

Timed Automata

Simon Wimmer

February 4, 2026

Abstract

Timed automata are a widely used formalism for modeling real-time systems, which is employed in a class of successful model checkers such as UPPAAL [LPY97], HyTech [HHWt97] or Kronos [Yov97]. This work formalizes the theory for the subclass of diagonal-free timed automata, which is sufficient to model many interesting problems. We first define the basic concepts and semantics of diagonal-free timed automata. Based on this, we prove two types of decidability results for the language emptiness problem.

The first is the classic result of Alur and Dill [AD90, AD94], which uses a finite partitioning of the state space into so-called *regions*.

Our second result focuses on an approach based on *Difference Bound Matrices (DBMs)*, which is practically used by model checkers. We prove the correctness of the basic forward analysis operations on DBMs. One of these operations is the Floyd-Warshall algorithm for the all-pairs shortest paths problem. To obtain a finite search space, a widening operation has to be used for this kind of analysis. We use Patricia Bouyer's [Bou04] approach to prove that this widening operation is correct in the sense that DBM-based forward analysis in combination with the widening operation also decides language emptiness. The interesting property of this proof is that the first decidability result is reused to obtain the second one.

Contents

1	Miscellaneous	4
1.1	Lists	4
1.2	Streams	6
1.3	Mixed Material	10
2	Graphs	16
2.1	Basic Definitions and Theorems	16
2.2	Graphs with a Start Node	22
2.3	Subgraphs	23
2.4	Bundles	26

2.5	Directed Acyclic Graphs	27
2.6	Finite Graphs	27
2.7	Graph Invariants	28
2.8	Simulations and Bisimulations	30
2.9	CTL	39
3	Basic Definitions and Semantics	43
3.1	Syntactic Definition	43
3.2	Operational Semantics	45
3.3	Contracting Runs	47
3.4	Zone Semantics	48
3.5	From Clock Constraints to DBMs	51
3.6	Semantics Based on DBMs	56
4	Refinement to β-regions	58
4.1	Definition	58
4.2	Basic Properties	61
4.3	Approximation with β -regions	63
4.4	Computing β -Approximation	67
4.5	Auxiliary β -boundedness Theorems	68
5	The Classic Construction for Decidability	71
5.1	Definition of Regions	71
5.2	Basic Properties	73
5.3	Set of Regions	75
5.4	Compability With Clock Constraints	79
5.5	Compability with Resets	80
5.6	A Semantics Based on Regions	81
5.7	Correct Approximation of Zones with α -regions	84
5.8	Old Variant Using a Global Set of Regions	84
5.9	A Zone Semantics Abstracting with $Closure_\alpha$	87
5.10	New Variant	90
5.11	A Semantics Based on Localized Regions	91
5.12	A New Zone Semantics Abstracting with $Closure_{\alpha,l}$	93
6	Correctness of β-approximation from α-regions	95
6.1	Preparing Bouyer's Theorem	96
6.2	Bouyer's Main Theorem	99
6.3	Nice Corollaries of Bouyer's Theorem	99
6.4	A New Zone Semantics Abstracting with $Approx_\beta$	100

7	Simulation Graphs	105
7.1	Simulation Graphs	105
7.2	Poststability	109
7.3	Prestability	110
7.4	Double Simulation	111
7.5	Finite Graphs	113
7.6	Complete Simulation Graphs	114
7.7	Finite Complete Double Simulations	115
7.8	Encoding of Properties in Runs	118
7.9	Instantiation of Simulation Locales	139
8	Forward Analysis with DBMs and Widening	151
8.1	DBM-based Semantics with Normalization	153
8.2	Additional Useful Properties of the Normalized Semantics . .	162
8.3	Appendix: Standard Clock Numberings for Concrete Models	162

1 Miscellaneous

1.1 Lists

theory *More-List1*

imports

Main

Instantiate-Existentials

begin

1.1.1 First and Last Elements of Lists

lemma (**in** $-$) *hd-butlast-last-id*:

$hd\ xs \ \# \ tl\ (butlast\ xs) \ @ \ [last\ xs] = xs$ **if** $length\ xs > 1$
 $\langle proof \rangle$

1.1.2 *list-all*

lemma (**in** $-$) *list-all-map*:

assumes $inv: \bigwedge x. P\ x \implies \exists y. f\ y = x$

and $all: list-all\ P\ as$

shows $\exists as'. map\ f\ as' = as$

$\langle proof \rangle$

1.1.3 *list-all2*

lemma *list-all2-op-map-iff*:

$list-all2\ (\lambda a\ b. b = f\ a)\ xs\ ys \longleftrightarrow map\ f\ xs = ys$
 $\langle proof \rangle$

lemma *list-all2-last*:

$R\ (last\ xs)\ (last\ ys)$ **if** $list-all2\ R\ xs\ ys$ $xs \neq []$
 $\langle proof \rangle$

lemma *list-all2-set1*:

$\forall x \in set\ xs. \exists xa \in set\ as. P\ x\ xa$ **if** $list-all2\ P\ xs\ as$
 $\langle proof \rangle$

lemma *list-all2-swap*:

$list-all2\ P\ xs\ ys \longleftrightarrow list-all2\ (\lambda x\ y. P\ y\ x)\ ys\ xs$
 $\langle proof \rangle$

lemma *list-all2-set2*:

$\forall x \in set\ as. \exists xa \in set\ xs. P\ xa\ x$ **if** $list-all2\ P\ xs\ as$
 $\langle proof \rangle$

1.1.4 Distinct lists

lemma *distinct-length-le*: $\text{finite } s \implies \text{set } xs \subseteq s \implies \text{distinct } xs \implies \text{length } xs \leq \text{card } s$
<proof>

1.1.5 filter

lemma *filter-eq-appendD*:

$\exists xs' ys'. \text{filter } P xs' = xs \wedge \text{filter } P ys' = ys \wedge as = xs' @ ys' \text{ if } \text{filter } P as = xs @ ys$
<proof>

lemma *list-all2-elem-filter*:

assumes *list-all2* $P xs us \ x \in \text{set } xs$
shows $\text{length } (\text{filter } (P x) us) \geq 1$
<proof>

lemma *list-all2-replicate-elem-filter*:

assumes *list-all2* $P (\text{concat } (\text{replicate } n xs)) ys \ x \in \text{set } xs$
shows $\text{length } (\text{filter } (P x) ys) \geq n$
<proof>

1.1.6 Sublists

lemma *nths-split*:

$\text{nths } xs (A \cup B) = \text{nths } xs A @ \text{nths } xs B \text{ if } \forall i \in A. \forall j \in B. i < j$
<proof>

lemma *nths-nth*:

$\text{nths } xs \{i\} = [xs ! i] \text{ if } i < \text{length } xs$
<proof>

lemma *nths-shift*:

$\text{nths } (xs @ ys) S = \text{nths } ys \{x - \text{length } xs \mid x. x \in S\} \text{ if } \forall i \in S. \text{length } xs \leq i$
<proof>

lemma *nths-eq-ConsD*:

assumes $\text{nths } xs I = x \# as$

shows

$\exists ys zs.$

$xs = ys @ x \# zs \wedge \text{length } ys \in I \wedge (\forall i \in I. i \geq \text{length } ys)$

$\wedge \text{nths } zs (\{i - \text{length } ys - 1 \mid i. i \in I \wedge i > \text{length } ys\}) = as$

<proof>

lemma *nths-out-of-bounds*:
nths xs I = [] **if** $\forall i \in I. i \geq \text{length } xs$

<proof>

lemma *nths-eq-appendD*:
assumes *nths xs I = as @ bs*

shows

$\exists ys zs.$

$xs = ys @ zs \wedge \text{nths } ys I = as$

$\wedge \text{nths } zs \{i - \text{length } ys \mid i. i \in I \wedge i \geq \text{length } ys\} = bs$

<proof>

lemma *filter-nths-length*:

$\text{length } (\text{filter } P (\text{nths } xs I)) \leq \text{length } (\text{filter } P xs)$

<proof>

end

1.2 Streams

theory *Stream-More*

imports

Transition-Systems-and-Automata.Sequence-LTL

Instantiate-Existentials

HOL-Library.Rewrite

begin

lemma *list-all-stake-least*:

$\text{list-all } (\text{Not} \circ P) (\text{stake } (\text{LEAST } n. P (xs !! n)) xs) (\text{is } ?G) \text{ if } \exists n. P (xs !! n)$

<proof>

lemma *alw-stream-all2-mono*:

assumes $\text{stream-all2 } P xs ys \text{ alw } Q xs \wedge xs ys. \text{stream-all2 } P xs ys \implies Q xs \implies R ys$

shows $\text{alw } R ys$

<proof>

lemma *alw-ev-HLD-cycle*:

assumes $\text{stream-all2 } (\in) xs (\text{cycle } as) a \in \text{set } as$

shows $\text{infs } (\lambda x. x \in a) xs$

<proof>

lemma *alw-ev-mono*:

assumes $alw (ev \varphi) xs$ **and** $\bigwedge xs. \varphi xs \implies \psi xs$
shows $alw (ev \psi) xs$
<proof>

lemma *alw-ev-lockstep*:

assumes
 $alw (ev (holds P)) xs$ *stream-all2* $Q xs$ *as*
 $\bigwedge x a. P x \implies Q x a \implies R a$
shows
 $alw (ev (holds R)) as$
<proof>

1.2.1 sfilter, wait, nxt

Useful?

lemma *nxt-holds-iff-snth*: $(nxt \overset{\sim}{\sim} i) (holds P) xs \longleftrightarrow P (xs !! i)$
<proof>

Useful?

lemma *wait-LEAST*:

$wait (holds P) xs = (LEAST n. P (xs !! n))$ *<proof>*

lemma *sfilter-SCons-decomp*:

assumes $sfilter P xs = x \#\# zs$ $ev (holds P) xs$
shows $\exists ys' zs'. xs = ys' @- x \#\# zs' \wedge list-all (Not o P) ys' \wedge P x \wedge$
 $sfilter P zs' = zs$
<proof>

lemma *sfilter-SCons-decomp'*:

assumes $sfilter P xs = x \#\# zs$ $ev (holds P) xs$
shows
 $list-all (Not o P) (stake (wait (holds P) xs) xs) (\mathbf{is} \ ?G1)$
 $P x$
 $\exists zs'. xs = stake (wait (holds P) xs) xs @- x \#\# zs' \wedge sfilter P zs' =$
 $zs (\mathbf{is} \ ?G2)$
<proof>

lemma *sfilter-shift-decomp*:

assumes $sfilter P xs = ys @- zs$ $alw (ev (holds P)) xs$
shows $\exists ys' zs'. xs = ys' @- zs' \wedge filter P ys' = ys \wedge sfilter P zs' = zs$
<proof>

lemma *finite-sset-sfilter-decomp*:

assumes *finite* (*sset* (*sfilter* P xs)) *alw* (*ev* (*holds* P)) xs

obtains x ws ys zs **where** $xs = ws @- x \#\# ys @- x \#\# zs$ P x

<proof>

Useful?

lemma *sfilter-shd-LEAST*:

shd (*sfilter* P xs) = $xs !! (LEAST$ $n. P$ ($xs !! n$)) **if** *ev* (*holds* P) xs

<proof>

lemma *alw-nxt-holds-cong*:

(*nxt* $\overset{\sim}{\sim}$ n) (*holds* ($\lambda x. P$ $x \wedge Q$ x)) $xs = (nxt$ $\overset{\sim}{\sim}$ n) (*holds* Q) xs **if** *alw* (*holds* P) xs

<proof>

lemma *alw-wait-holds-cong*:

wait (*holds* ($\lambda x. P$ $x \wedge Q$ x)) $xs = wait$ (*holds* Q) xs **if** *alw* (*holds* P) xs

<proof>

lemma *alw-sfilter*:

sfilter ($\lambda x. P$ $x \wedge Q$ x) $xs = sfilter$ Q xs **if** *alw* (*holds* P) xs *alw* (*ev* (*holds* Q)) xs

<proof>

lemma *alw-ev-holds-mp*:

alw (*holds* P) $xs \implies ev$ (*holds* Q) $xs \implies ev$ (*holds* ($\lambda x. P$ $x \wedge Q$ x)) xs

<proof>

lemma *alw-ev-conjI*:

alw (*ev* (*holds* ($\lambda x. P$ $x \wedge Q$ x))) xs **if** *alw* (*holds* P) xs *alw* (*ev* (*holds* Q)) xs

<proof>

1.2.2 Useful?

lemma *alw-holds-pred-stream-iff*:

alw (*holds* P) $xs \longleftrightarrow pred-stream$ P xs

<proof>

lemma *alw-holds-sset*:

alw (*holds* P) $xs = (\forall x \in sset$ $xs. P$ x)

<proof>

lemma *pred-stream-sfilter*:

assumes *alw-ev*: $\text{alw } (ev \text{ (holds } P)) \text{ } xs$
shows *pred-stream* $P \text{ (sfilter } P \text{ } xs)$
 $\langle proof \rangle$

lemma *alw-ev-sfilter-mono*:

assumes *alw-ev*: $\text{alw } (ev \text{ (holds } P)) \text{ } xs$
and *mono*: $\bigwedge x. P \text{ } x \implies Q \text{ } x$
shows *pred-stream* $Q \text{ (sfilter } P \text{ } xs)$
 $\langle proof \rangle$

lemma *sset-sfilter*:

sset (sfilter P xs) \subseteq sset xs **if** *alw (ev (holds P)) xs*
 $\langle proof \rangle$

lemma *stream-all2-weaken*:

stream-all2 Q xs ys **if** *stream-all2 P xs ys* $\bigwedge x \text{ } y. P \text{ } x \text{ } y \implies Q \text{ } x \text{ } y$
 $\langle proof \rangle$

lemma *stream-all2-SCons1*:

stream-all2 P (x ## xs) ys = ($\exists z \text{ } zs. ys = z ## zs \wedge P \text{ } x \text{ } z \wedge \text{stream-all2 } P \text{ } xs \text{ } zs)$
 $\langle proof \rangle$

lemma *stream-all2-SCons2*:

stream-all2 P xs (y ## ys) = ($\exists z \text{ } zs. xs = z ## zs \wedge P \text{ } z \text{ } y \wedge \text{stream-all2 } P \text{ } zs \text{ } ys)$
 $\langle proof \rangle$

lemma *stream-all2-combine*:

stream-all2 R xs zs **if**
stream-all2 P xs ys *stream-all2 Q ys zs* $\bigwedge x \text{ } y \text{ } z. P \text{ } x \text{ } y \wedge Q \text{ } y \text{ } z \implies R \text{ } x \text{ } z$
 $\langle proof \rangle$

lemma *stream-all2-shift1*:

stream-all2 P (xs1 @- xs2) ys =
 $(\exists \text{ } ys1 \text{ } ys2. ys = ys1 @- ys2 \wedge \text{list-all2 } P \text{ } xs1 \text{ } ys1 \wedge \text{stream-all2 } P \text{ } xs2 \text{ } ys2)$
 $\langle proof \rangle$

lemma *stream-all2-shift2*:

stream-all2 P ys (xs1 @- xs2) =
 $(\exists \text{ } ys1 \text{ } ys2. ys = ys1 @- ys2 \wedge \text{list-all2 } P \text{ } ys1 \text{ } xs1 \wedge \text{stream-all2 } P \text{ } ys2 \text{ } xs2)$
 $\langle proof \rangle$

lemma *stream-all2-bisim*:
assumes *stream-all2* (\in) *xs* as *stream-all2* (\in) *ys* as *sset* as $\subseteq S$
shows *stream-all2* ($\lambda x y. \exists a. x \in a \wedge y \in a \wedge a \in S$) *xs ys*
 \langle *proof* \rangle

end

1.3 Mixed Material

theory *TA-Misc*
imports *Main HOL.Real*
begin

1.3.1 Reals

Properties of fractions **lemma** *frac-add-le-preservation*:

fixes $a d :: \text{real}$ **and** $b :: \text{nat}$
assumes $a < b$ $d < 1 - \text{frac } a$
shows $a + d < b$
 \langle *proof* \rangle

lemma *lt-1-contr*:
 $(a :: \text{int}) < b \implies b < a + 1 \implies \text{False}$ \langle *proof* \rangle

lemma *int-intv-frac-gt0*:
 $(a :: \text{int}) < b \implies b < a + 1 \implies \text{frac } b > 0$ \langle *proof* \rangle

lemma *floor-frac-add-preservation*:

fixes $a d :: \text{real}$
assumes $0 < d$ $d < 1 - \text{frac } a$
shows $\text{floor } a = \text{floor } (a + d)$
 \langle *proof* \rangle

lemma *frac-distr*:

fixes $a d :: \text{real}$
assumes $0 < d$ $d < 1 - \text{frac } a$
shows $\text{frac } (a + d) > 0$ $\text{frac } a + d = \text{frac } (a + d)$
 \langle *proof* \rangle

lemma *frac-add-leD*:

fixes $a d :: \text{real}$
assumes $0 < d$ $d < 1 - \text{frac } a$ $d < 1 - \text{frac } b$ $\text{frac } (a + d) \leq \text{frac } (b + d)$

shows $\text{frac } a \leq \text{frac } b$
<proof>

lemma *floor-frac-add-preservation'*:
fixes $a d :: \text{real}$
assumes $0 \leq d \ d < 1 - \text{frac } a$
shows $\text{floor } a = \text{floor } (a + d)$
<proof>

lemma *frac-add-leIFF*:
fixes $a d :: \text{real}$
assumes $0 \leq d \ d < 1 - \text{frac } a \ d < 1 - \text{frac } b$
shows $\text{frac } a \leq \text{frac } b \iff \text{frac } (a + d) \leq \text{frac } (b + d)$
<proof>

lemma *nat-intv-frac-gt0*:
fixes $c :: \text{nat}$ **fixes** $x :: \text{real}$
assumes $c < x \ x < \text{real } (c + 1)$
shows $\text{frac } x > 0$
<proof>

lemma *nat-intv-frac-decomp*:
fixes $c :: \text{nat}$ **and** $d :: \text{real}$
assumes $c < d \ d < c + 1$
shows $d = c + \text{frac } d$
<proof>

lemma *nat-intv-not-int*:
fixes $c :: \text{nat}$
assumes $\text{real } c < d \ d < c + 1$
shows $d \notin \mathbb{Z}$
<proof>

lemma *frac-nat-add-id*: $\text{frac } ((n :: \text{nat}) + (r :: \text{real})) = \text{frac } r$ — Found by sledgehammer
<proof>

lemma *floor-nat-add-id*: $0 \leq (r :: \text{real}) \implies r < 1 \implies \text{floor } (\text{real } (n :: \text{nat}) + r) = n$ *<proof>*

lemma *int-intv-frac-gt-0'*:
 $(a :: \text{real}) \in \mathbb{Z} \implies (b :: \text{real}) \in \mathbb{Z} \implies a \leq b \implies a \neq b \implies a \leq b - 1$
<proof>

lemma *int-lt-Suc-le*:

$(a :: \text{real}) \in \mathbb{Z} \implies (b :: \text{real}) \in \mathbb{Z} \implies a < b + 1 \implies a \leq b$
<proof>

lemma *int-lt-neq-Suc-lt*:

$(a :: \text{real}) \in \mathbb{Z} \implies (b :: \text{real}) \in \mathbb{Z} \implies a < b \implies a + 1 \neq b \implies a + 1 < b$
<proof>

lemma *int-lt-neq-prev-lt*:

$(a :: \text{real}) \in \mathbb{Z} \implies (b :: \text{real}) \in \mathbb{Z} \implies a - 1 < b \implies a \neq b \implies a < b$
<proof>

lemma *ints-le-add-frac1*:

fixes $a\ b\ x :: \text{real}$
assumes $0 < x\ x < 1\ a \in \mathbb{Z}\ b \in \mathbb{Z}\ a + x \leq b$
shows $a \leq b$
<proof>

lemma *ints-le-add-frac2*:

fixes $a\ b\ x :: \text{real}$
assumes $0 \leq x\ x < 1\ a \in \mathbb{Z}\ b \in \mathbb{Z}\ b \leq a + x$
shows $b \leq a$
<proof>

1.3.2 Ordering Fractions

lemma *distinct-twice-contradiction*:

$xs ! i = x \implies xs ! j = x \implies i < j \implies j < \text{length } xs \implies \neg \text{distinct } xs$
<proof>

lemma *distinct-nth-unique*:

$xs ! i = xs ! j \implies i < \text{length } xs \implies j < \text{length } xs \implies \text{distinct } xs \implies i = j$
<proof>

lemma (**in** *linorder*) *linorder-order-fun*:

fixes $S :: 'a \text{ set}$
assumes *finite* S
obtains $f :: 'a \Rightarrow \text{nat}$
where $(\forall x \in S. \forall y \in S. f\ x \leq f\ y \iff x \leq y)$ **and** $\text{range } f \subseteq \{0.. \text{card } S - 1\}$
<proof>

locale *enumerable* =
fixes $T :: 'a \text{ set}$
fixes $less :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** \prec 50)
assumes $finite: finite\ T$
assumes $total: \forall x \in T. \forall y \in T. x \neq y \longrightarrow (x \prec y) \vee (y \prec x)$
assumes $trans: \forall x \in T. \forall y \in T. \forall z \in T. (x :: 'a) \prec y \longrightarrow y \prec z \longrightarrow x \prec z$
assumes $asymmetric: \forall x \in T. \forall y \in T. x \prec y \longrightarrow \neg (y \prec x)$
begin

lemma *non-empty-set-has-least'*:
 $S \subseteq T \Longrightarrow S \neq \{\} \Longrightarrow \exists x \in S. \forall y \in S. x \neq y \longrightarrow \neg y \prec x$
<proof>

lemma *non-empty-set-has-least''*:
 $S \subseteq T \Longrightarrow S \neq \{\} \Longrightarrow \exists! x \in S. \forall y \in S. x \neq y \longrightarrow \neg y \prec x$
<proof>

abbreviation $least\ S \equiv THE\ t :: 'a. t \in S \wedge (\forall y \in S. t \neq y \longrightarrow \neg y \prec t)$

lemma *non-empty-set-has-least*:
 $S \subseteq T \Longrightarrow S \neq \{\} \Longrightarrow least\ S \in S \wedge (\forall y \in S. least\ S \neq y \longrightarrow \neg y \prec least\ S)$
<proof>

fun $f :: 'a \text{ set} \Rightarrow \text{nat} \Rightarrow 'a \text{ list}$
where
 $f\ S\ 0 = []$
 $f\ S\ (Suc\ n) = least\ S \# f\ (S - \{least\ S\})\ n$

inductive $sorted :: 'a \text{ list} \Rightarrow \text{bool}$ **where**
 Nil [iiff]: $sorted\ []$
| $Cons: \forall y \in set\ xs. x \prec y \Longrightarrow sorted\ xs \Longrightarrow sorted\ (x \# xs)$

lemma *f-set*:
 $S \subseteq T \Longrightarrow n = card\ S \Longrightarrow set\ (f\ S\ n) = S$
<proof>

lemma *f-distinct*:
 $S \subseteq T \Longrightarrow n = card\ S \Longrightarrow distinct\ (f\ S\ n)$
<proof>

lemma *f-sorted*:

$S \subseteq T \implies n = \text{card } S \implies \text{sorted } (f S n)$
 <proof>

lemma *sorted-nth-mono*:
 $\text{sorted } xs \implies i < j \implies j < \text{length } xs \implies xs!i \prec xs!j$
 <proof>

lemma *order-fun*:
fixes $S :: 'a \text{ set}$
assumes $S \subseteq T$
obtains $f :: 'a \Rightarrow \text{nat}$ **where** $\forall x \in S. \forall y \in S. f x < f y \iff x \prec y$
and $\text{range } f \subseteq \{0.. \text{card } S - 1\}$
 <proof>

end

lemma *finite-total-preorder-enumeration*:
fixes $X :: 'a \text{ set}$
fixes $r :: 'a \text{ rel}$
assumes *fin*: $\text{finite } X$
assumes *tot*: $\text{total-on } X r$
assumes *refl*: $\text{refl-on } X r$
assumes *trans*: $\text{trans } r$
obtains $f :: 'a \Rightarrow \text{nat}$ **where** $\forall x \in X. \forall y \in X. f x \leq f y \iff (x, y) \in r$
 <proof>

1.3.3 Finiteness

lemma *pairwise-finiteI*:
assumes $\text{finite } \{b. \exists a. P a b\}$ (**is finite** ?B)
assumes $\text{finite } \{a. \exists b. P a b\}$
shows $\text{finite } \{(a,b). P a b\}$ (**is finite** ?C)
 <proof>

lemma *finite-ex-and1*:
assumes $\text{finite } \{b. \exists a. P a b\}$ (**is finite** ?A)
shows $\text{finite } \{b. \exists a. P a b \wedge Q a b\}$ (**is finite** ?B)
 <proof>

lemma *finite-ex-and2*:
assumes $\text{finite } \{b. \exists a. Q a b\}$ (**is finite** ?A)
shows $\text{finite } \{b. \exists a. P a b \wedge Q a b\}$ (**is finite** ?B)
 <proof>

1.3.4 Numbering the elements of finite sets

lemma *upt-last-append*: $a \leq b \implies [a..<b] @ [b] = [a ..< Suc b]$ *<proof>*

lemma *map-of-zip-dom-to-range*:

$a \in \text{set } A \implies \text{length } B = \text{length } A \implies \text{the } (\text{map-of } (\text{zip } A B) a) \in \text{set } B$
<proof>

lemma *zip-range-id*:

$\text{length } A = \text{length } B \implies \text{snd } ` \text{set } (\text{zip } A B) = \text{set } B$
<proof>

lemma *map-of-zip-in-range*:

$\text{distinct } A \implies \text{length } B = \text{length } A \implies b \in \text{set } B \implies \exists a \in \text{set } A. \text{the } (\text{map-of } (\text{zip } A B) a) = b$
<proof>

lemma *distinct-zip-inj*:

$\text{distinct } ys \implies (a, b) \in \text{set } (\text{zip } xs \ ys) \implies (c, b) \in \text{set } (\text{zip } xs \ ys) \implies a = c$
<proof>

lemma *map-of-zip-distinct-inj*:

$\text{distinct } B \implies \text{length } A = \text{length } B \implies \text{inj-on } (\text{the } o \text{ map-of } (\text{zip } A B)) (\text{set } A)$
<proof>

lemma *nat-not-ge-1D*: $\neg \text{Suc } 0 \leq x \implies x = 0$ *<proof>*

lemma *standard-numbering*:

assumes *finite* A
obtains $v :: 'a \Rightarrow \text{nat}$ **and** n **where** *bij-betw* v A $\{1..n\}$
and $\forall c \in A. v \ c > 0$
and $\forall c. c \notin A \longrightarrow v \ c > n$
<proof>

1.3.5 Products

lemma *prod-set-fst-id*:

$x = y$ **if** $\forall a \in x. \text{fst } a = b \ \forall a \in y. \text{fst } a = b$ **snd** $` x = \text{snd } ` y$
<proof>

end

2 Graphs

```
theory Graphs
  imports
    More-List1 Stream-More
    HOL-Library.Rewrite
begin
```

2.1 Basic Definitions and Theorems

```
locale Graph-Defs =
  fixes E :: 'a ⇒ 'a ⇒ bool
begin
```

```
inductive steps where
  Single: steps [x] |
  Cons: steps (x # y # xs) if E x y steps (y # xs)
```

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

```
lemma steps-append:
  steps (xs @ tl ys) if steps xs steps ys last xs = hd ys
  <proof>
```

```
lemma steps-append':
  steps xs if steps as steps bs last as = hd bs as @ tl bs = xs
  <proof>
```

```
coinductive run where
  run (x ## y ## xs) if E x y run (y ## xs)
```

```
lemmas [intro] = run.intros
```

```
lemma steps-appendD1:
  steps xs if steps (xs @ ys) xs ≠ []
  <proof>
```

```
lemma steps-appendD2:
  steps ys if steps (xs @ ys) ys ≠ []
  <proof>
```

```
lemma steps-appendD3:
  steps (xs @ [x]) ∧ E x y if steps (xs @ [x, y])
  <proof>
```

lemma *steps-ConsD*:

steps xs **if** *steps (x # xs) xs* $\neq []$

<proof>

lemmas *stepsD = steps-ConsD steps-appendD1 steps-appendD2*

lemma *steps-alt-induct*[*consumes 1, case-names Single Snoc*]:

assumes

steps x ($\bigwedge x. P [x]$)

$\bigwedge y x xs. E y x \implies steps (xs @ [y]) \implies P (xs @ [y]) \implies P (xs @ [y,x])$

shows *P x*

<proof>

lemma *steps-appendI*:

steps (xs @ [x, y]) **if** *steps (xs @ [x])* *E x y*

<proof>

lemma *steps-append-single*:

assumes

steps xs E (last xs) x xs $\neq []$

shows *steps (xs @ [x])*

<proof>

lemma *extend-run*:

assumes

steps xs E (last xs) x run (x ## ys) xs $\neq []$

shows *run (xs @- x ## ys)*

<proof>

lemma *run-cycle*:

assumes *steps xs E (last xs) (hd xs) xs* $\neq []$

shows *run (cycle xs)*

<proof>

lemma *run-stl*:

run (stl xs) **if** *run xs*

<proof>

lemma *run-sdrop*:

run (sdrop n xs) **if** *run xs*

<proof>

lemma *run-reachable'*:

assumes $run (x \#\# xs) E^{**} x_0 x$
shows $pred-stream (\lambda x. E^{**} x_0 x) xs$
 $\langle proof \rangle$

lemma *run-reachable*:

assumes $run (x_0 \#\# xs)$
shows $pred-stream (\lambda x. E^{**} x_0 x) xs$
 $\langle proof \rangle$

lemma *run-decomp*:

assumes $run (xs @- ys) xs \neq []$
shows $steps xs \wedge run ys \wedge E (last xs) (shd ys)$
 $\langle proof \rangle$

lemma *steps-decomp*:

assumes $steps (xs @ ys) xs \neq [] \wedge ys \neq []$
shows $steps xs \wedge steps ys \wedge E (last xs) (hd ys)$
 $\langle proof \rangle$

lemma *steps-rotate*:

assumes $steps (x \# xs @ y \# ys @ [x])$
shows $steps (y \# ys @ x \# xs @ [y])$
 $\langle proof \rangle$

lemma *run-shift-coinduct*[*case-names run-shift, consumes 1*]:

assumes $R w$
and $\bigwedge w. R w \implies \exists u v x y. w = u @- x \#\# y \#\# v \wedge steps (u @ [x]) \wedge E x y \wedge R (y \#\# v)$
shows $run w$
 $\langle proof \rangle$

lemma *run-flat-coinduct*[*case-names run-shift, consumes 1*]:

assumes $R xss$
and
 $\bigwedge xs ys xss. R (xs \#\# ys \#\# xss) \implies xs \neq [] \wedge steps xs \wedge E (last xs) (hd ys) \wedge R (ys \#\# xss)$
shows $run (flat xss)$
 $\langle proof \rangle$

lemma *steps-non-empty*[*simp*]:

$\neg steps []$
 $\langle proof \rangle$

lemma *steps-non-empty*'[simp]:

$xs \neq []$ **if** $steps\ xs$
 $\langle proof \rangle$

lemma *steps-replicate*:

