ of\ In\ a \Rightarrow a \in set\ broadcast \longrightarrow g = [] \mid \_ \Rightarrow True$
begin

lemma *broadcast_receivers_unguarded*:
$\forall p{<}n\_ps.\ \forall l\ b\ g\ a\ f\ r\ l'.$
$(l, b, g, In\ a, f, r, l') \in Simple\_Network\_Language.trans\ (N\ p) \wedge a \in set\ broadcast \longrightarrow g = []$
⟨*proof*⟩

sublocale *conv*: *Prod_TA*
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
⟨*proof*⟩

sublocale *TA_Start_No_Ceiling prod_ta init m*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *sem_N_eq*:
sem.N p = *automaton_of* (*conv_automaton* (*automata* ! *p*)) **if** ‹*p* < *n_ps*›
⟨*proof*⟩

end

inductive_cases *step_u_elims*:
$A \vdash \langle L, s, u\rangle \rightarrow_{Del} \langle L', s', u'\rangle$
$A \vdash \langle L, s, u\rangle \rightarrow_{Internal\ a} \langle L', s', u'\rangle$
$A \vdash \langle L, s, u\rangle \rightarrow_{Bin\ a} \langle L', s'', u'\rangle$
$A \vdash \langle L, s, u\rangle \rightarrow_{Broad\ a} \langle L', s'', u'\rangle$

inductive_cases *step_u_elims'*:
$(broadcast, N, B) \vdash \langle L, s, u\rangle \rightarrow_{Del} \langle L', s', u'\rangle$

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

lemma (**in** *Prod_TA_Defs*) *states_lengthD*:
$length\ L = n\_ps$ **if** $L \in states$
⟨*proof*⟩

end
theory *Simple_Network_Language_Impl_Refine*
imports *Simple_Network_Language_Impl*
begin

unbundle *no_library_syntax*
notation *fun_rel_syn* (**infixr** $\rightarrow$ 60)
hide_const (**open**) *upd*

Expression evaluation **fun** *bval* :: $('a \Rightarrow 'b) \Rightarrow ('a,\ 'b :: linorder)\ bexp \Rightarrow bool$ **and** *eval*
where
$bval\ \_\ bexp.true \longleftrightarrow True$ |
$bval\ s\ (not\ e) \longleftrightarrow \neg\ bval\ s\ e$ |
$bval\ s\ (and\ e1\ e2) \longleftrightarrow bval\ s\ e1 \wedge bval\ s\ e2$ |
$bval\ s\ (bexp.or\ e1\ e2) \longleftrightarrow bval\ s\ e1 \vee bval\ s\ e2$ |
$bval\ s\ (imply\ e1\ e2) \longleftrightarrow bval\ s\ e1 \longrightarrow bval\ s\ e2$ |
$bval\ s\ (eq\ i\ x) \longleftrightarrow eval\ s\ i = eval\ s\ x$ |
$bval\ s\ (le\ i\ x) \longleftrightarrow eval\ s\ i \leq eval\ s\ x$ |
$bval\ s\ (lt\ i\ x) \longleftrightarrow eval\ s\ i < eval\ s\ x$ |
$bval\ s\ (ge\ i\ x) \longleftrightarrow eval\ s\ i \geq eval\ s\ x$ |
$bval\ s\ (gt\ i\ x) \longleftrightarrow eval\ s\ i > eval\ s\ x$
| $eval\ s\ (const\ c) = c$
| $eval\ s\ (var\ x) \quad = s\ x$
| $eval\ s\ (if\_then\_else\ b\ e1\ e2) = (if\ bval\ s\ b\ then\ eval\ s\ e1\ else\ eval\ s\ e2)$
| $eval\ s\ (binop\ f\ e1\ e2) = f\ (eval\ s\ e1)\ (eval\ s\ e2)$
| $eval\ s\ (unop\ f\ e) = f\ (eval\ s\ e)$

lemma *check_bexp_determ*:
$check\_bexp\ s\ b\ b1 \Longrightarrow check\_bexp\ s\ b\ b2 \Longrightarrow b1 = b2$
and *is_val_determ*: $is\_val\ s\ e\ v1 \Longrightarrow is\_val\ s\ e\ v2 \Longrightarrow v1 = v2$
⟨*proof*⟩

lemma *is_upd_determ*:
$s1 = s2$ **if** $is\_upd\ s\ f\ s1\ is\_upd\ s\ f\ s2$
⟨*proof*⟩

lemma *is_upds_determ*:
$s1 = s2$ **if** $is\_upds\ s\ fs\ s1\ is\_upds\ s\ fs\ s2$
⟨*proof*⟩

lemma *check_bexp_bval*:
$\forall x \in vars\_of\_bexp\ b.\ x \in dom\ s \Longrightarrow check\_bexp\ s\ b\ (bval\ (the\ o\ s)\ b)$
and *is_val_eval*:
$\forall x \in vars\_of\_exp\ e.\ x \in dom\ s \Longrightarrow is\_val\ s\ e\ (eval\ (the\ o\ s)\ e)$
⟨*proof*⟩

lemma *is_upd_dom*:
dom $s' = dom\ s$ **if** *is_upd* $s\ (x,\ e)\ s'\ x \in dom\ s$
⟨*proof*⟩

lemma *is_upds_dom*:
dom $s' = dom\ s$ **if** *is_upds* $s\ upds\ s' \forall (x,\ e) \in set\ upds.\ x \in dom\ s$
⟨*proof*⟩

definition
$mk\_upd \equiv \lambda(x,\ e)\ s.\ s(x \mapsto eval\ (the\ o\ s)\ e)$

definition *mk_upds* ::
$('a \Rightarrow ('b :: linorder)\ option) \Rightarrow ('a \times ('a,\ 'b)\ exp)\ list \Rightarrow ('a \Rightarrow 'b\ option)$ **where**
mk_upds $s\ upds = fold\ mk\_upd\ upds\ s$

lemma *is_upd_make_updI*:
is_upd $s\ upd\ (mk\_upd\ upd\ s)$ **if** $upd = (x,\ e) \forall x \in vars\_of\_exp\ e.\ x \in dom\ s$
⟨*proof*⟩

lemma *dom_mk_upd*:
$dom\ s \subseteq dom\ (mk\_upd\ upd\ s)$
⟨*proof*⟩

lemma *is_upds_make_updsI*:
is_upds $s\ upds\ (mk\_upds\ s\ upds)$ **if** $\forall\ (\_,\ e) \in set\ upds. \forall x \in vars\_of\_exp\ e.\ x \in dom\ s$
⟨*proof*⟩

Implementation auxiliaries **definition**
union_map_of $xs =$
$fold\ (\lambda\ (x,\ y)\ m.\ case\ m\ x\ of\ None \Rightarrow m(x \mapsto [y]) \mid Some\ ys \Rightarrow m(x \mapsto y\ \#\ ys))\ xs\ Map.empty$

lemma *union_map_of_alt_def*:
union_map_of $xs\ x = (let$
$ys = rev\ (map\ snd\ (filter\ (\lambda\ (x',\ y).\ x' = x)\ xs))$
$in\ if\ ys = []\ then\ None\ else\ Some\ ys)$
⟨*proof*⟩

lemma *in_union_map_ofI*:
$\exists\ ys.\ union\_map\_of\ xs\ x = Some\ ys \wedge y \in set\ ys$ **if** $(x,\ y) \in set\ xs$
⟨*proof*⟩

lemma *in_union_map_ofD*:
$(x,\ y) \in set\ xs$ **if** *union_map_of* $xs\ x = Some\ ys\ y \in set\ ys$
⟨*proof*⟩

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*]:

$L \in states \longleftrightarrow length\ L = n\_ps \wedge (\forall i{<}n\_ps.\ L\ !\ i \in states\_i\ i)$
⟨*proof*⟩

lemma *states_mem_compute′*:
$L \in states \longleftrightarrow length\ L = n\_ps \wedge (\forall i{<}n\_ps.\ L\ !\ i \in map\ states\_i\ [0..{<}n\_ps]\ !\ i)$
⟨*proof*⟩

end

context *Simple_Network_Impl_nat*
begin

Fundamentals **lemma** *mem_trans_N_iff*:
‹$t \in Simple\_Network\_Language.trans\ (N\ i) \longleftrightarrow t \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$›
if $i < n\_ps$
⟨*proof*⟩

lemma *L_i_len*:
‹$L\ !\ i < num\_states\ i$› **if** $i < n\_ps\ L \in states$
⟨*proof*⟩

lemma *L_i_simp*:
‹$[0..{<}num\_states\ i]\ !\ (L\ !\ i) = L\ !\ i$›
if $i < n\_ps\ L \in states$
⟨*proof*⟩

lemma *action_setD*:
‹$pred\_act\ (\lambda a'.\ a' < num\_actions)\ a$›
if ‹$(l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$› ‹$p < n\_ps$›
⟨*proof*⟩

More precise state sets **definition**
$states' \equiv \{(L,\ s).\ L \in states \wedge dom\ s = \{0..{<}n\_vs\} \wedge bounded\ bounds\ s\}$

lemma *states′_states*[*intro*, *dest*]:
$L \in states$ **if** $(L,\ s) \in states'$
⟨*proof*⟩

lemma *states′_dom*[*intro*, *dest*]:
$dom\ s = \{0..{<}n\_vs\}$ **if** $(L,\ s) \in states'$
⟨*proof*⟩

lemma *states′_bounded*[*intro*, *dest*]:
$bounded\ bounds\ s$ **if** $(L,\ s) \in states'$
⟨*proof*⟩

Implementation of invariants **definition** (**in** *Simple_Network_Impl_nat_defs*)
$invs\ i \equiv let\ m = default\_map\_of\ []\ (snd\ (snd\ (snd\ (automata\ !\ i))));$
$m' = map\ (\lambda\ j.\ m\ j)\ [0..{<}num\_states\ i]$
$in\ m'$

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*
⟨*proof*⟩

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*›
⟨*proof*⟩

lemma *invs_simp*:
invs1 ! *i* ! (*L* ! *i*) = *Simple_Network_Language.inv* (*N i*) (*L* ! *i*)
if *i* < *n_ps L* ∈ *states*
⟨*proof*⟩

lemma *inv_fun_inv_of′*:
(*inv_fun*, *inv_of prod_ta*) ∈ *inv_rel R states′* **if** *R* ⊆ *Id* ×*r S*
⟨*proof*⟩

lemma *inv_fun_alt_def*:
inv_fun = (λ(*L*, _). *concat* (*map* (λ*i*. *invs2* !! *i* !! (*L* ! *i*)) [*0*..<*n_ps*]))
⟨*proof*⟩

end

Implementation of transitions **context** *Simple_Network_Impl_nat_defs*
begin

definition
bounds_map ≡ *the o 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 (λj.
case m j of None ⇒ [] | Some xs ⇒ xs)

Filter for internal transitions.

definition
trans_i_map i j ≡
List.map_filter
(λ (b, g, a, m, l′). case a of Sil a ⇒ Some (b, g, a, m, l′) | _ ⇒ None)
(trans_map i j)

Compute valid internal successors given:

- a process *p*,
- initial location *l*,
- location vector *L*,
- and initial state *s*.

definition
int_trans_from_loc p l L s ≡
let trans = trans_i_map p l
in
List.map_filter (λ (b, g, a, f, r, l′).
let s′ = mk_upds s f in
if bval (the o s) b ∧ check_bounded s′ then Some (g, Internal a, r, (L[p := l′], s′))
else None
) trans

definition
int_trans_from_vec pairs L s ≡
concat (map (λ(p, l). int_trans_from_loc p l L s) pairs)

definition
int_trans_from_all L s ≡
concat (map (λp. int_trans_from_loc p (L ! p) L s) [0..<n_ps])

definition
int_trans_from ≡ *λ (L, s).*
let pairs = get_committed L in
if pairs = []
then int_trans_from_all L s
else int_trans_from_vec pairs L s

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

definition
all_actions_by_state t L ≡

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* (λ*a*. *a* ∉ *set broadcast*) [*0*..<*num_actions*]

lemma *mem_bin_actions_iff*:
a ∈ *set bin_actions* ⟷ *a* ∉ *local.broadcast* ∧ *a* < *num_actions*
⟨*proof*⟩

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* (λ*a*. *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* (λ*a*. *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
  ⟨proof⟩

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])
```

⟨*proof*⟩

Correctness for implementations of primitives **lemma** *dom_bounds_eq*:
dom bounds = {*0*..<*n_vs*}
⟨*proof*⟩

lemma *check_bounded_iff*:
Simple_Network_Language.bounded bounds s ⟷ *check_bounded s* **if** *dom s* = {*0*..<*n_vs*}
⟨*proof*⟩

lemma *get_committed_mem_iff*:
(*p*, *l*) ∈ *set* (*get_committed L*) ⟷ (*l* = *L* ! *p* ∧ *l* ∈ *committed* (*N p*) ∧ *p* < *n_ps*)
⟨*proof*⟩

lemma *get_committed_empty_iff*:
(∀ *p* < *n_ps*. *L* ! *p* ∉ *committed* (*N p*)) ⟷ *get_committed L* = []
⟨*proof*⟩

lemma *get_committed_distinct*: *distinct* (*get_committed L*)
⟨*proof*⟩

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*}
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *is_updsD*:
  $s' = mk\_upds\ s\ f$ **if** $is\_upds\ s\ f\ s'$
  $(l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)\ p < n\_ps$
  $dom\ s = \{0..<n\_vs\}$
⟨*proof*⟩

lemma *is_upds_make_upds_concatI2*:
  $is\_upds\ s\ (concat\ upds)\ (mk\_upds\ s\ (concat\ upds))$
  if $dom\ s = \{0..<n\_vs\}$
    $\forall\, upd \in set\ upds.\ \exists\, p < n\_ps.\ \exists\, l\ b\ g\ a\ r\ l'.$
      $(l,\ b,\ g,\ a,\ upd,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
  ⟨*proof*⟩

lemma *is_upds_concat_dom2*:
  assumes $is\_upds\ s\ (concat\ upds)\ s'$
    and $\forall\, f \in set\ upds.\ \exists\ p < n\_ps.\ \exists\ l\ b\ g\ a\ r\ l'.$
      $(l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
    and $dom\ s = \{0..<n\_vs\}$
  shows $dom\ s' = dom\ s$
  ⟨*proof*⟩

lemma *is_upds_dom3*:
  assumes $is\_upds\ s\ (concat\_map\ fs\ ps)\ s'$
    and $\forall\, p{\in}set\ ps.\ (L\ !\ p,\ bs\ p,\ gs\ p,\ a,\ fs\ p,\ rs\ p,\ ls'\ p) \in trans\ (N\ p)$
    and $set\ ps \subseteq \{0..<n\_ps\}$
    and $dom\ s = \{0..<n\_vs\}$
  shows $dom\ s' = dom\ s$
  ⟨*proof*⟩

lemma *is_upds_make_updsI3*:
  $is\_upds\ s\ (concat\_map\ fs\ ps)\ (mk\_upds\ s\ (concat\_map\ fs\ ps))$
  if $dom\ s = \{0..<n\_vs\}$
    and $\forall\, p{\in}set\ ps.\ (L\ !\ p,\ bs\ p,\ gs\ p,\ a,\ fs\ p,\ rs\ p,\ ls'\ p) \in trans\ (N\ p)$
    and $set\ ps \subseteq \{0..<n\_ps\}$
  for $s :: nat \Rightarrow int\ option$
  ⟨*proof*⟩

lemma *is_upds_concatD*:
  assumes
    $dom\ s = \{0..<n\_vs\}$ **and**
    $\forall\, p{\in}set\ ps.$
      $(ls\ p,\ bs\ p,\ gs\ p,\ a,\ fs\ p,\ rs\ p,\ ls'\ p)$
      $\in Simple\_Network\_Language.trans\ (N\ p)$ **and**
    $set\ ps \subseteq \{0..<n\_ps\}$ **and**
    $is\_upds\ s\ (concat\_map\ fs\ ps)\ s'$
  shows $s' = mk\_upds\ s\ (concat\_map\ fs\ ps)$
⟨*proof*⟩

context
  notes [*simp*] = *length_automata_eq_n_ps*
begin

lemma *trans_mapI*:
assumes
$(L\ !\ p,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
$p < n\_ps$
shows
$(g,\ a,\ f,\ r,\ l') \in set\ (trans\_map\ p\ (L\ !\ p))$
⟨*proof*⟩

lemma *trans_i_mapI*:
assumes
$(L\ !\ p,\ b,\ g,\ Sil\ a',\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
$p < n\_ps$
shows
$(b,\ g,\ a',\ f,\ r,\ l') \in set\ (trans\_i\_map\ p\ (L\ !\ p))$
⟨*proof*⟩

lemma *trans_mapI'*:
assumes
$(l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
$p < n\_ps$
shows
$(b,\ g,\ a,\ f,\ r,\ l') \in set\ (trans\_map\ p\ l)$
⟨*proof*⟩

lemma *trans_mapD*:
assumes
$(b,\ g,\ a,\ f,\ r,\ l') \in set\ (trans\_map\ p\ l)$
$p < n\_ps$
shows
$(l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
⟨*proof*⟩

lemma *trans_map_iff*:
assumes
$p < n\_ps$
shows
$(b,\ g,\ a,\ f,\ r,\ l') \in set\ (trans\_map\ p\ l)$
$\longleftrightarrow (l,\ b,\ g,\ a,\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
⟨*proof*⟩

lemma *trans_i_mapD*:
assumes
$(b,\ g,\ a',\ f,\ r,\ l') \in set\ (trans\_i\_map\ p\ (L\ !\ p))$
$p < n\_ps$
shows
$(L\ !\ p,\ b,\ g,\ Sil\ a',\ f,\ r,\ l') \in Simple\_Network\_Language.trans\ (N\ p)$
⟨*proof*⟩

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** $L\ !\ p\ \in committed\ (N\ p)\ p < n\_ps$

⟨*proof*⟩

lemma *check_bexp_bvalI*:
bval (*the o s*) *b* **if** *check_bexp s b True*
(*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
dom s = {*0..<n_vs*}
⟨*proof*⟩

lemma *check_bexp_bvalD*:
check_bexp s b True **if** *bval* (*the o s*) *b*
(*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *Simple_Network_Language.trans* (*N p*) *p* < *n_ps*
dom s = {*0..<n_vs*}
⟨*proof*⟩

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'*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *length_actions_by_state_preserv*[*simp*]:
shows *length* (*actions_by_state y xs acc*) = *length acc*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *length_all_actions_by_state_preserv*:
length (*all_actions_by_state M L*) = *num_actions*
⟨*proof*⟩

lemma *length_actions_by_state'_preserv*:
length (*actions_by_state'* *xs*) = *num_actions*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

Binary transitions **lemma** *bin_trans_from_correct*:
(*bin_trans_from*, *trans_bin*) ∈ *transition_rel* *states'*
⟨*proof*⟩

Broadcast transitions **lemma** *make_combs_alt_def*:
make_combs *p* *a* *xs* ≡
let
ys =
map (λ *i*. *map* (λ*t*. (*i*, *t*)) (*xs* ! *i* ! *a*))
(*filter*
(λ*i*. *xs* ! *i* ! *a* ≠ [] ∧ *i* ≠ *p*)
[*0*..<*n_ps*])
in *if* *ys* = [] *then* [] *else* *product_lists* *ys*
⟨*proof*⟩

lemma *list_all2_fst_aux*:
map *fst* *xs* = *ys* **if** *list_all2* (λ*x* *y*. *fst* *x* = *y*) *xs* *ys*
⟨*proof*⟩

lemma *broad_trans_from_correct*:
(*broad_trans_from*, *trans_broad*) ∈ *transition_rel* *states'*
⟨*proof*⟩

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*:

$\exists L\ s.\ ((Li,\ si),\ (L,\ s)) \in loc\_rel$ **if** *length* $Li = n\_ps$ *length* $si = n\_vs$
⟨*proof*⟩

lemma *state_rel_left_unique*:
$l \in states' \Longrightarrow (li,\ l) \in loc\_rel \Longrightarrow (li',\ l) \in loc\_rel \Longrightarrow li' = li$
⟨*proof*⟩

lemma *state_rel_right_unique*:
$l \in states' \Longrightarrow l' \in states' \Longrightarrow (li,\ l) \in loc\_rel \Longrightarrow (li,\ l') \in loc\_rel \Longrightarrow l' = l$
⟨*proof*⟩

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
  ⟨proof⟩

lemma mk_upds_mk_updsi:
  state_rel (mk_upds s upds) (mk_updsi si upds)
  if assms: state_rel s si ∀ (_, e) ∈ set upds. ∀ x ∈ vars_of_exp e. x < n_vs
    ∀ (x, e) ∈ set upds. x < n_vs
⟨proof⟩

lemma check_bounded_check_boundedi:
  check_bounded s = check_boundedi si if state_rel s si
  ⟨proof⟩
```

definition
valid_upd $\equiv \lambda(x,\ e).\ x < n\_vs \wedge (\forall\, x \in vars\_of\_exp\ e.\ x < n\_vs)$

definition
valid_check $b \equiv (\forall\, x \in vars\_of\_bexp\ b.\ x < n\_vs)$

context includes *lifting_syntax* **begin**
notation *rel_prod* (**infixr** $\times_R$ *56*)

definition *is_at_least_equality* **where**
is_at_least_equality $R \equiv \forall\, x\ y.\ R\ x\ y \longrightarrow x = y$ **for** R

named_theorems *is_at_least_equality*

lemma [*is_at_least_equality*]:
is_at_least_equality (=)
⟨*proof*⟩

lemma [*is_at_least_equality*]:
is_at_least_equality R **if** *is_equality* R **for** R
⟨*proof*⟩

lemma [*is_at_least_equality*]:
is_at_least_equality (*eq_onp* P)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *is_at_least_equality_rel_prod*[*is_at_least_equality*]:
is_at_least_equality (*R1* $\times_R$ *R2*)
if *is_at_least_equality R1 is_at_least_equality R2* **for** *R1 R2*
⟨*proof*⟩

lemma *is_at_least_equality_cong1*:
(*S1* ===> (=)) *f f* **if** *is_at_least_equality S1 is_at_least_equality S2* **for** *S1 f*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is_at_least_equality_Let*:
(*S* ===> ((=) ===> *R*) ===> *R*) *Let Let* **if** *is_at_least_equality S* **for** *R*
⟨*proof*⟩

lemma *map_transfer_length*:

fixes R S n
shows
$((R ===> S)$
$===> (\lambda x\ y.\ list\_all2\ R\ x\ y \wedge length\ x = n)$
$===> (\lambda x\ y.\ list\_all2\ S\ x\ y \wedge length\ x = n))$
$map\ map$
⟨*proof*⟩

lemma *upt_0_transfer*:
$(eq\_onp\ (\lambda x.\ x = 0) ===> eq\_onp\ (\lambda x.\ x = n) ===> list\_all2\ (eq\_onp\ (\lambda x.\ x < n)))\ upt\ upt$ **for** n
⟨*proof*⟩

lemma *upt_length_transfer*:
$(eq\_onp\ (\lambda x.\ x = 0) ===> eq\_onp\ (\lambda x.\ x = n)$
$===> (\lambda\ x\ y.\ list\_all2\ (eq\_onp\ (\lambda x.\ x < n))\ x\ y \wedge length\ x = n))\ upt\ upt$ **for** n
⟨*proof*⟩

lemma *case_prod_transfer_strong*:
fixes A B C
assumes $\bigwedge x\ y.\ A1\ x\ y \Longrightarrow A\ x\ y \bigwedge x\ y.\ B1\ x\ y \Longrightarrow B\ x\ y$
shows $((A ===> B ===> C) ===> A1 \times_R B1 ===> C)\ case\_prod\ case\_prod$
⟨*proof*⟩

lemma *concat_transfer_strong*:
fixes A B C
assumes $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y \bigwedge x\ y.\ C\ x\ y \Longrightarrow list\_all2\ (list\_all2\ A)\ x\ y$
shows $(C ===> list\_all2\ B)\ concat\ concat$
⟨*proof*⟩

lemma *map_transfer_strong*:
fixes A B C
assumes $\bigwedge xs\ ys.\ C\ xs\ ys \Longrightarrow list\_all2\ A\ xs\ ys$
shows $((A ===> B) ===> C ===> list\_all2\ B)\ map\ map$
⟨*proof*⟩

lemma *list_update_transfer'*:
fixes $A :: 'a \Rightarrow 'b \Rightarrow bool$
shows $(list\_all2\ A ===> eq\_onp\ (\lambda i.\ i<n\_ps) ===> A ===> list\_all2\ A)\ list\_update\ list\_update$
⟨*proof*⟩

lemma *list_update_transfer''*:
fixes $A :: 'a \Rightarrow 'b \Rightarrow bool$ **and** n
shows $((\lambda\ x\ y.\ list\_all2\ A\ x\ y \wedge length\ x = n) ===> eq\_onp\ (\lambda i.\ i<n) ===> A$
$===> (\lambda\ x\ y.\ list\_all2\ A\ x\ y \wedge length\ x = n))\ list\_update\ list\_update$
⟨*proof*⟩

lemma *list_update_transfer'''*:
fixes $A :: 'a \Rightarrow 'b \Rightarrow bool$ **and** n
shows $((\lambda\ x\ y.\ list\_all2\ A\ x\ y \wedge length\ x = n) ===> (=) ===> A$
$===> (\lambda\ x\ y.\ list\_all2\ A\ x\ y \wedge length\ x = n))\ list\_update\ list\_update$
⟨*proof*⟩

lemma *fold_transfer_strong*:
fixes *A B*
assumes $\bigwedge x\ y.\ A1\ x\ y \Longrightarrow A\ x\ y\ \bigwedge x\ y.\ B1\ x\ y \Longrightarrow B\ x\ y\ \bigwedge x\ y.\ B\ x\ y \Longrightarrow B2\ x\ y$
$\bigwedge x\ y.\ B\ x\ y \Longrightarrow B3\ x\ y$
shows $((A\ ===>\ B2\ ===>\ B1)\ ===>\ list\_all2\ A1\ ===>\ B\ ===>\ B3)\ fold\ fold$
⟨*proof*⟩

lemma *bval_bvali_transfer*[*transfer_rule*]:
$(state\_rel\ ===>\ eq\_onp\ valid\_check\ ===>\ (=))\ (\lambda\ s.\ bval\ (the\ o\ s))\ bvali$
⟨*proof*⟩

lemma *mk_upds_mk_updsi_transfer*[*transfer_rule*]:
$(state\_rel\ ===>\ list\_all2\ (eq\_onp\ valid\_upd)\ ===>\ state\_rel)\ mk\_upds\ mk\_updsi$
⟨*proof*⟩

lemma *check_bounded_transfer*[*transfer_rule*]:
$(state\_rel\ ===>\ (=))\ check\_bounded\ check\_boundedi$
⟨*proof*⟩

lemma *trans_map_transfer*:
$(eq\_onp\ (\lambda i.\ i<n\_ps)\ ===>\ (=)\ ===>$
$list\_all2\ ($
$eq\_onp\ valid\_check\ \times_R\ (=)\ \times_R\ eq\_onp\ (pred\_act\ (\lambda x.\ x < num\_actions))$
$\times_R\ list\_all2\ (eq\_onp\ valid\_upd)\ \times_R\ (=)$
$))\ trans\_map\ trans\_map$
⟨*proof*⟩