$steps\ (hd\ xs\ \# \text{concat}\ (\text{replicate}\ n\ (tl\ xs)))$ **if** $last\ xs = hd\ xs$ $steps\ xs\ n > 0$
 $\langle proof \rangle$

notation $E\ (\leftarrow \rightarrow \rightarrow [100, 100] 40)$

abbreviation *reaches* $(\leftarrow \rightarrow^* \rightarrow [100, 100] 40)$ **where** $reaches\ x\ y \equiv E^{**}\ x\ y$

abbreviation *reaches1* $(\leftarrow \rightarrow^+ \rightarrow [100, 100] 40)$ **where** $reaches1\ x\ y \equiv E^{++}\ x\ y$

lemma *steps-reaches*:

$hd\ xs \rightarrow^* last\ xs$ **if** $steps\ xs$
 $\langle proof \rangle$

lemma *steps-reaches'*:

$x \rightarrow^* y$ **if** $steps\ xs$ $hd\ xs = x$ $last\ xs = y$
 $\langle proof \rangle$

lemma *reaches-steps*:

$\exists xs. hd\ xs = x \wedge last\ xs = y \wedge steps\ xs$ **if** $x \rightarrow^* y$
 $\langle proof \rangle$

lemma *reaches-steps-iff*:

$x \rightarrow^* y \longleftrightarrow (\exists xs. hd\ xs = x \wedge last\ xs = y \wedge steps\ xs)$
 $\langle proof \rangle$

lemma *steps-reaches1*:

$x \rightarrow^+ y$ **if** $steps\ (x\ \# \ xs\ @ [y])$
 $\langle proof \rangle$

lemma *stepsI*:

$steps\ (x\ \# \ xs)$ **if** $x \rightarrow hd\ xs$ $steps\ xs$
 $\langle proof \rangle$

lemma *reaches1-steps*:

$\exists xs. steps\ (x\ \# \ xs\ @ [y])$ **if** $x \rightarrow^+ y$

$\langle proof \rangle$

lemma *reaches1-steps-iff*:

$x \rightarrow^+ y \iff (\exists xs. steps (x \# xs @ [y]))$
 $\langle proof \rangle$

lemma *reaches-steps-iff2*:

$x \rightarrow^* y \iff (x = y \vee (\exists vs. steps (x \# vs @ [y])))$
 $\langle proof \rangle$

lemma *reaches1-reaches-iff1*:

$x \rightarrow^+ y \iff (\exists z. x \rightarrow z \wedge z \rightarrow^* y)$
 $\langle proof \rangle$

lemma *reaches1-reaches-iff2*:

$x \rightarrow^+ y \iff (\exists z. x \rightarrow^* z \wedge z \rightarrow y)$
 $\langle proof \rangle$

lemma

$x \rightarrow^+ z$ **if** $x \rightarrow^* y$ $y \rightarrow^+ z$
 $\langle proof \rangle$

lemma

$x \rightarrow^+ z$ **if** $x \rightarrow^+ y$ $y \rightarrow^* z$
 $\langle proof \rangle$

lemma *steps-append2*:

$steps (xs @ x \# ys)$ **if** $steps (xs @ [x])$ $steps (x \# ys)$
 $\langle proof \rangle$

lemma *reaches1-steps-append*:

assumes $a \rightarrow^+ b$ $steps xs$ $hd xs = b$
shows $\exists ys. steps (a \# ys @ xs)$
 $\langle proof \rangle$

lemma *steps-last-step*:

$\exists a. a \rightarrow last xs$ **if** $steps xs$ $length xs > 1$
 $\langle proof \rangle$

lemma *steps-remove-cycleE*:

assumes $steps (a \# xs @ [b])$
obtains ys **where** $steps (a \# ys @ [b])$ $distinct ys$ $a \notin set ys$ $b \notin set ys$
 $set ys \subseteq set xs$
 $\langle proof \rangle$

lemma *reaches1-stepsE*:

assumes $a \rightarrow^+ b$

obtains xs **where** $steps (a \# xs @ [b])$ *distinct* xs $a \notin set\ xs$ $b \notin set\ xs$

<proof>

lemma *reaches-stepsE*:

assumes $a \rightarrow^* b$

obtains $a = b \mid xs$ **where** $steps (a \# xs @ [b])$ *distinct* xs $a \notin set\ xs$ $b \notin set\ xs$

<proof>

definition *sink where*

sink $a \equiv \nexists b. a \rightarrow b$

lemma *sink-or-cycle*:

assumes *finite* $\{b. reaches\ a\ b\}$

obtains b **where** $reaches\ a\ b$ *sink* $b \mid b$ **where** $reaches\ a\ b$ *reaches1* $b\ b$

<proof>

A directed graph where every node has at least one ingoing edge, contains a directed cycle.

lemma *directed-graph-indegree-ge-1-cycle'*:

assumes *finite* S $S \neq \{\}$ $\forall y \in S. \exists x \in S. E\ x\ y$

shows $\exists x \in S. \exists y. E\ x\ y \wedge E^{**}\ y\ x$

<proof>

lemma *directed-graph-indegree-ge-1-cycle*:

assumes *finite* S $S \neq \{\}$ $\forall y \in S. \exists x \in S. E\ x\ y$

shows $\exists x \in S. \exists y. x \rightarrow^+ x$

<proof>

Vertices of a graph

definition *vertices* = $\{x. \exists y. E\ x\ y \vee E\ y\ x\}$

lemma *reaches1-verts*:

assumes $x \rightarrow^+ y$

shows $x \in vertices$ **and** $y \in vertices$

<proof>

lemmas *graphI* =

steps.intros

steps-append-single

steps-reaches'
stepsI

end

2.2 Graphs with a Start Node

locale *Graph-Start-Defs* = *Graph-Defs* +

fixes $s_0 :: 'a$

begin

definition *reachable* **where**

$reachable = E^{**} s_0$

lemma *start-reachable*[*intro!*, *simp*]:

$reachable s_0$

<proof>

lemma *reachable-step*:

$reachable b$ **if** $reachable a \ E \ a \ b$

<proof>

lemma *reachable-reaches*:

$reachable b$ **if** $reachable a \ a \ \rightarrow^* \ b$

<proof>

lemma *reachable-steps-append*:

assumes $reachable a \ steps \ xs \ hd \ xs = a \ last \ xs = b$

shows $reachable b$

<proof>

lemmas $steps-reachable = reachable-steps-append[of \ s_0, \ simplified]$

lemma *reachable-steps-elem*:

$reachable y$ **if** $reachable x \ steps \ xs \ y \ \in \ set \ xs \ hd \ xs = x$

<proof>

lemma *reachable-steps*:

$\exists \ xs. \ steps \ xs \ \wedge \ hd \ xs = s_0 \ \wedge \ last \ xs = x$ **if** $reachable x$

<proof>

lemma *reachable-cycle-iff*:

$reachable x \ \wedge \ x \ \rightarrow^+ \ x \ \longleftrightarrow \ (\exists \ ws \ xs. \ steps \ (s_0 \ \# \ ws \ @ \ [x] \ @ \ xs \ @ \ [x]))$

<proof>

lemma *reachable-induct*[*consumes 1, case-names start step, induct pred: reachable*]:
assumes *reachable x*
and $P\ s_0$
and $\bigwedge a\ b. \text{reachable } a \implies P\ a \implies a \rightarrow b \implies P\ b$
shows $P\ x$
 $\langle \text{proof} \rangle$

lemmas *graphI-aggressive* =
tranclp-into-rtranclp
rtranclp.rtrancl-into-rtrancl
tranclp.trancl-into-trancl
rtranclp-into-tranclp2

lemmas *graphI-aggressive1* =
graphI-aggressive
steps-append'

lemmas *graphI-aggressive2* =
graphI-aggressive
stepsD
steps-reaches1
steps-reachable

lemmas *graphD* =
reaches1-steps

lemmas *graphD-aggressive* =
tranclpD

lemmas *graph-startI* =
reachable-reaches
start-reachable

end

2.3 Subgraphs

2.3.1 Edge-induced Subgraphs

locale *Subgraph-Defs* = $G: \text{Graph-Defs} +$
fixes $E' :: 'a \Rightarrow 'a \Rightarrow \text{bool}$
begin

```

sublocale  $G'$ : Graph-Defs  $E'$   $\langle$ proof $\rangle$ 

end

locale Subgraph-Start-Defs =  $G$ : Graph-Start-Defs +
  fixes  $E' :: 'a \Rightarrow 'a \Rightarrow \text{bool}$ 
begin

sublocale  $G'$ : Graph-Start-Defs  $E'$   $s_0$   $\langle$ proof $\rangle$ 

end

locale Subgraph = Subgraph-Defs +
  assumes subgraph[intro]:  $E' a b \implies E a b$ 
begin

lemma non-subgraph-cycle-decomp:
   $\exists c d. G.\text{reaches } a c \wedge E c d \wedge \neg E' c d \wedge G.\text{reaches } d b$  if
   $G.\text{reaches1 } a b \neg G'.\text{reaches1 } a b$  for  $a b$ 
   $\langle$ proof $\rangle$ 

lemma reaches:
   $G.\text{reaches } a b$  if  $G'.\text{reaches } a b$ 
   $\langle$ proof $\rangle$ 

lemma reaches1:
   $G.\text{reaches1 } a b$  if  $G'.\text{reaches1 } a b$ 
   $\langle$ proof $\rangle$ 

end

locale Subgraph-Start = Subgraph-Start-Defs + Subgraph
begin

lemma reachable-subgraph[intro]:  $G.\text{reachable } b$  if  $\langle G.\text{reachable } a \rangle \langle G'.\text{reaches } a b \rangle$  for  $a b$ 
   $\langle$ proof $\rangle$ 

lemma reachable:
   $G.\text{reachable } x$  if  $G'.\text{reachable } x$ 
   $\langle$ proof $\rangle$ 

end

```

2.3.2 Node-induced Subgraphs

locale *Subgraph-Node-Defs* = *Graph-Defs* +

fixes $V :: 'a \Rightarrow \text{bool}$

begin

definition E' **where** $E' x y \equiv E x y \wedge V x \wedge V y$

sublocale *Subgraph* $E E'$ $\langle \text{proof} \rangle$

lemma *subgraph'*:

$E' x y$ **if** $E x y V x V y$

$\langle \text{proof} \rangle$

lemma $E'-V1$:

$V x$ **if** $E' x y$

$\langle \text{proof} \rangle$

lemma $E'-V2$:

$V y$ **if** $E' x y$

$\langle \text{proof} \rangle$

lemma G' -reaches- V :

$V y$ **if** $G'.reaches x y V x$

$\langle \text{proof} \rangle$

lemma G' -steps- V -all:

$list-all V xs$ **if** $G'.steps xs V (hd xs)$

$\langle \text{proof} \rangle$

lemma G' -steps- V -last:

$V (last xs)$ **if** $G'.steps xs V (hd xs)$

$\langle \text{proof} \rangle$

lemmas *subgraphI* = $E'-V1 E'-V2 G'$ -reaches- V

lemmas *subgraphD* = $E'-V1 E'-V2 G'$ -reaches- V

end

locale *Subgraph-Node-Defs-Notation* = *Subgraph-Node-Defs*

begin

no-notation E ($\leftarrow \rightarrow \rightarrow$ [100, 100] 40)
notation E' ($\leftarrow \rightarrow \rightarrow$ [100, 100] 40)
no-notation $reaches$ ($\leftarrow \rightarrow^* \rightarrow$ [100, 100] 40)
notation $G'.reaches$ ($\leftarrow \rightarrow^* \rightarrow$ [100, 100] 40)
no-notation $reaches1$ ($\leftarrow \rightarrow^+ \rightarrow$ [100, 100] 40)
notation $G'.reaches1$ ($\leftarrow \rightarrow^+ \rightarrow$ [100, 100] 40)

end

2.3.3 The Reachable Subgraph

context *Graph-Start-Defs*
begin

interpretation *Subgraph-Node-Defs-Notation* E *reachable* $\langle proof \rangle$

sublocale *reachable-subgraph*: *Subgraph-Node-Defs* E *reachable* $\langle proof \rangle$

lemma *reachable-supgraph*:
 $x \rightarrow y$ **if** E x y *reachable* x
 $\langle proof \rangle$

lemma *reachable-reaches-equiv*: $reaches$ x y \longleftrightarrow $x \rightarrow^* y$ **if** *reachable* x **for**
 x y
 $\langle proof \rangle$

lemma *reachable-reaches1-equiv*: $reaches1$ x y \longleftrightarrow $x \rightarrow^+ y$ **if** *reachable* x
for x y
 $\langle proof \rangle$

lemma *reachable-steps-equiv*:
 $steps$ ($x \# xs$) \longleftrightarrow $G'.steps$ ($x \# xs$) **if** *reachable* x
 $\langle proof \rangle$

end

2.4 Bundles

bundle *graph-automation*
begin

lemmas [*intro*] = *Graph-Defs.graphI* *Graph-Start-Defs.graph-startI*
lemmas [*dest*] = *Graph-Start-Defs.graphD*

end

bundle *reaches-steps-iff* =
 Graph-Defs.reaches1-steps-iff [iff]
 Graph-Defs.reaches-steps-iff [iff]

bundle *graph-automation-aggressive*
begin

unbundle *graph-automation*

lemmas [*intro*] = *Graph-Start-Defs.graphI-aggressive*
lemmas [*dest*] = *Graph-Start-Defs.graphD-aggressive*

end

bundle *subgraph-automation*
begin

unbundle *graph-automation*

lemmas [*intro*] = *Subgraph-Node-Defs.subgraphI*
lemmas [*dest*] = *Subgraph-Node-Defs.subgraphD*

end

2.5 Directed Acyclic Graphs

locale *DAG* = *Graph-Defs* +
 assumes *acyclic*: $\neg E^{++} x x$
begin

lemma *topological-numbering*:
 fixes *S* **assumes** *finite S*
 shows $\exists f :: - \Rightarrow \text{nat. inj-on } f S \wedge (\forall x \in S. \forall y \in S. E x y \longrightarrow f x < f y)$
 $\langle \text{proof} \rangle$

end

2.6 Finite Graphs

locale *Finite-Graph* = *Graph-Defs* +
 assumes *finite-graph*: *finite vertices*

locale *Finite-DAG* = *Finite-Graph* + *DAG*
begin

lemma *finite-reachable*:
finite {*y*. *x* \rightarrow^* *y*} (**is** *finite* ?*S*)
 \langle *proof* \rangle

end

2.7 Graph Invariants

locale *Graph-Invariant* = *Graph-Defs* +
fixes *P* :: '*a* \Rightarrow *bool*
assumes *invariant*: *P* *a* \Longrightarrow *a* \rightarrow *b* \Longrightarrow *P* *b*
begin

lemma *invariant-steps*:
list-all *P* *as* **if** *steps* (*a* # *as*) *P* *a*
 \langle *proof* \rangle

lemma *invariant-reaches*:
P *b* **if** *a* \rightarrow^* *b* *P* *a*
 \langle *proof* \rangle

lemma *invariant-run*:
assumes *run*: *run* (*x* ## *xs*) **and** *P*: *P* *x*
shows *pred-stream* *P* (*x* ## *xs*)
 \langle *proof* \rangle

Every graph invariant induces a subgraph.

sublocale *Subgraph-Node-Defs* **where** *E* = *E* **and** *V* = *P* \langle *proof* \rangle

lemma *subgraph'*:
assumes *x* \rightarrow *y* *P* *x*
shows *E'* *x* *y*
 \langle *proof* \rangle

lemma *invariant-steps-iff*:
G'.*steps* (*v* # *vs*) \longleftrightarrow *steps* (*v* # *vs*) **if** *P* *v*
 \langle *proof* \rangle

lemma *invariant-reaches-iff*:
G'.*reaches* *u* *v* \longleftrightarrow *reaches* *u* *v* **if** *P* *u*
 \langle *proof* \rangle

```

lemma invariant-reaches1-iff:
   $G'.reaches1\ u\ v \iff reaches1\ u\ v$  if  $P\ u$ 
   $\langle proof \rangle$ 

end

locale Graph-Invariants = Graph-Defs +
  fixes  $P\ Q :: 'a \Rightarrow bool$ 
  assumes invariant:  $P\ a \implies a \rightarrow b \implies Q\ b$  and Q-P:  $Q\ a \implies P\ a$ 
begin

sublocale Pre: Graph-Invariant  $E\ P$ 
   $\langle proof \rangle$ 

sublocale Post: Graph-Invariant  $E\ Q$ 
   $\langle proof \rangle$ 

lemma invariant-steps:
  list-all  $Q\ as$  if steps  $(a \# as)\ P\ a$ 
   $\langle proof \rangle$ 

lemma invariant-run:
  assumes run: run  $(x \#\# xs)$  and  $P$ :  $P\ x$ 
  shows pred-stream  $Q\ xs$ 
   $\langle proof \rangle$ 

lemma invariant-reaches1:
   $Q\ b$  if  $a \rightarrow^+ b\ P\ a$ 
   $\langle proof \rangle$ 

end

locale Graph-Invariant-Start = Graph-Start-Defs + Graph-Invariant +
  assumes P-s0:  $P\ s_0$ 
begin

lemma invariant-steps:
  list-all  $P\ as$  if steps  $(s_0 \# as)$ 
   $\langle proof \rangle$ 

lemma invariant-reaches:
   $P\ b$  if  $s_0 \rightarrow^* b$ 
   $\langle proof \rangle$ 

```

lemmas *invariant-run* = *invariant-run*[*OF* - *P-s0*]

end

locale *Graph-Invariant-Strong* = *Graph-Defs* +

fixes *P* :: 'a \Rightarrow bool

assumes *invariant*: $a \rightarrow b \Longrightarrow P\ b$

begin

sublocale *inv*: *Graph-Invariant* \langle proof \rangle

lemma *P-invariant-steps*:

list-all *P as* **if** *steps* ($a \# as$)

\langle proof \rangle

lemma *steps-last-invariant*:

P (*last xs*) **if** *steps* ($x \# xs$) $xs \neq []$

\langle proof \rangle

lemmas *invariant-reaches* = *inv.invariant-reaches*

lemma *invariant-reaches1*:

P b **if** $a \rightarrow^+ b$

\langle proof \rangle

end

2.8 Simulations and Bisimulations

locale *Simulation-Defs* =

fixes *A* :: 'a \Rightarrow 'a \Rightarrow bool **and** *B* :: 'b \Rightarrow 'b \Rightarrow bool

and *sim* :: 'a \Rightarrow 'b \Rightarrow bool (**infixr** $\langle \sim \rangle$ 60)

begin

sublocale *A*: *Graph-Defs* *A* \langle proof \rangle

sublocale *B*: *Graph-Defs* *B* \langle proof \rangle

end

locale *Simulation* = *Simulation-Defs* +

assumes *A-B-step*: $\bigwedge a\ b\ a'. A\ a\ b \Longrightarrow a \sim a' \Longrightarrow (\exists b'. B\ a'\ b' \wedge b \sim b')$

begin

lemma *simulation-reaches*:

$\exists b'. B^{**} b b' \wedge a' \sim b' \text{ if } A^{**} a a' a \sim b$
 $\langle \text{proof} \rangle$

lemma *simulation-reaches1*:

$\exists b'. B^{++} b b' \wedge a' \sim b' \text{ if } A^{++} a a' a \sim b$
 $\langle \text{proof} \rangle$

lemma *simulation-steps*:

$\exists bs. B.steps (b \# bs) \wedge list-all2 (\lambda a b. a \sim b) as bs \text{ if } A.steps (a \# as)$
 $a \sim b$
 $\langle \text{proof} \rangle$

lemma *simulation-run*:

$\exists ys. B.run (y \#\# ys) \wedge stream-all2 (\sim) xs ys \text{ if } A.run (x \#\# xs) x \sim y$
 $\langle \text{proof} \rangle$

end

lemma (in *Subgraph*) *Subgraph-Simulation*:

Simulation $E' E (=)$
 $\langle \text{proof} \rangle$

locale *Simulation-Invariant* = *Simulation-Defs* +

fixes $PA :: 'a \Rightarrow bool$ **and** $PB :: 'b \Rightarrow bool$

assumes *A-B-step*: $\bigwedge a b a'. A a b \Longrightarrow PA a \Longrightarrow PB a' \Longrightarrow a \sim a' \Longrightarrow$
 $(\exists b'. B a' b' \wedge b \sim b')$

assumes *A-invariant*[intro]: $\bigwedge a b. PA a \Longrightarrow A a b \Longrightarrow PA b$

assumes *B-invariant*[intro]: $\bigwedge a b. PB a \Longrightarrow B a b \Longrightarrow PB b$

begin

definition *equiv'* $\equiv \lambda a b. a \sim b \wedge PA a \wedge PB b$

sublocale *Simulation* $A B equiv'$ $\langle \text{proof} \rangle$

sublocale *PA-invariant*: *Graph-Invariant* $A PA$ $\langle \text{proof} \rangle$

sublocale *PB-invariant*: *Graph-Invariant* $B PB$ $\langle \text{proof} \rangle$

lemma *simulation-reaches*:

$\exists b'. B^{**} b b' \wedge a' \sim b' \wedge PA a' \wedge PB b' \text{ if } A^{**} a a' a \sim b PA a PB b$
 $\langle \text{proof} \rangle$

lemma *simulation-steps*:

$\exists bs. B.steps (b \# bs) \wedge list-all2 (\lambda a b. a \sim b \wedge PA a \wedge PB b) as bs$
if $A.steps (a \# as) a \sim b PA a PB b$
 $\langle proof \rangle$

lemma *simulation-steps'*:

$\exists bs. B.steps (b \# bs) \wedge list-all2 (\lambda a b. a \sim b) as bs \wedge list-all PA as \wedge$
 $list-all PB bs$
if $A.steps (a \# as) a \sim b PA a PB b$
 $\langle proof \rangle$

context

fixes f

assumes $eq: a \sim b \implies b = f a$

begin

lemma *simulation-steps'-map*:

$\exists bs.$
 $B.steps (b \# bs) \wedge bs = map f as$
 $\wedge list-all2 (\lambda a b. a \sim b) as bs$
 $\wedge list-all PA as \wedge list-all PB bs$
if $A.steps (a \# as) a \sim b PA a PB b$
 $\langle proof \rangle$

end

end

locale *Simulation-Invariants = Simulation-Defs +*

fixes $PA QA :: 'a \Rightarrow bool$ **and** $PB QB :: 'b \Rightarrow bool$

assumes $A\text{-}B\text{-}step: \bigwedge a b a'. A a b \implies PA a \implies PB a' \implies a \sim a' \implies$
 $(\exists b'. B a' b' \wedge b \sim b')$

assumes $A\text{-}invariant[intro]: \bigwedge a b. PA a \implies A a b \implies QA b$

assumes $B\text{-}invariant[intro]: \bigwedge a b. PB a \implies B a b \implies QB b$

assumes $PA\text{-}QA[intro]: \bigwedge a. QA a \implies PA a$ **and** $PB\text{-}QB[intro]: \bigwedge a.$
 $QB a \implies PB a$

begin

sublocale *Pre: Simulation-Invariant A B (\sim) PA PB*

$\langle proof \rangle$

sublocale *Post: Simulation-Invariant A B (\sim) QA QB*

$\langle proof \rangle$

sublocale *A-invs: Graph-Invariants A PA QA*
⟨proof⟩

sublocale *B-invs: Graph-Invariants B PB QB*
⟨proof⟩

lemma *simulation-reaches1:*

$\exists b2. B.reaches1\ b1\ b2 \wedge a2 \sim b2 \wedge QB\ b2$ **if** $A.reaches1\ a1\ a2\ a1 \sim b1$
 $PA\ a1\ PB\ b1$
⟨proof⟩

lemma *reaches1-unique:*

assumes *unique:* $\bigwedge b2. a \sim b2 \implies QB\ b2 \implies b2 = b$
and that: $A.reaches1\ a\ a\ a \sim b\ PA\ a\ PB\ b$
shows $B.reaches1\ b\ b$
⟨proof⟩

end

locale *Bisimulation = Simulation-Defs +*

assumes *A-B-step:* $\bigwedge a\ b\ a'. A\ a\ b \implies a \sim a' \implies (\exists b'. B\ a'\ b' \wedge b \sim b')$

assumes *B-A-step:* $\bigwedge a\ a'\ b'. B\ a'\ b' \implies a \sim a' \implies (\exists b. A\ a\ b \wedge b \sim b')$

begin

sublocale *A-B: Simulation A B (\sim)* ⟨proof⟩

sublocale *B-A: Simulation B A $\lambda x y. y \sim x$* ⟨proof⟩

lemma *A-B-reaches:*

$\exists b'. B^{**}\ b\ b' \wedge a' \sim b'$ **if** $A^{**}\ a\ a'\ a \sim b$
⟨proof⟩

lemma *B-A-reaches:*

$\exists b'. A^{**}\ b\ b' \wedge b' \sim a'$ **if** $B^{**}\ a\ a'\ b \sim a$
⟨proof⟩

end

locale *Bisimulation-Invariant = Simulation-Defs +*

fixes $PA :: 'a \Rightarrow bool$ **and** $PB :: 'b \Rightarrow bool$

assumes *A-B-step:* $\bigwedge a\ b\ a'. A\ a\ b \implies a \sim a' \implies PA\ a \implies PB\ a' \implies$

$(\exists b'. B a' b' \wedge b \sim b')$
assumes *B-A-step*: $\bigwedge a a' b'. B a' b' \implies a \sim a' \implies PA a \implies PB a'$
 $\implies (\exists b. A a b \wedge b \sim b')$
assumes *A-invariant[intro]*: $\bigwedge a b. PA a \implies A a b \implies PA b$
assumes *B-invariant[intro]*: $\bigwedge a b. PB a \implies B a b \implies PB b$
begin

sublocale *PA-invariant*: *Graph-Invariant A PA* $\langle proof \rangle$

sublocale *PB-invariant*: *Graph-Invariant B PB* $\langle proof \rangle$

lemmas *B-steps-invariant[intro]* = *PB-invariant.invariant-reaches*

definition *equiv'* $\equiv \lambda a b. a \sim b \wedge PA a \wedge PB b$

sublocale *bisim*: *Bisimulation A B equiv'*
 $\langle proof \rangle$

sublocale *A-B*: *Simulation-Invariant A B (\sim) PA PB*
 $\langle proof \rangle$

sublocale *B-A*: *Simulation-Invariant B A $\lambda x y. y \sim x$ PB PA*
 $\langle proof \rangle$

context
fixes *f*
assumes *eq*: $a \sim b \iff b = f a$
and *inj*: $\forall a b. PB (f a) \wedge PA b \wedge f a = f b \implies a = b$
begin

lemma *list-all2-inj-map-eq*:
 $as = bs$ **if** *list-all2* $(\lambda a b. a = f b)$ $(map f as)$ *bs* *list-all* *PB* $(map f as)$
list-all *PA* *bs*
 $\langle proof \rangle$

lemma *steps-map-equiv*:
 $A.steps (a \# as) \iff B.steps (b \# map f as)$ **if** $a \sim b$ *PA* *a* *PB* *b*
 $\langle proof \rangle$

lemma *steps-map*:
 $\exists as. bs = map f as$ **if** $B.steps (f a \# bs)$ *PA* *a* *PB* $(f a)$
 $\langle proof \rangle$

lemma *reaches-equiv*:

$A.reaches\ a\ a' \longleftrightarrow B.reaches\ (f\ a)\ (f\ a')$ **if** $PA\ a\ PB\ (f\ a)$
 ⟨proof⟩

end

lemma *equiv'-D*:

$a \sim b$ **if** $A-B.equiv'\ a\ b$
 ⟨proof⟩

lemma *equiv'-rotate-1*:

$B-A.equiv'\ b\ a$ **if** $A-B.equiv'\ a\ b$
 ⟨proof⟩

lemma *equiv'-rotate-2*:

$A-B.equiv'\ a\ b$ **if** $B-A.equiv'\ b\ a$
 ⟨proof⟩

lemma *stream-all2-equiv'-D*:

$stream-all2\ (\sim)\ xs\ ys$ **if** $stream-all2\ A-B.equiv'\ xs\ ys$
 ⟨proof⟩

lemma *stream-all2-equiv'-D2*:

$stream-all2\ B-A.equiv'\ ys\ xs \implies stream-all2\ ((\sim)^{-1-1})\ ys\ xs$
 ⟨proof⟩

lemma *stream-all2-rotate-1*:

$stream-all2\ B-A.equiv'\ ys\ xs \implies stream-all2\ A-B.equiv'\ xs\ ys$
 ⟨proof⟩

lemma *stream-all2-rotate-2*:

$stream-all2\ A-B.equiv'\ xs\ ys \implies stream-all2\ B-A.equiv'\ ys\ xs$
 ⟨proof⟩

end

locale *Bisimulation-Invariants* = *Simulation-Defs* +

fixes $PA\ QA :: 'a \Rightarrow bool$ **and** $PB\ QB :: 'b \Rightarrow bool$

assumes $A-B-step: \bigwedge a\ b\ a'. A\ a\ b \implies a \sim a' \implies PA\ a \implies PB\ a' \implies$
 $(\exists b'. B\ a'\ b' \wedge b \sim b')$

assumes $B-A-step: \bigwedge a\ a'\ b'. B\ a'\ b' \implies a \sim a' \implies PA\ a \implies PB\ a'$
 $\implies (\exists b. A\ a\ b \wedge b \sim b')$

assumes $A-invariant[intro]: \bigwedge a\ b. PA\ a \implies A\ a\ b \implies QA\ b$

assumes $B-invariant[intro]: \bigwedge a\ b. PB\ a \implies B\ a\ b \implies QB\ b$

assumes $PA-QA[intro]: \bigwedge a. QA\ a \implies PA\ a$ **and** $PB-QB[intro]: \bigwedge a.$

$QB\ a \implies PB\ a$
begin

sublocale *PA-invariant: Graph-Invariant A PA* $\langle proof \rangle$

sublocale *PB-invariant: Graph-Invariant B PB* $\langle proof \rangle$

sublocale *QA-invariant: Graph-Invariant A QA* $\langle proof \rangle$

sublocale *QB-invariant: Graph-Invariant B QB* $\langle proof \rangle$

sublocale *Pre-Bisim: Bisimulation-Invariant A B (\sim) PA PB*
 $\langle proof \rangle$

sublocale *Post-Bisim: Bisimulation-Invariant A B (\sim) QA QB*
 $\langle proof \rangle$

sublocale *A-B: Simulation-Invariants A B (\sim) PA QA PB QB*
 $\langle proof \rangle$

sublocale *B-A: Simulation-Invariants B A $\lambda x\ y. y \sim x$ PB QB PA QA*
 $\langle proof \rangle$

context
fixes f
assumes $eq[simp]: a \sim b \iff b = f\ a$
and $inj: \forall a\ b. QB\ (f\ a) \wedge QA\ b \wedge f\ a = f\ b \implies a = b$
begin

lemmas $list-all2-inj-map-eq = Post-Bisim.list-all2-inj-map-eq[OF\ eq\ inj]$
lemmas $steps-map-equiv' = Post-Bisim.steps-map-equiv[OF\ eq\ inj]$

lemma *list-all2-inj-map-eq'*:
 $as = bs$ **if** $list-all2\ (\lambda a\ b. a = f\ b)\ (map\ f\ as)\ bs$ $list-all\ QB\ (map\ f\ as)$
 $list-all\ QA\ bs$
 $\langle proof \rangle$

lemma *steps-map-equiv*:
 $A.steps\ (a \# as) \iff B.steps\ (b \# map\ f\ as)$ **if** $a \sim b$ $PA\ a\ PB\ b$
 $\langle proof \rangle$

lemma *steps-map*:
 $\exists as. bs = map\ f\ as$ **if** $B.steps\ (f\ a \# bs)$ $PA\ a\ PB\ (f\ a)$
 $\langle proof \rangle$