lemma *trans_map_transfer′*:
$(eq\_onp\ (\lambda i.\ i<n\_ps)\ ===>\ (=)\ ===>$
$list\_all2\ (eq\_onp\ valid\_check\ \times_R\ (=)\ \times_R\ (=)\ \times_R\ list\_all2\ (eq\_onp\ valid\_upd)\ \times_R\ (=))$
$)\ trans\_map\ trans\_map$
⟨*proof*⟩

lemma *map_filter_transfer*[*transfer_rule*]:
$((S\ ===>\ rel\_option\ R)\ ===>\ list\_all2\ S\ ===>\ list\_all2\ R)\ List.map\_filter\ List.map\_filter$
⟨*proof*⟩

lemma *trans_i_map_transfer*[*transfer_rule*]:
$(eq\_onp\ (\lambda i.\ i<n\_ps)\ ===>\ (=)\ ===>$
$list\_all2\ (eq\_onp\ valid\_check\ \times_R\ (=)\ \times_R\ (=)\ \times_R\ list\_all2\ (eq\_onp\ valid\_upd)\ \times_R\ (=))$
$)\ trans\_i\_map\ trans\_i\_map$
⟨*proof*⟩

lemma *int_trans_from_loc_transfer*[*transfer_rule*]:
$(eq\_onp\ (\lambda i.\ i<n\_ps)\ ===>\ (=)\ ===>\ (\lambda x\ y.\ list\_all2\ (=)\ x\ y \wedge length\ x = n\_ps)\ ===>$
$state\_rel$
$===>\ list\_all2((=)\ \times_R\ (=)\ \times_R\ (=)\ \times_R\ (\lambda x\ y.\ list\_all2\ (=)\ x\ y \wedge length\ x = n\_ps)\ \times_R$
$state\_rel))$
$int\_trans\_from\_loc\ int\_trans\_from\_loc\_impl$
⟨*proof*⟩

lemma *n_ps_transfer*:
$eq\_onp\ (\lambda x.\ x = n\_ps)\ n\_ps\ n\_ps$
⟨*proof*⟩

lemma *zero_nat_transfer*:
(=) *0* (*0*::*nat*)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩ **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*⟩

lemma *eq_transfer*:
(*list_all2* (*eq_onp* (λ*x*. *x* < *n_ps*) ×$_R$ (=)) ===> *list_all2* (=) ===> (=)) (=) (=)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemmas *rel_elims* =
rel_prod.cases
rel_funD

lemmas *rel_intros* =

rel_funI

lemma *pairs_by_action_transfer'*:
((λx 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 ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ (λx 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
⟨*proof*⟩

lemma *trans_in_map_transfer*[*transfer_rule*]:
(*eq_onp* (λi. i<*n_ps*) ===> (=)
===> *list_all2* (
eq_onp valid_check ×$_R$ (=) ×$_R$ *eq_onp* (λa. a < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=))
) *trans_in_map trans_in_map*
⟨*proof*⟩

lemma *trans_in_map_transfer*[*transfer_rule*]:
(*eq_onp* (λi. 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*
⟨*proof*⟩

lemma *trans_out_map_transfer*[*transfer_rule*]:
(*eq_onp* (λi. i<*n_ps*) ===> (=)
===> *list_all2* (
eq_onp valid_check ×$_R$ (=) ×$_R$ *eq_onp* (λa. a < *num_actions*) ×$_R$ *list_all2* (*eq_onp valid_upd*) ×$_R$ (=)
)) *trans_out_map trans_out_map*
⟨*proof*⟩

lemma *trans_out_map_transfer*[*transfer_rule*]:
(*eq_onp* (λi. 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
⟨*proof*⟩

lemma *actions_by_state_transfer*[*transfer_rule*]:
(*eq_onp* (λi. i < *n_ps*) ===>
list_all2 ((=) ×$_R$ (=) ×$_R$ *eq_onp* (λi. i < n) ×$_R$ (=)) ===>
(λ x y. *list_all2* (*list_all2* (*eq_onp* (λi. i<*n_ps*) ×$_R$ (=) ×$_R$ (=) ×$_R$ *eq_onp* (λx. x < n) ×$_R$ (=))) x y ∧ *length* x = n) ===>
(λ x y. *list_all2* (*list_all2* (*eq_onp* (λi. i<*n_ps*) ×$_R$ (=) ×$_R$ (=) ×$_R$ *eq_onp* (λx. x < n) ×$_R$ (=))) x y ∧ *length* x = n)
)
actions_by_state actions_by_state **for** n

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) ×$_R$ (=)))
===>
(λ*x y*. *list_all2* (=) *x y* ∧ *length x* = *n_ps*)
===>
(λ *x y*. *list_all2* (*list_all2* (*eq_onp* (λ*i*. *i*<*n_ps*) ×$_R$ *eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*i*. *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
⟨*proof*⟩

lemma *all_actions_from_vec_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*) ×$_R$ (=)))
===>
list_all2 (*eq_onp* (λ*i*. *i* < *n_ps*) ×$_R$ (=))
===>
(λ *x y*. *list_all2* (*list_all2* (*eq_onp* (λ*i*. *i*<*n_ps*) ×$_R$ *eq_onp valid_check* ×$_R$ (=) ×$_R$ *eq_onp* (λ*i*. *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
⟨*proof*⟩

lemma *bin_actions_transfer*[*transfer_rule*]:
(*list_all2* (*eq_onp* (λ*x*. *x* < *num_actions*))) *bin_actions bin_actions*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *bin_trans_from_transfer*:
((λx y. list_all2 (=) x y ∧ length x = n_ps) ×$_R$ state_rel
===> list_all2 ((=) ×$_R$ (=) ×$_R$ (=) ×$_R$ ((λx y. list_all2 (=) x y ∧ length x = n_ps) ×$_R$
state_rel)))
bin_trans_from bin_trans_from_impl
⟨*proof*⟩

lemma *trans_map_transfer''*:
((=) ===> (=) ===>
list_all2 (eq_onp valid_check ×$_R$ (=) ×$_R$ eq_onp (pred_act (λx. x < num_actions)) ×$_R$
list_all2 (eq_onp valid_upd) ×$_R$ (=)))
trans_map trans_map
⟨*proof*⟩

lemma *compute_upds_transfer*:
(
(list_all2 (=) ×$_R$ (=) ×$_R$ list_all2 (=) ×$_R$ (λx y. list_all2 (=) x y ∧ length x = n) ×$_R$
state_rel)
===> list_all2 (list_all2
((=) ×$_R$ eq_onp valid_check ×$_R$ list_all2 (=) ×$_R$ (=) ×$_R$ list_all2 (eq_onp valid_upd)
×$_R$ list_all2 (=) ×$_R$ (=))) ===>
list_all2 (
list_all2 (=) ×$_R$ (=) ×$_R$ list_all2 (=) ×$_R$ (λx y. list_all2 (=) x y ∧ length x = n) ×$_R$
state_rel
)) compute_upds compute_upds_impl
⟨*proof*⟩

lemma *in_transfer*:
(eq_onp (λx. x < num_actions) ===> (=) ===> (=)) (∈) (∈)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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′
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

```
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])
  )
```

⟨*proof*⟩

lemma *concat_length_transfer*:
((λ *x y*. *list_all2* (*list_all2 A*) *x y* $\wedge$ *length x* = *n*) ===> *list_all2 A*) *concat concat* **for** *A n*
⟨*proof*⟩

lemma *broad_trans_from_transfer*:
((λx *y*. *list_all2* (=) *x y* $\wedge$ *length x* = *n_ps*) $\times_R$ *state_rel*
===> *list_all2* ((=) $\times_R$ (=) $\times_R$ (=) $\times_R$ ((λx *y*. *list_all2* (=) *x y* $\wedge$ *length x* = *n_ps*) $\times_R$ *state_rel*)))
broad_trans_from broad_trans_from_impl
⟨*proof*⟩

lemma *trans_from_transfer*:
((λx *y*. *list_all2* (=) *x y* $\wedge$ *length x* = *n_ps*) $\times_R$ *state_rel*
===> *list_all2* ((=) $\times_R$ (=) $\times_R$ (=) $\times_R$ ((λx *y*. *list_all2* (=) *x y* $\wedge$ *length x* = *n_ps*) $\times_R$ *state_rel*)))
trans_from trans_impl
⟨*proof*⟩

lemma *list_all2_swap*:
list_all2 S ys xs **if** *list_all2 R xs ys S* = (λx *y*. *R y x*) **for** *R S*
⟨*proof*⟩

lemma *swap_rel_prod*: (λ *x y*. (*R* $\times_R$ *S*) *y x*) = (λx *y*. *R y x*) $\times_R$ (λx *y*. *S y x*) **for** *R S*
⟨*proof*⟩

lemma *swap_eq*:
(λx *y*. *y* = *x*) = (=)
⟨*proof*⟩

lemma *trans_from_refine*:
(*trans_impl*, *trans_from*) $\in$ *fun_rel_syn loc_rel* (*list_rel* (*Id* $\times_r$ *Id* $\times_r$ *Id* $\times_r$ *loc_rel*))
⟨*proof*⟩

lemma *trans_from_correct*:
(*trans_from*, *trans_prod*) $\in$ *transition_rel states'*
⟨*proof*⟩

lemma *states'_alt_def*:
states' = {(*L*, *s*). *L* $\in$ *states* $\wedge$ *bounded bounds s*}
⟨*proof*⟩

lemma *states'_alt_def2*:
states' = {(*L*, *s*). *L* $\in$ *states* $\wedge$ *dom s* = {*0..<n_vs*} $\wedge$ *check_bounded s*}

⟨*proof*⟩

lemma *trans_prod_states'_inv*:
l' ∈ *states'* **if** (*l*, *g*, *a*, *r*, *l'*) ∈ *trans_prod l* ∈ *states'*
⟨*proof*⟩

lemma *prod_ta_states'_inv*:
l' ∈ *states'* **if** *prod_ta* ⊢ *l* $\longrightarrow^{g,a,r}$ *l' l* ∈ *states'*
⟨*proof*⟩

lemma *dom_eq_transfer* [*transfer_rule*]:
(*state_rel* ===> (=)) (λ*s*. *dom s* = {*0*..<*n_vs*}) (λ*s*. *length s* = *n_vs*)
⟨*proof*⟩

lemma *states'_memi_correct*:
(*states'_memi*, (λ*l*. *l* ∈ *states'*)) ∈ *loc_rel* → *bool_rel*
⟨*proof*⟩

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 $\bigwedge a\ b.\ B\ a\ b \Longrightarrow P\ a \Longrightarrow A\ a\ b$
and $\bigwedge a\ b.\ P\ a \Longrightarrow A\ a\ b \Longrightarrow P\ b$
shows *Bisimulation_Invariant A B* (=) *P P*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *all_prop_statesD*[*dest*]:
L ∈ *states* **if** *all_prop L s*
⟨*proof*⟩

end

context *Prod_TA*
begin

lemma *prod_action_step_not_eq_delay*:
a ≠ *Del* **if** *prod_ta* ⊢ ⟨(*L*, *s*), *u*⟩ →$_a$ ⟨(*L'*, *s'*), *u'*⟩
⟨*proof*⟩

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'*⟩
⟨*proof*⟩

lemma *prod_all_prop_inv'*:
all_prop L' s' **if** *all_prop L s prod_ta* ⊢' ⟨(*L*, *s*), *u*⟩ → ⟨(*L'*, *s'*), *u'*⟩
⟨*proof*⟩

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*)
⟨*proof*⟩

lemmas *prod_bisim_intro* = *prod_bisim.Bisimulation_Invariant_axioms*

end

8 The Final Semantics

State formulas

datatype $('n, 's, 'a, 'b)$ *sexp* =
  true |
  — Boolean connectives
  not $('n, 's, 'a, 'b)$ *sexp* | *and* $('n, 's, 'a, 'b)$ *sexp* $('n, 's, 'a, 'b)$ *sexp* |
  or $('n, 's, 'a, 'b)$ *sexp* $('n, 's, 'a, 'b)$ *sexp* | *imply* $('n, 's, 'a, 'b)$ *sexp* $('n, 's, 'a, 'b)$ *sexp* |
  — Does var a equal x?
  eq $'a$ $'b$ |
  le $'a$ $'b$ |
  lt $'a$ $'b$ |
  ge $'a$ $'b$ |
  gt $'a$ $'b$ |
  — Is procces i in location l?
  loc $'n$ $'s$

datatype $('n, 's, 'a, 'b)$ *formula* =
  EX $('n, 's, 'a, 'b)$ *sexp* | *EG* $('n, 's, 'a, 'b)$ *sexp*
| *AX* $('n, 's, 'a, 'b)$ *sexp* | *AG* $('n, 's, 'a, 'b)$ *sexp*
| *Leadsto* $('n, 's, 'a, 'b)$ *sexp* $('n, 's, 'a, 'b)$ *sexp*

fun *check_sexp* :: $(nat, 's, 'a, 'b :: linorder)$ *sexp* $\Rightarrow$ $'s$ *list* $\Rightarrow$ $('a \Rightarrow 'b) \Rightarrow$ *bool* **where**
  check_sexp sexp.true _ _ $\longleftrightarrow$ *True* |
  check_sexp (*not e*) *L s* $\longleftrightarrow \neg$ *check_sexp e L s* |
  check_sexp (*and e1 e2*) *L s* $\longleftrightarrow$ *check_sexp e1 L s* $\wedge$ *check_sexp e2 L s* |
  check_sexp (*sexp.or e1 e2*) *L s* $\longleftrightarrow$ *check_sexp e1 L s* $\vee$ *check_sexp e2 L s* |
  check_sexp (*imply e1 e2*) *L s* $\longleftrightarrow$ *check_sexp e1 L s* $\longrightarrow$ *check_sexp e2 L s* |
  check_sexp (*eq i x*) *L s* $\longleftrightarrow$ *s i* $=$ *x* |
  check_sexp (*le i x*) *L s* $\longleftrightarrow$ *s i* $\leq$ *x* |
  check_sexp (*lt i x*) *L s* $\longleftrightarrow$ *s i* $<$ *x* |
  check_sexp (*ge i x*) *L s* $\longleftrightarrow$ *s i* $\geq$ *x* |
  check_sexp (*gt i x*) *L s* $\longleftrightarrow$ *s i* $>$ *x* |
  check_sexp (*loc i x*) *L s* $\longleftrightarrow$ *L* ! *i* $=$ *x*

fun *locs_of_sexp* :: $('n, 's, 'a, 'b)$ *sexp* $\Rightarrow$ $'n$ *set* **where**
  locs_of_sexp (*not e*) $=$ *locs_of_sexp e* |
  locs_of_sexp (*and e1 e2*) $=$ *locs_of_sexp e1* $\cup$ *locs_of_sexp e2* |
  locs_of_sexp (*sexp.or e1 e2*) $=$ *locs_of_sexp e1* $\cup$ *locs_of_sexp e2* |
  locs_of_sexp (*imply e1 e2*) $=$ *locs_of_sexp e1* $\cup$ *locs_of_sexp e2* |
  locs_of_sexp (*loc i x*) $= \{i\}$ |
  locs_of_sexp _ $= \{\}$

fun *vars_of_sexp* :: $('n, 's, 'a, 'b)$ *sexp* $\Rightarrow$ $'a$ *set* **where**
  vars_of_sexp (*not e*) $=$ *vars_of_sexp e* |
  vars_of_sexp (*and e1 e2*) $=$ *vars_of_sexp e1* $\cup$ *vars_of_sexp e2* |
  vars_of_sexp (*sexp.or e1 e2*) $=$ *vars_of_sexp e1* $\cup$ *vars_of_sexp e2* |
  vars_of_sexp (*imply e1 e2*) $=$ *vars_of_sexp e1* $\cup$ *vars_of_sexp e2* |
  vars_of_sexp (*eq i x*) $= \{i\}$ |
  vars_of_sexp (*lt i x*) $= \{i\}$ |

```
  vars_of_sexp (le i x) = {i} |
  vars_of_sexp (ge i x) = {i} |
  vars_of_sexp (gt i x) = {i} |
  vars_of_sexp _ = {}

fun locs_of_formula :: ('n, 's, 'a, 'b) formula ⇒ '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* λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *A* $\vdash$ $\langle$*L*, *s*, *u*$\rangle$ $\rightarrow$ $\langle$*L'*, *s'*, *u'*$\rangle$ $\langle$*proof*$\rangle$

lemma *models_ctl_iff*:
A,a_0 $\models$ Φ $\longleftrightarrow$ *models_ctl* (*ctl_of* Φ) a_0
$\langle$*proof*$\rangle$

end

fun *check_sexpi* :: (*nat*, '*s*, *nat*, *int*) *sexp* $\Rightarrow$ '*s list* $\Rightarrow$ *int list* $\Rightarrow$ *bool* **where**
check_sexpi sexp.true _ _ $\longleftrightarrow$ *True* |
check_sexpi (*not* *e*) *L* *s* $\longleftrightarrow$ $\neg$ *check_sexpi* *e* *L* *s* |
check_sexpi (*and* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexpi* *e1* *L* *s* $\wedge$ *check_sexpi* *e2* *L* *s* |
check_sexpi (*sexp.or* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexpi* *e1* *L* *s* $\vee$ *check_sexpi* *e2* *L* *s* |
check_sexpi (*imply* *e1* *e2*) *L* *s* $\longleftrightarrow$ *check_sexpi* *e1* *L* *s* $\longrightarrow$ *check_sexpi* *e2* *L* *s* |
check_sexpi (*eq* *i* *x*) *L* *s* $\longleftrightarrow$ *s* ! *i* = *x* |
check_sexpi (*le* *i* *x*) *L* *s* $\longleftrightarrow$ *s* ! *i* $\leq$ *x* |
check_sexpi (*lt* *i* *x*) *L* *s* $\longleftrightarrow$ *s* ! *i* < *x* |
check_sexpi (*ge* *i* *x*) *L* *s* $\longleftrightarrow$ *s* ! *i* $\geq$ *x* |
check_sexpi (*gt* *i* *x*) *L* *s* $\longleftrightarrow$ *s* ! *i* > *x* |
check_sexpi (*loc* *i* *x*) *L* *s* $\longleftrightarrow$ *L* ! *i* = *x*

fun *hd_of_formulai* :: (*nat*, '*s*, *nat*, *int*) *formula* $\Rightarrow$ '*s list* $\Rightarrow$ *int list* $\Rightarrow$ *bool* **where**
hd_of_formulai (*formula.EX* φ) *L* *s* = *check_sexpi* 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*
$\langle$*proof*$\rangle$
lemma *equiv'_weak_cong*[*cong*]:
Bisimulation_Invariant.equiv' *R* *P* *Q* = *Bisimulation_Invariant.equiv'* *R* *P* *Q* **for** *R* *P* *Q*
$\langle$*proof*$\rangle$
lemma *equiv'_weak_cong2*[*cong*]:
Simulation_Invariant.equiv' *R* *P* *Q* = *Simulation_Invariant.equiv'* *R* *P* *Q* **for** *R* *P* *Q*
$\langle$*proof*$\rangle$

lemma *models_ctl_weak_cong*[*cong*]:
Graph_Defs.models_ctl E = *Graph_Defs.models_ctl E* **for** *E*
⟨*proof*⟩
lemma *models_path_weak_cong*[*cong*]:
Graph_Defs.models_path E = *Graph_Defs.models_path E* **for** *E*
⟨*proof*⟩
lemma *models_state_weak_cong*[*cong*]:
Graph_Defs.models_state E = *Graph_Defs.models_state E* **for** *E*
⟨*proof*⟩
lemma *models_ltl_weak_cong*[*cong*]:
Graph_Defs.models_ltlc E = *Graph_Defs.models_ltlc E* **for** *E*
⟨*proof*⟩
lemma *models_weak_cong*[*cong*]:
models E = *models E*
for *E* ⟨*proof*⟩

9 Instantiating the Model Checking Locale

locale *Simple_Network_Impl_nat_urge* =
Simple_Network_Impl_nat + **assumes** *no_urgency*: ∀ (_, *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
$k\_fun\ l\ c \equiv$
$\quad if\ (c > 0 \wedge c \leq m) \wedge fst\ l \in states\ then\ Max\ \{k\ !\ i\ !\ (fst\ l\ !\ i)\ !\ c \mid i\ .\ i < n\_ps\}\ else\ 0$

lemma (**in** −) *default_map_of_distinct*:
$(k,\ default\_map\_of\ x\ xs\ k) \in set\ xs \cup \{(k,\ x)\}$ **if** $distinct\ (map\ fst\ xs)$
⟨*proof*⟩

lemma *N_inv*:
$(L\ !\ i,\ inv\ (N\ i)\ (L\ !\ i)) \in set\ ((snd\ o\ snd\ o\ snd)\ (automata\ !\ i)) \cup \{(L\ !\ i,\ [])\}$
if $i < n\_ps$
⟨*proof*⟩

lemma (**in** −) *subset_nat_0_atLeastLessThan_conv*:
$S \subseteq \{0..<n::nat\} \longleftrightarrow (\forall\ x \in S.\ x < n)$
⟨*proof*⟩

lemma *k_ceiling_rule*:
$m \leq int\ (k\ !\ i\ !\ l\ !\ x)$
if $i < n\_ps\ (l,\ b,\ g,\ xx) \in set\ ((fst\ o\ snd\ o\ snd)\ (automata\ !\ i))$
$\quad (x,\ m) \in collect\_clock\_pairs\ g$
for $i\ l\ x\ g\ xx$
⟨*proof*⟩

lemma *k_ceiling_1*:
$\forall s.\ \forall L \in states.\ \forall (x,m) \in clkp\_set\ prod\_ta\ (L,\ s).\ m \leq k\_fun\ (L,\ s)\ x$
⟨*proof*⟩

lemma *k_fun_mono′*:
$k\_fun\ (L,\ s)\ c \leq k\_fun\ (L',\ s')\ c$ **if**
$\forall i < n\_ps.\ k\ !\ i\ !\ (L\ !\ i)\ !\ c \leq k\ !\ i\ !\ (L'\ !\ i)\ !\ c\ L \in states\ L' \in states$
⟨*proof*⟩

lemma *k_fun_mono*:
$‹Max\ \{k\ !\ i\ !\ (L\ !\ i)\ !\ c \mid i\ .\ i < n\_ps\} \leq Max\ \{k\ !\ i\ !\ (L'\ !\ i)\ !\ c \mid i\ .\ i < n\_ps\}›$
if $‹\forall i < n\_ps.\ k\ !\ i\ !\ (L\ !\ i)\ !\ c \leq k\ !\ i\ !\ (L'\ !\ i)\ !\ c›$
⟨*proof*⟩

lemma (**in** −) *fold_upds_aux1*:
$fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs\ !\ i = xs\ !\ i$ **if** $‹i \notin set\ ps›$
⟨*proof*⟩

lemma (**in** −) *fold_upds_aux2*:
$fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs\ !\ i = g\ i$ **if** $‹distinct\ ps›\ ‹i \in set\ ps›\ ‹i < length\ xs›$
⟨*proof*⟩

lemma (**in** −) *fold_upds_aux_length*:
$length\ (fold\ (\lambda p\ L.\ L[p := g\ p])\ ps\ xs) = length\ xs$
⟨*proof*⟩

lemma *prod_ta_step_statesD*:

assumes *prod_ta* $\vdash$ $(L,\ s) \longrightarrow^{g,a,r} (L',\ s')$
shows $L \in$ *states* $L' \in$ *states*
⟨*proof*⟩

lemma *k_ceiling_2*:
$\forall\, l\ g\ a\ r\ l'.\ \forall\ c \leq m.\ prod\_ta \vdash l \longrightarrow^{g,a,r} l' \wedge c \notin set\ r \longrightarrow k\_fun\ l'\ c \leq k\_fun\ l\ c$
⟨*proof*⟩

abbreviation F **where** $F \equiv \lambda(L,\ s).\ hd\_of\_formula\ formula\ L\ (the\ o\ s)$
abbreviation $Fi \equiv \lambda(L,\ s).\ hd\_of\_formulai\ formula\ L\ s$

lemma (**in** *Simple_Network_Impl_nat*) *check_sexp_check_sexpi*:
check_sexp e L (*the o s*) $\longleftrightarrow$ *check_sexpi e L s'*
if *state_rel s s' vars_of_sexp e* $\subseteq \{0..<n\_vs\}$
⟨*proof*⟩

lemma (**in** *Simple_Network_Impl_nat*) *hd_of_formula_hd_of_formulai*:
hd_of_formula φ *L* (*the o s*) $\longleftrightarrow$ *hd_of_formulai* φ *L s'*
if *state_rel s s' vars_of_formula* $\varphi \subseteq \{0..<n\_vs\}$
⟨*proof*⟩

lemma *F_Fi*:
F l $\longleftrightarrow$ *Fi l'* **if** $(l',\ l) \in loc\_rel$
⟨*proof*⟩

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) $s_0 i$
⟨*proof*⟩

lemma *statesI*:
$L \in$ *states* **if** *length* $L = n\_ps\ \forall i{<}n\_ps.\ L\ !\ i \in fst\ `\ set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$
⟨*proof*⟩

lemma L_0*_states*[*simp*, *intro*]:
$L_0 \in$ *states*
⟨*proof*⟩

lemma l_0*_states'*[*simp*, *intro*]:
$l_0 \in$ *states'*
⟨*proof*⟩

sublocale *reach*: *Reachability_Problem_Defs*
prod_ta
l_0
m
k_fun
F

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *clkp_set_statesD*:
fixes *x d*
assumes (*x*, *d*)∈*Closure.clkp_set prod_ta* (*L*, *s*)
shows *L* ∈ *states*
⟨*proof*⟩

sublocale *reach1*: *Reachability_Problem*
prod_ta
l_0
m
k_fun
F
⟨*proof*⟩

lemma *states′_superset*:
{l_0} ∪ *Normalized_Zone_Semantics_Impl_Refine.state_set trans_prod* ⊆ *states′*
(**is** {l_0} ∪ *?S* ⊆ *states′*)
⟨*proof*⟩

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* ≠ {}
⟨*proof*⟩

lemma (**in** *Simple_Network_Impl_nat*) *n_ps_gt_0*: *n_ps* > *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *k_impl_k_fun*:
k_impl (*L*, *s*) = *IArray* (*map* (*k_fun* (*L*, *s*)) [*0*..<*m+1*]) **if** *L* ∈ *states*
⟨*proof*⟩

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 l_0 = l_0
and $l_0 i$ = $l_0 i$
and *F* = *PR_CONST F*
and *n* = *m*
and *k* = *k_fun*
and *trans_impl* = *trans_impl*
and *states'* = *states'*
and *loc_rel* = *loc_rel*
⟨*proof*⟩

end

no_notation *UPPAAL_Model_Checking.models* (_,_ ⊨_ _ [61,61] 61)

context *Reachability_Problem_Impl*
begin

lemma *F_reachable_correct*:
op.F_reachable
⟷ (∃ *l'*. ∀ u_0. (∀ *c* ∈ {1..*n*}. u_0 *c* = 0) ⟶ (∃ *u'*. *conv_A A* ⊢' ⟨l_0, u_0⟩ →* ⟨*l'*, *u'*⟩ ∧ *F l'*))
⟨*proof*⟩

lemma *E_op''_F_reachable_correct*:
E_op''.F_reachable
⟷ (∃ *l'*. ∀ u_0. (∀ *c* ∈ {1..*n*}. u_0 *c* = 0) ⟶ (∃ *u'*. *conv_A A* ⊢' ⟨l_0, u_0⟩ →* ⟨*l'*, *u'*⟩ ∧ *F l'*))
⟨*proof*⟩

lemma *Ex_ev_impl_hnr*:
assumes ∀ u_0. (∀ *c* ∈ {1..*n*}. u_0 *c* = 0) ⟶ ¬ *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* (λ*r*. (*r* ⟷ (∀ u_0. (∀ *c* ∈ {1..*n*}. u_0 *c* = 0) ⟶ *Ex_ev* (λ(*l*, _). *F l*) (l_0, u_0))))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
⟨*proof*⟩

end

context *Simple_Network_Impl_nat_urge*
begin

sublocale *conv*: *Prod_TA_urge*
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
⟨*proof*⟩

abbreviation $A \equiv (set\ broadcast,\ map\ automaton\_of\ automata,\ map\_of\ bounds')$

interpretation *conv_eq_bisim*:
$(\lambda(l,\ u)\ (l',\ u').\ conv\_A\ prod\_ta \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda(L,\ s,\ u)\ (L',\ s',\ u').\ conv.prod\_ta \vdash' \langle (L,\ s),\ u\rangle \rightarrow \langle (L',\ s'),\ u'\rangle)$
$(\lambda((L,\ s),\ u)\ (L',\ s',\ u').\ L = L' \wedge u = u' \wedge s = s')$
$(\lambda((L,\ s),\ u).\ conv.all\_prop\ L\ s)$
$(\lambda(L,\ s,\ u).\ conv.all\_prop\ L\ s)$
⟨*proof*⟩

interpretation *Bisimulation_Invariant*
$(\lambda(l,\ u)\ (l',\ u').\ conv\_A\ prod\_ta \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda(L,\ s,\ u)\ (L',\ s',\ u').\ Simple\_Network\_Language.conv\ A \vdash \langle L,\ s,\ u\rangle \rightarrow \langle L',\ s',\ u'\rangle)$
$(\lambda((L,\ s),\ u)\ (L',\ s',\ u').\ L = L' \wedge u = u' \wedge s = s')$
$(\lambda((L,\ s),\ u).\ conv.all\_prop\ L\ s)$
$(\lambda(L,\ s,\ u).\ conv.all\_prop\ L\ s)$
⟨*proof*⟩

lemmas *prod_bisim* = *Bisimulation_Invariant_axioms*

lemmas *deadlock_iff* = *deadlock_iff*

lemma *conv_all_prop*:
conv.all_prop = *all_prop*
⟨*proof*⟩

definition *prop_of2* **where**
prop_of2 φ = *PropC* $(\lambda((L,\ s),\ \_).\ check\_sexp\ \varphi\ L\ (the\ o\ s))$

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,$(L_0,\ s_0,\ u_0) \models \Phi$ = (*case* Φ *of*
formula.EX $\varphi \Rightarrow$
Graph_Defs.Ex_ev
$(\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ prod\_ta \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda\ ((L,\ s),\ \_).\ check\_sexp\ \varphi\ L\ (the\ o\ s))$
| *formula.EG* $\varphi \Rightarrow$
Not o Graph_Defs.Alw_ev
$(\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ prod\_ta \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda\ ((L,\ s),\ \_).\ \neg\ check\_sexp\ \varphi\ L\ (the\ o\ s))$
| *formula.AX* $\varphi \Rightarrow$
Graph_Defs.Alw_ev
$(\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ prod\_ta \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle)$
$(\lambda\ ((L,\ s),\ \_).\ check\_sexp\ \varphi\ L\ (the\ o\ s))$
| *formula.AG* $\varphi \Rightarrow$

```
    Not o Graph_Defs.Ex_ev
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). ¬ check_sexp φ L (the o s))
| formula.Leadsto φ ψ ⇒
    Graph_Defs.leadsto
      (λ (l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
      (λ ((L, s), _). check_sexp φ L (the o s))
      (λ ((L, s), _). check_sexp ψ L (the o s))
) ((L₀, s₀), u₀)
 if L₀ ∈ states Simple_Network_Language.bounded bounds s₀
⟨proof⟩

end

context Simple_Network_Impl_nat_ceiling_start_state — slow: 70s
begin

definition Alw_ev_checker where
  Alw_ev_checker = dfs_map_impl'
    (impl.succs_P_impl' Fi) impl.a₀_impl impl.subsumes_impl (return ∘ fst)
    impl.state_copy_impl

definition leadsto_checker where
  leadsto_checker ψ = do {
    r ← leadsto_impl
    impl.state_copy_impl (impl.succs_P_impl' (λ (L, s). ¬ check_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 l_0
      then Alw_ev_checker
      else return False |
    formula.Leadsto _ ψ ⇒ leadsto_checker ψ
  )
```

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

lemma *all_prop_start*:
$all\_prop\ L_0\ (map\_of\ s_0)$
⟨*proof*⟩

lemma *deadlock_start_iff*:
Graph_Defs.deadlock
$(\lambda(L,\ s,\ u)\ (L',\ s',\ u').\ Simple\_Network\_Language.conv\ A \vdash \langle L,\ s,\ u\rangle \rightarrow \langle L',\ s',\ u'\rangle)\ (L_0,\ (map\_of\ s_0),\ u_0)$
$\longleftrightarrow$ *reach.deadlock* $(l_0,\ u_0)$
⟨*proof*⟩

lemma *F_Fi'*:
check_sexp $\psi\ L\ (the\ o\ s) \longleftrightarrow$ *check_sexpi* $\psi\ L'\ s'$
if $((L',\ s'),\ (L,\ s)) \in loc\_rel\ formula = Leadsto\ \varphi\ \psi$
⟨*proof*⟩

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⟩
```

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
  }
⟨proof⟩

lemma Alw_ev_checker_alt_def':
  Alw_ev_checker ≡
    do {
      x ← let
        key = return ∘ fst;
        sub = impl.subsumes_impl;
        copy = impl.state_copy_impl;
        start = impl.a₀_impl;
        succs =  impl.succs_P_impl' Fi
      in dfs_map_impl' succs start sub key copy;
      _ ← return ();
      return x
    }
  ⟨proof⟩

lemma leadsto_checker_alt_def':
  leadsto_checker ψ ≡
    do {
      r ← let
        key = return ∘ fst;
        sub = impl.subsumes_impl;
        copy = impl.state_copy_impl;
        start = impl.a₀_impl;
        final = impl.F_impl;
        final' = (impl.Q_impl (λ(L, s). ¬ check_sexpi ψ L s));
        succs =  impl.succs_P_impl' (λ(L, s). ¬ check_sexpi ψ L s);
        succs' =  impl.succs_impl';
        empty = impl.emptiness_check_impl;
        trace = impl.tracei
      in
        leadsto_impl copy succs start sub key succs' empty final final' trace;
      return (¬ r)
    }
  ⟨proof⟩

theorem k_impl_alt_def:
  k_impl = (λ(l, s). IArray (map (λc. MAX i ∈ {0..<n_ps}. k_i !! i !! (l ! i) !! c) [0..<m + 1]))
⟨proof⟩

definition
  trans_map_inner ≡ map (λi. union_map_of (fst (snd (snd (automata ! i))))) [0..<n_ps]

lemma trans_map_alt_def:
  trans_map i j = (case (IArray trans_map_inner !! i) j of None ⇒ [] | Some xs ⇒ xs)
  if i < n_ps
  ⟨proof⟩

schematic_goal succs_impl_alt_def:
```

impl.succs_impl ≡ *?impl*
⟨*proof*⟩

schematic_goal *succs_P_impl_alt_def*:
impl.succs_P_impl Pi ≡ *?impl*
if (*Pi*, *P*) ∈ *inv_rel loc_rel states'*
for *P Pi*
⟨*proof*⟩

lemmas *succs_P'_impl_alt_def* =
impl.succs_P_impl'_def[*OF impl.F_fun*, *unfolded succs_P_impl_alt_def*[*OF impl.F_fun*]]

schematic_goal *Alw_ev_checker_alt_def*:
Alw_ev_checker ≡ *?impl*
⟨*proof*⟩

lemma *ψ_compatibleI*:
(λ(*L*, *s*). ¬ *check_sexpi ψ L s*, λ(*L*, *s*). ¬ *check_sexp ψ L* (*the* ∘ *s*)) ∈ *inv_rel loc_rel states'*
if *formula* = *Leadsto φ ψ*
⟨*proof*⟩

lemma *Q_impl_alt_def*:
impl.Q_impl (λ(*L*, *s*). ¬ *check_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*
⟨*proof*⟩

schematic_goal *leadsto_checker_alt_def*:
leadsto_checker ψ ≡ *?impl* **if** *formula* = *Leadsto φ ψ*
⟨*proof*⟩

schematic_goal *reachability_checker_alt_def*:
reachability_checker ≡ *?impl*
⟨*proof*⟩

end

concrete_definition *reachability_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.reachability_checker_alt_def*

concrete_definition *Alw_ev_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.Alw_ev_checker_alt_def*

concrete_definition *leadsto_checker*
uses *Simple_Network_Impl_nat_ceiling_start_state.leadsto_checker_alt_def*

context *Simple_Network_Impl_nat_ceiling_start_state* — slow: 100s
begin

lemma *model_checker_unfold_leadsto*:
model_checker = (
case formula of Simple_Network_Language_Model_Checking.formula.EX xa ⇒ *reachability_checker*

```
  | Simple_Network_Language_Model_Checking.formula.EG xa ⇒
    if PR_CONST F l₀ then Alw_ev_checker else return False
  | Simple_Network_Language_Model_Checking.formula.AX xa ⇒
    (if PR_CONST F l₀ then Alw_ev_checker else return False) ⤜ (λr. return (¬ r))
  | Simple_Network_Language_Model_Checking.formula.AG xa ⇒
    reachability_checker ⤜ (λr. return (¬ r))
  | Simple_Network_Language_Model_Checking.formula.Leadsto φ ψ ⇒
    Simple_Network_Language_Model_Checking.leadsto_checker
      broadcast bounds' automata m num_states num_actions k L₀ s₀ formula ψ show_clock
show_state)
```

⟨*proof*⟩

lemmas *model_checker_def_refined* = *model_checker_unfold_leadsto*[*unfolded*
reachability_checker.refine[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*]
Alw_ev_checker.refine[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*]
]

end

concrete_definition *model_checker* **uses**
Simple_Network_Impl_nat_ceiling_start_state.model_checker_def_refined

definition *precond_mc* **where**
precond_mc
show_clock show_state broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
≡
if Simple_Network_Impl_nat_ceiling_start_state
broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
then
model_checker
broadcast bounds' automata m num_states num_actions k L₀ s₀ formula show_clock
show_state
⤜ (λ *x*. *return* (*Some x*))
else return None

abbreviation *N* **where**
N broadcast automata bounds ≡
Simple_Network_Language.conv
(*set broadcast*, *map automaton_of automata*, *map_of bounds*)

definition *has_deadlock* **where**
has_deadlock A a₀ ≡
Graph_Defs.deadlock (λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *A* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩) *a₀*

theorem *model_check*:
<*emp*> *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⟩

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
(_↓ [71] 71)
where
  Z↓ = {u | u d. (u ⊕ d) ∈ Z ∧ d ≥ (0::'t)}

definition zone_set_pre :: ('c, 't::time) zone ⇒ 'c list ⇒ ('c, 't) zone
where
  zone_set_pre Z r = {u . ([r → (0::'t)]u) ∈ Z}

definition zone_pre :: ('c, 't::time) zone ⇒ 'c list ⇒ ('c, 't) zone
 where
  zone_pre Z r = (zone_set_pre Z r)↓

lemma zone_time_pre_mono:
  A↓ ⊆ B↓ if A ⊆ B
  ⟨proof⟩
```

lemma *clock_set_split*:
$P\ (([r \to 0]u)\ x) \longleftrightarrow (x \notin set\ r \longrightarrow P\ (u\ x)) \wedge (x \in set\ r \longrightarrow P\ 0)$
⟨*proof*⟩

context *Regions_TA*
begin

definition
check_deadlock $l\ Z \equiv Z \subseteq$
$\bigcup\ \{(zone\_set\_pre\ \{u.\ u \vdash inv\_of\ A\ l'\}\ r \cap \{u.\ u \vdash g\} \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow} \mid g\ a\ r\ l'.$
$A \vdash l \longrightarrow^{g,a,r} l'\}$

lemma *V_zone_time_pre*:
$x \in (Z \cap V)^{\downarrow}$ **if** $x \in Z^{\downarrow}\ x \in V$
⟨*proof*⟩

lemma *check_deadlock_alt_def*:
check_deadlock $l\ Z = (Z \subseteq \bigcup\ \{$
$(zone\_set\_pre\ (\{u.\ u \vdash inv\_of\ A\ l'\} \cap V)\ r \cap \{u.\ \forall\ x \in set\ r.\ u\ x \geq 0\}$
$\cap \{u.\ u \vdash g\} \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow} \cap V$
$\mid g\ a\ r\ l'.\ A \vdash l \longrightarrow^{g,a,r} l'\})$ (**is** $\_ = (?L \subseteq ?R)$) **if** $Z \subseteq V$
⟨*proof*⟩

lemma *step_trans1*:
assumes $A \vdash_t \langle l, u\rangle \to_{(g,a,r)} \langle l', u'\rangle$
shows $u \in zone\_set\_pre\ \{u.\ u \vdash inv\_of\ A\ l'\}\ r \cap \{u.\ u \vdash g\}\ A \vdash l \longrightarrow^{g,a,r} l'$
⟨*proof*⟩

lemma *step_trans2*:
assumes $u \in zone\_set\_pre\ \{u.\ u \vdash inv\_of\ A\ l'\}\ r \cap \{u.\ u \vdash g\}\ A \vdash l \longrightarrow^{g,a,r} l'$
shows $\exists\ u'.\ A \vdash_t \langle l, u\rangle \to_{(g,a,r)} \langle l', u'\rangle$
⟨*proof*⟩

lemma *time_pre_zone*:
$u \in (Z \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow}$ **if** $A \vdash \langle l, u\rangle \to^{d} \langle l', u'\rangle\ u' \in Z$
⟨*proof*⟩

lemma *time_pre_zone'*:
$\exists\ d\ u'.\ u' \in Z \wedge A \vdash \langle l, u\rangle \to^{d} \langle l, u'\rangle$ **if** $u \in (Z \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow}$
⟨*proof*⟩

lemma *step_trans3*:
assumes $A \vdash' \langle l, u\rangle \to^{(g,a,r)} \langle l', u'\rangle$
shows $u \in (zone\_set\_pre\ \{u.\ u \vdash inv\_of\ A\ l'\}\ r \cap \{u.\ u \vdash g\} \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow}$
$A \vdash l \longrightarrow^{g,a,r} l'$
⟨*proof*⟩

lemma *step_trans4*:
assumes $u \in (zone\_set\_pre\ \{u.\ u \vdash inv\_of\ A\ l'\}\ r \cap \{u.\ u \vdash g\} \cap \{u.\ u \vdash inv\_of\ A\ l\})^{\downarrow}$
$A \vdash l \longrightarrow^{g,a,r} l'$
shows $\exists\ u'.\ A \vdash' \langle l, u\rangle \rightarrow^{(g,a,r)} \langle l', u'\rangle$
⟨*proof*⟩

lemma *check_deadlock_correct*:
$check\_deadlock\ l\ Z \longleftrightarrow (\forall\, u \in Z.\ \exists\, l'\ u'\ g\ a\ r.\ A \vdash' \langle l, u\rangle \rightarrow^{(g,a,r)} \langle l', u'\rangle)$
⟨*proof*⟩

lemma *step'_step_trans'_iff*:
$A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle \longleftrightarrow (\exists\, g\ a\ r.\ A \vdash' \langle l, u\rangle \rightarrow^{(g,a,r)} \langle l', u'\rangle)$
⟨*proof*⟩

lemma *check_deadlock_correct_step'*:
$check\_deadlock\ l\ Z \longleftrightarrow (\forall\, u \in Z.\ \exists\, l'\ u'.\ A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle)$
⟨*proof*⟩

Unused **lemma** *delay_step_zone*:
$u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv\_of\ A\ l\}$ **if** $A \vdash \langle l, u\rangle \rightarrow^{d} \langle l, u'\rangle\ u \in Z$
⟨*proof*⟩

lemma *delay_step_zone'*:
$\exists\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \rightarrow^{d} \langle l, u'\rangle$ **if** $u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv\_of\ A\ l\}$
⟨*proof*⟩

lemma *delay_step_zone''*:
$(\exists\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \rightarrow^{d} \langle l, u'\rangle) \longleftrightarrow u' \in Z^{\uparrow} \cap \{u.\ u \vdash inv\_of\ A\ l\}$
⟨*proof*⟩

lemma *delay_step_zone'''*:
$\{u' \mid u'\ d\ u.\ u \in Z \wedge A \vdash \langle l, u\rangle \rightarrow^{d} \langle l, u'\rangle\} = Z^{\uparrow} \cap \{u.\ u \vdash inv\_of\ A\ l\}$
⟨*proof*⟩

end

context *Regions_TA_Start_State*
begin

lemma *check_deadlock_deadlocked*:
$\neg\ check\_deadlock\ l\ Z \longleftrightarrow (\exists\, u{\in}Z.\ sim.sim.deadlocked\ (l, u))$
⟨*proof*⟩

lemma *deadlock_check'*:
$(\exists\, x_0{\in}a_0.\ \exists\, l\ u.\ sim.sim.reaches\ x_0\ (l, u) \wedge sim.sim.deadlocked\ (l, u)) \longleftrightarrow$
$(\exists\, l\ Z.\ reaches\ (l_0, Z_0)\ (l, Z) \wedge \neg\ check\_deadlock\ l\ Z)$
⟨*proof*⟩

lemma *deadlock_check*:
$(\exists\, x_0{\in}a_0.\ sim.sim.deadlock\ x_0) \longleftrightarrow (\exists\, l\ Z.\ reaches\ (l_0, Z_0)\ (l, Z) \wedge \neg\ check\_deadlock\ l\ Z)$
⟨*proof*⟩

end

10.3 Operations

Subset inclusion check for federations on DBMs **lemma**
$S \subseteq R \longleftrightarrow S \cap -R = \{\}$
⟨*proof*⟩

lemma
$A \subseteq B \cup C \longleftrightarrow A \cap -B \cap -C = \{\}$
⟨*proof*⟩

lemma
$(A \cup B) \cap (C \cup D) = A \cap C \cup A \cap D \cup B \cap C \cup B \cap D$
⟨*proof*⟩

lemma *Le_le_inf*[*intro*]:
Le $(x :: \_ :: linordered\_cancel\_ab\_monoid\_add) \preceq \infty$
⟨*proof*⟩

lemma *dbm_entry_val_mono*:
dbm_entry_val u a b e' **if** *dbm_entry_val u a b e* $e \leq e'$
⟨*proof*⟩

definition *and_entry* ::
nat ⇒ *nat* ⇒ ('*t*::{*linordered_cancel_ab_monoid_add*,*uminus*}) *DBMEntry* ⇒ '*t DBM* ⇒ '*t DBM* **where**
and_entry a b e M = (λ*i j*. *if i* = *a* ∧ *j* = *b then min* (*M i j*) *e else M i j*)

lemma *and_entry_mono*:
and_entry a b e M i j ≤ *M i j*
⟨*proof*⟩

abbreviation *clock_to_option a* ≡ (*if a* > *0 then Some a else None*)

definition
dbm_entry_val' *u a b e* ≡ *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* ∨ *j* > *0*) ∧ *i* ≤ *n* ∧ *j* ≤ *n* ∧ *m i j* ≠ ∞])

locale *Default_Nat_Clock_Numbering* =
fixes *n* :: *nat* **and** *v* :: *nat* ⇒ *nat*
assumes *v_is_id*: ∀ *c*. *c* > *0* ∧ *c* ≤ *n* ⟶ *v c* = *c* ∀ *c*. *c* > *n* ⟶ *v c* = *n* + *1 v 0* = *n* + *1*
begin

lemma *v_id*:
v c = *c* **if** *v c* ≤ *n*
⟨*proof*⟩

lemma *le_n*:
c ≤ *n* **if** *v c* ≤ *n*

⟨*proof*⟩

lemma *gt_0*:
$c > 0$ **if** $v\ c \leq n$
⟨*proof*⟩

lemma *le_n_iff*:
$v\ c \leq n \longleftrightarrow c \leq n \wedge c > 0$
⟨*proof*⟩

lemma *v_0*:
$v\ c = 0 \longleftrightarrow \mathit{False}$
⟨*proof*⟩

lemma *surj_on*: $\forall\ k \leq n.\ k > 0 \longrightarrow (\exists\ c.\ v\ c = k)$
⟨*proof*⟩

abbreviation *zone_of* ($[\![\_]\!]$) **where** *zone_of* $M \equiv [M]_{v,n}$

abbreviation
dbm_fed $S \equiv \bigcup\ m \in S.\ [\![m]\!]$

abbreviation
dbm_list $xs \equiv$ *dbm_fed* (*set* xs)

lemma *dbm_fed_singleton*:
dbm_fed $\{m\} = [m]_{v,n}$
⟨*proof*⟩

lemma *dbm_list_single*:
dbm_list $xs = [m]_{v,n}$ **if** *set* $xs = \{m\}$
⟨*proof*⟩

lemma *dbm_fed_superset_fold*:
$S \subseteq$ *dbm_list* $xs \longleftrightarrow$ *fold* $(\lambda m\ S.\ S \cap - ([m]_{v,n}))\ xs\ S = \{\}$
⟨*proof*⟩

lemma *dbm_fed_superset_fold′*:
dbm_fed $S \subseteq$ *dbm_list* $xs \longleftrightarrow$ *dbm_fed* (*fold* $f\ xs\ S$) $= \{\}$ **if**
$\bigwedge\ m\ S.\ m \in$ *set* $xs \Longrightarrow$ *dbm_fed* $(f\ m\ S) =$ *dbm_fed* $S \cap - ([m]_{v,n})$
⟨*proof*⟩

lemma *dbm_fed_superset_fold″*:
dbm_list $S \subseteq$ *dbm_list* $xs \longleftrightarrow$ *dbm_list* (*fold* $f\ xs\ S$) $= \{\}$ **if**
$\bigwedge\ m\ S.\ m \in$ *set* $xs \Longrightarrow$ *dbm_list* $(f\ m\ S) =$ *dbm_list* $S \cap - ([m]_{v,n})$
⟨*proof*⟩

lemma *neg_inf*:
$\{u.\ \neg$ *dbm_entry_val* $u\ a\ b\ e\} = \{\}$ **if** $e = (\infty :: \_\ \mathit{DBMEntry})$
⟨*proof*⟩

lemma *dbm_entry_val′_diff_shift*:
dbm_entry_val′ $(u \oplus d)\ c1\ c2\ (M\ c1\ c2)$ **if** *dbm_entry_val′* $u\ c1\ c2\ (M\ c1\ c2)\ 0 < c1\ 0 < c2$
⟨*proof*⟩

lemma *dbm_entry_val_iff_bounded_Le1*:
dbm_entry_val u (Some c1) None e $\longleftrightarrow$ *Le (u c1)* $\leq$ *e*
⟨*proof*⟩

lemma *dbm_entry_val_iff_bounded_Le2*:
dbm_entry_val u None (Some c2) e $\longleftrightarrow$ *Le (− u c2)* $\leq$ *e*
⟨*proof*⟩

lemma *dbm_entry_val_iff_bounded_Le3*:
dbm_entry_val u (Some c1) (Some c2) e $\longleftrightarrow$ *Le (u c1 − u c2)* $\leq$ *e*
⟨*proof*⟩

lemma *dbm_entry_val′_iff_bounded*:
dbm_entry_val′ u c1 c2 e $\longleftrightarrow$ *Le((if c1 > 0 then u c1 else 0) − (if c2 > 0 then u c2 else 0))* $\leq$ *e*
if *c1 > 0 ∨ c2 > 0*
⟨*proof*⟩

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* $\neq$ (∞ :: _ *DBMEntry*) *a > 0 ∨ b > 0*

⟨*proof*⟩

lemma *neg_unbounded*:
{*u*. ¬ *dbm_entry_val′ u i j e*} = {} **if** *e* = (∞ :: _ *DBMEntry*)
⟨*proof*⟩

lemma *and_entry_sound*:
u $\vdash_{v,n}$ *and_entry a b e M* **if** *dbm_entry_val′ u a b e u* $\vdash_{v,n}$ *M*
⟨*proof*⟩

lemma *DBM_val_bounded_mono*:
u $\vdash_{v,n}$ *M* **if** *u* $\vdash_{v,n}$ *M′* ∀ *i* $\leq$ *n*. ∀ *j* $\leq$ *n*. *M′ i j* $\leq$ *M i j*
⟨*proof*⟩

lemma *and_entry_entry*:
dbm_entry_val′ u a b e **if** *u* $\vdash_{v,n}$ *and_entry a b e M a* $\leq$ *n b* $\leq$ *n a > 0 ∨ b > 0*
⟨*proof*⟩

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* $\leq$ *n b* $\leq$ *n a > 0 ∨ b > 0*
⟨*proof*⟩

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* $\leq$ *n j* $\leq$ *n i > 0 ∨ j > 0*
⟨*proof*⟩

context
fixes $m :: nat \Rightarrow nat \Rightarrow ('a :: \{time\})\ DBMEntry$
assumes $Le\ 0\ \preceq\ m\ 0\ 0$
begin

private lemma A:
$-\ ([m]_{v,n}) =$
$(\bigcup\ (i, j) \in \{(i, j).\ i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$
$\{u.\ \neg\ dbm\_entry\_val\ u\ (Some\ i)\ (Some\ j)\ (m\ i\ j)\})$
$\cup\ (\bigcup\ i \in \{i.\ i > 0 \wedge i \leq n\}.\ \{u.\ \neg\ dbm\_entry\_val\ u\ (Some\ i)\ None\ (m\ i\ 0)\})$
$\cup\ (\bigcup\ j \in \{i.\ i > 0 \wedge i \leq n\}.\ \{u.\ \neg\ dbm\_entry\_val\ u\ None\ (Some\ j)\ (m\ 0\ j)\})$
⟨*proof*⟩ **lemma** B:
$S \cap -\ ([m]_{v,n}) =$
$(\bigcup\ (i, j) \in \{(i, j).\ i > 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n\}.$
$S \cap \{u.\ \neg\ dbm\_entry\_val\ u\ (Some\ i)\ (Some\ j)\ (m\ i\ j)\})$
$\cup\ (\bigcup\ i \in \{i.\ i > 0 \wedge i \leq n\}.\ S \cap \{u.\ \neg\ dbm\_entry\_val\ u\ (Some\ i)\ None\ (m\ i\ 0)\})$
$\cup\ (\bigcup\ j \in \{i.\ i > 0 \wedge i \leq n\}.\ S \cap \{u.\ \neg\ dbm\_entry\_val\ u\ None\ (Some\ j)\ (m\ 0\ j)\})$
⟨*proof*⟩ **lemma** *UNION_cong*:
$(\bigcup\ x \in S.\ f\ x) = (\bigcup\ x \in T.\ g\ x)$ **if** $S = T \bigwedge x.\ x \in T \Longrightarrow f\ x = g\ x$
⟨*proof*⟩ **lemma** *1*:
$S \cap -\ ([m]_{v,n}) =$
$(\bigcup\ (i, j) \in \{(i, j).\ (i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}.\ S \cap \{u.\ \neg\ dbm\_entry\_val'\ u\ i\ j\ (m\ i\ j)\})$