$\llbracket \bigwedge a b. a \sim b = (b = ?f a); \forall a b. QB (?f a) \wedge QA b \wedge ?f a = ?f b \longrightarrow a = b; QA ?a; QB (?f ?a) \rrbracket \Longrightarrow A.reaches ?a ?a' = B.reaches (?f ?a) (?f ?a')$
cannot be lifted directly: injectivity cannot be applied for the reflexive case.

lemma *reaches1-equiv*:

$A.reaches1 a a' \longleftrightarrow B.reaches1 (f a) (f a')$ **if** $PA a PB (f a)$

$\langle proof \rangle$

including *graph-automation-aggressive* $\langle proof \rangle$

including *graph-automation-aggressive* $\langle proof \rangle$

end

end

lemma *Bisimulation-Invariant-composition*:

assumes

Bisimulation-Invariant A B sim1 PA PB

Bisimulation-Invariant B C sim2 PB PC

shows

Bisimulation-Invariant A C $(\lambda a c. \exists b. PB b \wedge sim1 a b \wedge sim2 b c)$

$PA PC$

$\langle proof \rangle$

lemma *Bisimulation-Invariant-filter*:

assumes

Bisimulation-Invariant A B sim PA PB

$\bigwedge a b. sim a b \Longrightarrow PA a \Longrightarrow PB b \Longrightarrow FA a \longleftrightarrow FB b$

$\bigwedge a b. A a b \wedge FA b \longleftrightarrow A' a b$

$\bigwedge a b. B a b \wedge FB b \longleftrightarrow B' a b$

shows

Bisimulation-Invariant A' B' sim PA PB

$\langle proof \rangle$

lemma *Bisimulation-Invariants-filter*:

assumes

Bisimulation-Invariants A B sim PA QA PB QB

$\bigwedge a b. QA a \Longrightarrow QB b \Longrightarrow FA a \longleftrightarrow FB b$

$\bigwedge a b. A a b \wedge FA b \longleftrightarrow A' a b$

$\bigwedge a b. B a b \wedge FB b \longleftrightarrow B' a b$

shows

Bisimulation-Invariants A' B' sim PA QA PB QB

$\langle proof \rangle$

lemma *Bisimulation-Invariants-composition*:

assumes

Bisimulation-Invariants $A B \text{ sim1 } PA QA PB QB$

Bisimulation-Invariants $B C \text{ sim2 } PB QB PC QC$

shows

Bisimulation-Invariants $A C (\lambda a c. \exists b. PB b \wedge \text{sim1 } a b \wedge \text{sim2 } b c)$
 $PA QA PC QC$

$\langle \text{proof} \rangle$

lemma *Bisimulation-Invariant-Invariants-composition:*

assumes

Bisimulation-Invariant $A B \text{ sim1 } PA PB$

Bisimulation-Invariants $B C \text{ sim2 } PB QB PC QC$

shows

Bisimulation-Invariants $A C (\lambda a c. \exists b. PB b \wedge \text{sim1 } a b \wedge \text{sim2 } b c)$
 $PA PA PC QC$

$\langle \text{proof} \rangle$

lemma *Bisimulation-Invariant-Bisimulation-Invariants:*

assumes *Bisimulation-Invariant* $A B \text{ sim } PA PB$

shows *Bisimulation-Invariants* $A B \text{ sim } PA PA PB PB$

$\langle \text{proof} \rangle$

lemma *Bisimulation-Invariant-strengthen-post:*

assumes

Bisimulation-Invariant $A B \text{ sim } PA PB$

$\bigwedge a b. PA' a \implies PA b \implies A a b \implies PA' b$

$\bigwedge a. PA' a \implies PA a$

shows *Bisimulation-Invariant* $A B \text{ sim } PA' PB$

$\langle \text{proof} \rangle$

lemma *Bisimulation-Invariant-strengthen-post':*

assumes

Bisimulation-Invariant $A B \text{ sim } PA PB$

$\bigwedge a b. PB' a \implies PB b \implies B a b \implies PB' b$

$\bigwedge a. PB' a \implies PB a$

shows *Bisimulation-Invariant* $A B \text{ sim } PA PB'$

$\langle \text{proof} \rangle$

lemma *Simulation-Invariant-strengthen-post:*

assumes

Simulation-Invariant $A B \text{ sim } PA PB$

$\bigwedge a b. PA a \implies PA b \implies A a b \implies PA' b$

$\bigwedge a. PA' a \implies PA a$

shows *Simulation-Invariant* $A B \text{ sim } PA' PB$

$\langle proof \rangle$

lemma *Simulation-Invariant-strengthen-post'*:

assumes

Simulation-Invariant A B sim PA PB

$\bigwedge a b. PB a \implies PB b \implies B a b \implies PB' b$

$\bigwedge a. PB' a \implies PB a$

shows *Simulation-Invariant A B sim PA PB'*

$\langle proof \rangle$

lemma *Simulation-Invariants-strengthen-post*:

assumes

Simulation-Invariants A B sim PA QA PB QB

$\bigwedge a b. PA a \implies QA b \implies A a b \implies QA' b$

$\bigwedge a. QA' a \implies QA a$

shows *Simulation-Invariants A B sim PA QA' PB QB*

$\langle proof \rangle$

lemma *Simulation-Invariants-strengthen-post'*:

assumes

Simulation-Invariants A B sim PA QA PB QB

$\bigwedge a b. PB a \implies QB b \implies B a b \implies QB' b$

$\bigwedge a. QB' a \implies QB a$

shows *Simulation-Invariants A B sim PA QA PB QB'*

$\langle proof \rangle$

lemma *Bisimulation-Invariant-sim-replace*:

assumes *Bisimulation-Invariant A B sim PA PB*

and $\bigwedge a b. PA a \implies PB b \implies sim a b \longleftrightarrow sim' a b$

shows *Bisimulation-Invariant A B sim' PA PB*

$\langle proof \rangle$

end

2.9 CTL

theory *CTL*

imports *Graphs*

begin

lemmas [*simp*] = *holds.simps*

context *Graph-Defs*

begin

definition

$$Alw\text{-}ev\ \varphi\ x \equiv \forall\ xs.\ run\ (x\ \#\#\ xs) \longrightarrow ev\ (holds\ \varphi)\ (x\ \#\#\ xs)$$
definition

$$Alw\text{-}alw\ \varphi\ x \equiv \forall\ xs.\ run\ (x\ \#\#\ xs) \longrightarrow alw\ (holds\ \varphi)\ (x\ \#\#\ xs)$$
definition

$$Ex\text{-}ev\ \varphi\ x \equiv \exists\ xs.\ run\ (x\ \#\#\ xs) \wedge ev\ (holds\ \varphi)\ (x\ \#\#\ xs)$$
definition

$$Ex\text{-}alw\ \varphi\ x \equiv \exists\ xs.\ run\ (x\ \#\#\ xs) \wedge alw\ (holds\ \varphi)\ (x\ \#\#\ xs)$$
definition

$$leadsto\ \varphi\ \psi\ x \equiv Alw\text{-}alw\ (\lambda\ x.\ \varphi\ x \longrightarrow Alw\text{-}ev\ \psi\ x)\ x$$
definition

$$deadlocked\ x \equiv \neg (\exists\ y.\ x \rightarrow y)$$
definition

$$deadlock\ x \equiv \exists\ y.\ reaches\ x\ y \wedge deadlocked\ y$$
lemma *no-deadlockD*:
$$\neg\ deadlocked\ y\ \mathbf{if}\ \neg\ deadlock\ x\ reaches\ x\ y$$

<proof>

lemma *not-deadlockedE*:

assumes $\neg\ deadlocked\ x$
obtains y **where** $x \rightarrow y$

<proof>

lemma *holds-Not*:
$$holds\ (Not\ o\ \varphi) = (\lambda\ x.\ \neg\ holds\ \varphi\ x)$$

<proof>

lemma *Alw-alw-iff*:
$$Alw\text{-}alw\ \varphi\ x \longleftrightarrow \neg\ Ex\text{-}ev\ (Not\ o\ \varphi)\ x$$

<proof>

lemma *Ex-alw-iff*:
$$Ex\text{-}alw\ \varphi\ x \longleftrightarrow \neg\ Alw\text{-}ev\ (Not\ o\ \varphi)\ x$$

<proof>

lemma *leadsto-iff*:

$leadsto \varphi \psi x \longleftrightarrow \neg Ex\text{-}ev (\lambda x. \varphi x \wedge \neg Alw\text{-}ev \psi x) x$
 ⟨proof⟩

lemma *run-siterate-from*:

assumes $\forall y. x \rightarrow^* y \longrightarrow (\exists z. y \rightarrow z)$

shows $run (siterate (\lambda x. SOME y. x \rightarrow y) x) (\mathbf{is} run (siterate ?f x))$

⟨proof⟩ **including** *graph-automation-aggressive* ⟨proof⟩

lemma *extend-run'*:

$run\ zs \mathbf{if}\ steps\ xs\ run\ ys\ last\ xs = shd\ ys\ xs @- stl\ ys = zs$

⟨proof⟩

lemma *no-deadlock-run-extend*:

$\exists ys. run (x \#\# xs @- ys) \mathbf{if} \neg deadlock\ x\ steps (x \# xs)$

⟨proof⟩

include *graph-automation*

⟨proof⟩

lemma *Ex-ev*:

$Ex\text{-}ev \varphi x \longleftrightarrow (\exists y. x \rightarrow^* y \wedge \varphi y) \mathbf{if} \neg deadlock\ x$

⟨proof⟩

including *graph-automation* ⟨proof⟩

lemma *Alw-ev*:

$Alw\text{-}ev \varphi x \longleftrightarrow \neg (\exists xs. run (x \#\# xs) \wedge alw (holds (Not o \varphi)) (x \#\# xs))$

⟨proof⟩

lemma *leadsto-iff'*:

$leadsto \varphi \psi x \longleftrightarrow (\nexists y. x \rightarrow^* y \wedge \varphi y \wedge \neg Alw\text{-}ev \psi y) \mathbf{if} \neg deadlock\ x$

⟨proof⟩

end

context *Bisimulation-Invariant*

begin

context

fixes $\varphi :: 'a \Rightarrow bool$ **and** $\psi :: 'b \Rightarrow bool$

assumes *compatible*: $A\text{-}B.\text{equiv}' a\ b \Longrightarrow \varphi a \longleftrightarrow \psi b$

begin

lemma *ev-ψ-φ*:

ev (holds φ) xs if stream-all2 B-A.equiv' ys xs ev (holds ψ) ys
<proof>

lemma *ev-φ-ψ*:

ev (holds ψ) ys if stream-all2 A-B.equiv' xs ys ev (holds φ) xs
<proof>

lemma *Ex-ev-iff*:

A.Ex-ev φ a ↔ B.Ex-ev ψ b if A-B.equiv' a b
<proof>

lemma *Alw-ev-iff*:

A.Alw-ev φ a ↔ B.Alw-ev ψ b if A-B.equiv' a b
<proof>

end

context

fixes $\varphi :: 'a \Rightarrow \text{bool}$ **and** $\psi :: 'b \Rightarrow \text{bool}$

assumes *compatible1*: $A-B.equiv' a b \implies \varphi a \longleftrightarrow \psi b$

begin

lemma *Alw-alw-iff-strong*:

A.Alw-alw φ a ↔ B.Alw-alw ψ b if A-B.equiv' a b
<proof>

lemma *Ex-alw-iff*:

A.Ex-alw φ a ↔ B.Ex-alw ψ b if A-B.equiv' a b
<proof>

end

context

fixes $\varphi :: 'a \Rightarrow \text{bool}$ **and** $\psi :: 'b \Rightarrow \text{bool}$

and $\varphi' :: 'a \Rightarrow \text{bool}$ **and** $\psi' :: 'b \Rightarrow \text{bool}$

assumes *compatible1*: $A-B.equiv' a b \implies \varphi a \longleftrightarrow \psi b$

assumes *compatible2*: $A-B.equiv' a b \implies \varphi' a \longleftrightarrow \psi' b$

begin

lemma *Leadsto-iff*:

A.leadsto φ φ' a ↔ B.leadsto ψ ψ' b if A-B.equiv' a b
<proof>

end

lemma *deadlock-iff*:

$A.\text{deadlock } a \longleftrightarrow B.\text{deadlock } b$ **if** $a \sim b$ *PA a PB b*
<proof>

end

lemmas [*simp del*] = *holds.simps*

end

theory *Timed-Automata*

imports *library/Graphs Difference-Bound-Matrices.Zones*

begin

3 Basic Definitions and Semantics

3.1 Syntactic Definition

Clock constraints

datatype (*'c, 't*) *acconstraint* =
 LT 'c 't |
 LE 'c 't |
 EQ 'c 't |
 GT 'c 't |
 GE 'c 't

type-synonym (*'c, 't*) *cconstraint* = (*'c, 't*) *acconstraint list*

For an informal description of timed automata we refer to Bengtsson and Yi [BY03]. We define a timed automaton A

type-synonym

(*'c, 'time, 's*) *invassn* = *'s* \Rightarrow (*'c, 'time*) *cconstraint*

type-synonym

(*'a, 'c, 'time, 's*) *transition* = *'s* * (*'c, 'time*) *cconstraint* * *'a* * *'c list* * *'s*

type-synonym

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

definition *trans-of* :: (*'a, 'c, 'time, 's*) *ta* \Rightarrow (*'a, 'c, 'time, 's*) *transition set*
where

$trans\text{-of} \equiv fst$

definition $inv\text{-of} :: ('a, 'c, 'time, 's) ta \Rightarrow ('c, 'time, 's) invassn$ **where**
 $inv\text{-of} \equiv snd$

abbreviation $transition ::$

$('a, 'c, 'time, 's) ta \Rightarrow 's \Rightarrow ('c, 'time) cconstraint \Rightarrow 'a \Rightarrow 'c list \Rightarrow 's \Rightarrow$
 $bool$

$(\langle - \vdash - \rangle \longrightarrow \langle \cdot, \cdot \rangle \rightarrow [61,61,61,61,61,61] 61)$ **where**

$(A \vdash l \longrightarrow^{g,a,r} l') \equiv (l,g,a,r,l') \in trans\text{-of } A$

3.1.1 Collecting Information About Clocks

fun $constraint\text{-clk} :: ('c, 't) acconstraint \Rightarrow 'c$

where

$constraint\text{-clk} (LT\ c\ -) = c \mid$

$constraint\text{-clk} (LE\ c\ -) = c \mid$

$constraint\text{-clk} (EQ\ c\ -) = c \mid$

$constraint\text{-clk} (GE\ c\ -) = c \mid$

$constraint\text{-clk} (GT\ c\ -) = c$

definition $collect\text{-clks} :: ('c, 't) cconstraint \Rightarrow 'c\ set$

where

$collect\text{-clks}\ cc \equiv constraint\text{-clk}\ 'set\ cc$

fun $constraint\text{-pair} :: ('c, 't) acconstraint \Rightarrow ('c * 't)$

where

$constraint\text{-pair} (LT\ x\ m) = (x, m) \mid$

$constraint\text{-pair} (LE\ x\ m) = (x, m) \mid$

$constraint\text{-pair} (EQ\ x\ m) = (x, m) \mid$

$constraint\text{-pair} (GE\ x\ m) = (x, m) \mid$

$constraint\text{-pair} (GT\ x\ m) = (x, m)$

definition $collect\text{-clock-pairs} :: ('c, 't) cconstraint \Rightarrow ('c * 't)\ set$

where

$collect\text{-clock-pairs}\ cc = constraint\text{-pair}\ 'set\ cc$

definition $collect\text{-clkt} :: ('a, 'c, 't, 's) transition\ set \Rightarrow ('c * 't)\ set$

where

$collect\text{-clkt}\ S = \bigcup \{ collect\text{-clock-pairs}\ (fst\ (snd\ t)) \mid t . t \in S \}$

definition $collect\text{-clki} :: ('c, 't, 's) invassn \Rightarrow ('c * 't)\ set$

where

$collect\text{-clki}\ I = \bigcup \{ collect\text{-clock-pairs}\ (I\ x) \mid x. True \}$

definition $clkp\text{-set} :: ('a, 'c, 't, 's) ta \Rightarrow ('c * 't) \text{ set}$

where

$$clkp\text{-set } A = \text{collect-clki } (\text{inv-of } A) \cup \text{collect-clkt } (\text{trans-of } A)$$

definition $\text{collect-clkvt} :: ('a, 'c, 't, 's) \text{ transition set} \Rightarrow 'c \text{ set}$

where

$$\text{collect-clkvt } S = \bigcup \{ \text{set } ((fst \circ snd \circ snd \circ snd) t) \mid t . t \in S \}$$

abbreviation $clk\text{-set}$ **where** $clk\text{-set } A \equiv \text{fst } ' clkp\text{-set } A \cup \text{collect-clkvt } (\text{trans-of } A)$

inductive valid-abstraction

where

$$\begin{aligned} & \llbracket \forall (x, m) \in clkp\text{-set } A. m \leq k \ x \wedge x \in X \wedge m \in \mathbb{N}; \text{collect-clkvt } (\text{trans-of } A) \subseteq X; \text{finite } X \rrbracket \\ & \implies \text{valid-abstraction } A \ X \ k \end{aligned}$$

3.2 Operational Semantics

inductive $\text{clock-val-a } (\lrcorner \vdash_a \rightarrow [62, 62] \ 62)$ **where**

$$\begin{aligned} \llbracket u \ c < \ d \rrbracket & \implies u \vdash_a \text{LT } c \ d \mid \\ \llbracket u \ c \leq \ d \rrbracket & \implies u \vdash_a \text{LE } c \ d \mid \\ \llbracket u \ c = \ d \rrbracket & \implies u \vdash_a \text{EQ } c \ d \mid \\ \llbracket u \ c \geq \ d \rrbracket & \implies u \vdash_a \text{GE } c \ d \mid \\ \llbracket u \ c > \ d \rrbracket & \implies u \vdash_a \text{GT } c \ d \end{aligned}$$

inductive-cases $[\text{elim!}]$: $u \vdash_a \text{LT } c \ d$

inductive-cases $[\text{elim!}]$: $u \vdash_a \text{LE } c \ d$

inductive-cases $[\text{elim!}]$: $u \vdash_a \text{EQ } c \ d$

inductive-cases $[\text{elim!}]$: $u \vdash_a \text{GE } c \ d$

inductive-cases $[\text{elim!}]$: $u \vdash_a \text{GT } c \ d$

declare $\text{clock-val-a.intros}[\text{intro}]$

definition $\text{clock-val} :: ('c, 't) \text{ cval} \Rightarrow ('c, 't::\text{time}) \text{ cconstraint} \Rightarrow \text{bool } (\lrcorner \vdash \rightarrow [62, 62] \ 62)$

where

$$u \vdash cc = \text{list-all } (\text{clock-val-a } u) \ cc$$

lemma $\text{atomic-guard-continuous}$:

assumes $u \vdash_a \ g \ u \oplus t \vdash_a \ g \ 0 \leq (t'::'t::\text{time}) \ t' \leq t$

shows $u \oplus t' \vdash_a \ g$

$\langle \text{proof} \rangle$

lemma *guard-continuous*:

assumes $u \vdash g \ u \oplus t \vdash g \ 0 \leq t' \ t' \leq t$

shows $u \oplus t' \vdash g$

<proof>

inductive *step-t* ::

$(\ 'a, \ 'c, \ 't, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ cval \Rightarrow \ ('t::time) \Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ cval \Rightarrow \ bool$

$(\ \langle - \vdash \langle -, - \rangle \rightarrow \langle -, - \rangle \rangle \ [61,61,61] \ 61)$

where

$\llbracket u \oplus d \vdash \text{inv-of } A \ l; \ d \geq 0 \rrbracket \Longrightarrow A \vdash \langle l, u \rangle \rightarrow^d \langle l, u \oplus d \rangle$

lemmas $[intro] = \text{step-t.intros}$

context

notes *step-t.cases*[*elim!*] *step-t.intros*[*intro!*]

begin

lemma *step-t-determinacy1*:

$A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow A \vdash \langle l, u \rangle \rightarrow^d \langle l'', u'' \rangle \Longrightarrow l' = l''$
<proof>

lemma *step-t-determinacy2*:

$A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow A \vdash \langle l, u \rangle \rightarrow^d \langle l'', u'' \rangle \Longrightarrow u' = u''$
<proof>

lemma *step-t-cont1*:

$d \geq 0 \Longrightarrow e \geq 0 \Longrightarrow A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow A \vdash \langle l', u' \rangle \rightarrow^e \langle l'', u'' \rangle$
 $\Longrightarrow A \vdash \langle l, u \rangle \rightarrow^{d+e} \langle l'', u'' \rangle$
<proof>

end

inductive *step-a* ::

$(\ 'a, \ 'c, \ 't, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ ('t::time)) \ cval \Rightarrow \ 'a \Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ cval \Rightarrow \ bool$

$(\ \langle - \vdash \langle -, - \rangle \rightarrow \langle -, - \rangle \rangle \ [61,61,61] \ 61)$

where

$\llbracket A \vdash l \xrightarrow{g,a,r} l'; \ u \vdash g; \ u' \vdash \text{inv-of } A \ l'; \ u' = [r \rightarrow 0]u \rrbracket \Longrightarrow (A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle)$

inductive *step* ::

$(\langle a, 'c, 't, 's \rangle ta \Rightarrow 's \Rightarrow ('c, ('t::time)) cval \Rightarrow 's \Rightarrow ('c, 't) cval \Rightarrow bool$
 $(\langle - \vdash \langle -, - \rangle \rightarrow \langle -, - \rangle \rangle [61,61,61] 61)$

where

$step\text{-}a: A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle \Longrightarrow (A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle) \mid$
 $step\text{-}t: A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow (A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle)$

declare $step.intros[intro]$

declare $step.cases[elim]$

inductive

$steps :: (\langle a, 'c, 't, 's \rangle ta \Rightarrow 's \Rightarrow ('c, ('t::time)) cval \Rightarrow 's \Rightarrow ('c, 't) cval$
 $\Rightarrow bool$
 $(\langle - \vdash \langle -, - \rangle \rightarrow * \langle -, - \rangle \rangle [61,61,61] 61)$

where

$refl: A \vdash \langle l, u \rangle \rightarrow * \langle l, u \rangle \mid$
 $step: A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \Longrightarrow A \vdash \langle l', u' \rangle \rightarrow * \langle l'', u'' \rangle \Longrightarrow A \vdash \langle l, u \rangle \rightarrow * \langle l'', u'' \rangle$

declare $steps.intros[intro]$

3.3 Contracting Runs

inductive $step' ::$

$(\langle a, 'c, 't, 's \rangle ta \Rightarrow 's \Rightarrow ('c, ('t::time)) cval \Rightarrow 's \Rightarrow ('c, 't) cval \Rightarrow bool$
 $(\langle - \vdash'' \langle -, - \rangle \rightarrow \langle -, - \rangle \rangle [61,61,61] 61)$

where

$step': A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow A \vdash \langle l', u' \rangle \rightarrow_a \langle l'', u'' \rangle \Longrightarrow A \vdash' \langle l, u \rangle$
 $\rightarrow \langle l'', u'' \rangle$

lemmas $step'[intro]$

lemma $step'\text{-}altI:$

assumes

$A \vdash l \longrightarrow^{g,a,r} l' u \oplus d \vdash g u \oplus d \vdash \text{inv-of } A \ l \ 0 \leq d$
 $u' = [r \rightarrow 0](u \oplus d) \ u' \vdash \text{inv-of } A \ l'$

shows $A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$

$\langle proof \rangle$

inductive

$steps' :: (\langle a, 'c, 't, 's \rangle ta \Rightarrow 's \Rightarrow ('c, ('t::time)) cval \Rightarrow 's \Rightarrow ('c, 't) cval$
 $\Rightarrow bool$
 $(\langle - \vdash'' \langle -, - \rangle \rightarrow * \langle -, - \rangle \rangle [61,61,61] 61)$

where

$refl': A \vdash' \langle l, u \rangle \rightarrow * \langle l, u \rangle \mid$

$step': A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle \implies A \vdash' \langle l', u' \rangle \rightarrow^* \langle l'', u'' \rangle \implies A \vdash' \langle l, u \rangle \rightarrow^* \langle l'', u'' \rangle$

lemmas $steps'.intros[intro]$

lemma $steps'-altI$:

$A \vdash' \langle l, u \rangle \rightarrow^* \langle l'', u'' \rangle$ **if** $A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$ $A \vdash' \langle l', u' \rangle \rightarrow \langle l'', u'' \rangle$
 $\langle proof \rangle$

lemma $step-d-refl[intro]$:

$A \vdash \langle l, u \rangle \rightarrow^0 \langle l, u \rangle$ **if** $u \vdash inv\text{-of } A \ l$
 $\langle proof \rangle$

lemma $cval\text{-add}\text{-simp}$:

$(u \oplus d) \oplus d' = u \oplus (d + d')$ **for** $d \ d' :: 't :: time$
 $\langle proof \rangle$

context

notes $[elim!]$ = $step'.cases \ step\text{-}t.cases$

and $[intro!]$ = $step\text{-}t.intros$

begin

lemma $step\text{-}t\text{-}trans$:

$A \vdash \langle l, u \rangle \rightarrow^{d+d'} \langle l, u' \rangle$ **if** $A \vdash \langle l, u \rangle \rightarrow^d \langle l, u' \rangle$ $A \vdash \langle l, u' \rangle \rightarrow^{d'} \langle l, u' \rangle$
 $\langle proof \rangle$

lemma $steps'\text{-}complete$:

$\exists u'. A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$ **if** $A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$ $u \vdash inv\text{-of } A \ l$
 $\langle proof \rangle$

lemma $steps'\text{-}sound$:

$A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$ **if** $A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$
 $\langle proof \rangle$

lemma $steps\text{-}steps'\text{-}equiv$:

$(\exists u'. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle) \longleftrightarrow (\exists u'. A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle)$ **if** $u \vdash inv\text{-of } A \ l$
 $\langle proof \rangle$

end

3.4 Zone Semantics

datatype $'a \ action = Tau \ (\langle \tau \rangle) \mid Action \ 'a \ (\langle ! \rightarrow \rangle)$

inductive *step-z* ::

$(\text{'a}, \text{'c}, \text{'t}, \text{'s}) \text{ta} \Rightarrow \text{'s} \Rightarrow (\text{'c}, (\text{'t}::\text{time})) \text{zone} \Rightarrow \text{'a} \text{action} \Rightarrow \text{'s} \Rightarrow (\text{'c}, \text{'t}) \text{zone} \Rightarrow \text{bool}$
 $(\langle \cdot \vdash \langle \cdot, \cdot \rangle \rightsquigarrow_{\cdot} \langle \cdot, \cdot \rangle) [61,61,61,61] 61)$

where

step-t-z:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l, Z^{\uparrow} \cap \{u. u \vdash \text{inv-of } A \ l\} \rangle \mid$

step-a-z:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{1a} \langle l', \text{zone-set } (Z \cap \{u. u \vdash g\}) \ r \cap \{u. u \vdash \text{inv-of } A \ l'\} \rangle$
if $A \vdash l \longrightarrow^{g,a,r} l'$

lemmas *step-z.intros*[*intro*]

inductive-cases *step-t-z-E*[*elim*]: $A \vdash \langle l, u \rangle \rightsquigarrow_{\tau} \langle l', u' \rangle$

inductive-cases *step-a-z-E*[*elim*]: $A \vdash \langle l, u \rangle \rightsquigarrow_{1a} \langle l', u' \rangle$

3.4.1 Zone Semantics for Compressed Runs

definition

step-z' :: $(\text{'a}, \text{'c}, \text{'t}, \text{'s}) \text{ta} \Rightarrow \text{'s} \Rightarrow (\text{'c}, (\text{'t}::\text{time})) \text{zone} \Rightarrow \text{'s} \Rightarrow (\text{'c}, \text{'t}) \text{zone} \Rightarrow \text{bool}$
 $(\langle \cdot \vdash \langle \cdot, \cdot \rangle \rightsquigarrow \langle \cdot, \cdot \rangle) [61,61,61] 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z'' \rangle \equiv (\exists Z' a. A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l, Z' \rangle \wedge A \vdash \langle l, Z' \rangle \rightsquigarrow_{1a} \langle l', Z'' \rangle)$

abbreviation

steps-z :: $(\text{'a}, \text{'c}, \text{'t}, \text{'s}) \text{ta} \Rightarrow \text{'s} \Rightarrow (\text{'c}, (\text{'t}::\text{time})) \text{zone} \Rightarrow \text{'s} \Rightarrow (\text{'c}, \text{'t}) \text{zone} \Rightarrow \text{bool}$
 $(\langle \cdot \vdash \langle \cdot, \cdot \rangle \rightsquigarrow^* \langle \cdot, \cdot \rangle) [61,61,61] 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow^* \langle l', Z'' \rangle \equiv (\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z'' \rangle)^{**} (l, Z) (l', Z'')$

context

notes [*elim!*] = *step.cases step'.cases step-t.cases step-z.cases*

begin

lemma *step-t-z-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l', Z' \rangle \implies \forall u' \in Z'. \exists u \in Z. \exists d. A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$
<proof>

lemma *step-a-z-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{1a} \langle l', Z' \rangle \implies \forall u' \in Z'. \exists u \in Z. \exists d. A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$

$\langle \text{proof} \rangle$

lemma *step-z-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies \forall u' \in Z'. \exists u \in Z. A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-a-z-complete*:

$A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle \implies u \in Z \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow_{1a} \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *step-t-z-complete*:

$A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \implies u \in Z \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow_\tau \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *step-z-complete*:

$A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \implies u \in Z \implies \exists Z' a. A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

end

lemma *step-z-sound'*:

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \implies \forall u' \in Z'. \exists u \in Z. A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-z-complete'*:

$A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle \implies u \in Z \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *steps-z-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z' \rangle \implies u' \in Z' \implies \exists u \in Z. A \vdash' \langle l, u \rangle \rightarrow_* \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *steps-z-complete*:

$A \vdash' \langle l, u \rangle \rightarrow_* \langle l', u' \rangle \implies u \in Z \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *ta-zone-sim*:

Simulation

$(\lambda(l, u) (l', u')). A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$

$(\lambda(l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z'' \rangle)$
 $(\lambda(l, u) (l', Z). u \in Z \wedge l = l')$
 $\langle proof \rangle$

lemma *steps'-iff*:

$(\lambda(l, u) (l', u'). A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle)^{**} (l, u) (l', u') \longleftrightarrow A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$
 $\langle proof \rangle$

lemma *steps-z-complete*:

$A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle \implies u \in Z \implies \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow^* \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle proof \rangle$

end

3.5 From Clock Constraints to DBMs

theory *TA-DBM-Operations*

imports *Timed-Automata Difference-Bound-Matrices.DBM-Operations*
begin

fun *abstra* ::

$(\text{'c}, \text{'t}::\{\text{linordered-cancel-ab-monoid-add, uminus}\}) \text{acconstraint} \Rightarrow \text{'t DBM}$
 $\Rightarrow (\text{'c} \Rightarrow \text{nat}) \Rightarrow \text{'t DBM}$

where

$\text{abstra } (EQ \ c \ d) \ M \ v =$
 $(\lambda \ i \ j . \text{if } i = 0 \wedge j = v \ c \ \text{then } \min (M \ i \ j) (Le \ (-d)) \ \text{else if } i = v \ c \wedge j = 0 \ \text{then } \min (M \ i \ j) (Le \ d) \ \text{else } M \ i \ j) \ |$
 $\text{abstra } (LT \ c \ d) \ M \ v =$
 $(\lambda \ i \ j . \text{if } i = v \ c \wedge j = 0 \ \text{then } \min (M \ i \ j) (Lt \ d) \ \text{else } M \ i \ j) \ |$
 $\text{abstra } (LE \ c \ d) \ M \ v =$
 $(\lambda \ i \ j . \text{if } i = v \ c \wedge j = 0 \ \text{then } \min (M \ i \ j) (Le \ d) \ \text{else } M \ i \ j) \ |$
 $\text{abstra } (GT \ c \ d) \ M \ v =$
 $(\lambda \ i \ j . \text{if } i = 0 \wedge j = v \ c \ \text{then } \min (M \ i \ j) (Lt \ (-d)) \ \text{else } M \ i \ j) \ |$
 $\text{abstra } (GE \ c \ d) \ M \ v =$
 $(\lambda \ i \ j . \text{if } i = 0 \wedge j = v \ c \ \text{then } \min (M \ i \ j) (Le \ (-d)) \ \text{else } M \ i \ j)$

fun *abstr* :: $(\text{'c}, \text{'t}::\{\text{linordered-cancel-ab-monoid-add, uminus}\}) \text{cconstraint} \Rightarrow \text{'t DBM} \Rightarrow (\text{'c} \Rightarrow \text{nat}) \Rightarrow \text{'t DBM}$

where

$\text{abstr } cc \ M \ v = \text{fold } (\lambda \ ac \ M . \text{abstra } ac \ M \ v) \ cc \ M$

lemma *collect-clks-Cons[simp]*:

$collect-clks (ac \# cc) = insert (constraint-clk ac) (collect-clks cc)$
 $\langle proof \rangle$

lemma *abstr-id1*:

$c \notin collect-clks cc \implies clock-numbering' v n \implies \forall c \in collect-clks cc. v c \leq n$
 $\implies abstr cc M v 0 (v c) = M 0 (v c)$
 $\langle proof \rangle$

lemma *abstr-id2*:

$c \notin collect-clks cc \implies clock-numbering' v n \implies \forall c \in collect-clks cc. v c \leq n$
 $\implies abstr cc M v (v c) 0 = M (v c) 0$
 $\langle proof \rangle$

This lemma is trivial because we constrained our theory to difference constraints.

lemma *abstra-id3*:

assumes *clock-numbering v*
shows $abstra ac M v (v c1) (v c2) = M (v c1) (v c2)$
 $\langle proof \rangle$

lemma *abstr-id3*:

$clock-numbering v \implies abstr cc M v (v c1) (v c2) = M (v c1) (v c2)$
 $\langle proof \rangle$

lemma *abstra-id3'*:

assumes $\forall c. 0 < v c$
shows $abstra ac M v 0 0 = M 0 0$
 $\langle proof \rangle$

lemma *abstr-id3'*:

$clock-numbering v \implies abstr cc M v 0 0 = M 0 0$
 $\langle proof \rangle$

lemma *clock-numberingD*:

assumes *clock-numbering v v c = 0*
shows A
 $\langle proof \rangle$

lemma *dbm-abstra-soundness*:

$\llbracket u \vdash_a ac; u \vdash_{v,n} M; clock-numbering' v n; v (constraint-clk ac) \leq n \rrbracket$

\implies *DBM-val-bounded* v u (*abstra ac* M v) n
 \langle *proof* \rangle

lemma *dbm-abstr-soundness'*:

$\llbracket u \vdash cc; u \vdash_{v,n} M; \text{clock-numbering}' v n; \forall c \in \text{collect-clks } cc. v c \leq n \rrbracket$
 \implies *DBM-val-bounded* v u (*abstr cc* M v) n
 \langle *proof* \rangle

lemmas *dbm-abstr-soundness* = *dbm-abstr-soundness'*[*OF - DBM-triv*]

lemma *dbm-abstra-completeness*:

$\llbracket \text{DBM-val-bounded } v u (\text{abstra ac } M v) n; \forall c. v c > 0; v (\text{constraint-clk } ac) \leq n \rrbracket$
 $\implies u \vdash_a ac$
 \langle *proof* \rangle

lemma *abstra-mono*:

abstra ac M v i $j \leq M$ i j
 \langle *proof* \rangle

lemma *abstra-subset*:

$[\text{abstra ac } M v]_{v,n} \subseteq [M]_{v,n}$
 \langle *proof* \rangle

lemma *abstr-subset*:

$[\text{abstr cc } M v]_{v,n} \subseteq [M]_{v,n}$
 \langle *proof* \rangle

lemma *dbm-abstra-zone-eq*:

assumes *clock-numbering'* v n v (*constraint-clk* ac) $\leq n$
shows $[\text{abstra ac } M v]_{v,n} = \{u. u \vdash_a ac\} \cap [M]_{v,n}$
 \langle *proof* \rangle

lemma [*simp*]:

$u \vdash \square$
 \langle *proof* \rangle

lemma *clock-val-Cons*:

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

lemma *abstra-commute*:

$abstra\ ac1\ (abstra\ ac2\ M\ v)\ v = abstra\ ac2\ (abstra\ ac1\ M\ v)\ v$
 $\langle proof \rangle$

lemma *dbm-abstr-completeness-aux*:

$\llbracket DBM\text{-val-bounded}\ v\ u\ (abstr\ cc\ (abstra\ ac\ M\ v)\ v)\ n; \forall c. v\ c > 0; v$
 $(constraint\text{-clk}\ ac) \leq n \rrbracket$
 $\implies u \vdash_a ac$
 $\langle proof \rangle$

lemma *dbm-abstr-completeness*:

$\llbracket DBM\text{-val-bounded}\ v\ u\ (abstr\ cc\ M\ v)\ n; \forall c. v\ c > 0; \forall c \in collect\text{-clks}$
 $cc. v\ c \leq n \rrbracket$
 $\implies u \vdash cc$
 $\langle proof \rangle$

lemma *dbm-abstr-zone-eq*:

assumes $clock\text{-numbering}'\ v\ n\ \forall c \in collect\text{-clks}\ cc. v\ c \leq n$
shows $[abstr\ cc\ (\lambda i\ j. \infty)\ v]_{v,n} = \{u. u \vdash cc\}$
 $\langle proof \rangle$

lemma *dbm-abstr-zone-eq2*:

assumes $clock\text{-numbering}'\ v\ n\ \forall c \in collect\text{-clks}\ cc. v\ c \leq n$
shows $[abstr\ cc\ M\ v]_{v,n} = [M]_{v,n} \cap \{u. u \vdash cc\}$
 $\langle proof \rangle$

abbreviation *global-clock-numbering* ::

$('a, 'c, 't, 's)\ ta \Rightarrow ('c \Rightarrow nat) \Rightarrow nat \Rightarrow bool$

where

$global\text{-clock-numbering}\ A\ v\ n \equiv$
 $clock\text{-numbering}'\ v\ n \wedge (\forall c \in clk\text{-set}\ A. v\ c \leq n) \wedge (\forall k \leq n. k > 0 \longrightarrow$
 $(\exists c. v\ c = k))$

lemma *dbm-int-all-abstra*:

assumes $dbm\text{-int-all}\ M\ snd\ (constraint\text{-pair}\ ac) \in \mathbb{Z}$
shows $dbm\text{-int-all}\ (abstra\ ac\ M\ v)$
 $\langle proof \rangle$

lemma *dbm-int-all-abstr*:

assumes $dbm\text{-int-all}\ M\ \forall (x, m) \in collect\text{-clock-pairs}\ g. m \in \mathbb{Z}$
shows $dbm\text{-int-all}\ (abstr\ g\ M\ v)$
 $\langle proof \rangle$

lemma *dbm-int-all-abstr'*:

assumes $\forall (x, m) \in \text{collect-clock-pairs } g. m \in \mathbb{Z}$

shows *dbm-int-all* (*abstr* g $(\lambda i j. \infty)$ v)

<proof>

lemma *dbm-int-all-inv-abstr*:

assumes $\forall (x, m) \in \text{clkp-set } A. m \in \mathbb{N}$

shows *dbm-int-all* (*abstr* (*inv-of* A l) $(\lambda i j. \infty)$ v)

<proof>

lemma *dbm-int-all-guard-abstr*:

assumes $\forall (x, m) \in \text{clkp-set } A. m \in \mathbb{N} \ A \vdash l \longrightarrow^{g,a,r} l'$

shows *dbm-int-all* (*abstr* g $(\lambda i j. \infty)$ v)

<proof>

lemma *dbm-int-abstra*:

assumes *dbm-int* M n *snd* (*constraint-pair* ac) $\in \mathbb{Z}$

shows *dbm-int* (*abstra* ac M v) n

<proof>

lemma *dbm-int-abstr*:

assumes *dbm-int* M n $\forall (x, m) \in \text{collect-clock-pairs } g. m \in \mathbb{Z}$

shows *dbm-int* (*abstr* g M v) n

<proof>

lemma *dbm-int-abstr'*:

assumes $\forall (x, m) \in \text{collect-clock-pairs } g. m \in \mathbb{Z}$

shows *dbm-int* (*abstr* g $(\lambda i j. \infty)$ v) n

<proof>

lemma *int-zone-dbm*:

assumes *clock-numbering'* v n

$\forall (-, d) \in \text{collect-clock-pairs } cc. d \in \mathbb{Z} \ \forall c \in \text{collect-clks } cc. v \ c \leq n$

obtains M **where** $\{u. u \vdash cc\} = [M]_{v,n}$

and $\forall i \leq n. \forall j \leq n. M \ i \ j \neq \infty \longrightarrow \text{get-const } (M \ i \ j) \in \mathbb{Z}$

<proof>

lemma *dbm-int-inv-abstr*:

assumes $\forall (x, m) \in \text{clkp-set } A. m \in \mathbb{N}$

shows *dbm-int* (*abstr* (*inv-of* A l) $(\lambda i j. \infty)$ v) n

<proof>

lemma *dbm-int-guard-abstr*:

assumes $\forall (x, m) \in \text{clkp-set } A. m \in \mathbb{N} \ A \vdash l \longrightarrow^{g,a,r} l'$

shows *dbm-int* (*abstr g* ($\lambda i j. \infty$) *v*) *n*
 ⟨*proof*⟩

lemma *collect-clks-id*: *collect-clks cc = fst ' collect-clock-pairs cc*
 ⟨*proof*⟩

end

3.6 Semantics Based on DBMs

theory *DBM-Zone-Semantics*
imports *TA-DBM-Operations*
begin

no-notation *infinity* ($\langle \infty \rangle$)
hide-const (**open**) *D*

3.6.1 Single Step

inductive *step-z-dbm* ::

(*'a, 'c, 't, 's*) *ta* \Rightarrow *'s* \Rightarrow *'t* :: {*linordered-cancel-ab-monoid-add, uminus*}
DBM
 \Rightarrow (*'c* \Rightarrow *nat*) \Rightarrow *nat* \Rightarrow *'a action* \Rightarrow *'s* \Rightarrow *'t DBM* \Rightarrow *bool*
 ($\langle - \vdash \langle -, - \rangle \rightsquigarrow_{-, -, -} \langle -, - \rangle$) [61,61,61,61] 61)

where

step-t-z-dbm:

D-inv = *abstr (inv-of A l) (λi j. ∞) v* \Longrightarrow *A* \vdash $\langle l, D \rangle \rightsquigarrow_{v, n, \tau} \langle l, \text{And } (up$

D) D-inv |

step-a-z-dbm:

A \vdash *l* $\xrightarrow{g, a, r}$ *l'*

\Longrightarrow *A* \vdash $\langle l, D \rangle \rightsquigarrow_{v, n, |a} \langle l', \text{And } (\text{reset}' (\text{And } D (\text{abstr } g (\lambda i j. \infty) v)) n r$

$v \ 0) (\text{abstr } (\text{inv-of } A \ l') (\lambda i j. \infty) v)$

inductive-cases *step-z-t-cases*: *A* \vdash $\langle l, D \rangle \rightsquigarrow_{v, n, \tau} \langle l', D' \rangle$

inductive-cases *step-z-a-cases*: *A* \vdash $\langle l, D \rangle \rightsquigarrow_{v, n, |a} \langle l', D' \rangle$

lemmas *step-z-cases* = *step-z-a-cases step-z-t-cases*

declare *step-z-dbm.intros*[*intro*]

lemma *step-z-dbm-preserves-int-all*:

fixes *D D'* :: (*'t* :: {*time, ring-1*} *DBM*)

assumes *A* \vdash $\langle l, D \rangle \rightsquigarrow_{v, n, a} \langle l', D' \rangle$ *global-clock-numbering A v n* $\forall (x, m)$
 \in *clkp-set A. m* \in \mathbb{N}

dbm-int-all D

shows *dbm-int-all* D'
 $\langle \text{proof} \rangle$

lemma *step-z-dbm-preserves-int*:

fixes $D D' :: ('t :: \{\text{time}, \text{ring-1}\} \text{DBM})$
assumes $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle$ *global-clock-numbering* $A v n \forall (x, m) \in \text{clkp-set } A. m \in \mathbb{N}$
 $\text{dbm-int } D n$
shows *dbm-int* $D' n$
 $\langle \text{proof} \rangle$

lemma *up-correct*:

assumes *clock-numbering'* $v n$
shows $[\text{up } M]_{v,n} = [M]_{v,n}^\uparrow$
 $\langle \text{proof} \rangle$

lemma *step-z-dbm-sound*:

assumes $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle$ *global-clock-numbering* $A v n$
shows $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_a \langle l', [D']_{v,n} \rangle$
 $\langle \text{proof} \rangle$

lemma *step-z-dbm-DBM*:

assumes $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_a \langle l', Z \rangle$ *global-clock-numbering* $A v n$
obtains D' **where** $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle Z = [D']_{v,n}$
 $\langle \text{proof} \rangle$

lemma *step-z-computable*:

assumes $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_a \langle l', Z \rangle$ *global-clock-numbering* $A v n$
obtains D' **where** $Z = [D']_{v,n}$
 $\langle \text{proof} \rangle$

lemma *step-z-dbm-complete*:

assumes *global-clock-numbering* $A v n A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$
and $u \in [(D)]_{v,n}$
shows $\exists D' a. A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle \wedge u' \in [D']_{v,n}$
 $\langle \text{proof} \rangle$

3.6.2 Additional Useful Properties

lemma *step-z-equiv*:

assumes *global-clock-numbering* $A v n A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_a \langle l', Z \rangle [D]_{v,n} = [M]_{v,n}$
shows $A \vdash \langle l, [M]_{v,n} \rangle \rightsquigarrow_a \langle l', Z \rangle$
 $\langle \text{proof} \rangle$

lemma *step-z-dbm-equiv*:

assumes *global-clock-numbering* $A \ v \ n \ A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle [D]_{v,n}$
 $= [M]_{v,n}$
shows $\exists M'. A \vdash \langle l, M \rangle \rightsquigarrow_{v,n,a} \langle l', M' \rangle \wedge [D']_{v,n} = [M']_{v,n}$
<proof>

lemma *step-z-empty*:

assumes $A \vdash \langle l, \{\} \rangle \rightsquigarrow_a \langle l', Z \rangle$
shows $Z = \{\}$
<proof>

lemma *step-z-dbm-empty*:

assumes *global-clock-numbering* $A \ v \ n \ A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,a} \langle l', D' \rangle [D]_{v,n}$
 $= \{\}$
shows $[D']_{v,n} = \{\}$
<proof>

end

theory *Regions-Beta*

imports

TA-Misc

Difference-Bound-Matrices.DBM-Normalization

Difference-Bound-Matrices.DBM-Operations

Difference-Bound-Matrices.Zones

begin

4 Refinement to β -regions

4.1 Definition

type-synonym *'c ceiling* = (*'c* \Rightarrow *nat*)

datatype *intv* =

Const nat |

Intv nat |

Greater nat

datatype *intv'* =

Const' int |

Intv' int |

Greater' int |

Smaller' int

type-synonym $t = \text{real}$

inductive $\text{valid-intv} :: \text{nat} \Rightarrow \text{intv} \Rightarrow \text{bool}$

where

$0 \leq d \Longrightarrow d \leq c \Longrightarrow \text{valid-intv } c \text{ (Const } d) \mid$
 $0 \leq d \Longrightarrow d < c \Longrightarrow \text{valid-intv } c \text{ (Intv } d) \mid$
 $\text{valid-intv } c \text{ (Greater } c)$

inductive $\text{valid-intv}' :: \text{int} \Rightarrow \text{int} \Rightarrow \text{intv}' \Rightarrow \text{bool}$

where

$\text{valid-intv}' l \text{ - (Smaller' } (-l)) \mid$
 $-l \leq d \Longrightarrow d \leq u \Longrightarrow \text{valid-intv}' l u \text{ (Const' } d) \mid$
 $-l \leq d \Longrightarrow d < u \Longrightarrow \text{valid-intv}' l u \text{ (Intv' } d) \mid$
 $\text{valid-intv}' - u \text{ (Greater' } u)$

inductive $\text{intv-elem} :: 'c \Rightarrow ('c, t) \text{ cval} \Rightarrow \text{intv} \Rightarrow \text{bool}$

where

$u x = d \Longrightarrow \text{intv-elem } x u \text{ (Const } d) \mid$
 $d < u x \Longrightarrow u x < d + 1 \Longrightarrow \text{intv-elem } x u \text{ (Intv } d) \mid$
 $c < u x \Longrightarrow \text{intv-elem } x u \text{ (Greater } c)$

inductive $\text{intv}'\text{-elem} :: 'c \Rightarrow 'c \Rightarrow ('c, t) \text{ cval} \Rightarrow \text{intv}' \Rightarrow \text{bool}$

where

$u x - u y < c \Longrightarrow \text{intv}'\text{-elem } x y u \text{ (Smaller' } c) \mid$
 $u x - u y = d \Longrightarrow \text{intv}'\text{-elem } x y u \text{ (Const' } d) \mid$
 $d < u x - u y \Longrightarrow u x - u y < d + 1 \Longrightarrow \text{intv}'\text{-elem } x y u \text{ (Intv' } d) \mid$
 $c < u x - u y \Longrightarrow \text{intv}'\text{-elem } x y u \text{ (Greater' } c)$

abbreviation $\text{total-preorder } r \equiv \text{refl } r \wedge \text{trans } r$

inductive $\text{isConst} :: \text{intv} \Rightarrow \text{bool}$

where

isConst (Const -)

inductive $\text{isIntv} :: \text{intv} \Rightarrow \text{bool}$

where

isIntv (Intv -)

inductive $\text{isGreater} :: \text{intv} \Rightarrow \text{bool}$

where

$\text{isGreater (Greater -)}$

declare $\text{isIntv.intros[intro!]} \text{ isConst.intros[intro!]} \text{ isGreater.intros[intro!]}$

declare *isIntv.cases*[elim!] *isConst.cases*[elim!] *isGreater.cases*[elim!]

inductive *valid-region* :: '*c set* ⇒ ('*c* ⇒ *nat*) ⇒ ('*c* ⇒ *intv*) ⇒ ('*c* ⇒ '*c* ⇒ *intv'*) ⇒ '*c rel* ⇒ *bool*

where

$\llbracket X_0 = \{x \in X. \exists d. I x = Intv d\}; r \subseteq X_0 \times X_0; refl\text{-on } X_0 r; trans r; total\text{-on } X_0 r;$

$\forall x \in X. valid\text{-intv } (k x) (I x);$

$\forall x \in X. \forall y \in X. isGreater (I x) \vee isGreater (I y) \longrightarrow valid\text{-intv}' (k y) (k x) (J x y)$

$\implies valid\text{-region } X k I J r$

inductive-set *region* for *X I J r*

where

$\forall x \in X. u x \geq 0 \implies \forall x \in X. intv\text{-elem } x u (I x) \implies X_0 = \{x \in X. \exists d. I x = Intv d\} \implies$

$\forall x \in X_0. \forall y \in X_0. (x, y) \in r \iff frac (u x) \leq frac (u y) \implies$

$\forall x \in X. \forall y \in X. isGreater (I x) \vee isGreater (I y) \longrightarrow intv'\text{-elem } x y u (J x y)$

$\implies u \in region X I J r$

Defining the unique element of a partition that contains a valuation

definition *part* ($\langle[-]\rangle$ [61,61] 61) **where** *part* *v* $\mathcal{R} \equiv THE R. R \in \mathcal{R} \wedge v \in R$

First we need to show that the set of regions is a partition of the set of all clock assignments. This property is only claimed by P. Bouyer.

inductive-cases[elim!]: *intv-elem* *x u* (*Const d*)

inductive-cases[elim!]: *intv-elem* *x u* (*Intv d*)

inductive-cases[elim!]: *intv-elem* *x u* (*Greater d*)

inductive-cases[elim!]: *valid-intv* *c* (*Greater d*)

inductive-cases[elim!]: *valid-intv* *c* (*Const d*)

inductive-cases[elim!]: *valid-intv* *c* (*Intv d*)

inductive-cases[elim!]: *intv'\text{-elem}* *x y u* (*Const' d*)

inductive-cases[elim!]: *intv'\text{-elem}* *x y u* (*Intv' d*)

inductive-cases[elim!]: *intv'\text{-elem}* *x y u* (*Greater' d*)

inductive-cases[elim!]: *intv'\text{-elem}* *x y u* (*Smaller' d*)

inductive-cases[elim!]: *valid-intv'* *l u* (*Greater' d*)

inductive-cases[elim!]: *valid-intv'* *l u* (*Smaller' d*)

inductive-cases[elim!]: *valid-intv'* *l u* (*Const' d*)

inductive-cases[elim!]: *valid-intv'* *l u* (*Intv' d*)

declare *valid-intv.intros*[intro]

declare *valid-intv'.intros*[intro]

```

declare intv-elim.intros[intro]
declare intv'-elim.intros[intro]

declare region.cases[elim]
declare valid-region.cases[elim]

```

4.2 Basic Properties

First we show that all valid intervals are distinct

lemma *valid-intv-distinct*:

```

valid-intv c I  $\implies$  valid-intv c I'  $\implies$  intv-elim x u I  $\implies$  intv-elim x u I'
 $\implies I = I'$ 
<proof>

```

lemma *valid-intv'-distinct*:

```

 $-c \leq d \implies$  valid-intv' c d I  $\implies$  valid-intv' c d I'  $\implies$  intv'-elem x y u I
 $\implies$  intv'-elem x y u I'
 $\implies I = I'$ 
<proof>

```

From this we show that all valid regions are distinct

lemma *valid-regions-distinct*:

```

valid-region X k I J r  $\implies$  valid-region X k I' J' r'  $\implies v \in$  region X I J
 $r \implies v \in$  region X I' J' r'
 $\implies$  region X I J r = region X I' J' r'
<proof>

```

locale *Beta-Regions* =

```

fixes X :: 'c set and k :: 'c  $\Rightarrow$  nat
assumes finite: finite X
assumes non-empty: X  $\neq$  {}
begin

```

definition

```

 $\mathcal{R} \equiv \{ \textit{region X I J r} \mid \textit{I J r. valid-region X k I J r} \}$ 

```

definition *V* :: ('*c*, *t*) *cval set where*

```

 $V \equiv \{ v . \forall x \in X. v x \geq 0 \}$ 

```

lemma *R-regions-distinct*:

```

 $\llbracket R \in \mathcal{R}; v \in R; R' \in \mathcal{R}; R \neq R' \rrbracket \implies v \notin R'$ 
<proof>

```

Secondly, we also need to show that every valuations belongs to a region

which is part of the partition.

definition *intv-of* :: $\text{nat} \Rightarrow t \Rightarrow \text{intv}$ **where**

intv-of c $v \equiv$
 if $(v > c)$ then *Greater* c
 else if $(\exists x :: \text{nat}. x = v)$ then $(\text{Const} (\text{nat} (\text{floor } v)))$
 else $(\text{Intv} (\text{nat} (\text{floor } v)))$

definition *intv'-of* :: $\text{int} \Rightarrow \text{int} \Rightarrow t \Rightarrow \text{intv}'$ **where**

intv'-of l u $v \equiv$
 if $(v > u)$ then *Greater'* u
 else if $(v < l)$ then *Smaller'* l
 else if $(\exists x :: \text{int}. x = v)$ then $(\text{Const}' (\text{floor } v))$
 else $(\text{Intv}' (\text{floor } v))$

lemma *region-cover*:

$\forall x \in X. v x \geq 0 \implies \exists R. R \in \mathcal{R} \wedge v \in R$
 $\langle \text{proof} \rangle$

lemma *region-cover-V*: $v \in V \implies \exists R. R \in \mathcal{R} \wedge v \in R$ $\langle \text{proof} \rangle$

Note that we cannot show that every region is non-empty anymore. The problem are regions fixing differences between an 'infeasible' constant.

We can show that there is always exactly one region a valid valuation belongs to. Note that we do not need non-emptiness for that.

lemma *regions-partition*:

$\forall x \in X. 0 \leq v x \implies \exists! R \in \mathcal{R}. v \in R$
 $\langle \text{proof} \rangle$

lemma *region-unique*:

$v \in R \implies R \in \mathcal{R} \implies [v]_{\mathcal{R}} = R$
 $\langle \text{proof} \rangle$

lemma *regions-partition'*:

$\forall x \in X. 0 \leq v x \implies \forall x \in X. 0 \leq v' x \implies v' \in [v]_{\mathcal{R}} \implies [v']_{\mathcal{R}} = [v]_{\mathcal{R}}$
 $\langle \text{proof} \rangle$

lemma *regions-closed*:

$R \in \mathcal{R} \implies v \in R \implies t \geq 0 \implies [v \oplus t]_{\mathcal{R}} \in \mathcal{R}$
 $\langle \text{proof} \rangle$

lemma *regions-closed'*:

$R \in \mathcal{R} \implies v \in R \implies t \geq 0 \implies (v \oplus t) \in [v \oplus t]_{\mathcal{R}}$
 $\langle \text{proof} \rangle$

lemma *valid-regions-I-cong*:

valid-region $X k I J r \implies \forall x \in X. I x = I' x$
 $\implies \forall x \in X. \forall y \in X. (isGreater (I x) \vee isGreater (I y)) \longrightarrow J x y =$
 $J' x y$
 $\implies region X I J r = region X I' J' r \wedge valid-region X k I' J' r$
 $\langle proof \rangle$

fun *intv-const* :: *intv* \Rightarrow *nat*

where

intv-const (*Const* d) = d |
intv-const (*Intv* d) = d |
intv-const (*Greater* d) = d

fun *intv'-const* :: *intv'* \Rightarrow *int*

where

intv'-const (*Smaller'* d) = d |
intv'-const (*Const'* d) = d |
intv'-const (*Intv'* d) = d |
intv'-const (*Greater'* d) = d

lemma *finite- \mathcal{R} -aux*:

fixes $P A B$ **assumes** *finite* $\{x. A x\}$ *finite* $\{x. B x\}$
shows *finite* $\{(I, J) \mid I J. P I J r \wedge A I \wedge B J\}$
 $\langle proof \rangle$

lemma *finite- \mathcal{R}* :

notes $[[simproc\ add:\ finite-Collect]]$
shows *finite* \mathcal{R}
 $\langle proof \rangle$

end

4.3 Approximation with β -regions

locale *Beta-Regions'* = *Beta-Regions* +

fixes $v n$ *not-in-X*
assumes *clock-numbering*: $\forall c. v c > 0 \wedge (\forall x. \forall y. v x \leq n \wedge v y \leq n \wedge$
 $v x = v y \longrightarrow x = y)$
 $\forall k :: nat \leq n. k > 0 \longrightarrow (\exists c \in X. v c = k) \forall c \in$
 $X. v c \leq n$
assumes *not-in-X*: *not-in-X* $\notin X$
begin

definition $v' \equiv \lambda i. \text{if } 0 < i \wedge i \leq n \text{ then (THE } c. c \in X \wedge v c = i) \text{ else not-in-}X$

lemma $v-v'$:

$\forall c \in X. v'(v c) = c$

$\langle \text{proof} \rangle$

abbreviation

$vabstr (S :: ('a, t) \text{ zone}) M \equiv S = [M]_{v,n} \wedge (\forall i \leq n. \forall j \leq n. M i j \neq \infty \rightarrow \text{get-const } (M i j) \in \mathbb{Z})$

definition *normalized*:

$\text{normalized } M \equiv$

$(\forall i j. 0 < i \wedge i \leq n \wedge 0 < j \wedge j \leq n \wedge M i j \neq \infty \rightarrow$

$Lt (- (\text{real}((k o v') j))) \leq M i j \wedge M i j \leq Le ((k o v') i))$

$\wedge (\forall i \leq n. i > 0 \rightarrow (M i 0 \leq Le ((k o v') i) \vee M i 0 = \infty) \wedge Lt (- ((k o v') i)) \leq M 0 i)$

definition *apx-def*:

$\text{Approx}_\beta Z \equiv \bigcap \{S. \exists U M. S = \bigcup U \wedge U \subseteq \mathcal{R} \wedge Z \subseteq S \wedge vabstr S M \wedge \text{normalized } M\}$

definition

$\text{normalized}' M \equiv$

$(\forall i j. 0 < i \wedge i \leq n \wedge 0 < j \wedge j \leq n \wedge M i j \neq \infty \wedge i \neq j \rightarrow$

$Lt (- (\text{real}((k o v') j))) \leq M i j \wedge M i j \leq Le ((k o v') i))$

$\wedge (\forall i \leq n. i > 0 \rightarrow (M i 0 \leq Le ((k o v') i) \vee M i 0 = \infty) \wedge Lt (- ((k o v') i)) \leq M 0 i)$

lemma *normalized'-normalized*:

assumes $\forall i \leq n. M i i = 0$ *normalized'* M

shows *normalized* M

$\langle \text{proof} \rangle$

lemma *normalized-normalized'*:

normalized' M **if** *normalized* M

$\langle \text{proof} \rangle$

lemma *apx-min*:

$S = \bigcup U \implies U \subseteq \mathcal{R} \implies S = [M]_{v,n} \implies \forall i \leq n. \forall j \leq n. M i j \neq \infty \rightarrow \text{get-const } (M i j) \in \mathbb{Z}$

$\implies \text{normalized } M \implies Z \subseteq S \implies \text{Approx}_\beta Z \subseteq S$

$\langle \text{proof} \rangle$

lemma \mathcal{R} -union: $\bigcup \mathcal{R} = V$ \langle proof \rangle

definition V -dbm where

V -dbm $\equiv \lambda i j. \text{if } i = 0 \text{ then } Le \ 0 \text{ else } \infty$

lemma v -not-eq-0:

$v \ c \neq 0$
 \langle proof \rangle

lemma V -dbm-eq- V : $[V\text{-dbm}]_{v,n} = V$

\langle proof \rangle

lemma V -dbm-int:

$\forall i \leq n. \forall j \leq n. V\text{-dbm } i \ j \neq \infty \implies \text{get-const } (V\text{-dbm } i \ j) \in \mathbb{Z}$
 \langle proof \rangle

lemma normalized- V -dbm:

normalized V -dbm
 \langle proof \rangle

lemma all-dbm: $\exists M. \text{vabstr } (\bigcup \mathcal{R}) \ M \wedge \text{normalized } M$

\langle proof \rangle

lemma \mathcal{R} -int:

$R \in \mathcal{R} \implies R' \in \mathcal{R} \implies R \neq R' \implies R \cap R' = \{\}$ \langle proof \rangle

lemma aux1:

$u \in R \implies R \in \mathcal{R} \implies U \subseteq \mathcal{R} \implies u \in \bigcup U \implies R \subseteq \bigcup U$ \langle proof \rangle

lemma aux2: $x \in \bigcap U \implies U \neq \{\} \implies \exists S \in U. x \in S$ \langle proof \rangle

lemma aux2': $x \in \bigcap U \implies U \neq \{\} \implies \forall S \in U. x \in S$ \langle proof \rangle

lemma apx-subset: $Z \subseteq \text{Approx}_\beta Z$ \langle proof \rangle

lemma aux3:

$\forall X \in U. \forall Y \in U. X \cap Y \in U \implies S \subseteq U \implies S \neq \{\} \implies \text{finite } S$
 $\implies \bigcap S \in U$
 \langle proof \rangle

lemma empty-zone-dbm:

$\exists M :: t \text{ DBM}. \text{vabstr } \{\} \ M \wedge \text{normalized } M \wedge (\forall k \leq n. M \ k \ k \leq Le \ 0)$
 \langle proof \rangle

lemma *DBM-set-diag*:

assumes $[M]_{v,n} \neq \{\}$

shows $[M]_{v,n} = [(\lambda i j. \text{if } i = j \text{ then } Le\ 0 \text{ else } M\ i\ j)]_{v,n}$

<proof>

lemma *apx-min'*:

$S = \bigcup U \implies U \subseteq \mathcal{R} \implies S = [M]_{v,n} \implies \forall i \leq n. \forall j \leq n. M\ i\ j \neq \infty$
 $\longrightarrow \text{get-const } (M\ i\ j) \in \mathbb{Z}$

$\implies \text{normalized}' M \implies Z \subseteq S \implies \text{Approx}_\beta Z \subseteq S$

<proof>

lemma *valid-dbms-int*:

$\forall X \in \{S. \exists M. \text{vabstr } S\ M\}. \forall Y \in \{S. \exists M. \text{vabstr } S\ M\}. X \cap Y \in \{S. \exists M. \text{vabstr } S\ M\}$

<proof>

lemma *split-min'*:

$P\ (\min\ i\ j) = ((\min\ i\ j = i \longrightarrow P\ i) \wedge (\min\ i\ j = j \longrightarrow P\ j))$

<proof>

lemma *normalized-and-preservation*:

$\text{normalized } M1 \implies \text{normalized } M2 \implies \text{normalized } (\text{And } M1\ M2)$

<proof>

lemma *valid-dbms-int'*:

$\forall X \in \{S. \exists M. \text{vabstr } S\ M \wedge \text{normalized } M\}. \forall Y \in \{S. \exists M. \text{vabstr } S\ M \wedge \text{normalized } M\}.$

$X \cap Y \in \{S. \exists M. \text{vabstr } S\ M \wedge \text{normalized } M\}$

<proof>

lemma *apx-in*:

$Z \subseteq V \implies \text{Approx}_\beta Z \in \{S. \exists U\ M. S = \bigcup U \wedge U \subseteq \mathcal{R} \wedge Z \subseteq S \wedge \text{vabstr } S\ M \wedge \text{normalized } M\}$

<proof>

lemma *apx-empty*:

$\text{Approx}_\beta \{\} = \{\}$

<proof>

end

4.4 Computing β -Approximation

4.4.1 Computation

context *Beta-Regions'*
begin

lemma *dbm-regions*:

vabstr $S M \Longrightarrow \text{normalized}' M \Longrightarrow [M]_{v,n} \neq \{\} \Longrightarrow [M]_{v,n} \subseteq V \Longrightarrow \exists U \subseteq \mathcal{R}. S = \bigcup U$
 $\langle \text{proof} \rangle$

lemma *dbm-regions'*:

vabstr $S M \Longrightarrow \text{normalized}' M \Longrightarrow S \subseteq V \Longrightarrow \exists U \subseteq \mathcal{R}. S = \bigcup U$
 $\langle \text{proof} \rangle$

lemma *dbm-regions''*:

dbm-int $M n \Longrightarrow \text{normalized}' M \Longrightarrow [M]_{v,n} \subseteq V \Longrightarrow \exists U \subseteq \mathcal{R}. [M]_{v,n} = \bigcup U$
 $\langle \text{proof} \rangle$

lemma *DBM-le-subset'*:

assumes $\forall i \leq n. \forall j \leq n. i \neq j \longrightarrow M i j \leq M' i j$
and $\forall i \leq n. M' i i \geq \text{Le } 0$
and $u \in [M]_{v,n}$
shows $u \in [M']_{v,n}$
 $\langle \text{proof} \rangle$

lemma *neg-diag-empty-spec*:

assumes $i \leq n \ M i i < 0$
shows $[M]_{v,n} = \{\}$
 $\langle \text{proof} \rangle$

lemma *canonical-empty-zone-spec*:

assumes *canonical* $M n$
shows $[M]_{v,n} = \{\} \longleftrightarrow (\exists i \leq n. M i i < 0)$
 $\langle \text{proof} \rangle$

lemma *norm-set-diag*:

assumes *canonical* $M n \ [M]_{v,n} \neq \{\}$
obtains M' **where** $[M]_{v,n} = [M']_{v,n} \ [norm \ M \ (k \ o \ v') \ n]_{v,n} = [norm \ M' \ (k \ o \ v') \ n]_{v,n}$
 $\forall i \leq n. M' i i = 0 \ \text{canonical } M' n$
 $\langle \text{proof} \rangle$

lemma *norm-normalizes'*:
notes *any-le-inf*[intro]
shows *normalized'* (*norm M (k o v') n*)
⟨*proof*⟩

lemma *norm-normalizes*:
assumes $\forall i \leq n. M\ i\ i = 0$
shows *normalized* (*norm M (k o v') n*)
⟨*proof*⟩

lemma *norm-int-preservation*:
fixes *M :: real DBM*
assumes *dbm-int M n i ≤ n j ≤ n norm M (k o v') n i j ≠ ∞*
shows *get-const (norm M (k o v') n i j) ∈ ℤ*
⟨*proof*⟩

lemma *norm-V-preservation'*:
notes *any-le-inf*[intro]
assumes $[M]_{v,n} \subseteq V$ *canonical M n [M]_{v,n} ≠ {}*
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq V$
⟨*proof*⟩

lemma *norm-V-preservation*:
assumes $[M]_{v,n} \subseteq V$ *canonical M n*
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq V$ (**is** $[?M]_{v,n} \subseteq V$)
⟨*proof*⟩

lemma *norm-min*:
assumes *normalized' M1 [M]_{v,n} ⊆ [M1]_{v,n}*
canonical M n [M]_{v,n} ≠ {} [M]_{v,n} ⊆ V
shows $[norm\ M\ (k\ o\ v')\ n]_{v,n} \subseteq [M1]_{v,n}$ (**is** $[?M2]_{v,n} \subseteq [M1]_{v,n}$)
⟨*proof*⟩

lemma *apx-norm-eq*:
assumes *canonical M n [M]_{v,n} ⊆ V dbm-int M n*
shows $Approx_{\beta} ([M]_{v,n}) = [norm\ M\ (k\ o\ v')\ n]_{v,n}$
⟨*proof*⟩

end

4.5 Auxiliary β -boundedness Theorems

context *Beta-Regions'*

begin

lemma β -boundedness-diag-lt:

fixes $m :: int$

assumes $- k y \leq m \ m \leq k x \ x \in X \ y \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x - u y < m\}$

$\langle proof \rangle$

lemma β -boundedness-diag-eq:

fixes $m :: int$

assumes $- k y \leq m \ m \leq k x \ x \in X \ y \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x - u y = m\}$

$\langle proof \rangle$

lemma β -boundedness-lt:

fixes $m :: int$

assumes $m \leq k x \ x \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x < m\}$

$\langle proof \rangle$

lemma β -boundedness-gt:

fixes $m :: int$

assumes $m \leq k x \ x \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x > m\}$

$\langle proof \rangle$

lemma β -boundedness-eq:

fixes $m :: int$

assumes $m \leq k x \ x \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x = m\}$

$\langle proof \rangle$

lemma β -boundedness-diag-le:

fixes $m :: int$

assumes $- k y \leq m \ m \leq k x \ x \in X \ y \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x - u y \leq m\}$

$\langle proof \rangle$

lemma β -boundedness-le:

fixes $m :: int$

assumes $m \leq k x \ x \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x \leq m\}$

$\langle proof \rangle$

lemma β -boundedness-ge:

fixes $m :: int$

assumes $m \leq k x \ x \in X$

shows $\exists U \subseteq \mathcal{R}. \bigcup U = \{u \in V. u x \geq m\}$

$\langle proof \rangle$

lemma β -boundedness-diag-lt':

fixes $m :: int$

shows

$- k y \leq (m :: int) \implies m \leq k x \implies x \in X \implies y \in X \implies Z \subseteq \{u \in V. u x - u y < m\}$

$\implies Approx_{\beta} Z \subseteq \{u \in V. u x - u y < m\}$

$\langle proof \rangle$

lemma β -boundedness-diag-le':

fixes $m :: int$

shows

$- k y \leq (m :: int) \implies m \leq k x \implies x \in X \implies y \in X \implies Z \subseteq \{u \in V. u x - u y \leq m\}$

$\implies Approx_{\beta} Z \subseteq \{u \in V. u x - u y \leq m\}$

$\langle proof \rangle$

lemma β -boundedness-lt':

fixes $m :: int$

shows

$m \leq k x \implies x \in X \implies Z \subseteq \{u \in V. u x < m\} \implies Approx_{\beta} Z \subseteq \{u \in V. u x < m\}$

$\langle proof \rangle$

lemma β -boundedness-gt':

fixes $m :: int$

shows

$m \leq k x \implies x \in X \implies Z \subseteq \{u \in V. u x > m\} \implies Approx_{\beta} Z \subseteq \{u \in V. u x > m\}$

$\langle proof \rangle$

lemma obtains-dbm-le:

fixes $m :: int$

assumes $x \in X \ m \leq k x$

obtains M **where** $vabstr \ \{u \in V. u x \leq m\} \ M$ *normalized* M

$\langle proof \rangle$

lemma β -boundedness-le':

fixes $m :: int$
shows
 $m \leq k \ x \implies x \in X \implies Z \subseteq \{u \in V. u \ x \leq m\} \implies Approx_\beta Z \subseteq \{u \in V. u \ x \leq m\}$
 <proof>

lemma *obtains-dbm-ge*:
fixes $m :: int$
assumes $x \in X \ m \leq k \ x$
obtains M **where** $vabstr \ \{u \in V. u \ x \geq m\} \ M$ *normalized* M
 <proof>

lemma *β -boundedness-ge'*:
fixes $m :: int$
shows $m \leq k \ x \implies x \in X \implies Z \subseteq \{u \in V. u \ x \geq m\} \implies Approx_\beta Z \subseteq \{u \in V. u \ x \geq m\}$
 <proof>

end

end

5 The Classic Construction for Decidability

theory *Regions*
imports *Timed-Automata TA-Misc*
begin

The following is a formalization of regions in the correct version of Patricia Bouyer et al.

5.1 Definition of Regions

type-synonym *'c ceiling* = (*'c* \Rightarrow *nat*)

datatype *intv* =
Const nat |
Intv nat |
Greater nat

type-synonym *t* = *real*

inductive *valid-intv* :: *nat* \Rightarrow *intv* \Rightarrow *bool*

where

$$\begin{aligned} 0 \leq d &\implies d \leq c \implies \text{valid-intv } c \text{ (Const } d) \mid \\ 0 \leq d &\implies d < c \implies \text{valid-intv } c \text{ (Intv } d) \mid \\ &\text{valid-intv } c \text{ (Greater } c) \end{aligned}$$

inductive *intv-elem* :: 'c \Rightarrow ('c,t) *cval* \Rightarrow *intv* \Rightarrow *bool*

where

$$\begin{aligned} u \ x = d &\implies \text{intv-elem } x \ u \text{ (Const } d) \mid \\ d < u \ x &\implies u \ x < d + 1 \implies \text{intv-elem } x \ u \text{ (Intv } d) \mid \\ c < u \ x &\implies \text{intv-elem } x \ u \text{ (Greater } c) \end{aligned}$$

abbreviation *total-preorder* $r \equiv \text{refl } r \wedge \text{trans } r$

inductive *valid-region* :: 'c *set* \Rightarrow ('c \Rightarrow *nat*) \Rightarrow ('c \Rightarrow *intv*) \Rightarrow 'c *rel* \Rightarrow *bool*

where

$$\begin{aligned} \llbracket X_0 = \{x \in X. \exists d. I \ x = \text{Intv } d\}; r \subseteq X_0 \times X_0; \text{refl-on } X_0 \ r; \text{trans } r; \\ \text{total-on } X_0 \ r; \\ \forall x \in X. \text{valid-intv } (k \ x) \ (I \ x) \rrbracket \\ \implies \text{valid-region } X \ k \ I \ r \end{aligned}$$

inductive-set *region* **for** $X \ I \ r$

where

$$\begin{aligned} \forall x \in X. u \ x \geq 0 &\implies \forall x \in X. \text{intv-elem } x \ u \ (I \ x) \implies X_0 = \{x \in X. \\ \exists d. I \ x = \text{Intv } d\} &\implies \\ \forall x \in X_0. \forall y \in X_0. (x, y) \in r &\iff \text{frac } (u \ x) \leq \text{frac } (u \ y) \\ \implies u \in \text{region } X \ I \ r & \end{aligned}$$

Defining the unique element of a partition that contains a valuation

definition *part* ($\langle[-]\rangle$ [61,61] 61) **where** *part* $v \ \mathcal{R} \equiv \text{THE } R. R \in \mathcal{R} \wedge v \in R$

inductive-set *Succ* **for** $\mathcal{R} \ R$ **where**

$$u \in R \implies R \in \mathcal{R} \implies R' \in \mathcal{R} \implies t \geq 0 \implies R' = [u \oplus t]_{\mathcal{R}} \implies R' \in \text{Succ } \mathcal{R} \ R$$

First we need to show that the set of regions is a partition of the set of all clock assignments. This property is only claimed by P. Bouyer.

inductive-cases[*elim!*]: *intv-elem* $x \ u \text{ (Const } d)$

inductive-cases[*elim!*]: *intv-elem* $x \ u \text{ (Intv } d)$

inductive-cases[*elim!*]: *intv-elem* $x \ u \text{ (Greater } d)$

inductive-cases[*elim!*]: *valid-intv* $c \text{ (Greater } d)$

inductive-cases[*elim!*]: *valid-intv* $c \text{ (Const } d)$

inductive-cases[*elim!*]: *valid-intv* $c \text{ (Intv } d)$

declare *valid-intv.intros*[*intro*]
declare *intv-elem.intros*[*intro*]
declare *Succ.intros*[*intro*]

declare *Succ.cases*[*elim*]

declare *region.cases*[*elim*]
declare *valid-region.cases*[*elim*]

5.2 Basic Properties

First we show that all valid intervals are distinct.

lemma *valid-intv-distinct*:

$valid-intv\ c\ I \implies valid-intv\ c\ I' \implies intv-elem\ x\ u\ I \implies intv-elem\ x\ u\ I'$
 $\implies I = I'$

<proof>

From this we show that all valid regions are distinct.

lemma *valid-regions-distinct*:

$valid-region\ X\ I\ r \implies valid-region\ X\ k\ I'\ r' \implies v \in region\ X\ I\ r \implies v$
 $\in region\ X\ I'\ r'$
 $\implies region\ X\ I\ r = region\ X\ I'\ r'$

<proof>

lemma *\mathcal{R} -regions-distinct*:

$\llbracket \mathcal{R} = \{region\ X\ I\ r \mid I\ r.\ valid-region\ X\ k\ I\ r\}; R \in \mathcal{R}; v \in R; R' \in \mathcal{R};$
 $R \neq R' \rrbracket \implies v \notin R'$

<proof>

Secondly, we also need to show that every valuations belongs to a region which is part of the partition.

definition *intv-of* $:: nat \Rightarrow t \Rightarrow intv$ **where**

intv-of $k\ c \equiv$

if $(c > k)$ *then* *Greater* k

else if $(\exists x :: nat.\ x = c)$ *then* $(Const\ (nat\ (floor\ c)))$

else $(Intv\ (nat\ (floor\ c)))$

lemma *region-cover*:

$\forall x \in X.\ u\ x \geq 0 \implies \exists R.\ R \in \{region\ X\ I\ r \mid I\ r.\ valid-region\ X\ k\ I\ r\}$
 $\wedge u \in R$

<proof>

lemma *intv-not-empty*:

obtains d **where** $\text{intv-elem } x (v(x := d)) (I x)$
 $\langle \text{proof} \rangle$

fun $\text{get-intv-val} :: \text{intv} \Rightarrow \text{real} \Rightarrow \text{real}$
where

$\text{get-intv-val } (\text{Const } d) \quad - = d \mid$
 $\text{get-intv-val } (\text{Intv } d) \quad f = d + f \mid$
 $\text{get-intv-val } (\text{Greater } d) \quad - = d + 1$

lemma $\text{region-not-empty-aux}$:

assumes $0 < f \ f < 1 \ 0 < g \ g < 1$
shows $\text{frac } (\text{get-intv-val } (\text{Intv } d) f) \leq \text{frac } (\text{get-intv-val } (\text{Intv } d') g) \longleftrightarrow$
 $f \leq g$
 $\langle \text{proof} \rangle$

lemma region-not-empty :

assumes $\text{finite } X \ \text{valid-region } X \ k \ I \ r$
shows $\exists u. u \in \text{region } X \ I \ r$
 $\langle \text{proof} \rangle$

Now we can show that there is always exactly one region a valid valuation belongs to.

lemma regions-partition :

$\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\} \Longrightarrow \forall x \in X. 0 \leq u \ x \Longrightarrow$
 $\exists! R \in \mathcal{R}. u \in R$
 $\langle \text{proof} \rangle$

lemma region-unique :

$\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\} \Longrightarrow u \in R \Longrightarrow R \in \mathcal{R} \Longrightarrow$
 $[u]_{\mathcal{R}} = R$
 $\langle \text{proof} \rangle$

lemma $\text{regions-partition}'$:

$\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\} \Longrightarrow \forall x \in X. 0 \leq v \ x \Longrightarrow$
 $\forall x \in X. 0 \leq v' \ x \Longrightarrow v' \in [v]_{\mathcal{R}}$
 $\Longrightarrow [v']_{\mathcal{R}} = [v]_{\mathcal{R}}$
 $\langle \text{proof} \rangle$

lemma regions-closed :

$\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\} \Longrightarrow R \in \mathcal{R} \Longrightarrow v \in R \Longrightarrow$
 $t \geq 0 \Longrightarrow [v \oplus t]_{\mathcal{R}} \in \mathcal{R}$
 $\langle \text{proof} \rangle$

lemma $\text{regions-closed}'$:

$\mathcal{R} = \{\text{region } X I r \mid I r. \text{valid-region } X k I r\} \implies R \in \mathcal{R} \implies v \in R \implies$
 $t \geq 0 \implies (v \oplus t) \in [v \oplus t]_{\mathcal{R}}$
 <proof>

lemma *valid-regions-I-cong*:

$\text{valid-region } X k I r \implies \forall x \in X. I x = I' x \implies \text{region } X I r = \text{region}$
 $X I' r \wedge \text{valid-region } X k I' r$
 <proof>

fun *intv-const* :: *intv* \Rightarrow *nat*

where

$\text{intv-const } (\text{Const } d) = d \mid$
 $\text{intv-const } (\text{Intv } d) = d \mid$
 $\text{intv-const } (\text{Greater } d) = d$

lemma *finite- \mathcal{R}* :

notes [[*simproc add: finite-Collect*]] *finite-subset*[*intro*]
fixes *X k*
defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$
assumes *finite X*
shows *finite \mathcal{R}*
 <proof>

lemma *SuccI2*:

$\mathcal{R} = \{\text{region } X I r \mid I r. \text{valid-region } X k I r\} \implies v \in R \implies R \in \mathcal{R} \implies$
 $t \geq 0 \implies R' = [v \oplus t]_{\mathcal{R}}$
 $\implies R' \in \text{Succ } \mathcal{R} R$
 <proof>

5.3 Set of Regions

The first property Bouyer shows is that these regions form a 'set of regions'.

For the unbounded region in the upper right corner, the set of successors only contains itself.

lemma *Succ-refl*:

$\mathcal{R} = \{\text{region } X I r \mid I r. \text{valid-region } X k I r\} \implies \text{finite } X \implies R \in \mathcal{R} \implies$
 $R \in \text{Succ } \mathcal{R} R$
 <proof>

lemma *Succ-refl'*:

$\mathcal{R} = \{\text{region } X I r \mid I r. \text{valid-region } X k I r\} \implies \text{finite } X \implies \forall x \in X.$
 $\exists c. I x = \text{Greater } c$
 $\implies \text{region } X I r \in \mathcal{R} \implies \text{Succ } \mathcal{R} (\text{region } X I r) = \{\text{region } X I r\}$

<proof>

Defining the closest successor of a region. Only exists if at least one interval is upper-bounded.

definition

$succ \mathcal{R} R =$
 $(SOME R'. R' \in Succ \mathcal{R} R \wedge (\forall u \in R. \forall t \geq 0. (u \oplus t) \notin R \longrightarrow (\exists t' \leq t. (u \oplus t') \in R' \wedge 0 \leq t'))))$

inductive $isConst :: intv \Rightarrow bool$

where

$isConst (Const -)$

inductive $isIntv :: intv \Rightarrow bool$

where

$isIntv (Intv -)$

inductive $isGreater :: intv \Rightarrow bool$

where

$isGreater (Greater -)$

declare $isIntv.intros[intro!] isConst.intros[intro!] isGreater.intros[intro!]$

declare $isIntv.cases[elim!] isConst.cases[elim!] isGreater.cases[elim!]$

What Bouyer states at the end. However, we have to be a bit more precise than in her statement.

lemma *closest-prestable-1:*

fixes $I X k r$

defines $\mathcal{R} \equiv \{region X I r \mid I r. valid-region X k I r\}$

defines $R \equiv region X I r$

defines $Z \equiv \{x \in X . \exists c. I x = Const c\}$

assumes $Z \neq \{\}$

defines $I' \equiv \lambda x. if x \notin Z then I x else if intv-const (I x) = k x then Greater (k x) else Intv (intv-const (I x))$

defines $r' \equiv r \cup \{(x,y) . x \in Z \wedge y \in X \wedge intv-const (I x) < k x \wedge isIntv (I' y)\}$

assumes *finite X*

assumes *valid-region X k I r*

shows $\forall v \in R. \forall t > 0. \exists t' \leq t. (v \oplus t') \in region X I' r' \wedge t' \geq 0$

and $\forall v \in region X I' r'. \forall t \geq 0. (v \oplus t) \notin R$

and $\forall x \in X. \neg isConst (I' x)$

and $\forall v \in R. \forall t < 1. \forall t' \geq 0. (v \oplus t') \in region X I' r' \longrightarrow \{x. x \in X \wedge (\exists c. I x = Intv c \wedge v x + t \geq c + 1)\}$

$= \{x. x \in X \wedge (\exists c. I' x = \text{Intv } c \wedge (v \oplus t') x + (t - t') \geq c + 1)\}$

$\langle \text{proof} \rangle$

lemma *closest-valid-1:*

fixes $I X k r$
defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$
defines $R \equiv \text{region } X I r$
defines $Z \equiv \{x \in X . \exists c. I x = \text{Const } c\}$
assumes $Z \neq \{\}$
defines $I' \equiv \lambda x. \text{if } x \notin Z \text{ then } I x \text{ else if } \text{intv-const } (I x) = k x \text{ then } \text{Greater } (k x) \text{ else } \text{Intv } (\text{intv-const } (I x))$
defines $r' \equiv r \cup \{(x,y) . x \in Z \wedge y \in X \wedge \text{intv-const } (I x) < k x \wedge \text{isIntv } (I' y)\}$
assumes *finite* X
assumes *valid-region* $X k I r$
shows *valid-region* $X k I' r'$

$\langle \text{proof} \rangle$

lemma *closest-prestable-2:*

fixes $I X k r$
defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$
defines $R \equiv \text{region } X I r$
assumes $\forall x \in X. \neg \text{isConst } (I x)$
defines $X_0 \equiv \{x \in X. \text{isIntv } (I x)\}$
defines $M \equiv \{x \in X_0. \forall y \in X_0. (x, y) \in r \longrightarrow (y, x) \in r\}$
defines $I' \equiv \lambda x. \text{if } x \notin M \text{ then } I x \text{ else } \text{Const } (\text{intv-const } (I x) + 1)$
defines $r' \equiv \{(x,y) \in r. x \notin M \wedge y \notin M\}$
assumes *finite* X
assumes *valid-region* $X k I r$
assumes $M \neq \{\}$
shows $\forall v \in R. \forall t \geq 0. (v \oplus t) \notin R \longrightarrow (\exists t' \leq t. (v \oplus t') \in \text{region } X I' r' \wedge t' \geq 0)$
and $\forall v \in \text{region } X I' r'. \forall t \geq 0. (v \oplus t) \notin R$
and $\forall v \in R. \forall t'. \{x. x \in X \wedge (\exists c. I' x = \text{Intv } c \wedge (v \oplus t') x + (t - t') \geq \text{real } (c + 1))\}$
 $= \{x. x \in X \wedge (\exists c. I x = \text{Intv } c \wedge v x + t \geq \text{real } (c + 1))\} - M$
and $\exists x \in X. \text{isConst } (I' x)$

$\langle \text{proof} \rangle$

lemma *closest-valid-2:*

fixes $I X k r$
defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

```

defines  $R \equiv \text{region } X \ I \ r$ 
assumes  $\forall x \in X. \neg \text{isConst } (I \ x)$ 
defines  $X_0 \equiv \{x \in X. \text{isIntv } (I \ x)\}$ 
defines  $M \equiv \{x \in X_0. \forall y \in X_0. (x, y) \in r \longrightarrow (y, x) \in r\}$ 
defines  $I' \equiv \lambda x. \text{if } x \notin M \text{ then } I \ x \text{ else } \text{Const } (\text{intv-const } (I \ x) + 1)$ 
defines  $r' \equiv \{(x, y) \in r. x \notin M \wedge y \notin M\}$ 
assumes finite  $X$ 
assumes valid-region  $X \ k \ I \ r$ 
assumes  $M \neq \{\}$ 
shows valid-region  $X \ k \ I' \ r'$ 
<proof>

```

5.3.1 Putting the Proof for the 'Set of Regions' Property Together

Misc lemma *total-finite-trans-max*:

```

 $X \neq \{\} \implies \text{finite } X \implies \text{total-on } X \ r \implies \text{trans } r \implies \exists x \in X. \forall y \in X. x \neq y \longrightarrow (y, x) \in r$ 
<proof>

```

lemma *card-mono-strict-subset*:

```

 $\text{finite } A \implies \text{finite } B \implies \text{finite } C \implies A \cap B \neq \{\} \implies C = A - B \implies \text{card } C < \text{card } A$ 
<proof>

```

Proof First we show that a shift by a non-negative integer constant means that any two valuations from the same region are being shifted to the same region.

lemma *int-shift-equiv*:

```

fixes  $X \ k$  fixes  $t :: \text{int}$ 
defines  $\mathcal{R} \equiv \{\text{region } X \ I \ r \mid I \ r. \text{valid-region } X \ k \ I \ r\}$ 
assumes  $v \in R \ v' \in R \ R \in \mathcal{R} \ t \geq 0$ 
shows  $(v' \oplus t) \in [v \oplus t]_{\mathcal{R}}$  <proof>

```

Now, we can use the 'immediate' induction proposed by P. Bouyer for shifts smaller than one. The induction principle is not at all obvious: the induction is over the set of clocks for which the valuation is shifted beyond the current interval boundaries. Using the two successor operations, we can see that either the set of these clocks remains the same ($Z =$) or strictly decreases ($Z =$).

lemma *set-of-regions-lt-1*:

```

fixes  $X \ k \ I \ r \ t \ v$ 
defines  $\mathcal{R} \equiv \{\text{region } X \ I \ r \mid I \ r. \text{valid-region } X \ k \ I \ r\}$ 

```

defines $C \equiv \{x. x \in X \wedge (\exists c. I x = Intv\ c \wedge v\ x + t \geq c + 1)\}$
assumes *valid-region* $X\ k\ I\ r\ v \in region\ X\ I\ r\ v' \in region\ X\ I\ r\ finite\ X$
 $0 \leq t\ t < 1$
shows $\exists t' \geq 0. (v' \oplus t') \in [v \oplus t]_{\mathcal{R}}$ *<proof>*

Finally, we can put the two pieces together: for a non-negative shift t , we first shift $\lfloor t \rfloor$ and then *frac* t .

lemma *set-of-regions*:

fixes $X\ k$
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid-region\ X\ k\ I\ r\}$
assumes $R \in \mathcal{R}\ v \in R\ R' \in Succ\ \mathcal{R}\ R\ finite\ X$
shows $\exists t \geq 0. [v \oplus t]_{\mathcal{R}} = R'$ *<proof>*