⟨*proof*⟩ **lemma** *UNION_remove*:
$(\bigcup\ x \in S.\ f\ x) = (\bigcup\ x \in T.\ g\ x)$
if $T \subseteq S \bigwedge x.\ x \in T \Longrightarrow f\ x = g\ x \bigwedge x.\ x \in S - T \Longrightarrow f\ x = \{\}$
⟨*proof*⟩ **lemma** *2*:
$(\bigcup (i, j) \in \{(i, j).(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n\}.\ S \cap \{u.\ \neg\ dbm\_entry\_val'\ u\ i\ j\ (m\ i\ j)\})$
$= (\bigcup (i, j) \in \{(i, j).(i > 0 \vee j > 0) \wedge i \leq n \wedge j \leq n \wedge m\ i\ j \neq \infty\}.$
$S \cap \{u.\ dbm\_entry\_val'\ u\ j\ i\ (neg\_dbm\_entry\ (m\ i\ j))\})$
⟨*proof*⟩

lemma *dbm_list_subtract*:
$dbm\_list\ xs \cap -\ ([m]_{v,n}) = dbm\_list\ (dbm\_minus\ n\ xs\ m)$
⟨*proof*⟩

end — Context for fixed DBM

end — Simplifier setup

lemma *dbm_list_empty_check*:
$dbm\_list\ xs = \{\} \longleftrightarrow list\_all\ (\lambda m.\ [m]_{v,n} = \{\})\ xs$
⟨*proof*⟩

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*
⟨*proof*⟩

end

Down

Auxiliary **lemma** *dbm_entry_le_iff*:
$Le\ a \leq Le\ b \longleftrightarrow a \leq b$
$Le\ a \leq Lt\ b \longleftrightarrow a < b$
$Lt\ a \leq Le\ b \longleftrightarrow a \leq b$
$Lt\ a \leq Lt\ b \longleftrightarrow a \leq b$
$0 \leq Le\ a \longleftrightarrow 0 \leq a$
$0 \leq Lt\ a \longleftrightarrow 0 < a$
$Le\ a \leq 0 \longleftrightarrow a \leq 0$
$Lt\ a \leq 0 \longleftrightarrow a \leq 0$
$\infty \leq x \longleftrightarrow x = \infty$
$x \leq \infty \longleftrightarrow True$
⟨*proof*⟩

lemma *dbm_entry_lt_iff*:
$Le\ a < Le\ b \longleftrightarrow a < b$
$Le\ a < Lt\ b \longleftrightarrow a < b$
$Lt\ a < Le\ b \longleftrightarrow a \leq b$
$Lt\ a < Lt\ b \longleftrightarrow a < b$
$0 < Le\ a \longleftrightarrow 0 < a$
$0 < Lt\ a \longleftrightarrow 0 < a$
$Le\ a < 0 \longleftrightarrow a < 0$
$Lt\ a < 0 \longleftrightarrow a \leq 0$
$x < \infty \longleftrightarrow x \neq \infty$
$\infty < x \longleftrightarrow False$
⟨*proof*⟩

lemmas [*dbm_entry_simps*] = *dbm_entry_le_iff*(1−9) *dbm_entry_lt_iff*(1−9)

lemma *Le_le_sum_iff*:
$Le\ (y :: \_ :: time) \leq e \longleftrightarrow 0 \leq e + Le\ (-\ y)$
⟨*proof*⟩

lemma *dense′*:
$\exists\, c{\geq}a.\ c \leq b$ **if** $a \leq b$ **for** $a :: \_ :: time$
⟨*proof*⟩

lemma *aux1*:
$-\ c \leq (a :: \_ :: time)$ **if** $a \geq 0\ c \geq 0$
⟨*proof*⟩

lemma *dbm_entries_dense*:
$\exists\, d{\geq}0.\ Le\ (-\ d) \leq l \wedge Le\ (d :: \_ :: time) \leq r$ **if** $0 \leq l\ l \leq r$
⟨*proof*⟩

lemma *dbm_entries_dense′_aux*:
$\exists\, d{\geq}0.\ l + Le\ d \geq 0 \wedge 0 \leq r + Le\ (-\ d :: \_ :: time)$ **if** $l \leq 0\ l + r \geq 0\ r \geq 0$

⟨*proof*⟩

lemma *dbm_entries_dense'*:
∃ *d*≥*0*. *l* + *Le d* ≥ *0* ∧ *0* ≤ *r* + *Le* (− *d* :: _ :: *time*) **if** *l* ≤ *0 l* + *r* ≥ *0*
⟨*proof*⟩

lemma (**in** *time*) *non_trivial_pos*: ∃ *x*. *x* > *0*
⟨*proof*⟩

lemma *dbm_entries_dense_pos*:
∃ *d*>*0*. *Le* (*d* :: _ :: *time*) ≤ *e* **if** *e* > *0*
⟨*proof*⟩

lemma *le_minus_iff*:
− *x* ≤ (*y* :: _ :: *time*) ⟷ *0* ≤ *y* + *x*
⟨*proof*⟩

lemma *lt_minus_iff*:
− *x* < (*y* :: _ :: *time*) ⟷ *0* < *y* + *x*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟧
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dbm_entry_val'_delay2*:
dbm_entry_val' *u* (*0* :: *nat*) *c2* (*m c1 c2*) **if**

dbm_entry_val' $(u \oplus d)$ *c1 c2* $(m$ *c1 c2*$)$ $d \geq 0$
c1 > 0 *c2* > 0 *c1* $\leq n$ *c2* $\leq n$
$\forall\ c \leq n.\ c > 0 \longrightarrow u\ c \geq 0$
⟨*proof*⟩

lemma *dbm_entry_val'_nonneg_bound*:
dbm_entry_val' u $(0 :: nat)$ c $(Le\ 0)$ **if** $u\ c \geq 0\ c > 0$
⟨*proof*⟩

lemma *neg_diag_empty_spec*:
$[\![M]\!] = \{\}$ **if** $i \leq n$ $M\ i\ i < 0$
⟨*proof*⟩

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* $\leq n$
⟨*proof*⟩

context
fixes $M :: ('t::time)\ DBM$
assumes $[\![M]\!] \neq \{\}$
begin

lemma *non_empty_diag_0_0*: $M\ 0\ 0 \geq 0$
⟨*proof*⟩

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*⟩

lemma *non_empty_cycle_free*:
cycle_free M n
⟨*proof*⟩

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** $u \in [\![M]\!]\ u\ c = -\ r$
⟨*proof*⟩

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*⟩

end

end

Definition definition
down $:: nat \Rightarrow ('t::linordered\_cancel\_ab\_monoid\_add)\ DBM \Rightarrow 't\ DBM$
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*} ⟨*proof*⟩

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*⟩

lemma *down_sound*: *u* ∈ ⟦*M*⟧↓ **if** *u* ∈ ⟦*down n M*⟧ *canonical M n*
⟨*proof*⟩

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*⟩

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 $I \subseteq \{0..n\}$
and $a \leq n$ $b \leq n$
shows *canonical_subs n* $(I \cup \{a,b\})$ *(repair_pair n M a b)* $\vee$ $(\exists i{\leq}n.$ *repair_pair n M a b i i* $< 0)$
⟨*proof*⟩

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*
⟨*proof*⟩

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** $a \leq n$ $b \leq n$
⟨*proof*⟩

context
fixes *c1 c2 c x* :: *nat*
notes [*simp*] = *dbm_entry_val'_iff_bounded dbm_entry_simps DBM.add algebra_simps*
begin

lemma *dbm_entry_val'_diag_iff*: *dbm_entry_val' u c c e* $\longleftrightarrow$ $e \geq 0$ **if** $c > 0$
⟨*proof*⟩

lemma *dbm_entry_val'_inf*: *dbm_entry_val' u c1 c2* ∞ $\longleftrightarrow$ *True*
⟨*proof*⟩

lemma *dbm_entry_val'_reset_1*:
dbm_entry_val' $(u(x := d))$ *x c e* $\longleftrightarrow$ *dbm_entry_val' u 0 c* $(e +$ *Le* $(-d))$
if $d \geq 0$ $c \neq x$ $c > 0$ $x > 0$
⟨*proof*⟩

lemma *dbm_entry_val'_reset_2*:
dbm_entry_val' $(u(x := d))$ *c x e* $\longleftrightarrow$ *dbm_entry_val' u c* $(0 ::$ *nat*$)$ $(e +$ *Le d*$)$
if $d \geq 0$ $c \neq x$ $c > 0$ $x > 0$
⟨*proof*⟩

lemma *dbm_entry_val'_reset_2'*:
dbm_entry_val' $(u(x := d))$ *0 x e* $\longleftrightarrow$ *Le* $(-$ *d*$) \leq e$ **if** $d \geq 0$ $x > 0$
⟨*proof*⟩

lemma *dbm_entry_val'_reset_3*:
dbm_entry_val' $(u(x := d))$ *c1 c2 e* $\longleftrightarrow$ *dbm_entry_val' u c1 c2 e* **if** $c1 \neq x$ $c2 \neq x$ **for** *e*
⟨*proof*⟩

end

Correctness context
fixes $M :: ('t::time)\ DBM$
begin

lemma *free_complete*: $u(x := d) \in [\![free\ n\ M\ x]\!]$
if *assms*: $u \in [\![M]\!]\ d \geq 0\ x > 0\ \forall\, c \leq n.\ M\ c\ c \geq 0$
⟨*proof*⟩

lemma *free_sound*: $\exists\, d \geq 0.\ u(x := d) \in [\![M]\!]\ u\ x \geq 0$
if *assms*: $u \in [\![free\ n\ M\ x]\!]\ x > 0\ x \leq n\ canonical\ M\ n\ M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$
⟨*proof*⟩

lemma *free_correct*:
$[\![free\ n\ M\ x]\!] = \{u(x := d) \mid u\ d.\ u \in [\![M]\!] \wedge d \geq 0\}$
if $x > 0\ x \leq n\ \forall\, c \leq n.\ M\ c\ c \geq 0\ \forall\ u \in [\![M]\!].\ u\ x \geq 0\ canonical\ M\ n$
$M\ 0\ x \leq 0\ M\ 0\ 0 \leq 0$
⟨*proof*⟩

lemma *pre_reset_correct_aux*:
$\{u.\ (u(x := (0::'t))) \in [\![M]\!]\} \cap \{u.\ u\ x \geq 0\} = \{u(x := d) \mid u\ d.\ u \in [\![M]\!] \wedge u\ x = 0 \wedge d \geq 0\}$
⟨*proof*⟩

lemma *restrict_zero_correct*:
$[\![restrict\_zero\ n\ M\ x]\!] = \{u.\ u \in [\![M]\!] \wedge u\ x = 0\}$ **if** $0 < x\ x \leq n$
⟨*proof*⟩

lemma *restrict_zero_canonical*:
$canonical\ (restrict\_zero\ n\ M\ x)\ n \vee check\_diag\ n\ (uncurry\ (restrict\_zero\ n\ M\ x))$
if $canonical\ M\ n\ x \leq n$
⟨*proof*⟩

end

10.4 Structural Properties

lemma *free_canonical*:
$canonical\ (free\ n\ M\ x)\ n$ **if** $canonical\ M\ n\ M\ x\ x \geq 0$
⟨*proof*⟩

lemma *free_diag*:
$free\ n\ M\ x\ i\ i = M\ i\ i$
⟨*proof*⟩

lemma *check_diag_free*:
$check\_diag\ n\ (uncurry\ (free\ n\ M\ x))$ **if** $check\_diag\ n\ (uncurry\ M)$
⟨*proof*⟩

lemma
$\forall\, i{\leq}n.\ (free\ n\ M\ x)\ i\ i \leq 0$ **if** $\forall\, i{\leq}n.\ M\ i\ i \leq 0$
⟨*proof*⟩

lemma *canonical_nonneg_diag_non_empty*:
assumes $canonical\ M\ n\ \forall\, i{\leq}n.\ 0 \leq M\ i\ i$
shows $[M]_{v,n} \neq \{\}$

⟨*proof*⟩

lemma *V_structuralI*:
⟦*M*⟧ ⊆ *V* **if** ∀ *i* ≤ *n*. *i* > 0 ⟶ *M* 0 *i* ≤ 0
⟨*proof*⟩

lemma *canonical_V_non_empty_iff*:
assumes *canonical M n M* 0 0 ≤ 0
shows ⟦*M*⟧ ⊆ *V* ∧ ⟦*M*⟧ ≠ {} ⟷ (∀ *i* ≤ *n*. *i* > 0 ⟶ *M* 0 *i* ≤ 0) ∧ (∀ *i* ≤ *n*. *M i i* ≥ 0)
⟨*proof*⟩

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 ⟶ *free n M x* 0 *i* ≤ 0) ∧ (∀ *i* ≤ *n*. *free n M x i i* ≥ 0)
⟨*proof*⟩

lemma
assumes (∀ *i* ≤ *n*. *i* > 0 ⟶ *M* 0 *i* ≤ 0) ∧ (∀ *i* ≤ *n*. *M i i* ≥ 0) ⟹
(∀ *i* ≤ *n*. *i* > 0 ⟶ *f M* 0 *i* ≤ 0) ∧ (∀ *i* ≤ *n*. *f M i i* ≥ 0)
assumes *canonical M n canonical* (*f M*) *n*
assumes *M* 0 0 ≤ 0 *f M* 0 0 ≤ 0
assumes *check_diag*: *check_diag n* (*uncurry M*) ⟹ *check_diag n* (*uncurry* (*f M*))
assumes ⟦*M*⟧ ⊆ *V*
shows ⟦*f M*⟧ ⊆ *V*
⟨*proof*⟩

lemma
⟦*free n M x*⟧ ⊆ *V* **if** *assms*: *x* > 0 *canonical M n M* 0 0 ≤ 0 0 ≤ *M x x* ⟦*M*⟧ ⊆ *V*
⟨*proof*⟩

lemma
down n M i i = *M i i*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *check_diag_empty*:
⟦*M*⟧ = {} **if** *check_diag n* (*uncurry M*)
⟨*proof*⟩

lemma *restrict_zero_mono*:
restrict_zero n M x i j ≤ *M i j* **if** *i* ≤ *n j* ≤ *n*
⟨*proof*⟩

lemma *restrict_zero_diag*:
check_diag n (*uncurry* (*restrict_zero n M x*)) **if** *check_diag n* (*uncurry M*)
⟨*proof*⟩

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*⟩

lemma *zone_set_pre_Cons*:
zone_set_pre ⟦*M*⟧ (*x* # *r*) = *zone_set_pre* {*u*. (*u*(*x* := (0::′*t*::*time*))) ∈ ⟦*M*⟧} *r*
⟨*proof*⟩

lemma *pre_reset_list_Cons*:
pre_reset_list n M (*x* # *r*) = *pre_reset_list n* (*pre_reset n M x*) *r*
⟨*proof*⟩

lemma *pre_reset_diag*:
check_diag n (*uncurry* (*pre_reset n M x*)) **if** *check_diag n* (*uncurry M*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *check_diag_repair_pair*:
assumes *check_diag n* (*uncurry M*)
shows *check_diag n* (*uncurry* (*repair_pair n M i j*))
⟨*proof*⟩

lemma *check_diag_and_entry*:
assumes *check_diag n* (*uncurry M*)
shows *check_diag n* (*uncurry* (*and_entry a b e M*))
⟨*proof*⟩

lemma *canonical'_and_entry_repair*:
canonical' n (*and_entry_repair n i j e M*) **if** *canonical' n M i* ≤ *n j* ≤ *n*
⟨*proof*⟩

lemma *dbm_minus_canonical_canonical'*:
∀ *M* ∈ *set* (*dbm_minus_canonical n xs m*). *canonical' n M* **if** ∀ *M* ∈ *set xs*. *canonical' n M*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

context *Default_Nat_Clock_Numbering*
begin

lemma *canonical_empty_zone*:
$[M]_{v,n} = \{\} \longleftrightarrow (\exists\, i{\le}n.\ M\ i\ i < 0)$ **if** *canonical M n*
⟨*proof*⟩

lemma *check_diag_iff_empty*:
check_diag n (*uncurry M*) $\longleftrightarrow$ $[\![M]\!] = \{\}$ **if** *canonical′ n M*
⟨*proof*⟩

lemma *and_entry_repair_zone_equiv*:
$[\![$*and_entry_repair n a b e M*$]\!] = [\![$*and_entry a b e M*$]\!]$ **if** $a \le n\ b \le n$
⟨*proof*⟩

lemma *dbm_minus_rel*:
assumes *list_all2* ($\lambda x\ y.\ [\![x]\!] = [\![y]\!]$) *ms ms′*
shows *list_all2* ($\lambda x\ y.\ [\![x]\!] = [\![y]\!]$) (*dbm_minus n ms m*) (*dbm_minus_canonical n ms′ m*)
⟨*proof*⟩

lemma *dbm_minus_canonical_fold_canonical′*:
$\forall\, M \in$ *set* (*fold* ($\lambda m\ S.$ *dbm_minus_canonical n S m*) *xs ms*). *canonical′ n M*
if $\forall\, M \in$ *set ms. canonical′ n M* **for** *ms* **and** *xs* :: (′*t* ::*time*) *DBM list*
⟨*proof*⟩

lemma *not_check_diag_nonnegD*:
$M\ i\ i \ge 0$ **if** $\neg$ *check_diag n* (*uncurry M*) $i \le n$
⟨*proof*⟩

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 $[\![M]\!] \subseteq (\bigcup m{\in}$*set xs.* $[\![m]\!]) \longleftrightarrow$ *dbm_subset_fed n M xs*
⟨*proof*⟩

lemma *dbm_minus_canonical_check_fed_equiv*:
dbm_list (*dbm_minus_canonical_check n S m*) = *dbm_list* (*dbm_minus_canonical n S m*)
⟨*proof*⟩

lemma *dbm_minus_canonical_dbm_minus*:
dbm_list (*dbm_minus_canonical n xs m*) = *dbm_list* (*dbm_minus n xs m*)
⟨*proof*⟩

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′* $0 \le m\ 0\ 0$
⟨*proof*⟩

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* (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_check n S m*) *xs* [*M*]))
⟨*proof*⟩

lemma *subset_if_pointwise_le*:
⟦*M*⟧ ⊆ ⟦*M'*⟧ **if** *pointwise_cmp* (≤) *n M M'*
⟨*proof*⟩

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*⟩

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* ($\lambda x.\ \neg\ P\ x$) *xs*) $\longleftrightarrow$ (*filter* ($\lambda x.\ \neg\ P\ x$) *xs*) = []
⟨*proof*⟩

lemma *dbm_subset_fed_upd_alt_def*:
dbm_subset_fed_upd n M xs ≡
let xs = *filter* ($\lambda M.\ \neg$ *check_diag n M*) *xs*
in if xs = [] *then check_diag n M*
else if list_ex ($\lambda M'.$ *dbm_subset' n M M'*) *xs then True*
else fold ($\lambda m\ S.$ *dbm_minus_canonical_check_upd n S m*) *xs* $[M]$ = []
⟨*proof*⟩

definition
V_dbm' n = ($\lambda(i, j).$ (*if* $i = j \vee i = 0 \wedge j > 0 \vee i > n \vee j > n$ *then 0 else* ∞))

definition
down_upd :: *nat* ⇒ *_ DBM'* ⇒ *_ DBM'*
where
down_upd n M ≡ $\lambda(i, j)$.
if $i = 0 \wedge j > 0 \wedge i \leq n \wedge j \leq n$ *then Min* ({*Le 0*} ∪ {*M* (*k*, *j*) | *k*. $1 \leq k \wedge k \leq n$}) *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* $i = x \wedge j \neq x \wedge i \leq n \wedge j \leq n$
then ∞ *else if* $i \neq x \wedge j = x \wedge i \leq n \wedge j \leq n$ *then M* (*i*, *0*) *else M* (*i*, *j*)

definition
pre_reset_upd n M x ≡ *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*
⟨*proof*⟩

lemma *fold_min_transfer*:
(*(list_all2 (rel_DBMEntry ri)) ===> rel_DBMEntry ri ===> rel_DBMEntry ri) (fold min) (fold min)*
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

lemma *check_diag_transfer*[*transfer_rule*]:
(eq_onp (λx. x = n) ===> RI2 ===> (=)) (λn M. check_diag n (uncurry M)) check_diag
⟨*proof*⟩

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

⟨*proof*⟩

lemma *le_rel_DBMEntry_iff*:
a ≤ *b* ⟷ *x* ≤ *y* **if** *rel_DBMEntry ri a x rel_DBMEntry ri b y*
⟨*proof*⟩

lemma *dbm_subset'_transfer*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *RI2* ===> (=))
(λ *n M M'*. *dbm_subset' n* (*uncurry M*) (*uncurry M'*)) *dbm_subset'*
⟨*proof*⟩

lemma *dbm_subset_fed_transfer*:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *list_all2 RI2* ===> (=)) *dbm_subset_fed_check dbm_subset_fed_upd*
⟨*proof*⟩

lemma *V_dbm_transfer*[*transfer_rule*]:
RI2 V_dbm (*V_dbm' n*)
⟨*proof*⟩

lemma *down_transfer*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> *RI2*) *down down_upd*
⟨*proof*⟩

lemma *free_upd*[*transfer_rule*]:
(*eq_onp* (λ*x*. *x* = *n*) ===> *RI2* ===> (=) ===> *RI2*) *free free_upd*
⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *diag_unbounded_dbm*:
unbounded_dbm i i = *0*
⟨*proof*⟩

lemma *down_diag*:
$down\ n\ M\ i\ i = M\ i\ i$
⟨*proof*⟩

definition
$abstr\_FW\ n\ cc\ M\ v \equiv FW\ (abstr\ cc\ M\ v)\ n$

definition
$abstr\_FW\_upd\ n\ cc\ M \equiv FW'\ (abstr\_upd\ cc\ M)\ n$

lemma *abstr_mono*:
$abstr\ cc\ M\ v\ i\ j \leq M\ i\ j$
⟨*proof*⟩

lemma *abstr_FW_mono*:
$abstr\_FW\ n\ cc\ M\ v\ i\ j \leq M\ i\ j$ **if** $i \leq n\ j \leq n$
⟨*proof*⟩

lemma *abstr_FW_diag_preservation*:
$\forall\, k{\leq}n.\ abstr\_FW\ n\ cc\ M\ v\ k\ k \leq 0$ **if** $\forall\, k{\leq}n.\ M\ k\ k \leq 0$
⟨*proof*⟩

lemma *FW_canonical*:
$canonical'\ n\ (FW\ M\ n)$
⟨*proof*⟩

lemma *abstr_FW_canonical*:
$canonical'\ n\ (abstr\_FW\ n\ cc\ M\ v)$
⟨*proof*⟩

lemma *down_check_diag*:
$check\_diag\ n\ (uncurry\ (down\ n\ M))$ **if** $check\_diag\ n\ (uncurry\ M)$
⟨*proof*⟩

context *Default_Nat_Clock_Numbering*
begin

lemma *clock_numbering*:
$\forall\ c.\ v\ c > 0 \wedge (\forall\, x.\ \forall\, y.\ v\ x \leq n \wedge v\ y \leq n \wedge v\ x = v\ y \longrightarrow x = y)$
⟨*proof*⟩

lemma *V_dbm_correct*:
$[\![V\_dbm]\!] = V$
⟨*proof*⟩

lemma *abstr_correct*:
$[\![abstr\ cc\ M\ v]\!] = [\![M]\!] \cap \{u.\ u \vdash cc\}$ **if** $\forall\, c{\in}collect\_clks\ cc.\ 0 < c \wedge c \leq n$
⟨*proof*⟩

lemma *unbounded_dbm_correct*:
$[\![unbounded\_dbm]\!] = UNIV$
⟨*proof*⟩

lemma *abstr_correct'*:
⟦*abstr cc unbounded_dbm v*⟧ = {*u*. *u* ⊢ *cc*} **if** ∀ *c*∈*collect_clks cc*. 0 < *c* ∧ *c* ≤ *n*
⟨*proof*⟩

lemma *abstr_FW_correct*:
⟦*abstr_FW n cc M v*⟧ = ⟦*M*⟧ ∩ {*u*. *u* ⊢ *cc*} **if** ∀ *c*∈*collect_clks cc*. 0 < *c* ∧ *c* ≤ *n*
⟨*proof*⟩

lemma *abstr_FW_correct'*:
⟦*abstr_FW n cc unbounded_dbm v*⟧ = {*u*. *u* ⊢ *cc*} **if** ∀ *c*∈*collect_clks cc*. 0 < *c* ∧ *c* ≤ *n*
⟨*proof*⟩

lemma *down_V*:
⟦*down n M*⟧ ⊆ *V*
⟨*proof*⟩

lemma *down_correct'*:
⟦*down n M*⟧ = ⟦*M*⟧$^{\downarrow}$ ∩ *V* **if** *canonical M n*
⟨*proof*⟩

lemma *down_correct*:
⟦*down n M*⟧ = ⟦*M*⟧$^{\downarrow}$ ∩ *V* **if** *canonical' n M*
⟨*proof*⟩

lemma *pre_reset_diag_preservation*:
pre_reset n M x i i ≤ *M i i* **if** *i* ≤ *n*
⟨*proof*⟩

lemma *pre_reset_list_diag*:
pre_reset_list n M r i i ≤ *M i i* **if** *i* ≤ *n*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

end

end

lemma *RI2_trivial_transfer*[*transfer_rule*]: (*RI2 n*) (*curry* (*conv_M M*)) *M*
⟨*proof*⟩

end
theory *Deadlock_Impl*
imports *Deadlock Munta_Base.Abstract_Term*
begin

Misc **lemma** *constraint_clk_conv_ac*:
constraint_clk (*conv_ac ac*) = *constraint_clk ac*
⟨*proof*⟩

lemma *constraint_clk_conv_cc*:
collect_clks (*conv_cc cc*) = *collect_clks cc*
⟨*proof*⟩

lemma *atLeastLessThan_alt_def*:
{*a*..<*b*} = {*k*. *a* ≤ *k* ∧ *k* < *b*}
⟨*proof*⟩

lemma *atLeastLessThan_Suc_alt_def*:
{*a*..<*Suc b*} = {*k*. *a* ≤ *k* ∧ *k* ≤ *b*}
⟨*proof*⟩

lemma (**in** *Graph_Defs*) *deadlock_if_deadlocked*:
deadlock y **if** *deadlocked y*
⟨*proof*⟩

10.8 Functional Refinement

Elementary list operations **lemma** *map_conv_rev_fold*:
map f xs = *rev* (*fold* (λ *a xs*. *f a* # *xs*) *xs* [])
⟨*proof*⟩

lemma *concat_map_conv_rev_fold*:
concat (*map f xs*) = *rev* (*fold* (λ *xs ys*. *rev* (*f xs*) @ *ys*) *xs* [])
⟨*proof*⟩

lemma *concat_conv_fold_rev*:
concat xss = *fold* (@) (*rev xss*) []
⟨*proof*⟩

lemma *filter_conv_rev_fold*:
filter P xs = *rev* (*fold* (λ*x xs*. *if P x then x* # *xs else xs*) *xs* [])
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *upd_pairs_Cons*:
upd_pairs ((*p*, *q*) # *xs*) *f* = *upd_pairs xs* (*f*(*p*:=*q*))
⟨*proof*⟩

lemma *upd_pairs_no_upd*:
assumes *p* ∉ *fst* ‘ *set xs*
shows *upd_pairs xs f p* = *f p*
⟨*proof*⟩

lemma *upd_pairs_upd*:
assumes (*p*, *y*) ∈ *set xs distinct* (*map fst xs*)
shows *upd_pairs xs f p* = *y*
⟨*proof*⟩

lemma *upd_pairs_append*:
upd_pairs (*xs* @ *ys*) = *upd_pairs ys o upd_pairs xs*
⟨*proof*⟩

lemma *upd_pairs_commute_single*:
upd_pairs xs (*f*(*a* := *b*)) = (*upd_pairs xs f*)(*a* := *b*) **if** *a* ∉ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *upd_pairs_append′*:
upd_pairs (*xs* @ *ys*) = *upd_pairs xs o upd_pairs ys* **if** *fst* ‘ *set xs* ∩ *fst* ‘ *set ys* = {}
⟨*proof*⟩

definition
upd_pairs′ xs = fold (λ(*p*,*q*) *f*. *f*(*p*:=*f q*)) *xs*

lemma *upd_pairs′_Nil*:
upd_pairs′ [] *f* = *f*

⟨*proof*⟩

lemma *upd_pairs'_Cons*:
upd_pairs' ((*p*, *q*) # *xs*) *f* = *upd_pairs'* *xs* (*f*(*p*:=*f q*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *upd_pairs_map*:
upd_pairs (*map f xs*) = *fold* (λ*pq g*. *let* (*p*,*q*) = *f pq in g*(*p*:=*q*)) *xs*
⟨*proof*⟩

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*⟩

lemma *down_upd_alt_def1*:
down_upd n M =
fold (λ*j 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*⟩

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*⟩

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*⟩

lemma *free_upd_free_upd1*:
free_upd n M c = *free_upd1 n M c* **if** *c* > 0 *c* ≤ *n*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pre_reset_upd_pre_reset_upd1*:
pre_reset_upd n M c = *pre_reset_upd1 n M c* **if** *c* > 0 *c* ≤ *n*

⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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* ⟨*proof*⟩

sepref_definition *down_impl* **is**
RETURN o PR_CONST (*down_upd n*) :: *mtx_assn*^{d} →$_a$ *mtx_assn*
⟨*proof*⟩

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} →$_a$ *mtx_assn*
⟨*proof*⟩

sepref_definition *free_impl* **is**
uncurry (*RETURN oo PR_CONST* (*free_upd1 n*)) ::
[λ(_, *i*). *i*≤*n*]$_a$ *mtx_assn*^{d} ∗$_a$ *nat_assn*^{k} → *mtx_assn*
⟨*proof*⟩

sepref_definition *and_entry_impl* **is**
uncurry2 (*uncurry* (λ*x*. *RETURN ooo and_entry_upd x*)) ::
[λ(((*i*, *j*),_),_). *i*≤*n* ∧ *j* ≤ *n*]$_a$ *nat_assn*^{k} ∗$_a$ *nat_assn*^{k} ∗$_a$ *id_assn*^{k} ∗$_a$ *mtx_assn*^{d} → *mtx_assn*
⟨*proof*⟩

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 $\to$ *mtx_assn*
⟨*proof*⟩

lemmas [*sepref_fr_rules*] = *free_impl.refine restrict_zero_impl.refine*

sepref_definition *pre_reset_impl* **is**
uncurry (*RETURN oo PR_CONST* (*pre_reset_upd1 n*)) ::
$[\lambda(\_, i).\ i{\leq}n]_a$ mtx_assn^d $*_a$ $(clock\_assn)^k$ $\to$ *mtx_assn*
⟨*proof*⟩

lemmas [*sepref_fr_rules*] = *pre_reset_impl.refine*

sepref_definition *pre_reset_list_impl* **is**
uncurry (*RETURN oo PR_CONST* (*pre_reset_list_upd1 n*)) ::
mtx_assn^d $*_a$ $(list\_assn\ (clock\_assn))^k$ $\to_a$ *mtx_assn*
⟨*proof*⟩

sepref_definition *and_entry_repair_impl* **is**
uncurry2 (*uncurry* (λ*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 $\to$ *mtx_assn*
⟨*proof*⟩ **definition**
$V\_dbm'' = V\_dbm'\ n$

We use V_dbm'' because we cannot register V_dbm' twice with the refinement framework: we view it as a function first, and as a DBM later.

lemma $V\_dbm'\_alt\_def$:
$V\_dbm'\ n = op\_amtx\_new\ (Suc\ n)\ (Suc\ n)\ (V\_dbm'')$
⟨*proof*⟩

notation *fun_rel_syn* (**infixr** $\to$ *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''))))
$\in$ $unit\_assn^k$ $\to_a$ *pure* ($nat\_rel \times_r nat\_rel \to Id$)
⟨*proof*⟩

sepref_register $V\_dbm'' :: nat \times nat \Rightarrow \_$ *DBMEntry*

Necessary to solve side conditions of *op_amtx_new*

lemma $V\_dbm''\_bounded$:
$mtx\_nonzero\ V\_dbm'' \subseteq \{0..<Suc\ n\} \times \{0..<Suc\ n\}$
⟨*proof*⟩

We need to pre-process the lemmas due to a failure of *TRADE*

lemma $V\_dbm''\_bounded\_1$:
$(a, b) \in mtx\_nonzero\ V\_dbm'' \Longrightarrow a < Suc\ n$
⟨*proof*⟩

lemma *V_dbm''_bounded_2*:
(*a*, *b*) ∈ *mtx_nonzero V_dbm''* ⟹ *b* < *Suc n*
⟨*proof*⟩

lemmas [*sepref_opt_simps*] = *V_dbm''_def*

sepref_definition *V_dbm_impl* **is**
uncurry0 (*RETURN* (*PR_CONST* (*V_dbm' n*))) :: *unit_assn*^{k} →$_a$ *mtx_assn*
⟨*proof*⟩

This form of 'type casting' is used to assert a bound on the natural number.

private definition *make_clock* :: *nat* ⇒ *nat* **where** [*sepref_opt_simps*]:
make_clock x = *x*

lemma *make_clock*[*sepref_import_param*]:
(*make_clock*, *make_clock*) ∈ [λ*i*. *i* ≤ *n*]$_f$ *nat_rel* → *nbn_rel* (*Suc n*)
⟨*proof*⟩ **definition** *get_entries m* =
[(*make_clock i*, *make_clock j*).
i←[*0..<Suc n*], *j*←[*0..<Suc n*], (*i* > *0* ∨ *j* > *0*) ∧ *i* ≤ *n* ∧ *j* ≤ *n* ∧ *m* (*i*, *j*) ≠ ∞]

private lemma *get_entries_alt_def*:
get_entries m = [(*make_clock i*, *make_clock j*).
i←[*0..<Suc n*], *j*←[*0..<Suc n*], (*i* > *0* ∨ *j* > *0*) ∧ *m* (*i*, *j*) ≠ ∞]
⟨*proof*⟩ **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* ⇒ *'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*) ∈ *Id*→*Id* ⟨*proof*⟩

lemmas [*sepref_fr_rules*] = *and_entry_repair_impl.refine*

sepref_definition *upd_entry_impl* **is**
uncurry2 (*uncurry* (λ*x*. *RETURN ooo PR_CONST upd_entry x*)) ::
[λ(((*i*, *j*),_),_). *i*≤*n* ∧ *j* ≤ *n*]$_a$
nat_assn^k ∗$_a$ *nat_assn*^{k} ∗$_a$ *mtx_assn*^{k} ∗$_a$ *mtx_assn*^{k} → *mtx_assn*
⟨*proof*⟩

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*) ⟹ *intf_of_assn* (*list_assn*(*AA*)) *TYPE*(*'aa list*)

⟨*proof*⟩

lemmas [*sepref_fr_rules*] = *upd_entry_impl.refine*

sepref_definition *upd_entries_impl* **is**
uncurry2 (*uncurry* (λ*x*. *RETURN ooo PR_CONST upd_entries x*)) ::
[λ(((*i*, *j*),_),_). *i*≤*n* ∧ *j* ≤ *n*]$_a$
nat_assn^k ∗$_a$ *nat_assn*^{k} ∗$_a$ *mtx_assn*^{k} ∗$_a$ (*list_assn mtx_assn*)^k → *list_assn mtx_assn*
⟨*proof*⟩

lemma [*sepref_import_param*]: ((=), (=)) ∈ *Id*→*Id*→*Id* ⟨*proof*⟩

sepref_definition *get_entries_impl* **is**
RETURN o PR_CONST get_entries ::
mtx_assn^k →$_a$ *list_assn* ((*clock_assn*) ×$_a$ (*clock_assn*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩ **include** *lifting_syntax*
⟨*proof*⟩

lemmas [*sepref_fr_rules*] =
V_dbm_impl.refine abstr_FW_impl.refine pre_reset_list_impl.refine
down_impl.refine dbm_subset_fed_impl.refine

sepref_definition *check_deadlock_impl* **is**
uncurry (*RETURN oo PR_CONST check_deadlock_dbm*) ::
$location\_assn^k *_a (mtx\_assn\ n)^d \to_a bool\_assn$
⟨*proof*⟩

end

10.11 Checking a Property on the Reachable Set

locale *Worklist_Map2_Impl_check* =
Worklist_Map2_Impl_finite +
fixes *Q* :: *'a* ⇒ *bool*
fixes *Qi*
assumes *Q_refine*: (*Qi*,*RETURN o PR_CONST Q*) ∈ $A^d \to_a bool\_assn$
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* (λ*b*. ¬*b*)
(λ *passed' _*.
do {
passed' ← *SPEC* (λ*l*. *set l = passed'*);
nfoldli passed' (λ*b*. ¬*b*)
(λ*v' _*.
if Q v' then RETURN True else RETURN False
)
False
}

)
False;
RETURN r
}

lemma *check_passed_correct*:
check_passed ≤ *SPEC* (λ *r*. (*r* ⟷ (∃ *a*. *reachable a* ∧ ¬ *empty a* ∧ *Q a*)))
⟨*proof*⟩

schematic_goal *pw_algo_map2_refine*:
(*?x*, *uncurry0* (*PR_CONST pw_algo_map2*)) ∈
unit_assn^k →$_a$ *bool_assn* ×$_a$ *hm.hms_assn'* *K* (*lso_assn A*)
⟨*proof*⟩

sepref_register *pw_algo_map2*

sepref_register *PR_CONST Q*

sepref_thm *check_passed_impl* **is**
uncurry0 check_passed :: *unit_assn*^{k} →$_a$ *bool_assn*
⟨*proof*⟩

concrete_definition (**in** −) *check_passed_impl*
uses *Worklist_Map2_Impl_check.check_passed_impl.refine_raw* **is** (*uncurry0 ?f*,_)∈_

lemma *check_passed_impl_hnr*:
(*uncurry0* (*check_passed_impl succsi* 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*
⟨*proof*⟩

end

10.12 Complete Deadlock Checker

Miscellaneous Properties **context** *E_From_Op_Bisim*
begin

More general variant of *?F_rel* = (λ(*l*, *D*). *?F l* ∧ ¬ *check_diag n D*) ⟹ (∃ *l' D'*. *E*** 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'*))
⟨*proof*⟩

end

context *Regions_TA*
begin

lemma *check_deadlock_anti_mono*:
check_deadlock l Z **if** *check_deadlock l Z' Z* ⊆ *Z'*
⟨*proof*⟩

lemma *global_clock_numbering*:
global_clock_numbering A v n
⟨*proof*⟩

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*⟩ ⇝*β* ⟨*l'*, *Z'*⟩ ∧ *Z'* ≠ {}) (λ(*l*, *D*) (*l'*, *D'*). ∃ *a*. *?A* ⊢' ⟨*l*, *D*⟩ ⇝*Na* ⟨*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*⟩ ⇝*β* ⟨*l'*, *Z'*⟩)
(λ (*l*, *D*) (*l'*, *D'*). ∃ *a*. *A* ⊢' ⟨*l*, *D*⟩ ⇝*N*(*a*) ⟨*l'*, *D'*⟩)
(λ (*l*, *Z*) (*l'*, *D*). *l* = *l'* ∧ *Z* = [*D*]*v,n*)
(λ _. *True*) (λ (*l*, *D*). *valid_dbm D*)
⟨*proof*⟩

lemma *steps_z_beta_reaches*:
reaches (*l*, *Z*) (*l'*, *Z'*) **if** *A* ⊢ ⟨*l*, *Z*⟩ ⇝*β** ⟨*l'*, *Z'*⟩ *Z'* ≠ {}
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

lemma *reaches_steps_z_beta_iff*:
reaches (*l*, *Z*) (*l'*, *Z'*) ⟷ *A* ⊢ ⟨*l*, *Z*⟩ ⇝*β** ⟨*l'*, *Z'*⟩ ∧ *Z'* ≠ {} **if** *Z* ≠ {}
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

end

lemma *dbm_int_init_dbm*:
dbm_int (*curry init_dbm*) *n*
⟨*proof*⟩

context *TA_Start*
begin

lemma *wf_dbm_canonical'D*:
Deadlock.canonical' *n* (*curry* (*conv_M D*)) **if** *wf_dbm D*
⟨*proof*⟩

lemma *subsumes_dbm_subsetD*:
dbm_subset n M M' **if** *subsumes n* (*l*, *M*) (*l'*, *M'*) ¬ *check_diag n M*
⟨*proof*⟩

lemma *subsumes_loc_eqD*:
l = *l'* **if** *subsumes n* (*l*, *M*) (*l'*, *M'*) ¬ *check_diag n M*
⟨*proof*⟩

lemma *init_dbm_zone*:
[*curry* (*init_dbm* :: *real DBM'*)]*v,n* = {*u*. ∀ *c*∈{*1*..*n*}. *u c* = *0*}
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *not_check_deadlock_non_empty*:
Z ≠ {} **if** ¬ *TA.check_deadlock l Z*
⟨*proof*⟩

end

context *TA_Impl*
begin

lemma *op_E_from_op_iff*:
op.E_from_op = *E_op''.E_from_op*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *not_check_deadlock_non_empty*:
Z ≠ {} **if** ¬ *TA.check_deadlock l Z*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *deadlock_check_diag*:
$\neg$ *check_diag n M* **if** $\neg$ *check_deadlock_dbm l M E_op''.E_from_op*** a_0 (l, M)
⟨*proof*⟩

lemma *norm_final_bisim*:
Bisimulation_Invariant
$(\lambda(l, D)\ (l', D').\ \exists a.\ step\_z\_norm''\ (conv\_A\ A)\ l\ D\ a\ l'\ D')$
E_op''.E_from_op
$(\lambda\ (l, M)\ (l', D').\ l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = [M]_{v,n})$
$(\lambda(l, y).\ valid\_dbm\ y)$ *wf_state*
⟨*proof*⟩

interpretation *bisim*:
Bisimulation_Invariant
$\lambda(l, Z)\ (l', Z').\ step\_z\_beta'\ (conv\_A\ A)\ l\ Z\ l'\ Z'$
$\lambda a\ b.\ op.E\_from\_op\ a\ b$
$\lambda(l, Z)\ (l', D').\ l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = Z$
$\lambda\_.\ True\ \lambda(l, M).\ l \in states \wedge wf\_state\ (l, M)$
⟨*proof*⟩

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*⟩

lemma *deadlocked_if_not_is_start_in_states*:
deadlocked (l_0, Z_0) **if** $\neg$ *is_start_in_states*
⟨*proof*⟩

lemma *deadlock_if_not_is_start_in_states*:
deadlock (l_0, Z_0) **if** $\neg$ *is_start_in_states*
⟨*proof*⟩

sepref_definition *is_start_in_states_impl* **is**
uncurry0 (*RETURN is_start_in_states*) :: $unit\_assn^k \rightarrow_a bool\_assn$
⟨*proof*⟩

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*
$<\lambda r.\ \uparrow((r \longrightarrow l_0 \in$ *Simulation_Graphs_TA.state_set A*$)$
$\wedge\ (\neg r \longrightarrow deadlocked\ (l_0, \lambda\_\ .\ 0))$
$)>_t$
⟨*proof*⟩

lemma *deadlock_if_deadlocked*:
deadlock y **if** *deadlocked y*

⟨*proof*⟩

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*$_0$)))
)>$_t$
⟨*proof*⟩

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*}
⟨*proof*⟩

interpretation *bisim*:
Bisimulation_Invariant
λ(*l*, *Z*) (*l'*, *Z'*). *step_z_beta'* (*conv_A A*) *l Z l' Z'*
λ*a b*. *op.E_from_op a b*
λ(*l*, *Z*) (*l'*, *D'*). *l'* = *l* ∧ [*curry* (*conv_M D'*)]$_{v,n}$ = *Z*
λ_. *True* λ(*l*, *M*). *l* ∈ *states* ∧ *wf_state* (*l*, *M*)
⟨*proof*⟩

lemma *check_deadlock*:
(∃ *a*. *E_op''.E_from_op*** *a*$_0$ *a* ∧
¬ (*case a of* (*l*, *M*) ⇒ *check_diag n M*) ∧ (*case a of* (*l*, *M*) ⇒ ¬ *check_deadlock_dbm l M*))
⟷ (∃ *u*$_0$. (∀ *c* ∈ {*1..n*}. *u*$_0$ *c* = *0*) ∧ *deadlock* (*l*$_0$, *u*$_0$)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

lemma *check_deadlock'*:
(∄ *a*. *E_op''.E_from_op*** *a*$_0$ *a* ∧
¬ (*case a of* (*l*, *M*) ⇒ *check_diag n M*) ∧ (*case a of* (*l*, *M*) ⇒ ¬ *check_deadlock_dbm l M*))
⟷ (∀ *u*$_0$. (∀ *c* ∈ {*1..n*}. *u*$_0$ *c* = *0*) ⟶ ¬ *deadlock* (*l*$_0$, *u*$_0$)) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

context
assumes *F* = (λ _. *False*)
begin

interpretation *Worklist_Map2_Impl_check*
op.E_from_op a$_0$ *F_rel subsumes n succs* λ (*l*, *M*). *check_diag n M subsumes'*
λ (*l*, *M*). *F l state_assn'*
succs_impl a$_0$*_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*
⟨*proof*⟩

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⟩

end


context Reachability_Problem_precompiled
begin

lemma F_is_False_iff:
  PR_CONST F = (λ_. False) ⟷ final = []
  ⟨proof⟩

lemma F_impl_False: F_impl = (λ_. return False) if final = []
  ⟨proof⟩

definition deadlock_checker where
  deadlock_checker ≡
    let
      key = return ∘ fst;
      sub = subsumes_impl;
      copy = state_copy_impl;
      start = a₀_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 (∃ u₀. (∀ c∈{1..m}. u₀ c = 0) ∧ deadlock
(0, u₀))))
     ∈ unit_assn^k →ₐ bool_assn
  ⟨proof⟩

schematic_goal deadlock_checker_alt_def:
  deadlock_checker ≡ ?impl
  ⟨proof⟩

end

concrete_definition deadlock_checker_impl
  uses Reachability_Problem_precompiled.deadlock_checker_alt_def

export_code deadlock_checker_impl in SML_imp module_name TA

definition [code]:
  check_deadlock n m k I T ≡
    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⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

context
assumes *F_fun_False*: ⋀*x*. *F_fun x* = *False* **and** *F_False*: *F* = (λ_. *False*)
begin
lemma *F_impl_is_False*:
F_impl = (λ_. *return False*)
⟨*proof*⟩

lemma *deadlock_checker_correct*:
(*uncurry0 deadlock_checker*, *uncurry0* (*Refine_Basic.RETURN* (*deadlock* (l_0, λ_. *0*))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
⟨*proof*⟩

lemmas *deadlock_checker_hoare* = *deadlock_checker_correct*[*to_hnr*, *unfolded hn_refine_def*, *simplified*]

end

end

context *UPPAAL_Reachability_Problem_precompiled′*
begin

- *equiv.defs.states′*
- *EA* is *EA*, the state automaton constructed from UPPAAL network (*equiv*/*N*)
- *A* is *A*, the product automaton constructed from UPPAAL network (*equiv*/*N*)

context
fixes φ
assumes *formula_is_false*: *formula* = *formula.EX* (*and* φ (*not* φ))
begin

lemma *F_is_FalseI*:
shows *PR_CONST* ($\lambda(x, y)$. *F x y*) = (λ_. *False*)
⟨*proof*⟩

lemma *final_fun_is_False*:
final_fun a = *False*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *deadlock_checker_alt_def_refined*:
impl.deadlock_checker ≡ *?impl*
⟨*proof*⟩

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 s_0 *num_actions clock_ceiling*
then
deadlock_checker
num_processes num_clocks max_steps I T prog bounds program s_0 *num_actions clock_ceiling*
$\gg=$ (λx. *return* (*Some x*))
else return None

theorem *deadlock_check*:
<emp>
precond_dc p m k max_steps I T prog 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* ∧
r = *has_deadlock* (*conv* (*N p I P T prog bounds*)) *max_steps* (*repeat 0 p*, s_0, λ_ . *0*))
| *None* ⇒ ↑(¬ *UPPAAL_Reachability_Problem_precompiled′ p m max_steps I T prog bounds P* s_0 *na k*)
$>_t$
⟨*proof*⟩

export_code *precond_dc* **checking** *SML*

end

11 Renaming Identifiers in Simple Networks

theory *Simple_Network_Language_Renaming*
imports *Simple_Network_Language_Model_Checking*
begin

The part justifies a "renaming" step to normalize identifiers in the input.

unbundle *no_library_syntax*
notation (*input*) *TAG* (_ [] _ [*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*
⟨*proof*⟩

lemma *TAG_mp′*:
assumes *TAG t x x* ⟹ *y*
shows *y*
⟨*proof*⟩

lemmas *all_mono_rule* = *all_mono*[*THEN mp*, *OF impI*, *rotated*]

lemma *imp_mono_rule*:
assumes *P1* ⟶ *P2*
and *Q1* ⟹ *P1*
and *Q1* ⟹ *P2* ⟹ *Q2*
shows *Q1* ⟶ *Q2*
⟨*proof*⟩

lemma *Ball_mono*:
assumes ∀ *x* ∈ *S*. *P x* ⋀*x*. *x* ∈ *S* ⟹ *P x* ⟹ *Q x*
shows ∀ *x* ∈ *S*. *Q x*
⟨*proof*⟩

method *prop_monos* =
  erule all_mono_rule
  | *erule* (*1*) *imp_mono_rule*
  | *erule disj_mono*[*rule_format*, *rotated 2*]

locale *Prod_TA_Defs_urge* =
  fixes *A* :: ('*a*, '*s*, '*c*, '*t* :: {*zero*}, '*x*, '*v* :: *linorder*) *nta* **and** *urge* :: '*c*
begin

definition
  add_reset ≡ λ(*C*, *U*, *T*, *I*).
    (*C*, {}, (λ(*l*, *b*, *g*, *a*, *f*, *r*, *l'*). (*l*, *b*, *g*, *a*, *f*, *urge* # *r*, *l'*)) ‘ *T*, *I*)

definition
  add_inv ≡ λ(*C*, *U*, *T*, *I*). (*C*, *U*, *T*,
    (λ*l*. *if l* ∈ *U then acconstraint.LE urge 0* # *I l else I l*))

definition *A_urge* ≡
  (*fst A*, *map add_reset* (*map add_inv* (*fst* (*snd A*))), *snd* (*snd A*))

definition
  N_urge i ≡ *map add_reset* (*map add_inv* (*fst* (*snd A*))) ! *i*

end

locale *Prod_TA_sem_urge* =
  Prod_TA_Defs_urge A urge + *Prod_TA_sem A*
  for *A* :: ('*a*, '*s*, '*c*, '*t* :: {*zero*, *time*}, '*x*, '*v* :: *linorder*) *nta* **and** *urge* :: '*c* +
  assumes *urge_not_in_invariants*:
    ∀(_, _, _, *I*) ∈ *set* (*fst* (*snd A*)). ∀*l*. *urge* ∉ *constraint_clk* ‘ *set* (*I l*)
  assumes *urge_not_in_guards*:
    ∀(_, _, *T*, _) ∈ *set* (*fst* (*snd A*)). ∀(*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *T*. *urge* ∉ *constraint_clk* ‘ *set g*
  assumes *urge_not_in_resets*:
    ∀(_, _, *T*, _) ∈ *set* (*fst* (*snd A*)). ∀(*l*, *b*, *g*, *a*, *f*, *r*, *l'*) ∈ *T*. *urge* ∉ *set r*
begin

lemma *N_urge_simp*:
  N_urge i = *add_reset* (*add_inv* (*N i*)) **if** *i* < *n_ps*
  ⟨*proof*⟩

lemma *inv_add_reset*:
  inv (*add_reset A'*) = *inv A'*
  ⟨*proof*⟩

lemma *inv_add_inv*:
  inv (*add_inv* (*C*, *U*, *T*, *I*)) = (λ*l*. *if l* ∈ *U then acconstraint.LE urge 0* # *I l else I l*)
  ⟨*proof*⟩

definition
  is_urgent L ≡ ∃ *p*<*n_ps*. *L* ! *p* ∈ *urgent* (*N p*)

lemma *map_nth'*:
  map ((!) *xs*) [*0*..<*n*] = *xs* **if** *n* = *length xs*
  ⟨*proof*⟩

lemma *A_urge_split*:
A_urge = (*broadcast*, *map N_urge* [*0*..<*n_ps*], *bounds*)
⟨*proof*⟩

lemma *inv_add_inv'*:
inv (*add_inv A'*) =
(λ*l*. *if l* ∈ *urgent A' then acconstraint.LE urge 0* # *inv A' l else inv A' l*)
⟨*proof*⟩

lemma *A_split'*:
A = (*fst A*, *fst* (*snd A*), *snd* (*snd A*))
⟨*proof*⟩

lemma *is_urgent_simp*:
(∃ *p*<*n_ps*. *L* ! *p* ∈ *urgent* (*map N* [*0*..<*n_ps*] ! *p*)) ⟷ *is_urgent L*
⟨*proof*⟩

lemma *urgent_N_urge*:
urgent (*N_urge p*) = {} **if** *p* < *n_ps*
⟨*proof*⟩

lemma *committed_N_urge*:
committed (*N_urge p*) = *committed* (*N p*) **if** *p* < *n_ps*
⟨*proof*⟩

lemma *is_urgent_simp2*:
(∃ *p*<*n_ps*. *L* ! *p* ∈ *urgent* (*map N_urge* [*0*..<*n_ps*] ! *p*)) ⟷ *False*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inv_add_invD*:
u ⊢ *inv A' l* **if** *u* ⊢ *inv* (*add_inv A'*) *l*
⟨*proof*⟩

lemma *inv_N_urge*:
inv (*N_urge p*) = *inv* (*add_inv* (*N p*)) **if** *p* < *n_ps*
⟨*proof*⟩