5.4 Compability With Clock Constraints

definition *ccval* ($\langle \{-\} \rangle [100]$) **where** $ccval\ cc \equiv \{v. v \vdash cc\}$

definition *acompatible*

where

acompatible $\mathcal{R}\ ac \equiv \forall R \in \mathcal{R}. R \subseteq \{v. v \vdash_a ac\} \vee \{v. v \vdash_a ac\} \cap R = \{\}$

lemma *acompatibleD*:

assumes *acompatible* $\mathcal{R}\ ac\ R \in \mathcal{R}\ u \in R\ v \in R\ u \vdash_a ac$
shows $v \vdash_a ac$
<proof>

lemma *ccompatible1*:

fixes $X\ k$ **fixes** $c :: real$
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid-region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *acompatible* $\mathcal{R}\ (EQ\ x\ c)$ *<proof>*

lemma *ccompatible2*:

fixes $X\ k$ **fixes** $c :: real$
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid-region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *acompatible* $\mathcal{R}\ (LT\ x\ c)$ *<proof>*

lemma *ccompatible3*:

fixes $X\ k$ **fixes** $c :: real$
defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid-region\ X\ k\ I\ r\}$
assumes $c \leq k\ x\ c \in \mathbb{N}\ x \in X$
shows *acompatible* $\mathcal{R}\ (LE\ x\ c)$ *<proof>*

lemma *ccompatible4*:

fixes $X k$ **fixes** $c :: \text{real}$

defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

assumes $c \leq k$ $x c \in \mathbb{N}$ $x \in X$

shows *acompatible* \mathcal{R} $(GT\ x\ c)$ $\langle \text{proof} \rangle$

lemma *ccompatible5*:

fixes $X k$ **fixes** $c :: \text{real}$

defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

assumes $c \leq k$ $x c \in \mathbb{N}$ $x \in X$

shows *acompatible* \mathcal{R} $(GE\ x\ c)$ $\langle \text{proof} \rangle$

lemma *acompatible*:

fixes $X k$ **fixes** $c :: \text{real}$

defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

assumes $c \leq k$ $x c \in \mathbb{N}$ $x \in X$ *constraint-pair* $ac = (x, c)$

shows *acompatible* \mathcal{R} ac $\langle \text{proof} \rangle$

definition *ccompatible*

where

ccompatible $\mathcal{R}\ cc \equiv \forall R \in \mathcal{R}. R \subseteq \{\{cc\} \vee \{cc\} \cap R = \{\}$

lemma *ccompatible*:

fixes $X k$ **fixes** $c :: \text{nat}$

defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

assumes $\forall (x, m) \in \text{collect-clock-pairs } cc. m \leq k\ x \wedge x \in X \wedge m \in \mathbb{N}$

shows *ccompatible* $\mathcal{R}\ cc$ $\langle \text{proof} \rangle$

5.5 Compability with Resets

definition *region-set*

where

region-set $R\ x\ c = \{v(x := c) \mid v. v \in R\}$

lemma *region-set-id*:

fixes $X k$

defines $\mathcal{R} \equiv \{\text{region } X I r \mid I r. \text{valid-region } X k I r\}$

assumes $R \in \mathcal{R}$ $v \in R$ *finite* X $0 \leq c$ $c \leq k$ $x \in X$

shows $[v(x := c)]_{\mathcal{R}} = \text{region-set } R\ x\ c$ $[v(x := c)]_{\mathcal{R}} \in \mathcal{R}$ $v(x := c) \in [v(x := c)]_{\mathcal{R}}$
 $\langle \text{proof} \rangle$

definition *region-set'*

where

$region\text{-set}' R r c = \{[r \rightarrow c]v \mid v. v \in R\}$

lemma *region-set'-id*:

fixes $X k$ **and** $c :: nat$

defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid\text{-region}\ X\ k\ I\ r\}$

assumes $R \in \mathcal{R}\ v \in R\ finite\ X\ 0 \leq c\ \forall\ x \in set\ r. c \leq k\ x\ set\ r \subseteq X$

shows $[[r \rightarrow c]v]_{\mathcal{R}} = region\text{-set}'\ R\ r\ c \wedge [[r \rightarrow c]v]_{\mathcal{R}} \in \mathcal{R} \wedge [r \rightarrow c]v \in [[r \rightarrow c]v]_{\mathcal{R}}$ *<proof>*

This is the only additional lemma necessary to make local α -closures work.

lemma *region-set-subst*:

fixes $X k k'$ **and** $c :: nat$

defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r. valid\text{-region}\ X\ k\ I\ r\}$

defines $\mathcal{R}' \equiv \{region\ X\ I\ r \mid I\ r. valid\text{-region}\ X\ k'\ I\ r\}$

assumes $R \in \mathcal{R}\ v \in R\ finite\ X\ 0 \leq c\ set\ cs \subseteq X\ \forall\ y. y \notin set\ cs \longrightarrow k\ y \geq k'\ y$

shows $[[cs \rightarrow c]v]_{\mathcal{R}'} \supseteq region\text{-set}'\ R\ cs\ c\ [[cs \rightarrow c]v]_{\mathcal{R}'} \in \mathcal{R}'\ [cs \rightarrow c]v \in [[cs \rightarrow c]v]_{\mathcal{R}'}$ *<proof>*

5.6 A Semantics Based on Regions

5.6.1 Single step

inductive *step-r* ::

$(\langle a, 'c, t, 's \rangle ta \Rightarrow \langle 'c, t \rangle zone\ set \Rightarrow 's \Rightarrow \langle 'c, t \rangle zone \Rightarrow 's \Rightarrow \langle 'c, t \rangle zone \Rightarrow bool$

$\langle \langle -, - \rangle \vdash \langle -, - \rangle \rightsquigarrow \langle -, - \rangle \rangle [61,61,61,61] 61$

where

step-t-r:

$\llbracket \mathcal{R} = \{region\ X\ I\ r \mid I\ r. valid\text{-region}\ X\ k\ I\ r\}; valid\text{-abstraction}\ A\ X\ k; R \in \mathcal{R}; R' \in Succ\ \mathcal{R}\ R;$

$R' \subseteq \{\!\{inv\text{-of}\ A\ l\}\!\} \implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l, R' \rangle \mid$

step-a-r:

$\llbracket \mathcal{R} = \{region\ X\ I\ r \mid I\ r. valid\text{-region}\ X\ k\ I\ r\}; valid\text{-abstraction}\ A\ X\ k; A \vdash l \longrightarrow^{g,a,r} l'; R \in \mathcal{R} \rrbracket$

$\implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l', region\text{-set}'\ (R \cap \{u. u \vdash g\})\ r\ 0 \cap \{u. u \vdash inv\text{-of}\ A\ l'\} \rangle$

inductive-cases[*elim!*]: $A, \mathcal{R} \vdash \langle l, u \rangle \rightsquigarrow \langle l', u' \rangle$

declare *step-r.intros*[*intro*]

lemma *region-cover'*:

assumes $\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\}$ **and** $\forall x \in X. 0 \leq v \ x$
shows $v \in [v]_{\mathcal{R}} \ [v]_{\mathcal{R}} \in \mathcal{R}$
 $\langle \text{proof} \rangle$

lemma *step-r-complete-aux*:

fixes $R \ r \ A \ l' \ g$
defines $R' \equiv \text{region-set}' (R \cap \{u. u \vdash g\}) \ r \ 0 \cap \{u. u \vdash \text{inv-of } A \ l'\}$
assumes $\mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\}$
and *valid-abstraction* $A \ X \ k$
and $u \in R$
and $R \in \mathcal{R}$
and $A \vdash l \longrightarrow^{g,a,r} l'$
and $u \vdash g$
and $[r \rightarrow 0]u \vdash \text{inv-of } A \ l'$
shows $R = R \cap \{u. u \vdash g\} \wedge R' = \text{region-set}' R \ r \ 0 \wedge R' \in \mathcal{R}$
 $\langle \text{proof} \rangle$

lemma *step-r-complete*:

$\llbracket A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle; \mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\};$
valid-abstraction $A \ X \ k;$
 $\forall x \in X. u \ x \geq 0 \rrbracket \implies \exists R'. A, \mathcal{R} \vdash \langle l, ([u]_{\mathcal{R}}) \rangle \rightsquigarrow \langle l', R' \rangle \wedge u' \in R' \wedge R' \in \mathcal{R}$
 $\langle \text{proof} \rangle$

Compare this to lemma *step-z-sound*. This version is weaker because for regions we may very well arrive at a successor for which not every valuation can be reached by the predecessor. This is the case for e.g. the region with only Greater (k x) bounds.

lemma *step-r-sound*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l', R' \rangle \implies \mathcal{R} = \{\text{region } X \ I \ r \mid I \ r. \text{ valid-region } X \ k \ I \ r\}$
 $\implies R' \neq \{\}$ $\implies (\forall u \in R. \exists u' \in R'. A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle)$
 $\langle \text{proof} \rangle$

5.6.2 Multi Step

inductive

steps-r :: $(l'a, l'c, t, l's) \text{ ta} \Rightarrow (l'c, t) \text{ zone set} \Rightarrow l's \Rightarrow (l'c, t) \text{ zone} \Rightarrow l's \Rightarrow (l'c, t) \text{ zone} \Rightarrow \text{bool}$
 $(\langle -, - \rangle \vdash \langle -, - \rangle \rightsquigarrow^* \langle -, - \rangle) [61,61,61,61,61,61] \ 61)$

where

refl: $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l, R \rangle \mid$
step: $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l', R' \rangle \implies A, \mathcal{R} \vdash \langle l', R' \rangle \rightsquigarrow \langle l'', R'' \rangle \implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l'', R'' \rangle$

declare *steps-r.intros*[*intro*]

lemma *steps-alt*:

$A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle \implies A \vdash \langle l', u' \rangle \rightarrow \langle l'', u'' \rangle \implies A \vdash \langle l, u \rangle \rightarrow^* \langle l'', u'' \rangle$
 $\langle \text{proof} \rangle$

lemma *emptiness-preservation*: $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l', R' \rangle \implies R = \{\} \implies R' = \{\}$
 $\langle \text{proof} \rangle$

lemma *emptiness-preservation-steps*: $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l', R' \rangle \implies R = \{\} \implies R' = \{\}$
 $\langle \text{proof} \rangle$

Note how it is important to define the multi-step semantics “the right way round”. This is also the direction Bouyer implies for her implicit induction.

lemma *steps-r-sound*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l', R' \rangle \implies \mathcal{R} = \{\text{region } X \text{ } I \text{ } r \mid I \text{ } r. \text{ valid-region } X \text{ } k \text{ } I \text{ } r\}$
 $\implies R' \neq \{\} \implies u \in R \implies \exists u' \in R'. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *steps-r-sound'*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l', R' \rangle \implies \mathcal{R} = \{\text{region } X \text{ } I \text{ } r \mid I \text{ } r. \text{ valid-region } X \text{ } k \text{ } I \text{ } r\}$
 $\implies R' \neq \{\} \implies (\exists u' \in R'. \exists u \in R. A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle)$
 $\langle \text{proof} \rangle$

lemma *single-step-r*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l', R' \rangle \implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l', R' \rangle$
 $\langle \text{proof} \rangle$

lemma *steps-r-alt*:

$A, \mathcal{R} \vdash \langle l', R' \rangle \rightsquigarrow^* \langle l'', R'' \rangle \implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow \langle l', R' \rangle \implies A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^* \langle l'', R'' \rangle$
 $\langle \text{proof} \rangle$

lemma *single-step*:

$x1 \vdash \langle x2, x3 \rangle \rightarrow \langle x4, x5 \rangle \implies x1 \vdash \langle x2, x3 \rangle \rightarrow^* \langle x4, x5 \rangle$
 $\langle \text{proof} \rangle$

lemma *steps-r-complete*:

$\llbracket A \vdash \langle l, u \rangle \rightarrow^* \langle l', u' \rangle; \mathcal{R} = \{\text{region } X \text{ } I \text{ } r \mid I \text{ } r. \text{ valid-region } X \text{ } k \text{ } I \text{ } r\};$

valid-abstraction $A \ X \ k;$
 $\forall x \in X. u \ x \geq 0 \implies \exists R'. A, \mathcal{R} \vdash \langle l, ([u]_{\mathcal{R}}) \rangle \rightsquigarrow^* \langle l', R' \rangle \wedge u' \in R'$
 $\langle \text{proof} \rangle$

end
theory *Closure*
 imports *Regions*
begin

5.7 Correct Approximation of Zones with α -regions

lemma *subset-int-mono*: $A \subseteq B \implies A \cap C \subseteq B \cap C$ $\langle \text{proof} \rangle$

lemma *zone-set-mono*:
 $A \subseteq B \implies \text{zone-set } A \ r \subseteq \text{zone-set } B \ r$
 $\langle \text{proof} \rangle$

lemma *zone-delay-mono*:
 $A \subseteq B \implies A^\uparrow \subseteq B^\uparrow$
 $\langle \text{proof} \rangle$

lemma *step-z-mono*:
 $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Z \subseteq W \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_a \langle l', W' \rangle \wedge Z' \subseteq W'$
 $\langle \text{proof} \rangle$

5.8 Old Variant Using a Global Set of Regions

Shared Definitions for Local and Global Sets of Regions *locale*
Alpha-defs =
 fixes $X :: 'c \ \text{set}$
begin

definition $V :: ('c, t) \ \text{cval set}$ **where** $V \equiv \{v. \forall x \in X. v \ x \geq 0\}$

lemma *up-V*: $Z \subseteq V \implies Z^\uparrow \subseteq V$
 $\langle \text{proof} \rangle$

lemma *reset-V*: $Z \subseteq V \implies (\text{zone-set } Z \ r) \subseteq V$
 $\langle \text{proof} \rangle$

lemma *step-z-V*: $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \subseteq V$

<proof>

end

This is the classic variant using a global clock ceiling k and thus a global set of regions. It is also the version that is necessary to prove the classic extrapolation correct. It is preserved here for comparison with P. Bouyer's proofs and to outline the only slight adoptions that are necessary to obtain the new version.

locale *AlphaClosure-global* =

Alpha-defs X **for** $X :: 'c$ *set* +

fixes k \mathcal{R}

defines $\mathcal{R} \equiv \{region\ X\ I\ r \mid I\ r.\ valid-region\ X\ k\ I\ r\}$

assumes *finite*: *finite* X

begin

lemmas *set-of-regions-spec* = *set-of-regions*[*OF* - - - *finite*, *of* - k , *folded* \mathcal{R} -*def*]

lemmas *region-cover-spec* = *region-cover*[*of* X - k , *folded* \mathcal{R} -*def*]

lemmas *region-unique-spec* = *region-unique*[*of* \mathcal{R} X k , *folded* \mathcal{R} -*def*, *simplified*]

lemmas *regions-closed'-spec* = *regions-closed'*[*of* \mathcal{R} X k , *folded* \mathcal{R} -*def*, *simplified*]

lemma *valid-regions-distinct-spec*:

$R \in \mathcal{R} \implies R' \in \mathcal{R} \implies v \in R \implies v \in R' \implies R = R'$

<proof>

definition *cla* ($\langle Closure_\alpha \rightarrow [71] 71 \rangle$)

where

$cla\ Z = \bigcup \{R \in \mathcal{R}. R \cap Z \neq \{\}\}$

The Nice and Easy Properties Proved by Bouyer **lemma** *closure-constraint-id*:

$\forall (x, m) \in collect-clock-pairs\ g.\ m \leq real\ (k\ x) \wedge x \in X \wedge m \in \mathbf{N} \implies Closure_\alpha\ \{g\} = \{g\} \cap V$

<proof>

lemma *closure-id'*:

$Z \neq \{\} \implies Z \subseteq R \implies R \in \mathcal{R} \implies Closure_\alpha\ Z = R$

<proof>

lemma *closure-id*:

$Closure_\alpha\ Z \neq \{\} \implies Z \subseteq R \implies R \in \mathcal{R} \implies Closure_\alpha\ Z = R$

$\langle \text{proof} \rangle$

lemma *closure-update-mono*:

$Z \subseteq V \implies \text{set } r \subseteq X \implies \text{zone-set } (\text{Closure}_\alpha Z) r \subseteq \text{Closure}_\alpha(\text{zone-set } Z r)$

$\langle \text{proof} \rangle$

lemma *SuccI3*:

$R \in \mathcal{R} \implies v \in R \implies t \geq 0 \implies (v \oplus t) \in R' \implies R' \in \mathcal{R} \implies R' \in \text{Succ } \mathcal{R} R$

$\langle \text{proof} \rangle$

lemma *closure-delay-mono*:

$Z \subseteq V \implies (\text{Closure}_\alpha Z)^\dagger \subseteq \text{Closure}_\alpha (Z^\dagger)$

$\langle \text{proof} \rangle$

lemma *region-V*: $R \in \mathcal{R} \implies R \subseteq V$ $\langle \text{proof} \rangle$

lemma *closure-V*:

$\text{Closure}_\alpha Z \subseteq V$

$\langle \text{proof} \rangle$

lemma *closure-V-int*:

$\text{Closure}_\alpha Z = \text{Closure}_\alpha (Z \cap V)$

$\langle \text{proof} \rangle$

lemma *closure-constraint-mono*:

$\text{Closure}_\alpha g = g \implies g \cap (\text{Closure}_\alpha Z) \subseteq \text{Closure}_\alpha (g \cap Z)$

$\langle \text{proof} \rangle$

lemma *closure-constraint-mono'*:

assumes $\text{Closure}_\alpha g = g \cap V$

shows $g \cap (\text{Closure}_\alpha Z) \subseteq \text{Closure}_\alpha (g \cap Z)$

$\langle \text{proof} \rangle$

lemma *cla-empty-iff*:

$Z \subseteq V \implies Z = \{\} \iff \text{Closure}_\alpha Z = \{\}$

$\langle \text{proof} \rangle$

lemma *closure-involutive-aux*:

$U \subseteq \mathcal{R} \implies \text{Closure}_\alpha \bigcup U = \bigcup U$

$\langle \text{proof} \rangle$

lemma *closure-involutive-aux'*:

$\exists U. U \subseteq \mathcal{R} \wedge \text{Closure}_\alpha Z = \bigcup U$
 ⟨proof⟩

lemma *closure-involutive*:

$\text{Closure}_\alpha \text{Closure}_\alpha Z = \text{Closure}_\alpha Z$
 ⟨proof⟩

lemma *closure-involutive'*:

$Z \subseteq \text{Closure}_\alpha W \implies \text{Closure}_\alpha Z \subseteq \text{Closure}_\alpha W$
 ⟨proof⟩

lemma *closure-subst*:

$Z \subseteq V \implies Z \subseteq \text{Closure}_\alpha Z$
 ⟨proof⟩

lemma *cla-mono'*:

$Z' \subseteq V \implies Z \subseteq Z' \implies \text{Closure}_\alpha Z \subseteq \text{Closure}_\alpha Z'$
 ⟨proof⟩

lemma *cla-mono*:

$Z \subseteq Z' \implies \text{Closure}_\alpha Z \subseteq \text{Closure}_\alpha Z'$
 ⟨proof⟩

5.9 A Zone Semantics Abstracting with Closure_α

5.9.1 Single step

inductive *step-z-alpha* ::

$(\text{'a}, \text{'c}, t, \text{'s}) \text{ta} \Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow \text{'a} \text{action} \Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow$
bool

$(\text{'-} \vdash \langle -, - \rangle \rightsquigarrow_{\alpha(-)} \langle -, - \rangle) [61,61,61] \ 61)$

where

step-alpha: $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', \text{Closure}_\alpha Z' \rangle$

inductive-cases[*elim!*]: $A \vdash \langle l, u \rangle \rightsquigarrow_{\alpha(a)} \langle l', u' \rangle$

declare *step-z-alpha.intros*[*intro*]

definition

step-z-alpha' :: $(\text{'a}, \text{'c}, t, \text{'s}) \text{ta} \Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow$
bool

$(\text{'-} \vdash \langle -, - \rangle \rightsquigarrow_\alpha \langle -, - \rangle) [61,61,61] \ 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow_\alpha \langle l', Z'' \rangle = (\exists Z' a. A \vdash \langle l, Z \rangle \rightsquigarrow_\tau \langle l, Z' \rangle \wedge A \vdash \langle l, Z' \rangle)$

$\rightsquigarrow_{\alpha(1a)} \langle l', Z'' \rangle$

Single-step soundness and completeness follows trivially from *cla-empty-iff*.

lemma *step-z-alpha-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle$
 $\rightsquigarrow_a \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-z-alpha'-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow$
 $\langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-z-alpha-complete'*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Z \subseteq V \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z'' \rangle \wedge$
 $Z' \subseteq Z''$
 $\langle \text{proof} \rangle$

lemma *step-z-alpha-complete*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle$
 $\rightsquigarrow_{\alpha(a)} \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-z-alpha'-complete'*:

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \implies Z \subseteq V \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle \wedge Z'$
 $\subseteq Z''$
 $\langle \text{proof} \rangle$

lemma *step-z-alpha'-complete*:

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha}$
 $\langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 $\langle \text{proof} \rangle$

5.9.2 Multi step

abbreviation

steps-z-alpha :: (*'a*, *'c*, *t*, *'s*) *ta* \Rightarrow *'s* \Rightarrow (*'c*, *t*) *zone* \Rightarrow *'s* \Rightarrow (*'c*, *t*) *zone*
 \Rightarrow *bool*
 $(\langle \vdash \langle -, - \rangle \rightsquigarrow_{\alpha^*} \langle -, - \rangle \rangle [61,61,61] 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z'' \rangle \equiv (\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle)^{**}$
 $(l, Z) (l', Z'')$

P. Bouyer's calculation for $Post (Closure_{\alpha} Z, e) \subseteq Closure_{\alpha} Post (Z, e)$

This is now obsolete as we argue solely with monotonicity of *steps-z* w.r.t $Closure_\alpha$

lemma calc:

$$\begin{aligned} & \text{valid-abstraction } A \ X \ k \implies Z \subseteq V \implies A \vdash \langle l, Closure_\alpha Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \\ & \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z'' \rangle \wedge Z' \subseteq Z'' \\ & \langle \text{proof} \rangle \end{aligned}$$

Turning P. Bouyers argument for multiple steps into an inductive proof is not direct. With this initial argument we can get to a point where the induction hypothesis is applicable. This breaks the "information hiding" induced by the different variants of steps.

lemma steps-z-alpha-closure-involutive'-aux:

$$\begin{aligned} & A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Closure_\alpha Z \subseteq Closure_\alpha W \implies \text{valid-abstraction} \\ & A \ X \ k \implies Z \subseteq V \\ & \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_a \langle l', W' \rangle \wedge Closure_\alpha Z' \subseteq Closure_\alpha W' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma steps-z-alpha-closure-involutive'-aux':

$$\begin{aligned} & A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Closure_\alpha Z \subseteq Closure_\alpha W \implies \text{valid-abstraction} \\ & A \ X \ k \implies Z \subseteq V \implies W \subseteq Z \\ & \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_a \langle l', W' \rangle \wedge Closure_\alpha Z' \subseteq Closure_\alpha W' \wedge W' \\ & \subseteq Z' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma steps-z-alpha-V: $A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \subseteq V$
 $\langle \text{proof} \rangle$

lemma steps-z-alpha-closure-involutive':

$$\begin{aligned} & A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z' \rangle \implies A \vdash \langle l', Z' \rangle \rightsquigarrow_\tau \langle l', Z'' \rangle \implies A \vdash \langle l', Z'' \rangle \rightsquigarrow_{1a} \\ & \langle l'', Z''' \rangle \\ & \implies \text{valid-abstraction } A \ X \ k \implies Z \subseteq V \\ & \implies \exists W'''. A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l'', W''' \rangle \wedge Closure_\alpha Z''' \subseteq Closure_\alpha W''' \wedge \\ & W''' \subseteq Z''' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma steps-z-alpha-closure-involutive:

$$\begin{aligned} & A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies Z \subseteq V \\ & \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z'' \rangle \wedge Closure_\alpha Z' \subseteq Closure_\alpha Z'' \wedge Z'' \subseteq \\ & Z' \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *steps-z-V*:

$$A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \subseteq V$$

<proof>

lemma *steps-z-alpha-sound*:

$$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies Z \subseteq V \implies Z' \neq \{\}$$

$$\implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z'' \rangle \wedge Z'' \neq \{\} \wedge Z'' \subseteq Z'$$

<proof>

lemma *step-z-alpha-mono*:

$$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z' \rangle \implies Z \subseteq W \implies W \subseteq V \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_{\alpha(a)} \langle l', W' \rangle \wedge Z' \subseteq W'$$

<proof>

end

5.10 New Variant

New Definitions `hide-const collect-clkt collect-clki clkp-set valid-abstraction`

definition *collect-clkt* :: ('a, 'c, 't, 's) transition set \Rightarrow 's \Rightarrow ('c *'t) set

where

$$\text{collect-clkt } S \ l = \bigcup \{ \text{collect-clock-pairs } (\text{fst } (\text{snd } t)) \mid t . t \in S \wedge \text{fst } t = l \}$$

definition *collect-clki* :: ('c, 't, 's) invassn \Rightarrow 's \Rightarrow ('c *'t) set

where

$$\text{collect-clki } I \ s = \text{collect-clock-pairs } (I \ s)$$

definition *clkp-set* :: ('a, 'c, 't, 's) ta \Rightarrow 's \Rightarrow ('c *'t) set

where

$$\text{clkp-set } A \ s = \text{collect-clki } (\text{inv-of } A) \ s \cup \text{collect-clkt } (\text{trans-of } A) \ s$$

lemma *collect-clkt-alt-def*:

$$\text{collect-clkt } S \ l = \bigcup (\text{collect-clock-pairs } \text{'(fst o snd) ' } \{ t . t \in S \wedge \text{fst } t = l \})$$

<proof>

inductive *valid-abstraction*

where

$$\llbracket \forall l. \forall (x, m) \in \text{clkp-set } A \ l. m \leq k \ l \ x \wedge x \in X \wedge m \in \mathbb{N}; \text{collect-clkt} \rrbracket$$

$(\text{trans-of } A) \subseteq X; \text{ finite } X;$
 $\forall l \text{ g a r } l' c. A \vdash l \xrightarrow{g,a,r} l' \wedge c \notin \text{set } r \longrightarrow k l' c \leq k l c$
 \Downarrow
 $\implies \text{valid-abstraction } A X k$

locale *AlphaClosure* =
Alpha-defs X **for** $X :: 'c \text{ set} +$
fixes $k :: 's \Rightarrow 'c \Rightarrow \text{nat}$ **and** \mathcal{R}
defines $\mathcal{R} l \equiv \{\text{region } X I r \mid I r. \text{valid-region } X (k l) I r\}$
assumes *finite*: *finite* X
begin

5.11 A Semantics Based on Localized Regions

5.11.1 Single step

inductive *step-r* ::
 $('a, 'c, t, 's) \text{ ta} \Rightarrow - \Rightarrow 's \Rightarrow ('c, t) \text{ zone} \Rightarrow 'a \text{ action} \Rightarrow 's \Rightarrow ('c, t) \text{ zone}$
 $\Rightarrow \text{bool}$

$(\langle -, - \vdash \langle -, - \rangle \rightsquigarrow_{-} \langle -, - \rangle) [61,61,61,61,61] 61)$

where

step-t-r:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{\tau} \langle l, R' \rangle$ **if**

$\text{valid-abstraction } A X (\lambda x. \text{real } o k x) R \in \mathcal{R} l R' \in \text{Succ } (\mathcal{R} l) R R' \subseteq$
 $\{\text{inv-of } A l\} \mid$

step-a-r:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{1a} \langle l', R' \rangle$ **if**

$\text{valid-abstraction } A X (\lambda x. \text{real } o k x) A \vdash l \xrightarrow{g,a,r} l' R \in \mathcal{R} l$

$R \subseteq \{\text{g}\} \text{ region-set}' R r 0 \subseteq R' R' \subseteq \{\text{inv-of } A l'\} R' \in \mathcal{R} l'$

inductive-cases[*elim!*]: $A, \mathcal{R} \vdash \langle l, u \rangle \rightsquigarrow_a \langle l', u' \rangle$

declare *step-r.intros*[*intro*]

inductive *step-r'* ::

$('a, 'c, t, 's) \text{ ta} \Rightarrow - \Rightarrow 's \Rightarrow ('c, t) \text{ zone} \Rightarrow 'a \Rightarrow 's \Rightarrow ('c, t) \text{ zone} \Rightarrow \text{bool}$
 $(\langle -, - \vdash \langle -, - \rangle \rightsquigarrow_{-} \langle -, - \rangle) [61,61,61,61,61] 61)$

where

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R'' \rangle$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{\tau} \langle l, R' \rangle$ $A, \mathcal{R} \vdash \langle l, R' \rangle \rightsquigarrow_{1a} \langle l', R'' \rangle$

lemmas $\mathcal{R}\text{-def}' = \text{meta-eq-to-obj-eq}[OF \mathcal{R}\text{-def}]$

lemmas $\text{region-cover}' = \text{region-cover}'[OF \mathcal{R}\text{-def}]$

abbreviation $\text{part}'' (\langle [-] \rangle [61, 61] 61)$ **where** $\text{part}'' u l1 \equiv \text{part } u (\mathcal{R} l1)$
no-notation $\text{part} (\langle [-] \rangle [61, 61] 61)$

lemma *step-r-complete-aux*:

fixes $R u r A l' g$
defines $R' \equiv [[r \rightarrow 0]u]_{l'}$
assumes *valid-abstraction* $A X (\lambda x. \text{real } o k x)$
and $u \in R$
and $R \in \mathcal{R} l$
and $A \vdash l \xrightarrow{g, a, r} l'$
and $u \vdash g$
and $[r \rightarrow 0]u \vdash \text{inv-of } A l'$
shows $R = R \cap \{u. u \vdash g\} \wedge \text{region-set}' R r 0 \subseteq R' \wedge R' \in \mathcal{R} l' \wedge R' \subseteq \{\text{inv-of } A l'\}$
 $\langle \text{proof} \rangle$

lemma *step-t-r-complete*:

assumes
 $A \vdash \langle l, u \rangle \xrightarrow{d} \langle l', u' \rangle$ *valid-abstraction* $A X (\lambda x. \text{real } o k x) \forall x \in X. u$
 $x \geq 0$
shows $\exists R'. A, \mathcal{R} \vdash \langle l, ([u]_l) \rangle \rightsquigarrow_{\tau} \langle l', R' \rangle \wedge u' \in R' \wedge R' \in \mathcal{R} l'$
 $\langle \text{proof} \rangle$

lemma *step-a-r-complete*:

assumes
 $A \vdash \langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ *valid-abstraction* $A X (\lambda x. \text{real } o k x) \forall x \in X. u$
 $x \geq 0$
shows $\exists R'. A, \mathcal{R} \vdash \langle l, ([u]_l) \rangle \rightsquigarrow_{1a} \langle l', R' \rangle \wedge u' \in R' \wedge R' \in \mathcal{R} l'$
 $\langle \text{proof} \rangle$

lemma *step-r-complete*:

assumes
 $A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$ *valid-abstraction* $A X (\lambda x. \text{real } o k x) \forall x \in X. u$
 $x \geq 0$
shows $\exists R' a. A, \mathcal{R} \vdash \langle l, ([u]_l) \rangle \rightsquigarrow_a \langle l', R' \rangle \wedge u' \in R' \wedge R' \in \mathcal{R} l'$
 $\langle \text{proof} \rangle$

Compare this to lemma *step-z-sound*. This version is weaker because for regions we may very well arrive at a successor for which not every valuation can be reached by the predecessor. This is the case for e.g. the region with only Greater (k x) bounds.

lemma *step-t-r-sound*:

assumes $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{\tau} \langle l', R' \rangle$
shows $\forall u \in R. \exists u' \in R'. \exists d \geq 0. A \vdash \langle l, u \rangle \xrightarrow{d} \langle l', u' \rangle$

$\langle \text{proof} \rangle$

lemma *step-a-r-sound*:

assumes $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{1a} \langle l', R' \rangle$

shows $\forall u \in R. \exists u' \in R'. A \vdash \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$

$\langle \text{proof} \rangle$

lemma *step-r-sound*:

assumes $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$

shows $\forall u \in R. \exists u' \in R'. A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle$

$\langle \text{proof} \rangle$

lemma *step-r'-sound*:

assumes $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$

shows $\forall u \in R. \exists u' \in R'. A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$