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*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *trans_N_urgeE*:
assumes $(l, b, g, a, f, r, l') \in trans\ (N\_urge\ p)\ p < n\_ps$
obtains $(l, b, g, a, f, tl\ r, l') \in trans\ (N\ p)\ r = urge\ \#\ tl\ r$
⟨*proof*⟩

lemma *urge_not_in_inv*:
$urge \notin constraint\_clk\ `\ set\ (inv\ (N\ p)\ l)$ **if** $p < n\_ps$
⟨*proof*⟩

lemma *urge_not_in_guards'*:
assumes $(l, b, g, a, f, r, l') \in trans\ (N\ p)\ p < n\_ps$
shows $urge \notin constraint\_clk\ `\ set\ g$
⟨*proof*⟩

lemma *urge_not_in_resets'*:
assumes $(l, b, g, a, f, r, l') \in trans\ (N\ p)\ p < n\_ps$
shows $urge \notin set\ r$
⟨*proof*⟩

lemma *clk_upd_clock_val_a_simp*:
$u(c := d) \vdash_a ac \longleftrightarrow u \vdash_a ac$ **if** $c \neq constraint\_clk\ ac$
⟨*proof*⟩

lemma *clk_upd_clock_val_simp*:
$u(c := d) \vdash cc \longleftrightarrow u \vdash cc$ **if** $c \notin constraint\_clk\ `\ set\ cc$
⟨*proof*⟩

lemma *inv_N_urgeI*:
assumes $u \vdash inv\ (N\ p)\ l\ p < n\_ps$
shows $u(urge := 0) \vdash inv\ (N\_urge\ p)\ l$
⟨*proof*⟩

lemma *inv_N_urge_updI*:
assumes $u \vdash inv\ (N\ p)\ l\ p < n\_ps$
shows $u(urge := d) \vdash inv\ (N\ p)\ l$
⟨*proof*⟩

lemma *inv_N_urge_upd_simp*:
assumes $p < n\_ps$
shows $u(urge := d) \vdash inv\ (N\ p)\ l \longleftrightarrow u \vdash inv\ (N\ p)\ l$
⟨*proof*⟩

lemma *inv_N_urge_updD*:
assumes $u(urge := d) \vdash inv\ (N\ p)\ l\ p < n\_ps$
shows $u \vdash inv\ (N\ p)\ l$
⟨*proof*⟩

lemma *inv_N_urgeD*:
assumes $u \vdash inv\ (N\_urge\ p)\ l\ p < n\_ps$
shows $u(urge := d) \vdash inv\ (N\ p)\ l$
⟨*proof*⟩

lemma *inv_N_urge_urges*:
assumes $\forall p{<}n\_ps.\ u(urge := 0) \oplus d \vdash inv\ (N\_urge\ p)\ (L\ !\ p)\ is\_urgent\ L$
shows $d \leq 0$
⟨*proof*⟩

lemma *trans_urge_upd_iff*:
assumes $(l,\ b,\ g,\ a,\ f,\ r,\ l') \in trans\ (N\ p)\ p < n\_ps$
shows $u(urge := d) \vdash g \longleftrightarrow u \vdash g$
⟨*proof*⟩

lemma *cval_add_0*[*simp*]:
$u \oplus (0 :: 'tt :: time) = u$
⟨*proof*⟩

lemma *clock_val_reset_delay*:
$u(c := 0) \oplus (d :: 'tt :: time) = (u \oplus d)(c := d)$
⟨*proof*⟩

lemma *clock_set_upd_simp*:
$[r \rightarrow d]u(c := d') = [r \rightarrow d]u$ **if** $c \in set\ r$
⟨*proof*⟩

lemma *fun_upd_twist2*:
$f(a := x,\ b := x) = f(b := x,\ a := x)$
⟨*proof*⟩

lemma *clock_set_upd_simp*:
$[c\ \#\ r \rightarrow d]u(c := d') = [c\ \#\ r \rightarrow d]u$
⟨*proof*⟩

lemma *clock_set_commute_single*:
$[r \rightarrow d]u(c := d') = ([r \rightarrow d]u)(c := d')$ **if** $c \notin set\ r$
⟨*proof*⟩

lemma *clock_set_upd_simp2*:
$([xs\ @\ c\ \#\ ys \rightarrow d]u)(c := d') = ([xs\ @\ ys \rightarrow d]u)(c := d')$
⟨*proof*⟩

lemma *clock_set_upd_simp3*:
$([xs \rightarrow d]u)(c := d') = ([filter\ (\lambda x.\ x \neq c)\ xs \rightarrow d]u)(c := d')$
⟨*proof*⟩

interpretation *urge*: *Prod_TA_sem A_urge* ⟨*proof*⟩

lemma *urge_n_ps_eq*:
$urge.n\_ps = n\_ps$
⟨*proof*⟩

lemma *urge_N_simp*[*simp*]:
$urge.N = N\_urge$
⟨*proof*⟩

lemma *urge_states_eq*:
$urge.states = states$

⟨*proof*⟩

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*⟩

lemmas *urge_bisim = Bisimulation_Invariant_axioms*

end

context *Simple_Network_Impl_Defs*
begin

lemma *dom_bounds*: *dom bounds = fst ‘ set bounds'*
⟨*proof*⟩

lemma *mem_trans_N_iff*:
$t \in$ *Simple_Network_Language.trans* $(N\ i) \longleftrightarrow t \in set\ (fst\ (snd\ (snd\ (automata\ !\ i))))$
if $i < n\_ps$
⟨*proof*⟩

lemma *length_automata_eq_n_ps*:
length automata = n_ps
⟨*proof*⟩

lemma *N_p_trans_eq*:
Simple_Network_Language.trans $(N\ p) = set\ (fst\ (snd\ (snd\ (automata\ !\ p))))$ **if** $p < n\_ps$
⟨*proof*⟩

lemma *loc_set_compute*:
$loc\_set = \bigcup\ ((\lambda(\_, \_, t, \_).\ \bigcup\ ((\lambda(l, \_, \_, \_, \_, \_, l').\ \{l, l'\})\ ‘\ set\ t))\ ‘\ set\ automata)$
⟨*proof*⟩

lemma *var_set_compute*:
var_set =
$(\bigcup S \in (\lambda t.\ (fst \circ snd)\ ‘\ set\ t)\ ‘\ ((\lambda(\_, \_, t, \_).\ t)\ ‘\ set\ automata).\ \bigcup b \in S.\ vars\_of\_bexp\ b)$
$\cup$
$(\bigcup S \in (\lambda t.\ (fst \circ snd\ o\ snd \circ snd \circ snd)\ ‘\ set\ t)\ ‘\ ((\lambda(\_, \_, t, \_).\ t)\ ‘\ set\ automata).$
$\bigcup f \in S.\ \bigcup\ (x, e) \in set\ f.\ \{x\} \cup vars\_of\_exp\ e)$
⟨*proof*⟩

lemma *states_loc_setD*:
$set\ L \subseteq loc\_set$ **if** $L \in states$
⟨*proof*⟩

end

context *Simple_Network_Impl*
begin

lemma *sem_bounds_eq*: *sem.bounds* = *bounds*
⟨*proof*⟩

lemma *n_ps_eq*[*simp*]:
sem.n_ps = *n_ps*
⟨*proof*⟩

lemma *sem_loc_set_eq*:
sem.loc_set = *loc_set*
⟨*proof*⟩

lemma *sem_states_eq*:
sem.states = *states*
⟨*proof*⟩

end

locale *Simple_Network_Rename_Defs* =
Simple_Network_Impl_Defs automata broadcast bounds' **for** *automata* ::
('*s list* × '*s list* × (('*a* :: *countable*) *act*, '*s*, '*c*, '*t*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, '*t*) *cconstraint*) *list*) *list*
and *broadcast bounds'* +
fixes *renum_acts* :: '*a* ⇒ *nat*
and *renum_vars* :: '*x* ⇒ *nat*
and *renum_clocks* :: '*c* ⇒ *nat*
and *renum_states* :: *nat* ⇒ '*s* ⇒ *nat*
begin

definition
map_cconstraint f g xs = *map* (*map_acconstraint f g*) *xs*

definition
renum_cconstraint = *map_cconstraint renum_clocks id*

definition
renum_act = *map_act renum_acts*

definition
renum_bexp = *map_bexp renum_vars*

definition
renum_exp = *map_exp renum_vars*

definition
renum_upd = (λ(*x*, *upd*). (*renum_vars x*, *renum_exp upd*))

abbreviation
renum_upds ≡ *map renum_upd*

definition
renum_reset = *map renum_clocks*

definition *renum_automaton* **where**
renum_automaton i ≡ 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*
⟨*proof*⟩

definition *renum_label* **where**
renum_label = *map_label renum_acts*

sublocale *renum*: *Simple_Network_Impl_Defs*
map_index renum_automaton automata
map renum_acts broadcast
map (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds′*
⟨*proof*⟩

lemma *renum_n_ps_simp*[*simp*]:
renum.n_ps = *n_ps*
⟨*proof*⟩

end

locale *Simple_Network_Rename_Defs_int* =
Simple_Network_Rename_Defs automata +
Simple_Network_Impl automata
for *automata* ::
('*s list* × '*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'*
⟨*proof*⟩

end

lemma
fixes *f* :: *'b* :: *countable* ⇒ *nat*
assumes *inj_on f S finite S infinite* (*UNIV* :: *'b set*)
shows *extend_bij_inj*: *inj* (*extend_bij f S*) **and** *extend_bij_surj*: *surj* (*extend_bij f S*)
and *extend_bij_extends*: ∀ *x* ∈ *S*. *extend_bij f S x* = *f x*
⟨*proof*⟩

lemma *default_map_of_map*:
default_map_of y (*map* (λ(*a*, *b*). (*f a*, *g b*)) *xs*) (*f a*) = *g* (*default_map_of x xs a*)
if *inj f y* = *g x*
⟨*proof*⟩

lemma *default_map_of_map_2*:
default_map_of y (*map* (λ(*a*, *b*). (*a*, *g b*)) *xs*) *a* = *g* (*default_map_of x xs a*) **if** *y* = *g x*
⟨*proof*⟩

lemma *map_of_map'*:
map_of (*map* (λ(*k*, *v*). (*k*, *f k v*)) *xs*)
= (λ*k*. *case map_of xs k of Some v* ⇒ *Some* (*f k v*) | _ ⇒ *None*)
⟨*proof*⟩

lemma *default_map_of_map_3*:
default_map_of y (*map* (λ(*a*, *b*). (*a*, *g a b*)) *xs*) *a* = *g a* (*default_map_of x xs a*)
if ⋀*k*. *y* = *g k x*
⟨*proof*⟩

lemma *dom_map_of_map*:
dom (*map_of* (*map* (λ (*a*, *b*). (*f a*, *g b*)) *xs*)) = *f* ‘ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *inj_image_eqI*:
S = *T* **if** *inj f f* ‘ *S* = *f* ‘ *T*
⟨*proof*⟩

lemmas [*finite_intros*] = *finite_set*

lemma *map_of_NoneI*:
map_of xs x = *None* **if** *x* ∉ *fst* ‘ *set xs*
⟨*proof*⟩

lemma *bij_f_the_inv_f*:
f (*the_inv f x*) = *x* **if** *bij f*
⟨*proof*⟩

fun *map_sexp* ::
(*nat* ⇒ *'s* ⇒ *'s1*) ⇒ (*'a* ⇒ *'a1*) ⇒ (*'b* ⇒ *'b1*) ⇒ (*nat*, *'s*, *'a*, *'b*) *sexp*
⇒ (*nat*, *'s1*, *'a1*, *'b1*) *sexp*

where
map_sexp _ _ _ sexp.true = sexp.true |
map_sexp f g h (not e) = not (map_sexp f g h e) |
map_sexp f g h (and e1 e2) = and (map_sexp f g h e1) (map_sexp f g h e2) |
map_sexp f g h (sexp.or e1 e2) = sexp.or (map_sexp f g h e1) (map_sexp f g h e2) |
map_sexp f g h (imply e1 e2) = imply (map_sexp f g h e1) (map_sexp f g h e2) |
map_sexp f g h (eq i x) = eq (g i) (h x) |
map_sexp f g h (lt i x) = lt (g i) (h x) |
map_sexp f g h (le i x) = le (g i) (h x) |
map_sexp f g h (ge i x) = ge (g i) (h x) |
map_sexp f g h (gt i x) = gt (g i) (h x) |
map_sexp f g h (loc i x) = loc i (f i x)

fun *map_formula* ::
(nat ⇒ 's ⇒ 's1) ⇒ ('a ⇒ 'a1) ⇒ ('b ⇒ 'b1)
⇒ (nat, 's, 'a, 'b) formula ⇒ (nat, 's1, 'a1, 'b1) formula
where
map_formula f g h (formula.EX φ) = formula.EX (map_sexp f g h φ) |
map_formula f g h (EG φ) = EG (map_sexp f g h φ) |
map_formula f g h (AX φ) = AX (map_sexp f g h φ) |
map_formula f g h (AG φ) = AG (map_sexp f g h φ) |
map_formula f g h (Leadsto φ ψ) = Leadsto (map_sexp f g h φ) (map_sexp f g h ψ)

locale *Simple_Network_Impl_real* =
fixes *automata* ::
('s list × 's list × ('a act, 's, 'c, real, 'x, int) transition list
× ('s × ('c, real) cconstraint) list) list
and *broadcast* :: *'a list*
and *bounds'* :: *('x × (int × int)) list*
begin

sublocale *Simple_Network_Impl_Defs automata broadcast bounds' ⟨proof⟩*

abbreviation *sem ≡ (set broadcast, map automaton_of automata, map_of bounds')*

sublocale *Prod_TA_sem (set broadcast, map automaton_of automata, map_of bounds') ⟨proof⟩*

end

context *Simple_Network_Impl*
begin

lemma *sem_state_guard_eq*:
(fst ∘ snd) ‘ trans (sem.N p) = (fst ∘ snd) ‘ trans (N p) **if** *p < n_ps*
⟨proof⟩

lemma *sem_state_update_eq*:
(fst ∘ snd ∘ snd ∘ snd ∘ snd) ‘ trans (sem.N p) = (fst ∘ snd ∘ snd ∘ snd ∘ snd) ‘ trans (N p)
if *p < n_ps*
⟨proof⟩

lemma *sem_var_set_eq*:
sem.var_set = var_set

⟨*proof*⟩

end

locale *Simple_Network_Rename_Defs_real* =
Simple_Network_Rename_Defs automata +
Simple_Network_Impl_real automata
for *automata* ::
('*s list* × '*s list*
× (('*a* :: *countable*) *act*, '*s*, '*c*, *real*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, *real*) *cconstraint*) *list*) *list*
begin

sublocale *renum*: *Simple_Network_Impl_real*
map_index renum_automaton automata
map renum_acts broadcast
map (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds'*
⟨*proof*⟩

end

locale *Simple_Network_Rename'* =
Simple_Network_Rename_Defs **where** *automata* = *automata* **for** *automata* ::
('*s list* × '*s list*
× (('*a* :: *countable*) *act*, '*s*, '*c*, '*t*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, '*t*) *cconstraint*) *list*) *list* +
assumes *bij_renum_clocks*: *bij renum_clocks*
and *renum_states_inj*: ∀ *p*<*n_ps*. *inj* (*renum_states p*)
and *bij_renum_vars*: *bij renum_vars*
and *bounds'_var_set*: *fst* ‘ *set bounds'* = *var_set*
and *inj_renum_acts*: *inj renum_acts*

locale *Simple_Network_Rename_real* =
Simple_Network_Rename_Defs_real **where** *automata* = *automata* +
Simple_Network_Rename' **where** *automata* = *automata*
for *automata* ::
('*s list* × '*s list*
× (('*a* :: *countable*) *act*, '*s*, '*c*, *real*, '*x* :: *countable*, *int*) *transition list*
× (('*s* :: *countable*) × ('*c* :: *countable*, *real*) *cconstraint*) *list*) *list* +
assumes *urgency_removed*: ∀ *i* < *n_ps*. *urgent* (*N i*) = {}
begin

lemma *aux_1*:
(λ*x*. *case case x of*
(*l*, *g*) ⇒ (*renum_states p l*, *renum_cconstraint g*) *of*
(*s*, *cc*) ⇒ (*s*, *map conv_ac cc*))
= (λ (*l*, *g*). (*renum_states p l*, *map conv_ac* (*renum_cconstraint g*)))

⟨*proof*⟩

lemma *clocks_inv_inv*:
clocks_inv (*renum_clocks a*) = *a*
⟨*proof*⟩

lemma *map_u_renum_cconstraint_clock_valI*:
map_u u ⊢ *renum_cconstraint cc* **if** *u* ⊢ *cc*
⟨*proof*⟩

lemma *map_u_renum_cconstraint_clock_valD*:
u ⊢ *cc* **if** *map_u u* ⊢ *renum_cconstraint cc*
⟨*proof*⟩

lemma *inj_renum_states*: *inj* (*renum_states p*) **if** *p* < *n_ps*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *dom_the_inv_comp*:
dom (*m o the_inv f*) = *f* ‘ *dom m* **if** *inj f range f* = *UNIV*
⟨*proof*⟩

lemma *inj_renum_vars*:
inj renum_vars
⟨*proof*⟩

lemma *surj_renum_vars*:
surj renum_vars
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *dom_bounds_var_set*: *dom bounds* = *var_set*
⟨*proof*⟩

lemma *sem_states_loc_setD*: *L* ! *p* ∈ *loc_set* **if** *p* < *length automata L* ∈ *states* **for** *L p*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *match_assumption2*:
assumes *P a1 b1 a1* = *a b1* = *b* **shows** *P a b*
⟨*proof*⟩

lemma *inj_pair*:
assumes *inj f inj g*
shows *inj* (λ(*a*, *b*). (*f a*, *g b*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *renum_acconstraint_eq_convD*:
assumes *map_acconstraint renum_clocks id g* = *conv_ac g′*

obtains *g1* **where** $g = conv\_ac\ g1\ g' = map\_acconstraint\ renum\_clocks\ id\ g1$
⟨*proof*⟩

lemma *renum_cconstraint_eq_convD*:
assumes $renum\_cconstraint\ g = conv\_cc\ g'$
obtains *g1* **where** $g = conv\_cc\ g1\ g' = renum\_cconstraint\ g1$
⟨*proof*⟩

lemma *trans_sem_N_renumI*:
assumes (
$renum\_states\ p\ l, renum\_bexp\ b, renum\_cconstraint\ g, renum\_act\ a, renum\_upds\ f, renum\_reset\ r,$
$renum\_states\ p\ l'$)
$\in trans\ (renum.N\ p)\ p < n\_ps$
shows $(l, b, g, a, f, r, l') \in trans\ (N\ p)$
⟨*proof*⟩

lemma *trans_sem_N_renumI'*:
assumes $(renum\_states\ p\ l, b, g, a, f, r, l') \in trans\ (renum.N\ p)\ p < n\_ps$
shows $\exists\ b'\ g'\ a'\ f'\ r'\ l1.$
$TRANS\ ''orig''\ [] (l, b', g', a', f', r', l1) \in Simple\_Network\_Language.trans\ (N\ p)$
$\wedge\ ''renum\ bexp''\ [] b = renum\_bexp\ b'\ \wedge\ ''renum\ guard''\ [] g = renum\_cconstraint\ g'$
$\wedge\ ''renum\ action''\ [] a = renum\_act\ a'\ \wedge\ ''renum\ upds''\ [] f = renum\_upds\ f'$
$\wedge\ ''renum\ reset''\ [] r = renum\_reset\ r' \wedge ''renum\ loc''\ [] l' = renum\_states\ p\ l1$
⟨*proof*⟩

lemma *committed_renum_eq*:
$committed\ (renum.N\ p) = renum\_states\ p\ `\ committed\ (N\ p)$ **if** $p < n\_ps$
⟨*proof*⟩

lemma *urgent_renum_eq*:
$urgent\ (renum.N\ p) = renum\_states\ p\ `\ urgent\ (N\ p)$ **if** $p < n\_ps$
⟨*proof*⟩

lemma *renum_urgency_removed*:
$\forall i < n\_ps.\ urgent\ (renum.N\ i) = \{\}$
⟨*proof*⟩

lemma *check_bexp_renumD*:
$check\_bexp\ s\ b\ bv \Longrightarrow check\_bexp\ (s\ o\ vars\_inv)\ (renum\_bexp\ b)\ bv$
and *is_val_renumD*:
$is\_val\ s\ e\ v \Longrightarrow is\_val\ (s\ o\ vars\_inv)\ (renum\_exp\ e)\ v$
⟨*proof*⟩

method *solve_case* =
auto
intro: *check_bexp_is_val.intros*
simp: *renum_bexp_def renum_exp_def vars_inv_def the_inv_f_f*[*OF inj_renum_vars*];
fail
| *use is_val.simps* **in** *fastforce*

lemma *check_bexp_renumI*:
$check\_bexp\ (s\ o\ vars\_inv)\ (renum\_bexp\ b)\ bv \Longrightarrow check\_bexp\ s\ b\ bv$
and *is_val_renumI*:

is_val (s o vars_inv) (renum_exp e) v ⟹ is_val s e v
⟨*proof*⟩

lemma *renum_reset_map_u*:
[renum_reset r→0]map_u u = map_u ([r→0]u)
⟨*proof*⟩

lemma *bounded_renumI'*:
assumes *bounded bounds s'*
shows *bounded renum.bounds (s' o vars_inv)*
⟨*proof*⟩

lemma *bounded_renumD'*:
assumes *bounded renum.bounds (s' o vars_inv) dom s' ⊆ var_set*
shows *bounded bounds s'*
⟨*proof*⟩

lemma *is_upd_renumD*:
assumes *is_upd s f s'*
shows *is_upd (s o vars_inv) (renum_upd f) (s' o vars_inv)*
⟨*proof*⟩

lemma *is_upds_renumD*:
assumes *is_upds s1 ps s'*
shows *is_upds (s1 o vars_inv) (renum_upds ps) (s' o vars_inv)*
⟨*proof*⟩

lemma *is_upds_concat_renumD*:
assumes *is_upds s1 (concat ps) s'*
shows *is_upds (s1 o vars_inv) (concat_map renum_upds ps) (s' o vars_inv)*
⟨*proof*⟩

lemma *Ball_mono*:
assumes ∀ *x* ∈ *S*. *P x* ⋀*x*. *x* ∈ *S* ⟹ *P x* ⟹ *Q x*
shows ∀ *x* ∈ *S*. *Q x*
⟨*proof*⟩

lemma *atLeastLessThan_upperD*:
assumes *S* ⊆ {*l*..<*u*} *x* ∈ *S*
shows *x* < *u*
⟨*proof*⟩

lemma *imp_mono_rule*:
assumes *P1* ⟶ *P2*
and *Q1* ⟹ *P1*
and *Q1* ⟹ *P2* ⟹ *Q2*
shows *Q1* ⟶ *Q2*
⟨*proof*⟩

lemma *inj_id*:
inj id
⟨*proof*⟩

lemma *step_single_renumD*:

assumes *step_u sem L s u a L' s' u' L* ∈ *states dom s* ⊆ *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'*)
⟨*proof*⟩

lemma *inj_the_inv*:
inj (*the_inv f*) **if** *bij f*
⟨*proof*⟩

lemma *inj_vars_inv*:
inj vars_inv
⟨*proof*⟩

lemma *comp_vars_inv_upd_commute*:
(*s o vars_inv*)(*x* ↦ *y*) = *s*(*vars_inv x* ↦ *y*) *o vars_inv*
⟨*proof*⟩

lemma *is_upd_renumI*:
assumes *is_upd* (*s o vars_inv*) (*renum_upd f*) *s'*
shows *is_upd s f* (*s' o renum_vars*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *is_upds_renumI'*:
assumes *is_upds* (*s o vars_inv*) (*renum_upds f*) *s'*
shows *is_upds s f* (*s' o renum_vars*)
⟨*proof*⟩

lemma *is_upds_renumI''*:
assumes *is_upds s* (*renum_upds ps*) *s'*
shows *is_upds* (*s o renum_vars*) *ps* (*s' o renum_vars*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bounded_renumD1*:
assumes *bounded renum.bounds s′ dom* $(s' \circ renum\_vars) \subseteq var\_set$
shows *bounded bounds* $(s'\ o\ renum\_vars)$
⟨*proof*⟩

lemma *renum_reset_append*:
$renum\_reset\ xs\ @\ renum\_reset\ ys = renum\_reset\ (xs\ @\ ys)$
⟨*proof*⟩

lemma *if_eq_distrib*:
$(if\ i = j\ then\ f\ i\ a\ else\ f\ j\ b) = (f\ j\ (if\ i = j\ then\ a\ else\ b))$
⟨*proof*⟩

lemma *dom_comp_eq_vimage*:
$dom\ (s\ o\ f) = f\ -`\ dom\ s$
⟨*proof*⟩

lemma *dom_comp_vars_inv_eqD*:
assumes $dom\ s' = dom\ (s\ o\ vars\_inv)$
shows $dom\ (s'\ o\ renum\_vars) = dom\ s$
⟨*proof*⟩

lemma *sem_trans_upd_domD*:
assumes $(L\ !\ p,\ b,\ g',\ a,\ f,\ r,\ l1) \in trans\ (N\ p)\ p < n\_ps$
shows $fst\ `\ set\ f \subseteq var\_set$
⟨*proof*⟩

lemma *SilD*:
fixes *map_action*
assumes $Sil\ a = map\_act\ map\_action\ a1$
obtains a' **where** $a1 = Sil\ a'\ a = map\_action\ a'$
⟨*proof*⟩

lemma *InD*:
fixes *map_action*
assumes $In\ a = map\_act\ map\_action\ a1$
obtains a' **where** $a1 = In\ a'\ a = map\_action\ a'$
⟨*proof*⟩

lemma *OutD*:
fixes *map_action*
assumes $Out\ a = map\_act\ map\_action\ a1$
obtains a' **where** $a1 = Out\ a'\ a = map\_action\ a'$
⟨*proof*⟩

lemma *In_OutD*:
assumes $In\ a = renum\_act\ a1\ Out\ a = renum\_act\ a2$
obtains a' **where** $a = renum\_acts\ a'\ a1 = In\ a'\ a2 = Out\ a'$
⟨*proof*⟩

lemma *renum_sem_broadcast_eq*:
$renum.broadcast = renum\_acts\ `\ set\ broadcast$
⟨*proof*⟩

lemma *inj_eq_iff*:
$f\ x = f\ y \longleftrightarrow x = y$ **if** *inj f*
⟨*proof*⟩

lemma *trans_sem_N_renum_broadcastI*:
assumes
$\forall\, p{\in}set\ ps.\ (renum\_states\ p\ (L\ !\ p),\ bs\ p,\ gs\ p,\ In\ a,\ fs\ p,\ rs\ p,\ ls\ p) \in trans\ (renum.N\ p)$
$set\ ps \subseteq \{0..{<}n\_ps\}$
obtains $bs'\ gs'\ fs'\ rs'\ ls'\ a'$ **where**
$\forall\, p{\in}set\ ps.\ (L\ !\ p,\ bs'\ p,\ gs'\ p,\ In\ a',\ fs'\ p,\ rs'\ p,\ ls'\ p) \in trans\ (N\ p)$
$\forall\, p{\in}set\ ps.\ bs\ p = renum\_bexp\ (bs'\ p)$
$\forall\, p{\in}set\ ps.\ gs\ p = renum\_cconstraint\ (gs'\ p)$
$ps \neq [] \longrightarrow a = renum\_acts\ a'$
$\forall\, p{\in}set\ ps.\ fs\ p = renum\_upds\ (fs'\ p)$
$\forall\, p{\in}set\ ps.\ rs\ p = renum\_reset\ (rs'\ p)$
$\forall\, p{\in}set\ ps.\ ls\ p = renum\_states\ p\ (ls'\ p)$
⟨*proof*⟩

lemmas *all_mono_rule* = *all_mono*[*THEN mp*, *OF impI*, *rotated*]

method *prop_monos* =
erule all_mono_rule
| *erule* (*1*) *imp_mono_rule*
| *erule disj_mono*[*rule_format*, *rotated 2*]

lemma *inv_sem_N_renum_broadcastI*:
assumes
$\forall\, pa{<}n\_ps.$
$[renum\_reset\ r\ @\ concat\ (map\ rs\ ps){\to}0]map\_u\ u$
$\vdash inv\ (renum.N\ pa)$
$((fold\ (\lambda p\ L.\ L[p := ls\ p])\ ps\ (map\_index\ renum\_states\ L))\ [p := renum\_states\ p\ l1]\ !\ pa)$
$\forall\, p{\in}set\ ps.\ rs\ p = renum\_reset\ (rs'\ p)$
$\forall\, p{\in}set\ ps.\ ls\ p = renum\_states\ p\ (ls'\ p)$
$\forall\, p{\in}set\ ps.\ ls'\ p \in (\bigcup(l,\ b,\ g,\ a,\ r,\ u,\ l') \in trans\ (N\ p).\ \{l,\ l'\})$
$l1 \in (\bigcup(l,\ b,\ g,\ a,\ r,\ u,\ l') \in trans\ (N\ p).\ \{l,\ l'\})$
$L \in states$
shows
$\forall\, pa{<}n\_ps.$
$[r\ @\ concat\ (map\ rs'\ ps){\to}0]u \vdash inv\ (N\ pa)\ ((fold\ (\lambda p\ L.\ L[p := ls'\ p])\ ps\ L)\ [p := l1]\ !\ pa)$
⟨*proof*⟩

method *solve_triv* =
assumption
| *erule* (*1*) *bounded_renumD'*; *fail*
| *rule inv_renum_sem_D*[*OF* _ _ *sem_states_loc_setD*]; *simp*; *fail*
| *rule check_bexp_renumI*; *simp*; *fail*
| *rule map_u_renum_cconstraint_clock_valD*; *simp*; *fail*
| *rule is_upd_renumI is_upd_renumI'' is_upds_renumI' is_upds_renumI'' is_upds_concat_renumI''*,
simp; *fail*
| *simp*; *fail*
| *simp add*:
vars_inv_def bij_f_the_inv_f[*OF bij_renum_vars*] *renum_reset_map_u*[*symmetric*] *map_index_update*
renum_reset_append;
fail

| *subst nth_map*, (*simp*; *fail*)+; *fail*
| (*rule state_preservation_updI*, *blast*)+, *simp*; *fail*

lemma *trans_sem_upd_domI*:
assumes (*L* ! *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*)
⟨*proof*⟩

lemma *step_single_renumI*:
assumes
step_u renum.sem
(*map_index renum_states L*) (*s o vars_inv*) (*map_u u*) *a L' s' u'*
L ∈ *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*
⟨*proof*⟩

lemma *step_u_var_set_invariant*:
assumes *step_u sem L s u a L' s' u' dom s* = *var_set*
shows *dom s'* = *var_set*
⟨*proof*⟩

lemmas *step_u_invariants* = *states_inv step_u_var_set_invariant*

interpretation *single*: *Bisimulation_Invariant*
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*
⟨*proof*⟩

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*⟩

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*⟩

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*

$\times$ (('s :: *countable*) $\times$ ('c :: *countable*, *real*) *cconstraint*) *list*) *list* +
fixes Φ :: (*nat*, 's, 'x, *int*) *formula*
and s_0 :: ('x $\times$ *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*: $L_0 \in$ *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 $x \notin$ *renum_vars* ‘ *var_set*
shows *vars_inv x* $\notin$ *var_set*
⟨*proof*⟩

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*⟩

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* $\wedge$ *s'* = *s o vars_inv* $\wedge$ *u'* = *map_u u*
λ (*L*, *s*, *u*). *L* $\in$ *states* $\wedge$ *dom s* = *var_set* λ_. *True*
⟨*proof*⟩

lemma *start_equiv*:
A_B.equiv' a_0 $a_0{}'$
⟨*proof*⟩

lemma *check_sexp_equiv*:
assumes *A_B.equiv'* (*L*, *s*, *u*) (*L'*, *s'*, *u'*) *locs_of_sexp e* $\subseteq$ {*0*..<*n_ps*}
shows
check_sexp e L (*the* ∘ *s*) ⟷
check_sexp (*map_sexp renum_states renum_vars id e*) *L'* (*the* ∘ *s'*)
⟨*proof*⟩

lemma *models_iff*:
sem,a_0 $\models$ Φ = *renum.sem*,$a_0{}'$ $\models$ Φ' **if** *locs_of_formula* Φ $\subseteq$ {*0*..<*n_ps*}
⟨*proof*⟩

lemma *has_deadlock_iff*:
has_deadlock sem $a_0 \longleftrightarrow$ *has_deadlock renum.sem* a_0'
⟨*proof*⟩

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*⟩

lemma *Bisimulation_Invariants_Bisimulation_Invariant_iff*:
Bisimulation_Invariants A B sim PA PA PB PB $\longleftrightarrow$ *Bisimulation_Invariant A B sim PA PB*
⟨*proof*⟩

lemmas *Bisimulation_Invariant_composition* =
Bisimulation_Invariant_Invariants_composition[
THEN Bisimulation_Invariants_Bisimulation_Invariant,
OF _ Bisimulation_Invariant_Bisimulation_Invariants]

lemma *Bisimulation_Invariant_refl*:
Bisimulation_Invariant A A (=) *P P* **if** $\bigwedge a\ b.\ P\ a \Longrightarrow A\ a\ b \Longrightarrow P\ b$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *var_set_eq1*:
Prod_TA_Defs.var_set
(*set broadcast*, *map automaton_of* (*map conv_automaton automata*),
map_of bounds') = *var_set*
⟨*proof*⟩

lemma *urgency_removed'*:
∀ *i*<*n_ps*. *urgent*
(*Prod_TA_Defs.N* (*set broadcast*, *map automaton_of* (*map conv_automaton automata*), *map_of*

bounds') *i*)
= {}
⟨*proof*⟩

sublocale *real*: *Simple_Network_Rename_real* **where** *automata* = *map conv_automaton automata*
⟨*proof*⟩

lemma *sem_unfold1*:
real.sem = *sem*
⟨*proof*⟩

lemma *var_set_eq*:
real.var_set = *sem.var_set*
⟨*proof*⟩

lemma *map_acconstraint_conv_ac_commute*:
map_acconstraint renum_clocks id (*conv_ac ac*) = *conv_ac* (*map_acconstraint renum_clocks id ac*)
⟨*proof*⟩

lemma *map_ccconstraint_conv_cc_commute*:
renum_cconstraint (*conv_cc g*) = *conv_cc* (*renum_cconstraint g*)
⟨*proof*⟩

lemma *rename_conv_automaton_commute*:
real.renum_automaton n (*conv_automaton x*) = *conv_automaton* (*real.renum_automaton n x*)
⟨*proof*⟩

lemma *sem_unfold2*:
real.renum.sem = *renum.sem*
⟨*proof*⟩

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*
⟨*proof*⟩

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* ∉ *renum_vars* ‘ *var_set*
shows *vars_inv x* ∉ *var_set*
⟨*proof*⟩

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*⟩

lemma *start_equiv*:
renum_bisim.A_B.equiv' a_0 a_0'
⟨*proof*⟩

lemma *check_sexp_equiv*:
assumes *renum_bisim.A_B.equiv'* (*L*, *s*, *u*) (*L'*, *s'*, *u'*) *locs_of_sexp e* ⊆ {*0*..<*n_ps*}
shows
check_sexp e L (*the* ∘ *s*) ⟷
check_sexp (*map_sexp renum_states renum_vars id e*) *L'* (*the* ∘ *s'*)
⟨*proof*⟩

lemma *check_sexp_compatible*:
assumes *locs_of_sexp e* ⊆ {*0*..<*n_ps*}
shows *renum_bisim.compatible*
(λ(*L*, *s*, *u*). *check_sexp e L* (*the* ∘ *s*))
(λ(*L'*, *s'*, *u'*). *check_sexp* (*map_sexp renum_states renum_vars id e*) *L'* (*the* ∘ *s'*))
⟨*proof*⟩

lemma *has_deadlock_iff*:
has_deadlock sem a_0 ⟷ *has_deadlock renum.sem* a_0'
⟨*proof*⟩

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)
⟨*proof*⟩

lemmas *models_state_compatible* = *renum_bisim.models_state_compatible*[*OF state_formula_compatible*]

end

locale *Simple_Network_Rename_Formula_int* =
Simple_Network_Rename_Start_int **where** *automata* = *automata*
for *automata* ::
('*s list* × '*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**
Φ′ = *map_formula renum_states renum_vars id* Φ

lemma *models_iff*:
sem,a_0 ⊨ Φ = *renum.sem*,a_0′ ⊨ Φ′ **if** *locs_of_formula* Φ ⊆ {0..<*n_ps*}
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

definition (**in** *Prod_TA_Defs*)
act_set ≡ (⋃ p ∈ {0..<*n_ps*}. ⋃ (*l*, *e*, *g*, *a*, _) ∈ *trans* (*N p*). *act.set_act a*) ∪ *broadcast*

lemma (**in** *Simple_Network_Impl*) *act_set_compute*:
act_set =
$\bigcup$ ((λ(_, _, *t*, _). $\bigcup$ ((λ(*l*, *e*, *g*, *a*, _). *act.set_act a*) ‘ *set t*)) ‘ *set automata*) ∪ *set broadcast*
⟨*proof*⟩

lemma *set1_acconstraint_elim*:
assumes *c* ∈ *set1_acconstraint ac*
obtains *x* **where** (*c*, *x*) = *constraint_pair ac*
⟨*proof*⟩

lemma (**in** *Simple_Network_Impl*)
assumes (*x1*, *x2*, *T*, *I*) ∈ *set automata* (*l*, *b*, *g*, *a*, *f*, *r*, *l′*) ∈ *set T*
shows *clk_set′I1*[*intro*]: *c* ∈ *set r* ⟹ *c* ∈ *clk_set′*
and *clk_set′I2*[*intro*]: *ac* ∈ *set g* ⟹ *c* ∈ *set1_acconstraint ac* ⟹ *c* ∈ *clk_set′*
and *loc_setI1*[*intro*]: *l* ∈ *loc_set* **and** *loc_setI2*[*intro*]: *l′* ∈ *loc_set*
and *act_setI*[*intro*]: *a′* ∈ *set_act a* ⟹ *a′* ∈ *act_set*
and *var_setI1*[*intro*]: *v* ∈ *set_bexp b* ⟹ *v* ∈ *var_set*
and *var_setI2*[*intro*]: (*x*, *e*) ∈ *set f* ⟹ *x* ∈ *var_set*
and *var_setI3*[*intro*]: (*x*, *e*) ∈ *set f* ⟹ *v* ∈ *set_exp e* ⟹ *v* ∈ *var_set*
⟨*proof*⟩

lemma (**in** *Simple_Network_Impl*) *clk_set′I3*[*intro*]:
assumes (*x1*, *x2*, *T*, *I*) ∈ *set automata*
shows (*l*, *g′*) ∈ *set I* ⟹ *ac* ∈ *set g′* ⟹ *c* ∈ *set1_acconstraint ac* ⟹ *c* ∈ *clk_set′*
⟨*proof*⟩

definition
find_remove P = *map_option* (λ(*xs*, *x*, *ys*). (*x*, *xs* @ *ys*)) *o List.extract P*

fun *merge_pairs* :: (′*a* × ′*b list*) *list* ⇒ (′*a* × ′*b list*) *list* ⇒ (′*a* × ′*b list*) *list* **where**
merge_pairs [] *ys* = *ys* |
merge_pairs ((*k*, *v*) # *xs*) *ys* = (*case find_remove* (λ(*k′*, _). *k′* = *k*) *ys of*
None ⇒ (*k*, *v*) # *merge_pairs xs ys*
| *Some* ((_, *v′*), *ys*) ⇒ (*k*, *v* @ *v′*) # *merge_pairs xs ys*
)

definition
conv_urge urge ≡ λ(*committed*, *urgent*, *trans*, *inv*).
(*committed*,
[],
map (λ(*l*, *b*, *g*, *a*, *f*, *r*, *l′*). (*l*, *b*, *g*, *a*, *f*, *urge* # *r*, *l′*)) *trans*,
merge_pairs (*map* (λ*l*. (*l*, [*acconstraint.LE urge 0*])) *urgent*) *inv*)

lemma *map_of_distinct_upd2*:
assumes *x* ∉ *set* (*map fst xs*)
shows *map_of* (*xs* @ (*x*,*y*) # *ys*) = (*map_of* (*xs* @ *ys*))(*x* ↦ *y*)
⟨*proof*⟩

lemma *map_of_distinct_lookup*:

assumes $x \notin set\ (map\ fst\ xs)$
shows $map\_of\ (xs\ @\ (x,y)\ \#\ ys)\ x = Some\ y$
⟨*proof*⟩

lemma *find_remove_SomeD*:
assumes $find\_remove\ P\ xs = Some\ (x,\ ys)$
shows $\exists\ as\ bs.\ xs = as\ @\ x\ \#\ bs \wedge ys = as\ @\ bs \wedge (\forall\ x \in set\ as.\ \neg\ P\ x) \wedge P\ x$
⟨*proof*⟩

lemma *find_map*:
$find\ Q\ (map\ f\ xs) = map\_option\ f\ (find\ P\ xs)$ **if** $\forall\ x \in set\ xs.\ Q\ (f\ x) \longleftrightarrow P\ x$
⟨*proof*⟩

lemma *find_remove_map*:
$find\_remove\ Q\ (map\ f\ xs) = map\_option\ (\lambda(x,\ xs).\ (f\ x,\ map\ f\ xs))\ (find\_remove\ P\ xs)$
if $\forall\ x \in set\ xs.\ Q\ (f\ x) \longleftrightarrow P\ x$
⟨*proof*⟩

lemma *map_merge_pairs2*:
$map\ (\lambda(k,\ v).\ (g\ k,\ map\ f\ v))\ (merge\_pairs\ xs\ ys)$
$= merge\_pairs\ (map\ (\lambda(k,\ v).\ (g\ k,\ map\ f\ v))\ xs)\ (map\ (\lambda(k,\ v).\ (g\ k,\ map\ f\ v))\ ys)$
if *inj*: $inj\_on\ g\ (fst\ `\ (set\ xs \cup set\ ys))$
⟨*proof*⟩

lemma *map_merge_pairs*:
$map\ (\lambda(k,\ v).\ (k,\ map\ f\ v))\ (merge\_pairs\ xs\ ys)$
$= merge\_pairs\ (map\ (\lambda(k,\ v).\ (k,\ map\ f\ v))\ xs)\ (map\ (\lambda(k,\ v).\ (k,\ map\ f\ v))\ ys)$
⟨*proof*⟩

lemma *map_map_commute*:
$map\ f\ (map\ g\ xs) = map\ g\ (map\ f\ xs)$ **if** $\forall\ x \in set\ xs.\ f\ (g\ x) = g\ (f\ x)$
⟨*proof*⟩

lemma *conv_automaton_conv_urge_commute*:
$conv\_automaton\ (conv\_urge\ c\ A) = conv\_urge\ c\ (conv\_automaton\ A)$
⟨*proof*⟩

lemma *conv_automaton_conv_urge_commute′*:
$conv\_automaton\ o\ conv\_urge\ c = conv\_urge\ c\ o\ conv\_automaton$
⟨*proof*⟩

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*:
$Simple\_Network\_Language.conv\_A\ (automaton\_of\ A) = automaton\_of\ (conv\_automaton\ A)$
⟨*proof*⟩

locale *Simple_Network_Rename* =
Simple_Network_Rename_Defs_int_urge **where** *automata* = *automata* **for** *automata* ::

(*'s list* × *'s list*
× ((*'a* :: *countable*) *act*, *'s*, *'c*, *int*, *'x* :: *countable*, *int*) *transition list*
× ((*'s* :: *countable*) × (*'c* :: *countable*, *int*) *cconstraint*) *list*) *list* +
assumes *renum_states_inj*:
∀ *i*<*n_ps*. ∀ *x*∈*loc_set*. ∀ *y*∈*loc_set*. *renum_states i x* = *renum_states i y* ⟶ *x* = *y*
and *renum_clocks_inj*: *inj_on renum_clocks* (*insert urge clk_set'*)
and *renum_vars_inj*: *inj_on renum_vars var_set*
and *renum_acts_inj*: *inj_on renum_acts act_set*
and *infinite_types*:
infinite (*UNIV* :: *'c set*) *infinite* (*UNIV* :: *'x set*) *infinite* (*UNIV* :: *'s set*)
infinite (*UNIV* :: *'a set*)
and *bounds'_var_set*: *fst* ‘ *set bounds'* = *var_set*
and *loc_set_invs*: ⋃ ((λ*g*. *fst* ‘ *set g*) ‘ *set* (*map* (*snd o snd o snd*) *automata*)) ⊆ *loc_set*
and *loc_set_committed*: ⋃ ((*set o fst*) ‘ *set automata*) ⊆ *loc_set*
and *loc_set_urgent*: ⋃ ((*set o* (*fst o snd*)) ‘ *set automata*) ⊆ *loc_set*
and *urge_not_in_clk_set*: *urge* ∉ *clk_set'*
begin

lemma *clk_set'_finite*:
finite clk_set'
⟨*proof*⟩

lemmas [*finite_intros*] = *trans_N_finite*

lemma *set_act_finite*[*finite_intros*]:
finite (*set_act a*)
⟨*proof*⟩

lemma *loc_set_finite*:
finite loc_set
⟨*proof*⟩

lemma *var_set_finite*[*finite_intros*]:
finite var_set
⟨*proof*⟩

lemma *act_set_finite*:
finite act_set
⟨*proof*⟩

lemma *bij_extend_bij_renum_clocks*:
bij (*extend_bij renum_clocks* (*insert urge clk_set'*))
⟨*proof*⟩

lemma *renum_vars_bij_extends*[*simp*]:
extend_bij renum_vars var_set x = *renum_vars x* **if** *x* ∈ *var_set*
⟨*proof*⟩

lemma *bij_extend_bij_renum_states*:
bij (*extend_bij renum_vars var_set*)
⟨*proof*⟩

lemma *inj_extend_bij_renum_states*: *inj* (*extend_bij* (*renum_states p*) *loc_set*) **if** *p* < *n_ps*
⟨*proof*⟩

lemma *renum_states_extend*:
extend_bij (*renum_states p*) *loc_set l* = *renum_states p l* **if** *l* ∈ *loc_set p* < *n_ps*
⟨*proof*⟩

lemma *renum_acts_bij_extends*[*simp*]:
extend_bij renum_acts act_set x = *renum_acts x* **if** *x* ∈ *act_set*
⟨*proof*⟩

lemma *inj_extend_bij_renum_acts*: *inj* (*extend_bij renum_acts act_set*)
⟨*proof*⟩

lemma *constraint_clk_conv_ac*:
constraint_clk (*conv_ac ac*) = *constraint_clk ac*
⟨*proof*⟩

interpretation *urge*: *Prod_TA_sem_urge*
(*set broadcast*, *map* (*Simple_Network_Language.conv_A o automaton_of*) *automata*, *map_of bounds'*)
urge
⟨*proof*⟩

sublocale *rename*: *Simple_Network_Rename_Defs_int*
broadcast bounds'
extend_bij renum_acts act_set
extend_bij renum_vars var_set
extend_bij renum_clocks (*insert urge clk_set'*)
λ*p*. *extend_bij* (*renum_states p*) *loc_set*
map (*conv_urge urge*) *automata*
⟨*proof*⟩

sublocale *rename*: *Simple_Network_Rename_int*
broadcast bounds'
extend_bij renum_acts act_set
extend_bij renum_vars var_set
extend_bij renum_clocks (*insert urge clk_set'*)
λ*p*. *extend_bij* (*renum_states p*) *loc_set*
map (*conv_urge urge*) *automata*
⟨*proof*⟩

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⟩

lemma renum_urge_automaton_eq1:
  renum_automaton p (conv_urge urge (automata ! p))
  = conv_urge (renum_clocks urge) (renum_automaton p (automata ! p)) if p < n_ps
  ⟨proof⟩

lemma renum_urge_automaton_eq2:
  rename.real.renum_automaton p (conv_urge urge (automata ! p))
  = conv_urge
      (extend_bij renum_clocks (insert urge clk_set') urge)
      (rename.renum_automaton p (automata ! p))
  if p < n_ps
  ⟨proof⟩

lemma renum_urge_automaton_eq:
  rename.real.renum_automaton p (conv_urge urge (automata ! p)) =
   renum_automaton p (conv_urge urge (automata ! p)) if p < n_ps
  ⟨proof⟩

lemma rename_N_eq_sem:
  rename.renum.sem =
  Simple_Network_Language_Model_Checking.N
    (map renum_acts broadcast)
    (map_index renum_automaton (map (conv_urge urge) automata))
    (map (λ(a,p). (renum_vars a,p)) bounds')

  ⟨proof⟩

lemma map_of_merge_pairs:
  map_of (merge_pairs xs ys) = (λx.
  (if x ∈ fst ` set xs ∧ x ∈ fst ` set ys then Some (the (map_of xs x) @ the (map_of ys x))
   else if x ∈ fst ` set xs then map_of xs x
   else map_of ys x))
⟨proof⟩

lemma default_map_of_merge_pairs:
  default_map_of [] (merge_pairs xs ys) = (λx.
  (if x ∈ fst ` set xs then the (map_of xs x) @ default_map_of [] ys x
   else default_map_of [] ys x))
  ⟨proof⟩
```

lemma *automaton_of_conv_urge_commute*:
automaton_of (*conv_urge urge A*) = *urge.add_reset* (*urge.add_inv* (*automaton_of A*))
⟨*proof*⟩

lemma *urge_commute*:
rename.sem = *urge.A_urge*
⟨*proof*⟩

lemma *conv_automaton_of'*:
automaton_of ∘ *conv_automaton* = *Simple_Network_Language.conv_A o automaton_of*
⟨*proof*⟩

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*
⟨*proof*⟩

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'*)
⟨*proof*⟩

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
⟨*proof*⟩

lemma *urge_models_iff*:
sem,a_0 ⊨ Φ ⟷ *rename.sem*,a_0 ⊨ Φ
⟨*proof*⟩

lemma *urge_has_deadlock_iff*:
has_deadlock sem a_0 ⟷ *has_deadlock rename.sem* a_0
⟨*proof*⟩

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* (λ*P* (*L*, *s*, _). *check_sexp P L* (*the o s*)) φ)
(*map_state_formula* (λ*P* (*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* (λ*P* (*L*, *s*, _). *check_sexp P L* (*the o s*)) ψ)
(*map_path_formula* (λ*P* (*L*, *s*, _). *check_sexp P L* (*the o s*)) ψ)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *rename_states_eq*:
  rename.states = *states*
  ⟨*proof*⟩

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)
  ⟨*proof*⟩

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›
  ⟨*proof*⟩

lemma L_0*_dom*:
  length L_0 = *n_ps set* L_0 ⊆ *loc_set*
  ⟨*proof*⟩

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
⟨*proof*⟩

lemma *rename_a0_eq*:
rename.a0 = a0
⟨*proof*⟩

lemma *rename_a0'_eq*:
rename.a0' = a0'
⟨*proof*⟩

lemma *has_deadlock_iff*:
has_deadlock rename.renum.sem a0' ⟷ has_deadlock sem a0
⟨*proof*⟩

lemma *N_eq_sem*:
Simple_Network_Language_Model_Checking.N broadcast automata bounds' = sem
⟨*proof*⟩

lemma *rename_N_eq_sem'*:
Simple_Network_Language_Model_Checking.N
(map renum_acts broadcast)
(map_index renum_automaton automata)
(map (λ(a,p). (renum_vars a,p)) bounds')
= renum.sem
⟨*proof*⟩

lemmas *has_deadlock_iff' =*
has_deadlock_iff[unfolded rename_N_eq_sem, folded N_eq_sem, unfolded a0_def a0'_def]

lemmas *start_equiv = start_equiv[of a0, unfolded a0_def, simplified, OF L0_states]*

lemmas *urge_models_state_compatible =*
urge_models_state_compatible[THEN rel_funD, OF _ _ _ start_equiv,
of a0, unfolded a0_def, simplified, OF L0_states]

lemmas *rename_models_state_compatible =*
rename.models_state_compatible[THEN rel_funD, OF _ rename.start_equiv,
unfolded rename_a0_eq a0_def rename_n_ps_eq rename_a0'_eq a0'_def]

lemmas *models_state_compatible =*
transp_on_equality[THEN transpD, OF urge_models_state_compatible rename_models_state_compatible]

lemmas *models_state_compatible' = models_state_compatible[unfolded rename_N_eq_sem, folded N_eq_sem]*

end

locale *Simple_Network_Rename_Formula* =
  Simple_Network_Rename_Start **where** *automata* = *automata*
  for *automata* ::
    ('*s list* × '*s list*
      × (('*a* :: *countable*) *act*, '*s*, '*c*, *int*, '*x* :: *countable*, *int*) *transition list*
      × (('*s* :: *countable*) × ('*c* :: *countable*, *int*) *cconstraint*) *list*) *list* +
  fixes Φ :: (*nat*, '*s*, '*x*, *int*) *formula*
  assumes *formula_dom*:
    set2_formula Φ ⊆ *loc_set*
    locs_of_formula Φ ⊆ {*0*..<*n_ps*}
    vars_of_formula Φ ⊆ *var_set*
begin

sublocale *rename*: *Simple_Network_Rename_Formula_int*
  broadcast bounds'
  extend_bij renum_acts act_set
  extend_bij renum_vars var_set
  extend_bij renum_clocks (*insert urge clk_set'*)
  λ*p*. *extend_bij* (*renum_states p*) *loc_set*
  s_0 L_0
  map (*conv_urge urge*) *automata*
  ⟨*proof*⟩

definition
  Φ' = *map_formula* (λ*p*. *renum_states p*) *renum_vars id* Φ

lemma *rename_Φ'_eq*:
  rename.Φ' = Φ'
  ⟨*proof*⟩

lemma *models_iff1*:
  rename.renum.sem,a_0' ⊨ Φ' ⟷ *rename.sem*,a_0 ⊨ Φ
  ⟨*proof*⟩

lemma *models_iff2*:
  rename.sem,a_0 ⊨ Φ ⟷ *sem*,a_0 ⊨ Φ
  ⟨*proof*⟩

lemma *models_iff*:
  rename.renum.sem,a_0' ⊨ Φ' ⟷ *sem*,a_0 ⊨ Φ
  ⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *final_fun_is_False*:
Fi a = *False*
⟨*proof*⟩

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*
⟨*proof*⟩

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 L$_0$ *s*$_0$ (*formula.EX* (*not sexp.true*)))
>$_t$
⟨*proof*⟩

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*:
∀ *i*<*n_ps*. ∀ *x*∈*loc_set*. ∀ *y*∈*loc_set*. *renum_states i x* = *renum_states i y* ⟶ *x* = *y*
and *renum_clocks_inj*: *inj_on renum_clocks* (*insert urge clk_set*′)

and *renum_vars_inj*: *inj_on renum_vars var_set*
and *renum_acts_inj*: *inj_on renum_acts act_set*
and *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 *L₀_states*: *L₀* ∈ *states*
and *s₀_dom*: *fst ‘ set s₀ = var_set* **and** *s₀_distinct*: *distinct* (*map fst s₀*)
assumes *formula_dom*:
set2_formula Φ ⊆ *loc_set*
locs_of_formula Φ ⊆ {*0..<n_ps*}
vars_of_formula Φ ⊆ *var_set*
begin

interpretation *Simple_Network_Rename_Formula*
⟨*proof*⟩

lemmas *Simple_Network_Rename_intro = Simple_Network_Rename_Formula_axioms*

end

lemma *is_result_assert_iff*:
is_result (*assert b m*) ⟷ *b*
⟨*proof*⟩

lemma *is_result_combine_Cons_iff*:
is_result (*combine* (*x # xs*)) ⟷ *is_result x* ∧ *is_result* (*combine xs*)
⟨*proof*⟩

lemma *is_result_combine_iff*:
is_result (*a <|> b*) ⟷ *is_result a* ∧ *is_result b*
⟨*proof*⟩

context *Simple_Network_Rename_Formula_String_Defs*
begin

lemma *check_renaming*:
Simple_Network_Rename_Formula_String
broadcast bounds'
renum_acts renum_vars renum_clocks renum_states
automata urge Φ *s₀ L₀* ⟷
is_result (*check_renaming urge* Φ *L₀ s₀*)

⟨*proof*⟩

end

context *Simple_Network_Impl_nat_defs*
begin

definition *check_precond1* **where**
check_precond1 =
combine [