$\langle \text{proof} \rangle$

5.12 A New Zone Semantics Abstracting with $Closure_{\alpha, l}$

definition *cla* ($\langle Closure_{\alpha, -} \rangle$) [71, 71] 71)

where

$cla \ l \ Z = \bigcup \{ R \in \mathcal{R} \mid R \cap Z \neq \{\} \}$

5.12.1 Single step

inductive *step-z-alpha* ::

$('a, 'c, t, 's) \ ta \Rightarrow 's \Rightarrow ('c, t) \ zone \Rightarrow 'a \ action \Rightarrow 's \Rightarrow ('c, t) \ zone \Rightarrow$
 $bool$

$(\vdash \langle -, - \rangle \rightsquigarrow_{\alpha(-)} \langle -, - \rangle)$ [61, 61, 61] 61)

where

step-alpha: $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \Longrightarrow A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Closure_{\alpha, l'} Z' \rangle$

inductive-cases[*elim!*]: $A \vdash \langle l, u \rangle \rightsquigarrow_{\alpha(a)} \langle l', u' \rangle$

declare *step-z-alpha.intros*[*intro*]

Single-step soundness and completeness follows trivially from *cla-empty-iff*.

lemma *step-z-alpha-sound*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z' \rangle \Longrightarrow Z \subseteq V \Longrightarrow Z' \neq \{\}$

$\Longrightarrow \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z'' \rangle \wedge Z'' \neq \{\}$

$\langle \text{proof} \rangle$

context

fixes $l\ l' :: 's$
begin

interpretation $alpha$: *AlphaClosure-global - k l' R l' <proof>*

lemma [*simp*]:
 $alpha.cla = cla\ l'$
<proof>

lemma *step-z-alpha-complete*:
 $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies Z \subseteq V \implies Z' \neq \{\}$
 $\implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
<proof>

end

5.12.2 Multi step

definition

$step\text{-}z\text{-}alpha' :: ('a, 'c, t, 's) ta \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow$
 $bool$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{\alpha} \langle -, - \rangle) [61,61,61] 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle = (\exists Z' a. A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l, Z' \rangle \wedge A \vdash \langle l, Z' \rangle$
 $\rightsquigarrow_{\alpha(1a)} \langle l', Z'' \rangle)$

abbreviation

$steps\text{-}z\text{-}alpha :: ('a, 'c, t, 's) ta \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow 's \Rightarrow ('c, t) zone$
 $\Rightarrow bool$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{\alpha^*} \langle -, - \rangle) [61,61,61] 61)$

where

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z'' \rangle \equiv (\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle)^{**}$
 $(l, Z) (l', Z'')$

P. Bouyer's calculation for $Post(Closure_{\alpha,l} Z, e) \subseteq Closure_{\alpha,l}(Post(Z, e))$

This is now obsolete as we argue solely with monotonicity of *steps-z* w.r.t $Closure_{\alpha,l}$

Turning P. Bouyer's argument for multiple steps into an inductive proof is not direct. With this initial argument we can get to a point where the induction hypothesis is applicable. This breaks the "information hiding" induced by the different variants of steps.

context

fixes $l\ l' :: 's$

begin

interpretation *alpha*: *AlphaClosure-global - k l R l* \langle *proof* \rangle

lemma [*simp*]: *alpha.cla = cla l* \langle *proof* \rangle

interpretation *alpha'*: *AlphaClosure-global - k l' R l'* \langle *proof* \rangle

lemma [*simp*]: *alpha'.cla = cla l'* \langle *proof* \rangle

lemma *steps-z-alpha-closure-involutive'-aux'*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies \text{Closure}_{\alpha, l} Z \subseteq \text{Closure}_{\alpha, l} W \implies \text{valid-abstraction}$

$A \ X \ k \implies Z \subseteq V$

$\implies W \subseteq Z \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_a \langle l', W' \rangle \wedge \text{Closure}_{\alpha, l'} Z' \subseteq$

$\text{Closure}_{\alpha, l'} W' \wedge W' \subseteq Z'$

\langle *proof* \rangle

end

lemma *step-z-alpha-mono*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z' \rangle \implies Z \subseteq W \implies W \subseteq V \implies \exists W'. A \vdash \langle l, W \rangle$

$\rightsquigarrow_{\alpha(a)} \langle l', W' \rangle \wedge Z' \subseteq W'$

\langle *proof* \rangle

end

end

theory *Approx-Beta*

imports *DBM-Zone-Semantics Regions-Beta Closure*

begin

no-notation *infinity* ($\langle \infty \rangle$)

6 Correctness of β -approximation from α -regions

Merging the locales for the two types of regions

locale *Regions-defs* =

Alpha-defs **for** $X :: 'c \text{ set}+$

fixes $v :: 'c \Rightarrow \text{nat}$ **and** $n :: \text{nat}$

begin

abbreviation $vabstr :: ('c, t) \text{ zone} \Rightarrow - \Rightarrow -$ **where**

$vabstr S M \equiv S = [M]_{v,n} \wedge (\forall i \leq n. \forall j \leq n. M i j \neq \infty \longrightarrow \text{get-const } (M i j) \in \mathbb{Z})$

definition $V' \equiv \{Z. Z \subseteq V \wedge (\exists M. vabstr Z M)\}$

end

locale $Regions\text{-}global =$

$Regions\text{-}defs X v n$ **for** $X :: 'c \text{ set}$ **and** $v n +$

fixes $k :: 'c \Rightarrow \text{nat}$ **and** $not\text{-}in\text{-}X$

assumes $finite: finite X$

assumes $clock\text{-}numbering: clock\text{-}numbering' v n \forall k \leq n. k > 0 \longrightarrow (\exists c \in X. v c = k)$

$\forall c \in X. v c \leq n$

assumes $not\text{-}in\text{-}X: not\text{-}in\text{-}X \notin X$

assumes $non\text{-}empty: X \neq \{\}$

begin

definition $\mathcal{R}\text{-}def: \mathcal{R} \equiv \{Regions.region X I r \mid I r. Regions.valid\text{-}region X k I r\}$

sublocale $alpha\text{-}interp:$

$AlphaClosure\text{-}global X k \mathcal{R} \langle proof \rangle$

sublocale $beta\text{-}interp: Beta\text{-}Regions' X k v n not\text{-}in\text{-}X$

rewrites $beta\text{-}interp.V = V$

$\langle proof \rangle$

abbreviation \mathcal{R}_β **where** $\mathcal{R}_\beta \equiv beta\text{-}interp.\mathcal{R}$

lemmas $\mathcal{R}_\beta\text{-}def = beta\text{-}interp.\mathcal{R}\text{-}def$

abbreviation $Approx_\beta \equiv beta\text{-}interp.Approx_\beta$

6.1 Preparing Bouyer's Theorem

lemma $region\text{-}dbm:$

assumes $R \in \mathcal{R}$

defines $v' \equiv \lambda i. THE c. c \in X \wedge v c = i$

obtains M

where $[M]_{v,n} = R$

and $\forall i \leq n. \forall j \leq n. M i 0 = \infty \wedge j > 0 \wedge i \neq j \longrightarrow M i j = \infty \wedge M j i = \infty$
and $\forall i \leq n. M i i = Le 0$
and $\forall i \leq n. \forall j \leq n. i > 0 \wedge j > 0 \wedge M i 0 \neq \infty \wedge M j 0 \neq \infty \longrightarrow$
 $(\exists d :: int.$
 $(-k (v' j) \leq d \wedge d \leq k (v' i) \wedge M i j = Le d \wedge M j i = Le (-d))$
 $\vee (-k (v' j) \leq d - 1 \wedge d \leq k (v' i) \wedge M i j = Lt d \wedge M j i = Lt$
 $(-d + 1)))$
and $\forall i \leq n. i > 0 \wedge M i 0 \neq \infty \longrightarrow$
 $(\exists d :: int. d \leq k (v' i) \wedge d \geq 0$
 $\wedge (M i 0 = Le d \wedge M 0 i = Le (-d) \vee M i 0 = Lt d \wedge M 0 i =$
 $Lt (-d + 1)))$
and $\forall i \leq n. i > 0 \longrightarrow (\exists d :: int. -k (v' i) \leq d \wedge d \leq 0 \wedge (M 0 i =$
 $Le d \vee M 0 i = Lt d))$
and $\forall i. \forall j. M i j \neq \infty \longrightarrow get-const (M i j) \in \mathbb{Z}$
and $\forall i \leq n. \forall j \leq n. M i j \neq \infty \wedge i > 0 \wedge j > 0 \longrightarrow$
 $(\exists d :: int. (M i j = Le d \vee M i j = Lt d) \wedge (-k (v' j)) \leq d \wedge d \leq k$
 $(v' i))$
 $\langle proof \rangle$

lemma *len-inf-elem*:

$(a, b) \in set (arcs i j xs) \implies M a b = \infty \implies len M i j xs = \infty$
 $\langle proof \rangle$

lemma *zone-diag-lt*:

assumes $a \leq n \ b \leq n$ **and** $C: v c1 = a \vee c2 = b$ **and** $not0: a > 0 \ b > 0$
shows $[(\lambda i j. if i = a \wedge j = b then Lt d else \infty)]_{v,n} = \{u. u c1 - u c2 < d\}$
 $\langle proof \rangle$

lemma *zone-diag-le*:

assumes $a \leq n \ b \leq n$ **and** $C: v c1 = a \vee c2 = b$ **and** $not0: a > 0 \ b > 0$
shows $[(\lambda i j. if i = a \wedge j = b then Le d else \infty)]_{v,n} = \{u. u c1 - u c2 \leq d\}$
 $\langle proof \rangle$

lemma *zone-diag-lt-2*:

assumes $a \leq n$ **and** $C: v c = a$ **and** $not0: a > 0$
shows $[(\lambda i j. if i = a \wedge j = 0 then Lt d else \infty)]_{v,n} = \{u. u c < d\}$
 $\langle proof \rangle$

lemma *zone-diag-le-2*:

assumes $a \leq n$ **and** $C: v c = a$ **and** $not0: a > 0$
shows $[(\lambda i j. if i = a \wedge j = 0 then Le d else \infty)]_{v,n} = \{u. u c \leq d\}$

<proof>

lemma *zone-diag-lt-3:*

assumes $a \leq n$ **and** $C: v\ c = a$ **and** $not\ 0: a > 0$

shows $[(\lambda\ i\ j. \text{if } i = 0 \wedge j = a \text{ then } Lt\ d \text{ else } \infty)]_{v,n} = \{u. -\ u\ c < d\}$

<proof>

lemma *len-int-closed:*

$\forall\ i\ j. (M\ i\ j :: real) \in \mathbb{Z} \implies len\ M\ i\ j\ xs \in \mathbb{Z}$

<proof>

lemma *get-const-distr:*

$a \neq \infty \implies b \neq \infty \implies get\ const\ (a + b) = get\ const\ a + get\ const\ b$

<proof>

lemma *len-int-dbm-closed:*

$\forall\ (i, j) \in set\ (arcs\ i\ j\ xs). (get\ const\ (M\ i\ j) :: real) \in \mathbb{Z} \wedge M\ i\ j \neq \infty$

$\implies get\ const\ (len\ M\ i\ j\ xs) \in \mathbb{Z} \wedge len\ M\ i\ j\ xs \neq \infty$

<proof>

lemma *zone-diag-le-3:*

assumes $a \leq n$ **and** $C: v\ c = a$ **and** $not\ 0: a > 0$

shows $[(\lambda\ i\ j. \text{if } i = 0 \wedge j = a \text{ then } Le\ d \text{ else } \infty)]_{v,n} = \{u. -\ u\ c \leq d\}$

<proof>

lemma *dbm-lt':*

assumes $[M]_{v,n} \subseteq V\ M\ a\ b \leq Lt\ d\ a \leq n\ b \leq n\ v\ c1 = a\ v\ c2 = b\ a > 0\ b > 0$

shows $[M]_{v,n} \subseteq \{u \in V. u\ c1 - u\ c2 < d\}$

<proof>

lemma *dbm-lt'2:*

assumes $[M]_{v,n} \subseteq V\ M\ a\ 0 \leq Lt\ d\ a \leq n\ v\ c1 = a\ a > 0$

shows $[M]_{v,n} \subseteq \{u \in V. u\ c1 < d\}$

<proof>

lemma *dbm-lt'3:*

assumes $[M]_{v,n} \subseteq V\ M\ 0\ a \leq Lt\ d\ a \leq n\ v\ c1 = a\ a > 0$

shows $[M]_{v,n} \subseteq \{u \in V. -\ u\ c1 < d\}$

<proof>

lemma *dbm-le':*

assumes $[M]_{v,n} \subseteq V\ M\ a\ b \leq Le\ d\ a \leq n\ b \leq n\ v\ c1 = a\ v\ c2 = b\ a > 0\ b > 0$

shows $[M]_{v,n} \subseteq \{u \in V. u \text{ c1} - u \text{ c2} \leq d\}$
 $\langle \text{proof} \rangle$

lemma *dbm-le'2*:

assumes $[M]_{v,n} \subseteq V M a 0 \leq L e d a \leq n v c1 = a a > 0$
shows $[M]_{v,n} \subseteq \{u \in V. u \text{ c1} \leq d\}$
 $\langle \text{proof} \rangle$

lemma *dbm-le'3*:

assumes $[M]_{v,n} \subseteq V M 0 a \leq L e d a \leq n v c1 = a a > 0$
shows $[M]_{v,n} \subseteq \{u \in V. - u \text{ c1} \leq d\}$
 $\langle \text{proof} \rangle$

lemma *int-zone-dbm*:

assumes $\forall (-,d) \in \text{collect-clock-pairs } cc. d \in \mathbb{Z} \forall c \in \text{collect-clks } cc. v c \leq n$
obtains M **where** $\{u. u \vdash cc\} = [M]_{v,n}$ **and** *dbm-int* $M n$
 $\langle \text{proof} \rangle$

lemma *non-empty-dbm-diag-set'*:

assumes *clock-numbering'* $v n \forall i \leq n. \forall j \leq n. M i j \neq \infty \longrightarrow \text{get-const } (M i j) \in \mathbb{Z}$
 $[M]_{v,n} \neq \{\}$
obtains M' **where** $[M]_{v,n} = [M']_{v,n} \wedge (\forall i \leq n. \forall j \leq n. M' i j \neq \infty \longrightarrow \text{get-const } (M' i j) \in \mathbb{Z})$
 $\wedge (\forall i \leq n. M' i i = 0)$
 $\langle \text{proof} \rangle$

lemma *dbm-entry-int*:

$(x :: t \text{ DBMEntry}) \neq \infty \implies \text{get-const } x \in \mathbb{Z} \implies \exists d :: \text{int. } x = L e d \vee x = L t d$
 $\langle \text{proof} \rangle$

6.2 Bouyer's Main Theorem

theorem *region-zone-intersect-empty-approx-correct*:

assumes $R \in \mathcal{R} Z \subseteq V R \cap Z = \{\}$ *vabstr* $Z M$
shows $R \cap \text{Approx}_\beta Z = \{\}$
 $\langle \text{proof} \rangle$

6.3 Nice Corollaries of Bouyer's Theorem

lemma *R-V*: $\bigcup \mathcal{R} = V$ $\langle \text{proof} \rangle$

lemma *regions-beta-V*: $R \in \mathcal{R}_\beta \implies R \subseteq V$ *<proof>*

lemma *apx-V*: $Z \subseteq V \implies \text{Approx}_\beta Z \subseteq V$
<proof>

corollary *approx-beta-closure-alpha*:
 assumes $Z \subseteq V$ *vabstr* $Z M$
 shows $\text{Approx}_\beta Z \subseteq \text{Closure}_\alpha Z$
<proof>

corollary *approx-beta-closure-alpha'*: $Z \in V' \implies \text{Approx}_\beta Z \subseteq \text{Closure}_\alpha Z$
<proof>

We could prove this more directly too (without using $\text{Closure}_\alpha Z$), obviously

lemma *apx-empty-iff*:
 assumes $Z \subseteq V$ *vabstr* $Z M$
 shows $Z = \{\}$ $\longleftrightarrow \text{Approx}_\beta Z = \{\}$
<proof>

lemma *apx-empty-iff'*:
 assumes $Z \in V'$ **shows** $Z = \{\}$ $\longleftrightarrow \text{Approx}_\beta Z = \{\}$
<proof>

lemma *apx-V'*:
 assumes $Z \subseteq V$ **shows** $\text{Approx}_\beta Z \in V'$
<proof>

end

lemma *valid-abstraction-pairsD*:
 $\forall (x, m) \in \text{Timed-Automata.clkp-set } A. x \in X \wedge m \in \mathbf{N}$ **if** *valid-abstraction*
 $A X k$
 <proof>

6.4 A New Zone Semantics Abstracting with Approx_β

locale *Regions* =

Regions-defs $X v n$ **for** X **and** $v :: 'c \Rightarrow \text{nat}$ **and** $n :: \text{nat} +$
 fixes $k :: 's \Rightarrow 'c \Rightarrow \text{nat}$ **and** *not-in-X*
 assumes *finite*: *finite* X
 assumes *clock-numbering*:
 clock-numbering' $v n \forall k \leq n. k > 0 \longrightarrow (\exists c \in X. v c = k) \forall c \in X. v$
 $c \leq n$
 assumes *not-in-X*: *not-in-X* $\notin X$

assumes *non-empty*: $X \neq \{\}$
begin

definition \mathcal{R} -def: $\mathcal{R} \ l \equiv \{Regions.region \ X \ I \ r \mid I \ r. \ Regions.valid-region \ X \ (k \ l) \ I \ r\}$

definition \mathcal{R}_β -def:

$\mathcal{R}_\beta \ l \equiv \{Regions-Beta.region \ X \ I \ J \ r \mid I \ J \ r. \ Regions-Beta.valid-region \ X \ (k \ l) \ I \ J \ r\}$

sublocale

AlphaClosure $X \ k \ \mathcal{R} \ \langle proof \rangle$

abbreviation $Approx_\beta \ l \ Z \equiv \ Beta-Regions'.Approx_\beta \ X \ (k \ l) \ v \ n \ not-in-X \ Z$

6.4.1 Single Step

inductive *step-z-beta* ::

$(\ 'a, \ 'c, \ t, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ t) \ zone \Rightarrow \ 'a \ action \Rightarrow \ 's \Rightarrow \ ('c, \ t) \ zone \Rightarrow \ bool$

$(\ \langle - \ \vdash \ \langle -, - \rangle \rightsquigarrow_{\beta(-)} \ \langle -, - \rangle \ \rangle \ [61,61,61,61] \ 61)$

where

step-beta: $A \ \vdash \ \langle l, Z \rangle \rightsquigarrow_a \ \langle l', Z' \rangle \Longrightarrow A \ \vdash \ \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \ \langle l', Approx_\beta \ l' \ Z' \rangle$

inductive-cases[*elim!*]: $A \ \vdash \ \langle l, u \rangle \rightsquigarrow_{\beta(a)} \ \langle l', u' \rangle$

declare *step-z-beta.intros*[*intro*]

context

fixes $l' :: 's$

begin

interpretation *regions*: *Regions-global* - - - $k \ l'$

$\langle proof \rangle$

lemma *step-z-V'*:

assumes $A \ \vdash \ \langle l, Z \rangle \rightsquigarrow_a \ \langle l', Z' \rangle \ \text{valid-abstraction} \ A \ X \ k \ \forall \ c \in \text{clk-set} \ A. \ v \ c \leq n \ Z \in V'$

shows $Z' \in V'$

$\langle proof \rangle$

lemma step-z-alpha-sound:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies \forall c \in \text{clk-set } A. \ v \ c \leq n \implies Z \in V'$
 $\implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 ⟨proof⟩

lemma step-z-alpha-complete:

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies \forall c \in \text{clk-set } A. \ v \ c \leq n \implies Z \in V'$
 $\implies Z' \neq \{\} \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z'' \rangle \wedge Z'' \neq \{\}$
 ⟨proof⟩

lemma alpha-beta-step:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies \forall c \in \text{clk-set } A. \ v \ c \leq n \implies Z \in V'$
 $\implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', Z'' \rangle \wedge Z' \subseteq Z''$
 ⟨proof⟩

lemma alpha-beta-step':

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies \forall c \in \text{clk-set } A. \ v \ c \leq n \implies Z \in V' \implies W \subseteq V$
 $\implies Z \subseteq W \implies \exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_{\alpha(a)} \langle l', W' \rangle \wedge Z' \subseteq W'$
 ⟨proof⟩

lemma apx-mono:

$Z' \subseteq V \implies Z \subseteq Z' \implies \text{Approx}_{\beta} \ l' \ Z \subseteq \text{Approx}_{\beta} \ l' \ Z'$
 ⟨proof⟩

end

lemma step-z'-V':

assumes $A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle \text{ valid-abstraction } A \ X \ k \ \forall c \in \text{clk-set } A. \ v \ c \leq n \ Z \in V'$
shows $Z' \in V'$
 ⟨proof⟩

lemma steps-z-V':

$A \vdash \langle l, Z \rangle \rightsquigarrow_* \langle l', Z' \rangle \implies \text{valid-abstraction } A \ X \ k \implies \forall c \in \text{clk-set } A. \ v \ c \leq n \implies Z \in V' \implies Z' \in V'$
 ⟨proof⟩

6.4.2 Multi step

definition

$step\text{-}z\text{-}beta' :: ('a, 'c, t, 's) ta \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow$
 $bool$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{\beta} \langle -, - \rangle) [61,61,61] 61)$
where
 $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z'' \rangle = (\exists Z' a. A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l, Z' \rangle \wedge A \vdash \langle l, Z' \rangle$
 $\rightsquigarrow_{\beta(1a)} \langle l', Z'' \rangle)$

abbreviation

$steps\text{-}z\text{-}beta :: ('a, 'c, t, 's) ta \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow$
 $bool$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{\beta^*} \langle -, - \rangle) [61,61,61] 61)$
where

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z'' \rangle \equiv (\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z'' \rangle)^{**}$
 $(l, Z) (l', Z'')$

lemma $V'\text{-}V: Z \in V' \Longrightarrow Z \subseteq V \langle proof \rangle$

context

fixes $A :: ('a, 'c, t, 's) ta$
assumes $valid\text{-}ta: valid\text{-}abstraction A X k \forall c \in clk\text{-}set A. v c \leq n$
begin

interpretation $alpha: AlphaClosure\text{-}global - k l' \mathcal{R} l' \langle proof \rangle$

lemma $[simp]: alpha.cla l' = cla l' \langle proof \rangle$

lemma $step\text{-}z\text{-}alpha'\text{-}V:$

$Z' \subseteq V$ **if** $Z \subseteq V A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z' \rangle$
 $\langle proof \rangle$

lemma $step\text{-}z\text{-}beta'\text{-}V':$

$Z' \in V'$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle Z \in V'$
 $\langle proof \rangle$

lemma $steps\text{-}z\text{-}beta\text{-}V':$

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle \Longrightarrow Z \in V' \Longrightarrow Z' \in V'$
 $\langle proof \rangle$

Soundness lemma $alpha'\text{-}beta'\text{-}step:$

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle \Longrightarrow Z \in V' \Longrightarrow W \subseteq V \Longrightarrow Z \subseteq W \Longrightarrow \exists W'$
 $A \vdash \langle l, W \rangle \rightsquigarrow_{\alpha} \langle l', W' \rangle \wedge Z' \subseteq W'$
 $\langle proof \rangle$

lemma $alpha\text{-}beta\text{-}sim:$

Simulation-Invariant

$(\lambda(l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z'' \rangle)$
 $(\lambda(l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle)$
 $(\lambda(l, Z) (l', Z'). l = l' \wedge Z \subseteq Z') (\lambda(-, Z). Z \in V') (\lambda(-, Z). Z \subseteq V)$
 $\langle \text{proof} \rangle$

interpretation

Simulation-Invariant

$\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z'' \rangle$
 $\lambda (l, Z) (l', Z''). A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha} \langle l', Z'' \rangle$
 $\lambda (l, Z) (l', Z'). l = l' \wedge Z \subseteq Z'$
 $\lambda (-, Z). Z \in V' \lambda (-, Z). Z \subseteq V$
 $\langle \text{proof} \rangle$

lemma *alpha-beta-steps:*

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle \implies Z \in V' \implies \exists Z''. A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha^*} \langle l', Z'' \rangle$
 $\wedge Z' \subseteq Z''$
 $\langle \text{proof} \rangle$

end

Completeness lemma *step-z-beta-mono:*

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z' \rangle \implies Z \subseteq W \implies W \subseteq V \implies \exists W'. A \vdash \langle l, W \rangle$
 $\rightsquigarrow_{\beta(a)} \langle l', W' \rangle \wedge Z' \subseteq W'$
 $\langle \text{proof} \rangle$

lemma *step-z-beta'-V:*

$Z' \subseteq V$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle Z \subseteq V$
 $\langle \text{proof} \rangle$

lemma *steps-z-beta-V:*

$Z' \subseteq V$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle Z \subseteq V$
 $\langle \text{proof} \rangle$

lemma *step-z-beta'-mono:*

$\exists W'. A \vdash \langle l, W \rangle \rightsquigarrow_{\beta} \langle l', W' \rangle \wedge Z' \subseteq W'$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle Z \subseteq$
 $W W \subseteq V$
 $\langle \text{proof} \rangle$

lemma *steps-z-beta-mono*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta^*} \langle l', Z' \rangle \implies Z \subseteq W \implies W \subseteq V \implies \exists W'. A \vdash \langle l, W \rangle$
 $\rightsquigarrow_{\beta^*} \langle l', W' \rangle \wedge Z' \subseteq W'$
<proof>

end

end

theory *Simulation-Graphs*

imports

library/CTL

library/More-List1

begin

lemmas [*simp*] = *holds.simps*

7 Simulation Graphs

7.1 Simulation Graphs

locale *Simulation-Graph-Defs* = *Graph-Defs C* **for** $C :: 'a \Rightarrow 'a \Rightarrow \text{bool} +$

fixes $A :: 'a \text{ set} \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$

begin

sublocale *Steps*: *Graph-Defs A* *<proof>*

abbreviation *Steps* \equiv *Steps.steps*

abbreviation *Run* \equiv *Steps.run*

lemmas *Steps-appendD1* = *Steps.steps-appendD1*

lemmas *Steps-appendD2* = *Steps.steps-appendD2*

lemmas *steps-alt-induct* = *Steps.steps-alt-induct*

lemmas *Steps-appendI* = *Steps.steps-appendI*

lemmas *Steps-cases* = *Steps.steps.cases*

end

locale *Simulation-Graph-Poststable* = *Simulation-Graph-Defs* +
assumes *poststable*: $A\ S\ T \implies \forall s' \in T. \exists s \in S. C\ s\ s'$

locale *Simulation-Graph-Prestable* = *Simulation-Graph-Defs* +
assumes *prestable*: $A\ S\ T \implies \forall s \in S. \exists s' \in T. C\ s\ s'$

locale *Double-Simulation-Defs* =
fixes $C :: 'a \Rightarrow 'a \Rightarrow bool$ — Concrete step relation
and $A1 :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ — Step relation for the first abstraction layer
and $P1 :: 'a\ set \Rightarrow bool$ — Valid states of the first abstraction layer
and $A2 :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ — Step relation for the second abstraction layer
and $P2 :: 'a\ set \Rightarrow bool$ — Valid states of the second abstraction layer
begin

sublocale *Simulation-Graph-Defs* $C\ A2$ $\langle proof \rangle$

sublocale *pre-defs*: *Simulation-Graph-Defs* $C\ A1$ $\langle proof \rangle$

definition *closure* $a = \{x. P1\ x \wedge a \cap x \neq \{\}\}$

definition $A2'\ a\ b \equiv \exists x\ y. a = closure\ x \wedge b = closure\ y \wedge A2\ x\ y$

sublocale *post-defs*: *Simulation-Graph-Defs* $A1\ A2'$ $\langle proof \rangle$

lemma *closure-mono*:
 $closure\ a \subseteq closure\ b$ **if** $a \subseteq b$
 $\langle proof \rangle$

lemma *closure-intD*:
 $x \in closure\ a \wedge x \in closure\ b$ **if** $x \in closure\ (a \cap b)$
 $\langle proof \rangle$

end

locale *Double-Simulation* = *Double-Simulation-Defs* +
assumes *prestable*: $A1\ S\ T \implies \forall s \in S. \exists s' \in T. C\ s\ s'$
and *closure-poststable*: $s' \in closure\ y \implies A2\ x\ y \implies \exists s \in closure\ x. A1\ s\ s'$
and *P1-distinct*: $P1\ x \implies P1\ y \implies x \neq y \implies x \cap y = \{\}$
and *P1-finite*: $finite\ \{x. P1\ x\}$
and *P2-cover*: $P2\ a \implies \exists x. P1\ x \wedge x \cap a \neq \{\}$
begin

```

sublocale post: Simulation-Graph-Poststable A1 A2'
  ⟨proof⟩

sublocale pre: Simulation-Graph-Prestable C A1
  ⟨proof⟩

end

locale Finite-Graph = Graph-Defs +
  fixes  $x_0$ 
  assumes finite-reachable: finite {x. E** x0 x}

locale Simulation-Graph-Complete-Defs =
  Simulation-Graph-Defs C A for C :: 'a ⇒ 'a ⇒ bool and A :: 'a set ⇒ 'a
  set ⇒ bool +
  fixes  $P :: 'a \text{ set} \Rightarrow \text{bool}$  — well-formed abstractions

locale Simulation-Graph-Complete = Simulation-Graph-Complete-Defs +
  simulation: Simulation-Invariant C A (∈) λ -. True P
begin

lemmas complete = simulation.A-B-step
lemmas P-invariant = simulation.B-invariant

end

locale Simulation-Graph-Finite-Complete = Simulation-Graph-Complete +
  fixes  $a_0$ 
  assumes finite-abstract-reachable: finite {a. A** a0 a}
begin

sublocale Steps-finite: Finite-Graph A a0
  ⟨proof⟩

end

locale Double-Simulation-Complete = Double-Simulation +
  fixes  $a_0$ 
  assumes complete: C x y ⇒ x ∈ S ⇒ P2 S ⇒ ∃ T. A2 S T ∧ y ∈ T
  assumes P2-invariant: P2 a ⇒ A2 a a' ⇒ P2 a'
  and  $P2\text{-}a_0: P2 a_0$ 
begin

```

sublocale *Simulation-Graph-Complete C A2 P2*
⟨*proof*⟩

sublocale *P2-invariant: Graph-Invariant-Start A2 a₀ P2*
⟨*proof*⟩

end

locale *Double-Simulation-Finite-Complete = Double-Simulation-Complete*
+
assumes *finite-abstract-reachable: finite {a. A2** a₀ a}*
begin

sublocale *Simulation-Graph-Finite-Complete C A2 P2 a₀*
⟨*proof*⟩

end

locale *Simulation-Graph-Complete-Prestable = Simulation-Graph-Complete*
+ *Simulation-Graph-Prestable*
begin

sublocale *Graph-Invariant A P* ⟨*proof*⟩

end

locale *Double-Simulation-Complete-Bisim = Double-Simulation-Complete*
+
assumes *A1-complete: C x y \implies P1 S \implies x \in S \implies \exists T. A1 S T \wedge y \in T*
and *P1-invariant: P1 S \implies A1 S T \implies P1 T*
begin

sublocale *bisim: Simulation-Graph-Complete-Prestable C A1 P1*
⟨*proof*⟩

end

locale *Double-Simulation-Finite-Complete-Bisim =*
Double-Simulation-Finite-Complete + Double-Simulation-Complete-Bisim

locale *Double-Simulation-Complete-Bisim-Cover = Double-Simulation-Complete-Bisim*
+
assumes *P2-P1-cover: P2 a \implies x \in a \implies \exists a'. a \cap a' \neq {} \wedge P1 a' \wedge*