```
    assert (m > 0)
      (STR ''At least one clock'')
    assert (0 < length automata)
      (STR ''At least one automaton'')
    assert (∀ i<n_ps. let (_, _, trans, _) = (automata ! i) in ∀ (l, _, _, _, _, _, l') ∈ set trans.
      l < num_states i ∧ l' < num_states i)
      (STR ''Number of states is correct (transitions)'')
    assert (∀ i < n_ps. let (_, _, _, inv) = (automata ! i) in ∀ (x, _) ∈ set inv. x < num_states
i)
      (STR ''Number of states is correct (invariants)'')
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, _, _, f, _, _) ∈ set trans.
      ∀ (x, upd) ∈ set f. x < n_vs ∧ (∀ i ∈ vars_of_exp upd. i < n_vs))
      (STR ''Variable set bounded (updates)'')
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, b, _, _, _, _, _) ∈ set trans.
      ∀ i ∈ vars_of_bexp b. i < n_vs)
      (STR ''Variable set bounded (guards)'')
    assert (∀ i < n_vs. fst (bounds' ! i) = i)
      (STR ''Bounds first index'')
    assert (∀ a ∈ set broadcast. a < num_actions)
      (STR ''Broadcast actions bounded'')
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, _, a, _, _, _) ∈ set trans.
        pred_act (λa. a < num_actions) a)
      (STR ''Actions bounded (transitions)'')
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, g, _, _, r, _) ∈ set trans.
      (∀ c ∈ set r. 0 < c ∧ c ≤ m) ∧
      (∀ (c, x) ∈ collect_clock_pairs g. 0 < c ∧ c ≤ m ∧ x ∈ ℕ))
      (STR ''Clock set bounded (transitions)'')
    assert (∀ (_, _, _, inv) ∈ set automata. ∀ (l, g) ∈ set inv.
      (∀ (c, x) ∈ collect_clock_pairs g. 0 < c ∧ c ≤ m ∧ x ∈ ℕ))
      (STR ''Clock set bounded (invariants)'')
    assert (∀ (_, _, trans, _) ∈ set automata. ∀ (_, _, g, a, _, _, _) ∈ set trans.
      case a of In a ⇒ a ∈ set broadcast ⟶ g = [] | _ ⇒ True)
      (STR ''Broadcast receivers are unguarded'')
    assert (∀ (_, U, _, _)∈set automata. U = [])
      STR ''Urgency was removed''
  ]


lemma check_precond1:
  is_result check_precond1
  ⟷ Simple_Network_Impl_nat_urge broadcast bounds' automata m num_states num_actions
  ⟨proof⟩

context
  fixes k :: nat list list list
    and L₀ :: nat list
    and s₀ :: (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 s₀))
    (STR ''Initial state bounded''),
  assert (length L₀ = n_ps)
    (STR ''Length of initial state''),
  assert (∀ i < n_ps. L₀ ! 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 L₀ s₀ formula
  ⟨proof⟩

end

definition
  check_precond k L₀ s₀ formula ≡ check_precond1 <|> check_precond2 k L₀ s₀ formula

lemma check_precond:
  Simple_Network_Impl_nat_ceiling_start_state broadcast bounds' automata m num_states
     num_actions k L₀ s₀ formula ⟷ is_result (check_precond k L₀ s₀ formula)
  ⟨proof⟩

end

derive show acconstraint act sexp formula

fun shows_exp and shows_bexp where
  shows_exp (const c) = show c |
  shows_exp (var v) = show v |
  shows_exp (if_then_else b e1 e2) =
    shows_bexp b @ '' ? '' @ shows_exp e1 @ '' : '' @ shows_exp e2 |
  shows_exp (binop _ e1 e2) = ''binop '' @ shows_exp e1 @ '' '' @ shows_exp e2 |
```

```
  shows_exp (unop _ e) = ''unop '' @ shows_exp e |
  shows_bexp (bexp.lt a b) = shows_exp a @ '' < '' @ shows_exp b |
  shows_bexp (bexp.le a b) = shows_exp a @ '' <= '' @ shows_exp b |
  shows_bexp (bexp.eq a b) = shows_exp a @ '' = '' @ shows_exp b |
  shows_bexp (bexp.ge a b) = shows_exp a @ '' >= '' @ shows_exp b |
  shows_bexp (bexp.gt a b) = shows_exp a @ '' > '' @ shows_exp b |
  shows_bexp bexp.true = ''true'' |
  shows_bexp (bexp.not b) = ''! '' @ shows_bexp b |
  shows_bexp (bexp.and a b) = shows_bexp a @ '' && '' @ shows_bexp b |
  shows_bexp (bexp.or a b) = shows_bexp a @ '' || '' @ shows_bexp b |
  shows_bexp (bexp.imply a b) = shows_bexp a @ '' -> '' @ shows_bexp b

instantiation bexp :: (show, show) show
begin

definition shows_prec p (e :: (_, _) bexp) rest = shows_bexp e @ rest for p

definition shows_list (es) s =
  map shows_bexp es |> intersperse '', '' |> (λxs. ''['' @ concat xs @ '']'' @ s)
instance
  ⟨proof⟩

end

instantiation exp :: (show, show) show
begin

definition shows_prec p (e :: (_, _) exp) rest = shows_exp e @ rest for p

definition shows_list (es) s =
  map shows_exp es |> intersperse '', '' |> (λxs. ''['' @ concat xs @ '']'' @ s)
instance
  ⟨proof⟩

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
  ⟨proof⟩

end


definition rename_network where
 rename_network broadcast bounds' automata renum_acts renum_vars renum_clocks renum_states
≡
  let
   automata = map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states)
automata;
    broadcast = map renum_acts broadcast;
    bounds' = map (λ(a,b,c). (renum_vars a, b, c)) bounds'
  in
    (broadcast, automata, bounds')


definition
  show_clock inv_renum_clocks = show o inv_renum_clocks

definition
  show_locs inv_renum_states = show o map_index inv_renum_states

definition
  show_vars inv_renum_vars = show o map_index (λi v. show (inv_renum_vars i) @ ''='' @
show v)

definition
```

```
  show_state inv_renum_states inv_renum_vars ≡ λ(L, vs).
  let L = show_locs inv_renum_states L; vs = show_vars inv_renum_vars vs in
  ''<'' @ L @ ''>, <'' @ vs @ ''>''

definition do_rename_mc where
  do_rename_mc f dc broadcast bounds' automata k urge L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
≡
let
   _ = println (STR ''Checking renaming'');
   formula = (if dc then formula.EX (not sexp.true) else formula);
   renaming_valid = Simple_Network_Rename_Formula_String_Defs.check_renaming
      broadcast bounds' renum_acts renum_vars renum_clocks renum_states automata urge for-
mula L₀ s₀;
   _ = println (STR ''Renaming network'');
   (broadcast, automata, bounds') = rename_network
      broadcast bounds' (map (conv_urge urge) automata)
      renum_acts renum_vars renum_clocks renum_states;
   _ = trace_level 4 (λ_. return (STR ''Automata after renaming''));
   _ = map (λa. show a |> String.implode |> (λs. trace_level 4 (λ_. return s))) automata;
   _ = println (STR ''Renaming formula'');
   formula =
     (if dc then formula.EX (not sexp.true) else map_formula renum_states renum_vars id for-
mula);
    _ = println (STR ''Renaming state'');
   L₀ = map_index renum_states L₀;
   s₀ = map (λ(x, v). (renum_vars x, v)) s₀;
   show_clock = show o inv_renum_clocks;
   show_state = show_state inv_renum_states inv_renum_vars
in
  if is_result renaming_valid then do {
    let _ = println (STR ''Checking preconditions'');
    let r = Simple_Network_Impl_nat_defs.check_precond
      broadcast bounds' automata m num_states num_actions k L₀ s₀ formula;
    let _ = (case r of Result _ ⇒ ()
      | Error es ⇒
        let
          _ = println STR '''';
          _ = println STR ''The following pre−conditions were not 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⟩
```

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 s_0 L_0 *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* L_0) (*map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0)
(*formula.EX* (*not sexp.true*)))
$>_t$
⟨*proof*⟩

Code Setup for the Model Checker **lemmas** [*code*] =
reachability_checker_def
Alw_ev_checker_def
leadsto_checker_def
model_checker_def[*unfolded PR_CONST_def*]

lemmas [*code*] =
Prod_TA_Defs.n_ps_def
Simple_Network_Impl_Defs.n_vs_def
automaton_of_def
Simple_Network_Impl_nat_defs.pairs_by_action_impl_def
Simple_Network_Impl_nat_defs.all_actions_from_vec_def
Simple_Network_Impl_nat_defs.all_actions_by_state_def
Simple_Network_Impl_nat_defs.compute_upds_impl_def
Simple_Network_Impl_nat_defs.actions_by_state_def
Simple_Network_Impl_nat_defs.check_boundedi_def
Simple_Network_Impl_nat_defs.get_committed_def
Simple_Network_Impl_nat_defs.make_combs_def
Simple_Network_Impl_nat_defs.trans_map_def
Simple_Network_Impl_nat_defs.actions_by_state′_def
Simple_Network_Impl_nat_defs.bounds_map_def
Simple_Network_Impl_nat_defs.bin_actions_def
mk_updsi_def

lemma (**in** *Simple_Network_Impl_nat_defs*) *bounded_*s_0*_iff*:
bounded bounds (*map_of* s_0) ⟷ *bounded* (*map_of bounds′*) (*map_of* s_0)
⟨*proof*⟩

lemma *int_Nat_range_iff*:
(*n* :: *int*) ∈ ℕ ⟷ *n* ≥ *0* **for** *n*
⟨*proof*⟩

lemmas [*code*] =
Simple_Network_Impl_nat_ceiling_start_state_def
Simple_Network_Impl_nat_ceiling_start_state_axioms_def[
*unfolded Simple_Network_Impl_nat_defs.bounded_*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*:
$L \in states \longleftrightarrow length\ L = n\_ps \wedge (\forall\ i.\ i < n\_ps \longrightarrow$
$(\exists\ (l,\ b,\ g,\ a,\ r,\ u,\ l') \in fst\ (snd\ (snd\ (fst\ (snd\ A)\ !\ i)))).\ L\ !\ i = l \vee L\ !\ i = l')$
⟨*proof*⟩

lemmas [*code_unfold*] =
Prod_TA_Defs.states_mem_iff
Simple_Network_Impl.act_set_compute
Simple_Network_Impl_Defs.var_set_compute
Simple_Network_Impl_Defs.loc_set_compute
setcompr_eq_image
Simple_Network_Impl.length_automata_eq_n_ps[*symmetric*]

export_code *rename_mc* **in** *SML* **module_name** *Test*

Calculating the Clock Ceiling **context** *Simple_Network_Impl_nat_defs*
begin

definition *clkp_inv i l* ≡
$\bigcup g \in set\ (filter\ (\lambda(a,\ b).\ a = l)\ (snd\ (snd\ (snd\ (automata\ !\ i))))).\ collect\_clock\_pairs\ (snd\ g)$

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 = (⋃ (*l'*, *b*, *g*, *a*, *f*, *r*, _) ∈ *set* (*fst* (*snd* (*snd* (*automata* ! *i*))))).
if l' = *l then set r else* {})

context
fixes *q c* :: *nat*
begin

definition *n* ≡ *num_states q*

definition *V* ≡ λ *v*. *v* ≤ *n*

definition
bound_g l ≡
Max ({*0*} ∪ ⋃ ((λ (*x*, *d*). *if x* = *c then* {*d*} *else* {}) ‘ *clkp_set'' q l*))

definition
bound_inv l ≡
Max ({*0*} ∪ ⋃ ((λ (*x*, *d*). *if x* = *c then* {*d*} *else* {}) ‘ *clkp_inv q l*))

definition
bound l ≡ *max* (*bound_g l*) (*bound_inv l*)

definition
resets l ≡
fold
(λ (*l1*, *b*, *g*, *a*, *f*, *r*, *l'*) *xs*. *if l1* ≠ *l* ∨ *l'* ∈ *set xs* ∨ *c* ∈ *set r then xs else* (*l'* # *xs*))
(*fst* (*snd* (*snd* (*automata* ! *q*))))
[]

Edges in the direction nodes to single sink.

definition
E' *l* ≡ *resets l*

Turning around the edges to obtain a single source shortest paths problem.

definition
E l ≡ *if l* = *n then* [*0*..<*n*] *else filter* (λ *l'*. *l* ∈ *set* (*E' l'*)) [*0*..<*n*]

Weights already turned around.

definition
W l l' ≡ *if l* = *n then* − *bound l' else 0*

definition *G* **where**
G ≡ (| *gi_V* = *V*, *gi_E* = *E*, *gi_V0* = [*n*], ... = *W* |)

definition
local_ceiling_single ≡
let
w = *calc_shortest_scc_paths G n*
in
map (λ *x*. *case x of None* ⇒ *0* | *Some x* ⇒ *nat*(−*x*)) *w*

end

definition
local_ceiling ≡
rev $
fold
(λ *q xs*.
(λ *x*. *rev x* # *xs*) $
fold
(λ *l xs*.
(λ *x*. (*0* # *rev x*) # *xs*) $
fold
(λ *c xs*. *local_ceiling_single q c* ! *l* # *xs*)
[*1*..<*Suc m*]
[]
)
[*0*..<*n q*]
[]
)
[*0*..<*n_ps*]
[]

end

lemmas [*code*] =
Simple_Network_Impl_nat_defs.local_ceiling_def
Simple_Network_Impl_nat_defs.local_ceiling_single_def
Simple_Network_Impl_nat_defs.n_def
Simple_Network_Impl_nat_defs.G_def
Simple_Network_Impl_nat_defs.W_def
Simple_Network_Impl_nat_defs.V_def
Simple_Network_Impl_nat_defs.E'_def
Simple_Network_Impl_nat_defs.E_def
Simple_Network_Impl_nat_defs.resets_def
Simple_Network_Impl_nat_defs.bound_def
Simple_Network_Impl_nat_defs.bound_inv_def
Simple_Network_Impl_nat_defs.bound_g_def
Simple_Network_Impl_nat_defs.collect_resets_def
Simple_Network_Impl_nat_defs.clkp_set''_def
Simple_Network_Impl_nat_defs.clkp_inv_def

export_code *Simple_Network_Impl_nat_defs.local_ceiling* **checking** *SML_imp*

Calculating the Renaming **definition** *mem_assoc x = list_ex (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
⟨proof⟩

lemma *is_result_assert_msg_iff*:
is_result (assert_msg b m r) ⟷ is_result r ∧ b
⟨proof⟩

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' = (λ*i*. *Simple_Network_Impl.loc_set' automata i* |> *list_of_set*);
loc_set = *Prod_TA_Defs.loc_set*
(*set broadcast*, *map automaton_of automata*, *map_of bounds*);
loc_set_diff = (λ*i*. *loc_set* − *Simple_Network_Impl.loc_set' automata i* |> *list_of_set*);
loc_set = *list_of_set loc_set*;
var_set = *Prod_TA_Defs.var_set*
(*set broadcast*, *map automaton_of automata*, *map_of bounds*) |> *list_of_set*;
n_ps = *length automata*;
num_actions = *length action_set*;
m = *length* (*remdups clk_set*);
num_states_list = *map* (λ*i*. *loc_set' i* |> *remdups* |> *length*) [*0*..<*n_ps*];
num_states = (λ*i*. *num_states_list* ! *i*);
mk_renaming = *mk_renaming* (λ*x*. *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 = (λ*c*. *if c* = *0 then STR ''0'' else inv_renum_clocks* (*c* − *1*));
renum_states_list' ← *combine_map* (λ*i*. *mk_renaming'* (*loc_set' i*)) [*0*..<*n_ps*];
let renum_states_list = *map fst renum_states_list'*;
let renum_states_list = *map_index*
(λ*i m*. *extend_domain m* (*loc_set_diff i*) (*length* (*loc_set' i*))) *renum_states_list*;
let renum_states = (λ*i*. *renum_states_list* ! *i*);
let inv_renum_states = (λ*i*. *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) L₀ s₀ 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)]
⟨*proof*⟩

lemma *parse parse_bounds* (*STR* '''') = *Result* []
⟨*proof*⟩

definition [*consuming*]:
scan_var = *ta_var_ident*

abbreviation *seq_ignore_left_ws* (**infixr** ∗∗−− 60)
where *p* ∗∗−− *q* ≡ *token p* ∗−− *q* **for** *p q*

abbreviation *seq_ignore_right_ws* (**infixr** −−∗∗ 60)
where *p* −−∗∗ *q* ≡ *token p* −−∗ *q* **for** *p q*

abbreviation *seq_ws* (**infixr** −−− 60)
where *seq_ws p q* ≡ *token p* −− *q* **for** *p q*

definition *scan_acconstraint* **where** [*unfolded Let_def*, *consuming*]:
scan_acconstraint ≡
let scan =
(λ*s 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 ⋀*l2*. *ll_fuel l2* + (*if length s* > *0 then 1 else 0*) ≤ *ll_fuel l'* ⟹ *B l2* = *B' l2*
assumes *s* = *s' l*=*l'*
shows *scan_infix_pair A B s l* = *scan_infix_pair A' B' s' l'*
⟨*proof*⟩

lemma [*fundef_cong*]:
assumes ⋀*l2*. *ll_fuel l2* < *ll_fuel l* ⟹ *A l2* = *A' l2 l* = *l'*
shows *scan_parens' A l* = *scan_parens' A' l'*
⟨*proof*⟩

lemma *token_cong*[*fundef_cong*]:
assumes ⋀*l2*. *ll_fuel l2* ≤ *ll_fuel l* ⟹ *A l2* = *A' l2 l* = *l'*
shows *token A l* = *token A' l'*
⟨*proof*⟩

lemma *is_cparser_scan_parens'*[*parser_rules*]:
is_cparser (*scan_parens' a*)
⟨*proof*⟩

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'
  ⟨proof⟩

lemma [parser_rules]:
  is_cparser scan_6'
  ⟨proof⟩

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
  ⟨proof⟩

lemma parse scan_bexp (STR ''a < 3 && b>=2 || ~ (c <= 4)'')
= Result (sexp.or (and (lt STR ''a'' 3) (ge STR ''b'' 2)) (not (sexp.le STR ''c'' 4)))
  ⟨proof⟩

definition [consuming]: scan_prefix p head = exactly head **-- p for p

definition [consuming]: scan_formula ≡
  scan_prefix scan_bexp ''E<>'' with formula.EX ||
  scan_prefix scan_bexp ''E[]'' with EG ||
  scan_prefix scan_bexp ''A<>'' with AX ||
  scan_prefix scan_bexp ''A[]'' with AG ||
  scan_infix_pair scan_bexp scan_bexp ''-->'' with uncurry Leadsto


lemma is_cparser_token[parser_rules]:
```

```
  is_cparser (token a) if is_cparser a
  ⟨proof⟩

definition [consuming]: scan_action ≡
  (scan_var −−∗ token (exactly ''?'')) with In o String.implode ‖
  (scan_var −−∗ token (exactly ''!'')) with Out o String.implode ‖
  scan_var with Sil o String.implode

abbreviation orelse (infix orelse 58) where
  a orelse b ≡ default b a

definition
  parse_action s ≡ parse scan_action s orelse Sil (STR '''')

fun chop_sexp where
  chop_sexp clocks (and a b) (cs, es) =
    chop_sexp clocks a (cs, es) |> chop_sexp clocks b |
  chop_sexp clocks (eq a b) (cs, es) =
    (if a ∈ set clocks then (eq a b # cs, es) else (cs, eq a b # es)) |
  chop_sexp clocks (le a b) (cs, es) =
    (if a ∈ set clocks then (le a b # cs, es) else (cs, le a b # es)) |
  chop_sexp clocks (lt a b) (cs, es) =
    (if a ∈ set clocks then (lt a b # cs, es) else (cs, lt a b # es)) |
  chop_sexp clocks (ge a b) (cs, es) =
    (if a ∈ set clocks then (ge a b # cs, es) else (cs, ge a b # es)) |
  chop_sexp clocks (gt a b) (cs, es) =
    (if a ∈ set clocks then (gt a b # cs, es) else (cs, gt a b # es)) |
  chop_sexp clocks a (cs, es) = (cs, a # es)

fun sexp_to_acconstraint :: (String.literal, String.literal, String.literal, int) sexp ⇒ _ where
  sexp_to_acconstraint (lt a (b :: int)) = acconstraint.LT a b |
  sexp_to_acconstraint (le a b) = acconstraint.LE a b |
  sexp_to_acconstraint (eq a b) = acconstraint.EQ a b |
  sexp_to_acconstraint (ge a b) = acconstraint.GE a b |
  sexp_to_acconstraint (gt a b) = acconstraint.GT a b

no_notation top_assn (true)

fun sexp_to_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, L_0, s_0) ⇒
      do_preproc_mc dc ids_to_names (broadcast, automata, bounds) L_0 s_0 formula
```

Eliminate Gabow statistics

```
code_printing
  code_module Gabow_Skeleton_Statistics ⇀ (SML) ‹›
code_printing
  code_module AStatistics ⇀ (SML) ‹›
```

Delete "junk"

```
code_printing code_module Bits_Integer ⇀ (SML) ‹›

code_printing
  constant stat_newnode ⇀ (SML) (fn x => ()) _
| constant stat_start   ⇀ (SML) (fn x => ()) _
| constant stat_stop    ⇀ (SML) (fn x => ()) _



code_printing
  constant IArray.sub' ⇀ (SML) (Vector.sub o (fn (a, b) => (a, IntInf.toInt b)))

code_printing code_module Logging ⇀ (SML)
‹
structure Logging : sig
  val set_level : int -> unit
  val trace : int -> (unit -> string) -> unit
  val get_trace: unit -> (int * string) list
end = struct
  val level = Unsynchronized.ref 0;
  val messages : (int * string) list ref = Unsynchronized.ref [];
  fun set_level i = level := i;
  fun get_trace () = !messages;
  fun trace i f =
    if i > !level
    then ()
    else
      let
        val s = f ();
        val _ = messages := (i, s) :: !messages;
      in () end;
end
›
and (Eval) ‹›
code_reserved (Eval) Logging
code_reserved (SML) Logging


code_printing constant trace_level ⇀ (SML)
  Logging.trace (IntInf.toInt (integer'_of'_int _)) (_ ())
```

and (*Eval*)
(*fn* '_ => *fn s* => *writeln* (*s* ())) _ (_ ())

To disable state tracing:

SML Code Export Now we export the actual executable code for MUNTA. First for SML:

export_code *parse_convert_run Result Error*
in *SML* **module_name** *Model_Checker*
file_prefix *Simple_Model_Checker*

OCaml Code Export This is how to do it for OCaml:

export_code
convert_run Result Error String.explode int_of_integer nat_of_integer
JSON.Object JSON.Array JSON.String JSON.Int JSON.Nat JSON.Rat JSON.Boolean JSON.Null fract.Rat
in *OCaml* **module_name** *Model_Checker*
file_prefix *Simple_Model_Checker*

Running Munta Adding a bit of wrapper code, so that we can directly run MUNTA from within Isabelle:

definition *parse_convert_run_test* **where**
parse_convert_run_test dc s ≡ *do* {
x ← *parse_convert_run dc s*;
case x of
Error es ⇒ *do* {*let* _ = *map println es*; *return* (*STR* ''*Fail*'')}
| *Result r* ⇒ *return r*
}

⟨*ML*⟩

Now we can run MUNTA on a few examples:

⟨*ML*⟩

end

13 Build and Test Exported Program With MLton

theory *Munta_Compile_MLton*
imports *Munta_Model_Checker.Simple_Network_Language_Export_Code*
begin

— Produces a command for checking a single benchmark, e.g.: `./check_benchmark.sh 568 "Property is not satisfied" ./munta -m benchmarks/PM_all_5.muntax`
⟨*ML*⟩

Here is how to compile Munta with MLton and then run some benchmarks:

compile_generated_files *code/Simple_Model_Checker.ML* (**in** *Simple_Network_Language_Export_Code*)
external_files
‹*Unsynchronized.sml*›
‹*Writeln.sml*›

```
    ‹Util.sml›
    ‹Muntax.sml›
    ‹Mlton_Main.sml›
    ‹library.ML›
    ‹muntax.mlb›
    ‹check_benchmark.sh›
    (in ../ML)
  and
    ‹HDDI_02.muntax›
    ‹HDDI_02_broadcast.muntax›
    ‹HDDI_08_broadcast.muntax›
    ‹PM_all_1.muntax›
    ‹PM_all_2.muntax›
    ‹PM_all_3.muntax›
    ‹PM_all_4.muntax›
    ‹PM_all_5.muntax›
    ‹PM_all_6.muntax›
    ‹PM_all_urgent.muntax›
    ‹bridge.muntax›
    ‹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›
          (////////$ISABELLE_MLTON/////////$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 116245 false*)
val _ = *exec* ‹*Test PM_all_urgent*› (*mk_benchmark_check PM_all_urgent 8 true*)
val _ = *exec* ‹*Test bridge*› (*mk_benchmark_check bridge 73 true*)
val _ = *exec* ‹*Test csma_05*› (*mk_benchmark_check csma_05 8217 false*)
val _ = *exec* ‹*Test csma_06*› (*mk_benchmark_check csma_06 68417 false*)
val _ = *exec* ‹*Test fischer*› (*mk_benchmark_check fischer 4500 true*)
val _ = *exec* ‹*Test fischer_05*› (*mk_benchmark_check fischer_05 38579 false*)
val _ = *exec* ‹*Test hddi_08*› (*mk_benchmark_check hddi_08 466 false*)
val _ = *exec* ‹*Test light_switch*› (*mk_benchmark_check light_switch 2 true*)
— a test for the deadlock checker
val _ = *exec* ‹*Test hddi_08*› (*mk_benchmark_check_dc hddi_08 466 false*)
in () *end*›

end

References

[1] S. Wimmer. Timed automata. *Archive of Formal Proofs*, March 2016. https://isa-afp.org/entries/Timed_Automata.html, Formal proof development.

[2] S. Wimmer. Munta: A verified model checker for timed automata. In É. André and M. Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 236–243. Springer, 2019.

[3] S. Wimmer. *Trustworthy Verification of Realtime Systems*. PhD thesis, Technical University of Munich, Germany, 2020.

[4] S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS (1)*, volume 10805 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2018.

[5] S. Wimmer and P. Lammich. Difference bound matrices. *Archive of Formal Proofs*, August 2024. https://isa-afp.org/entries/Difference_Bound_Matrices.html, Formal proof development.

[6] S. Wimmer and P. Lammich. Worklist algorithms. *Archive of Formal Proofs*, August 2024. https://isa-afp.org/entries/Worklist_Algorithms.html, Formal proof development.