$x \in a'$

locale *Double-Simulation-Finite-Complete-Bisim-Cover* =
Double-Simulation-Finite-Complete-Bisim + *Double-Simulation-Complete-Bisim-Cover*

locale *Double-Simulation-Complete-Abstraction-Prop* =
Double-Simulation-Complete +
fixes $\varphi :: 'a \Rightarrow \text{bool}$ — The property we want to check
assumes φ -A1-compatible: $A1\ a\ b \Longrightarrow b \subseteq \{x. \varphi\ x\} \vee b \cap \{x. \varphi\ x\} = \{\}$
and φ -P2-compatible: $P2\ a \Longrightarrow a \cap \{x. \varphi\ x\} \neq \{\} \Longrightarrow P2\ (a \cap \{x. \varphi\ x\})$
and φ -A2-compatible: $A2^{**}\ a_0\ a \Longrightarrow a \cap \{x. \varphi\ x\} \neq \{\} \Longrightarrow A2^{**}\ a_0\ (a \cap \{x. \varphi\ x\})$
and $P2$ -non-empty: $P2\ a \Longrightarrow a \neq \{\}$

locale *Double-Simulation-Complete-Abstraction-Prop-Bisim* =
Double-Simulation-Complete-Abstraction-Prop + *Double-Simulation-Complete-Bisim*

locale *Double-Simulation-Finite-Complete-Abstraction-Prop* =
Double-Simulation-Complete-Abstraction-Prop + *Double-Simulation-Finite-Complete*

locale *Double-Simulation-Finite-Complete-Abstraction-Prop-Bisim* =
Double-Simulation-Finite-Complete-Abstraction-Prop + *Double-Simulation-Finite-Complete-Bisim*

7.2 Poststability

context *Simulation-Graph-Poststable*
begin

lemma *Steps-poststable*:
 $\exists\ xs. \text{steps}\ xs \wedge \text{list-all2}\ (\in)\ xs\ as \wedge \text{last}\ xs = x$ **if** $\text{Steps}\ as\ x \in \text{last}\ as$
<proof>

lemma *reaches-poststable*:
 $\exists\ x \in a. \text{reaches}\ x\ y$ **if** $\text{Steps.reaches}\ a\ b\ y \in b$
<proof>

lemma *Steps-steps-cycle*:
 $\exists\ x\ xs. \text{steps}\ (x \# xs\ @\ [x]) \wedge (\forall\ x \in \text{set}\ xs. \exists\ a \in \text{set}\ as \cup \{a\}. x \in a)$
 $\wedge x \in a$
if *assms*: $\text{Steps}\ (a \# as\ @\ [a])$ *finite* $a\ a \neq \{\}$
<proof>

end

7.3 Prestability

context *Simulation-Graph-Prestable*

begin

lemma *Steps-prestable:*

$\exists xs. \text{steps } (x \# xs) \wedge \text{list-all2 } (\in) (x \# xs) \text{ as } \mathbf{if} \text{ Steps as } x \in \text{hd as}$
 $\langle \text{proof} \rangle$

lemma *reaches-prestable:*

$\exists y. \text{reaches } x \ y \wedge y \in b \ \mathbf{if} \ \text{Steps.reaches } a \ b \ x \in a$
 $\langle \text{proof} \rangle$

Abstract cycles lead to concrete infinite runs.

lemma *Steps-run-cycle-buechi:*

$\exists xs. \text{run } (x \#\# xs) \wedge \text{stream-all2 } (\in) \ xs \ (\text{cycle } (as \ @ \ [a]))$
 $\mathbf{if} \ \text{assms: Steps } (a \# as \ @ \ [a]) \ x \in a$
 $\langle \text{proof} \rangle$

lemma *Steps-run-cycle-buechi'':*

$\exists xs. \text{run } (x \#\# xs) \wedge (\forall x \in \text{sset } xs. \exists a \in \text{set } as \cup \{a\}. x \in a) \wedge \text{infs}$
 $(\lambda x. x \in b) (x \#\# xs)$
 $\mathbf{if} \ \text{assms: Steps } (a \# as \ @ \ [a]) \ x \in a \ b \in \text{set } (a \# as \ @ \ [a])$
 $\langle \text{proof} \rangle$

lemma *Steps-run-cycle-buechi':*

$\exists xs. \text{run } (x \#\# xs) \wedge (\forall x \in \text{sset } xs. \exists a \in \text{set } as \cup \{a\}. x \in a) \wedge \text{infs}$
 $(\lambda x. x \in a) (x \#\# xs)$
 $\mathbf{if} \ \text{assms: Steps } (a \# as \ @ \ [a]) \ x \in a$
 $\langle \text{proof} \rangle$

lemma *Steps-run-cycle':*

$\exists xs. \text{run } (x \#\# xs) \wedge (\forall x \in \text{sset } xs. \exists a \in \text{set } as \cup \{a\}. x \in a)$
 $\mathbf{if} \ \text{assms: Steps } (a \# as \ @ \ [a]) \ x \in a$
 $\langle \text{proof} \rangle$

lemma *Steps-run-cycle:*

$\exists xs. \text{run } xs \wedge (\forall x \in \text{sset } xs. \exists a \in \text{set } as \cup \{a\}. x \in a) \wedge \text{shd } xs \in a$
 $\mathbf{if} \ \text{assms: Steps } (a \# as \ @ \ [a]) \ a \neq \{\}$
 $\langle \text{proof} \rangle$

Unused lemma *Steps-cycle-every-prestable':*

$\exists b \ y. C \ x \ y \wedge y \in b \wedge b \in \text{set } as \cup \{a\}$
 $\mathbf{if} \ \text{assms: Steps } (as \ @ \ [a]) \ x \in b \ b \in \text{set } as$

<proof>

lemma *Steps-cycle-first-prestable:*

$\exists b y. C x y \wedge x \in b \wedge b \in \text{set } as \cup \{a\}$ **if** *assms:* $\text{Steps } (a \# as @ [a]) x \in a$

<proof>

lemma *Steps-cycle-every-prestable:*

$\exists b y. C x y \wedge y \in b \wedge b \in \text{set } as \cup \{a\}$

if *assms:* $\text{Steps } (a \# as @ [a]) x \in b \wedge b \in \text{set } as \cup \{a\}$

<proof>

end

7.4 Double Simulation

context *Double-Simulation*

begin

lemma *closure-involutive:*

$\text{closure } (\bigcup (\text{closure } x)) = \text{closure } x$

<proof>

lemma *closure-finite:*

$\text{finite } (\text{closure } x)$

<proof>

lemma *closure-non-empty:*

$\text{closure } x \neq \{\}$ **if** $P2 x$

<proof>

lemma *P1-closure-id:*

$\text{closure } R = \{R\}$ **if** $P1 R \wedge R \neq \{\}$

<proof>

lemma *A2'-A2-closure:*

$A2' (\text{closure } x) (\text{closure } y)$ **if** $A2 x y$

<proof>

lemma *Steps-Union:*

$\text{post-defs.Steps } (\text{map } \text{closure } xs)$ **if** $\text{Steps } xs$

<proof>

lemma *closure-reaches:*

$post-defs.Steps.reaches (closure\ x) (closure\ y)$ **if** $Steps.reaches\ x\ y$
 ⟨proof⟩

lemma *post-Steps-non-empty*:

$x \neq \{\}$ **if** $post-defs.Steps (a \# as) x \in b\ b \in set\ as$
 ⟨proof⟩

lemma *Steps-run-cycle'*:

$\exists xs. run\ xs \wedge (\forall x \in sset\ xs. \exists a \in set\ as \cup \{a\}. x \in \bigcup a) \wedge shd\ xs \in \bigcup a$
if *assms*: $post-defs.Steps (a \# as @ [a])\ finite\ a\ a \neq \{\}$
 ⟨proof⟩

lemma *Steps-run-cycle*:

$\exists xs. run\ xs \wedge (\forall x \in sset\ xs. \exists a \in set\ as \cup \{a\}. x \in \bigcup (closure\ a)) \wedge shd\ xs \in \bigcup (closure\ a)$
if *assms*: $Steps (a \# as @ [a])\ P2\ a$
 ⟨proof⟩

lemma *Steps-run-cycle2*:

$\exists x\ xs. run\ (x \#\# xs) \wedge x \in \bigcup (closure\ a_0)$
 $\wedge (\forall x \in sset\ xs. \exists a \in set\ as \cup \{a\} \cup set\ bs. x \in \bigcup a)$
 $\wedge infs (\lambda x. x \in \bigcup a) (x \#\# xs)$
if *assms*: $post-defs.Steps (closure\ a_0 \# as @ a \# bs @ [a])\ a \neq \{\}$
 ⟨proof⟩

lemma *Steps-run-cycle''*:

$\exists x\ xs. run\ (x \#\# xs) \wedge x \in \bigcup (closure\ a_0)$
 $\wedge (\forall x \in sset\ xs. \exists a \in set\ as \cup \{a\} \cup set\ bs. x \in \bigcup (closure\ a))$
 $\wedge infs (\lambda x. x \in \bigcup (closure\ a)) (x \#\# xs)$
if *assms*: $Steps (a_0 \# as @ a \# bs @ [a])\ P2\ a$
 ⟨proof⟩

Unused lemma *post-Steps-P1*:

$P1\ x$ **if** $post-defs.Steps (a \# as) x \in b\ b \in set\ as$
 ⟨proof⟩

lemma *strong-compatibility-impl-weak*:

fixes $\varphi :: 'a \Rightarrow bool$ — The property we want to check
assumes φ -closure-compatible: $\bigwedge x\ a. x \in a \Longrightarrow \varphi\ x \longleftrightarrow (\forall x \in \bigcup (closure\ a). \varphi\ x)$
shows $\varphi\ x \Longrightarrow x \in a \Longrightarrow y \in a \Longrightarrow P1\ a \Longrightarrow \varphi\ y$
 ⟨proof⟩

end

7.5 Finite Graphs

context *Finite-Graph*
begin

7.5.1 Infinite Büchi Runs Correspond to Finite Cycles

lemma *run-finite-state-set*:
 assumes $run\ (x_0\ \#\#\ xs)$
 shows $finite\ (sset\ (x_0\ \#\#\ xs))$
 $\langle proof \rangle$

lemma *run-finite-state-set-cycle*:
 assumes $run\ (x_0\ \#\#\ xs)$
 shows
 $\exists\ ys\ zs.\ run\ (x_0\ \#\#\ ys\ @-\ cycle\ zs) \wedge set\ ys \cup set\ zs \subseteq \{x_0\} \cup sset\ xs$
 $\wedge\ zs \neq []$
 $\langle proof \rangle$

lemma *buechi-run-finite-state-set-cycle*:
 assumes $run\ (x_0\ \#\#\ xs)\ alw\ (ev\ (holds\ \varphi))\ (x_0\ \#\#\ xs)$
 shows
 $\exists\ ys\ zs.$
 $run\ (x_0\ \#\#\ ys\ @-\ cycle\ zs) \wedge set\ ys \cup set\ zs \subseteq \{x_0\} \cup sset\ xs$
 $\wedge\ zs \neq [] \wedge (\exists\ x \in set\ zs.\ \varphi\ x)$
 $\langle proof \rangle$

lemma *run-finite-state-set-cycle-steps*:
 assumes $run\ (x_0\ \#\#\ xs)$
 shows $\exists\ x\ ys\ zs.\ steps\ (x_0\ \#\ ys\ @\ x\ \#\ zs\ @\ [x]) \wedge \{x\} \cup set\ ys \cup set\ zs$
 $\subseteq \{x_0\} \cup sset\ xs$
 $\langle proof \rangle$

lemma *buechi-run-finite-state-set-cycle-steps*:
 assumes $run\ (x_0\ \#\#\ xs)\ alw\ (ev\ (holds\ \varphi))\ (x_0\ \#\#\ xs)$
 shows
 $\exists\ x\ ys\ zs.$
 $steps\ (x_0\ \#\ ys\ @\ x\ \#\ zs\ @\ [x]) \wedge \{x\} \cup set\ ys \cup set\ zs \subseteq \{x_0\} \cup sset\ xs$
 $\wedge (\exists\ y \in set\ (x\ \#\ zs).\ \varphi\ y)$

<proof>

lemma *cycle-steps-run*:

assumes *steps* ($x_0 \# ys @ x \# zs @ [x]$)

shows $\exists xs. run (x_0 \#\# xs) \wedge sset xs = \{x\} \cup set ys \cup set zs$

<proof>

lemma *buechi-run-lasso*:

assumes *run* ($x_0 \#\# xs$) *alw* (*ev* (*holds* φ)) ($x_0 \#\# xs$)

obtains x **where** *reaches* x_0 x *reaches1* x φ x

<proof>

including *graph-automation* *<proof>*

end

7.6 Complete Simulation Graphs

context *Simulation-Graph-Defs*

begin

definition *abstract-run* x $xs = x \#\# sscan (\lambda y a. SOME b. A a b \wedge y \in b) xs$ x

lemma *abstract-run-ctr*:

abstract-run x $xs = x \#\# abstract-run (SOME b. A x b \wedge shd xs \in b)$ (*stl* xs)

<proof>

end

context *Simulation-Graph-Complete*

begin

lemma *steps-complete*:

$\exists as. Steps (a \# as) \wedge list-all2 (\in) xs as$ **if** *steps* ($x \# xs$) $x \in a$ $P a$

<proof>

lemma *abstract-run-Run*:

Run (*abstract-run* a xs) **if** *run* ($x \#\# xs$) $x \in a$ $P a$

<proof>

lemma *abstract-run-abstract*:

stream-all2 (\in) ($x \#\# xs$) (*abstract-run* a xs) **if** *run* ($x \#\# xs$) $x \in a$ $P a$

$\langle proof \rangle$

lemma *run-complete:*

$\exists as. Run (a \#\# as) \wedge stream-all2 (\in) xs as$ **if** $run (x \#\# xs) x \in a P a$
 $\langle proof \rangle$

end

7.6.1 Runs in Finite Complete Graphs

context *Simulation-Graph-Finite-Complete*

begin

lemma *run-finite-state-set-cycle-steps:*

assumes $run (x_0 \#\# xs) x_0 \in a_0 P a_0$

shows $\exists x ys zs.$

$Steps (a_0 \# ys @ x \# zs @ [x]) \wedge (\forall a \in \{x\} \cup set ys \cup set zs. \exists x \in \{x_0\} \cup sset xs. x \in a)$

$\langle proof \rangle$

lemma *buechi-run-finite-state-set-cycle-steps:*

assumes $run (x_0 \#\# xs) x_0 \in a_0 P a_0 alw (ev (holds \varphi)) (x_0 \#\# xs)$

shows $\exists x ys zs.$

$Steps (a_0 \# ys @ x \# zs @ [x])$

$\wedge (\forall a \in \{x\} \cup set ys \cup set zs. \exists x \in \{x_0\} \cup sset xs. x \in a)$

$\wedge (\exists y \in set (x \# zs). \exists a \in y. \varphi a)$

$\langle proof \rangle$

lemma *buechi-run-finite-state-set-cycle-lasso:*

assumes $run (x_0 \#\# xs) x_0 \in a_0 P a_0 alw (ev (holds \varphi)) (x_0 \#\# xs)$

shows $\exists a. Steps.reaches a_0 a \wedge Steps.reaches1 a a \wedge (\exists y \in a. \varphi y)$

$\langle proof \rangle$

end

7.7 Finite Complete Double Simulations

context *Double-Simulation*

begin

lemma *Run-closure:*

$post-defs.Run (smap closure xs)$ **if** $Run xs$

$\langle proof \rangle$

lemma *closure-set-finite*:
finite (closure ' UNIV) (is finite ?S)
 ⟨*proof*⟩

lemma *A2'-empty-step*:
 $b = \{\}$ **if** $A2' a b a = \{\}$
 ⟨*proof*⟩

lemma *A2'-empty-invariant*:
Graph-Invariant A2' ($\lambda x. x = \{\}$)
 ⟨*proof*⟩

end

context *Double-Simulation-Complete*
begin

lemmas *P2-invariant-Steps = P2-invariant.invariant-steps*

interpretation *Steps-finite: Finite-Graph A2' closure a₀*
 ⟨*proof*⟩

theorem *infinite-run-cycle-iff'*:
assumes $\bigwedge x xs. run (x \#\# xs) \implies x \in \bigcup (closure a_0) \implies \exists y ys. y \in a_0 \wedge run (y \#\# ys)$
shows
 $(\exists x_0 xs. x_0 \in \bigcup (closure a_0) \wedge run (x_0 \#\# xs)) \longleftrightarrow$
 $(\exists as a bs. post-defs.Steps (closure a_0 \# as @ a \# bs @ [a]) \wedge a \neq \{\})$
 ⟨*proof*⟩

corollary *infinite-run-cycle-iff*:
 $(\exists x_0 xs. x_0 \in a_0 \wedge run (x_0 \#\# xs)) \longleftrightarrow$
 $(\exists as a bs. post-defs.Steps (closure a_0 \# as @ a \# bs @ [a]) \wedge a \neq \{\})$
if $\bigcup (closure a_0) = a_0 P2 a_0$
 ⟨*proof*⟩

context
fixes $\varphi :: 'a \Rightarrow bool$ — The property we want to check
assumes *φ -closure-compatible*: $P2 a \implies x \in \bigcup (closure a) \implies \varphi x \longleftrightarrow$
 $(\forall x \in \bigcup (closure a). \varphi x)$
begin

We need the condition $a \neq \{\}$ in the following theorem because we cannot prove a lemma like this:

lemma

$\exists bs. \text{Steps } bs \wedge \text{closure } a \# as = \text{map closure } bs \text{ if } \text{post-defs.Steps } (\text{closure } a \# as)$
<proof>

One possible fix would be to add the stronger assumption $A2 a b \implies P2 b$.

theorem *infinite-buechi-run-cycle-iff-closure*:

assumes

$\bigwedge x xs. \text{run } (x \#\# xs) \implies x \in \bigcup (\text{closure } a_0) \implies \text{alw } (\text{ev } (\text{holds } \varphi)) xs$
 $\implies \exists y ys. y \in a_0 \wedge \text{run } (y \#\# ys) \wedge \text{alw } (\text{ev } (\text{holds } \varphi)) ys$
and $\bigwedge a. P2 a \implies a \subseteq \bigcup (\text{closure } a)$

shows

$(\exists x_0 xs. x_0 \in \bigcup (\text{closure } a_0) \wedge \text{run } (x_0 \#\# xs) \wedge \text{alw } (\text{ev } (\text{holds } \varphi)) (x_0 \#\# xs))$
 $\longleftrightarrow (\exists as a bs. a \neq \{\} \wedge \text{post-defs.Steps } (\text{closure } a_0 \# as @ a \# bs @ [a]) \wedge (\forall x \in \bigcup a. \varphi x))$
<proof>

end

end

context *Double-Simulation-Finite-Complete*

begin

lemmas *P2-invariant-Steps = P2-invariant.invariant-steps*

theorem *infinite-run-cycle-iff'*:

assumes $P2 a_0 \bigwedge x xs. \text{run } (x \#\# xs) \implies x \in \bigcup (\text{closure } a_0) \implies \exists y ys. y \in a_0 \wedge \text{run } (y \#\# ys)$
shows $(\exists x_0 xs. x_0 \in a_0 \wedge \text{run } (x_0 \#\# xs)) \longleftrightarrow (\exists as a bs. \text{Steps } (a_0 \# as @ a \# bs @ [a]))$
<proof>

corollary *infinite-run-cycle-iff*:

$(\exists x_0 xs. x_0 \in a_0 \wedge \text{run } (x_0 \#\# xs)) \longleftrightarrow (\exists as a bs. \text{Steps } (a_0 \# as @ a \# bs @ [a]))$
if $\bigcup (\text{closure } a_0) = a_0 P2 a_0$
<proof>

context

fixes $\varphi :: 'a \Rightarrow \text{bool}$ — The property we want to check

assumes φ -closure-compatible: $x \in a \implies \varphi x \longleftrightarrow (\forall x \in \bigcup (\text{closure } a). \varphi x)$

begin

theorem *infinite-buechi-run-cycle-iff*:

$(\exists x_0 xs. x_0 \in a_0 \wedge \text{run } (x_0 \#\# xs) \wedge \text{alw } (\text{ev } (\text{holds } \varphi)) (x_0 \#\# xs))$
 $\longleftrightarrow (\exists as a bs. \text{Steps } (a_0 \# as @ a \# bs @ [a]) \wedge (\forall x \in \bigcup(\text{closure } a).$
 $\varphi x))$
if $\bigcup(\text{closure } a_0) = a_0$
<proof>

end

end

7.8 Encoding of Properties in Runs

This approach only works if we assume strong compatibility of the property. For weak compatibility, encoding in the automaton is likely the right way.

context *Double-Simulation-Complete-Abstraction-Prop*

begin

definition $C\text{-}\varphi x y \equiv C x y \wedge \varphi y$

definition $A1\text{-}\varphi a b \equiv A1 a b \wedge b \subseteq \{x. \varphi x\}$

definition $A2\text{-}\varphi S S' \equiv \exists S''. A2 S S'' \wedge S'' \cap \{x. \varphi x\} = S' \wedge S' \neq \{\}$

lemma *A2-φ-P2-invariant*:

$P2 a$ **if** $A2\text{-}\varphi^{**} a_0 a$
<proof>

sublocale *phi*: *Double-Simulation-Complete C-φ A1-φ P1 A2-φ P2 a₀*

<proof>

lemma *phi-run-iff*:

$\text{phi.run } (x \#\# xs) \wedge \varphi x \longleftrightarrow \text{run } (x \#\# xs) \wedge \text{pred-stream } \varphi (x \#\# xs)$
<proof>

end

context *Double-Simulation-Finite-Complete-Abstraction-Prop*

begin

sublocale *phi*: *Double-Simulation-Finite-Complete C-φ A1-φ P1 A2-φ P2*

a₀

<proof>

including *graph-automation-aggressive* \langle proof \rangle

corollary *infinite-run-cycle-iff*:

$(\exists x_0 xs. x_0 \in a_0 \wedge run (x_0 \#\# xs) \wedge pred-stream \varphi (x_0 \#\# xs)) \longleftrightarrow$
 $(\exists as a bs. phi.Steps (a_0 \# as @ a \# bs @ [a]))$
if $\bigcup (closure a_0) = a_0$ $a_0 \subseteq \{x. \varphi x\}$
 \langle proof \rangle

theorem *Alw-ev-mc*:

$(\forall x_0 \in a_0. Alw-ev (Not o \varphi) x_0) \longleftrightarrow \neg (\exists as a bs. phi.Steps (a_0 \# as$
 $@ a \# bs @ [a]))$
if $\bigcup (closure a_0) = a_0$ $a_0 \subseteq \{x. \varphi x\}$
 \langle proof \rangle

end

context *Simulation-Graph-Defs*

begin

definition *represent-run* $x as = x \#\# sscan (\lambda b x. SOME y. C x y \wedge y$
 $\in b) as x$

lemma *represent-run-ctr*:

$represent-run x as = x \#\# represent-run (SOME y. C x y \wedge y \in shd as)$
 $(stl as)$
 \langle proof \rangle

end

context *Simulation-Graph-Prestable*

begin

lemma *represent-run-Run*:

$run (represent-run x as) \mathbf{if} Run (a \#\# as) x \in a$
 \langle proof \rangle

lemma *represent-run-represent*:

$stream-all2 (\in) (represent-run x as) (a \#\# as) \mathbf{if} Run (a \#\# as) x \in a$
 \langle proof \rangle

end

context *Simulation-Graph-Complete-Prestable*

begin

lemma *step-bisim*:

$\exists y'. C\ x'\ y' \wedge (\exists a. P\ a \wedge y \in a \wedge y' \in a)$ **if** $C\ x\ y\ x \in a\ x' \in a\ P\ a$
<proof>

sublocale *steps-bisim*:

Bisimulation-Invariant $C\ C\ \lambda\ x\ y. \exists a. P\ a \wedge x \in a \wedge y \in a\ \lambda\ -. True\ \lambda\ -. True$
<proof>

lemma *runs-bisim*:

$\exists ys. run\ (y\ \#\#\ ys) \wedge stream-all2\ (\lambda\ x\ y. \exists a. x \in a \wedge y \in a \wedge P\ a)\ xs\ ys$
if $run\ (x\ \#\#\ xs)\ x \in a\ y \in a\ P\ a$
<proof>

lemma *runs-bisim'*:

$\exists ys. run\ (y\ \#\#\ ys)$ **if** $run\ (x\ \#\#\ xs)\ x \in a\ y \in a\ P\ a$
<proof>

context

fixes $Q :: 'a \Rightarrow bool$

assumes *compatible*: $Q\ x \Longrightarrow x \in a \Longrightarrow y \in a \Longrightarrow P\ a \Longrightarrow Q\ y$

begin

lemma *Alw-ev-compatible'*:

assumes $\forall xs. run\ (x\ \#\#\ xs) \longrightarrow ev\ (holds\ Q)\ (x\ \#\#\ xs)\ run\ (y\ \#\#\ xs)\ x \in a\ y \in a\ P\ a$
shows $ev\ (holds\ Q)\ (y\ \#\#\ xs)$
<proof>

lemma *Alw-ev-compatible*:

$Alw-ev\ Q\ x \longleftrightarrow Alw-ev\ Q\ y$ **if** $x \in a\ y \in a\ P\ a$
<proof>

end

lemma *steps-bisim*:

$\exists ys. steps\ (y\ \#\ ys) \wedge list-all2\ (\lambda\ x\ y. \exists a. x \in a \wedge y \in a \wedge P\ a)\ xs\ ys$
if $steps\ (x\ \#\ xs)\ x \in a\ y \in a\ P\ a$
<proof>

end

context *Subgraph-Node-Defs*
begin

lemma *subgraph-runD*:
 $run\ xs\ \mathbf{if}\ G'.run\ xs$
 $\langle proof \rangle$

lemma *subgraph-V-all*:
 $pred-stream\ V\ xs\ \mathbf{if}\ G'.run\ xs$
 $\langle proof \rangle$

lemma *subgraph-runI*:
 $G'.run\ xs\ \mathbf{if}\ pred-stream\ V\ xs\ run\ xs$
 $\langle proof \rangle$

lemma *subgraph-run-iff*:
 $G'.run\ xs\ \longleftrightarrow\ pred-stream\ V\ xs\ \wedge\ run\ xs$
 $\langle proof \rangle$

end

context *Double-Simulation-Finite-Complete-Abstraction-Prop-Bisim*
begin

sublocale *sim-complete: Simulation-Graph-Complete-Prestable C-φ A1-φ*
P1
 $\langle proof \rangle$

lemma *runs-closure-bisim*:
 $\exists y\ ys.\ y \in a_0 \wedge phi.run\ (y\ \#\#\ ys)\ \mathbf{if}\ phi.run\ (x\ \#\#\ xs)\ x \in \bigcup(phi.closure\ a_0)$
 $\langle proof \rangle$

lemma *infinite-run-cycle-iff'*:
 $(\exists x_0\ xs.\ x_0 \in a_0 \wedge phi.run\ (x_0\ \#\#\ xs)) = (\exists as\ a\ bs.\ phi.Steps\ (a_0\ \#\ as\ @\ a\ \#\ bs\ @\ [a]))$
 $\langle proof \rangle$

corollary *infinite-run-cycle-iff*:
 $(\exists x_0\ xs.\ x_0 \in a_0 \wedge run\ (x_0\ \#\#\ xs) \wedge pred-stream\ \varphi\ (x_0\ \#\#\ xs)) \longleftrightarrow$
 $(\exists as\ a\ bs.\ phi.Steps\ (a_0\ \#\ as\ @\ a\ \#\ bs\ @\ [a]))$
 $\mathbf{if}\ a_0 \subseteq \{x.\ \varphi\ x\}$
 $\langle proof \rangle$

theorem *Alw-ev-mc*:

$(\forall x_0 \in a_0. \text{Alw-ev } (\text{Not } o \ \varphi) \ x_0) \longleftrightarrow \neg (\exists \text{ as a bs. } \text{phi.Steps } (a_0 \# \text{ as} \\ @ \ a \# \text{ bs } @ \ [a]))$
if $a_0 \subseteq \{x. \varphi \ x\}$
<proof>

lemma *phi-Steps-Alw-ev*:

$\neg (\exists \text{ as a bs. } \text{phi.Steps } (a_0 \# \text{ as } @ \ a \# \text{ bs } @ \ [a])) \longleftrightarrow \text{phi.Steps.Alw-ev}$
 $(\lambda \ -. \ \text{False}) \ a_0$
<proof>

theorem *Alw-ev-mc'*:

$(\forall x_0 \in a_0. \text{Alw-ev } (\text{Not } o \ \varphi) \ x_0) \longleftrightarrow \text{phi.Steps.Alw-ev } (\lambda \ -. \ \text{False}) \ a_0$
if $a_0 \subseteq \{x. \varphi \ x\}$
<proof>

end

context *Graph-Start-Defs*

begin

interpretation *Bisimulation-Invariant E E (=) reachable reachable*

including *graph-automation <proof>*

lemma *Alw-alw-iff-default*:

$\text{Alw-alw } \varphi \ x \longleftrightarrow \text{Alw-alw } \psi \ x$ **if** $\bigwedge x. \text{reachable } x \implies \varphi \ x \longleftrightarrow \psi \ x$
reachable x
<proof>

lemma *Alw-ev-iff-default*:

$\text{Alw-ev } \varphi \ x \longleftrightarrow \text{Alw-ev } \psi \ x$ **if** $\bigwedge x. \text{reachable } x \implies \varphi \ x \longleftrightarrow \psi \ x$
reachable x
<proof>

end

context *Double-Simulation-Complete-Bisim-Cover*

begin

lemma *P2-closure-subs*:

$a \subseteq \bigcup (\text{closure } a)$ **if** $P2 \ a$
<proof>

lemma (in *Double-Simulation-Complete*) *P2-Steps-last*:

P2 (last as) **if** Steps as $a_0 = \text{hd as}$

<proof>

lemma (in *Double-Simulation*) *compatible-closure*:

assumes compatible: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and $\forall x \in a. P x$

shows $\forall x \in \bigcup(\text{closure } a). P x$

<proof>

lemma *compatible-closure-all-iff*:

assumes compatible: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$ **and** *P2 a*

shows $(\forall x \in a. P x) \longleftrightarrow (\forall x \in \bigcup(\text{closure } a). P x)$

<proof>

lemma *compatible-closure-ex-iff*:

assumes compatible: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$ **and** *P2 a*

shows $(\exists x \in a. P x) \longleftrightarrow (\exists x \in \bigcup(\text{closure } a). P x)$

<proof>

lemma (in *Double-Simulation-Complete-Bisim*) *no-deadlock-closureI*:

$\forall x_0 \in \bigcup(\text{closure } a_0). \neg \text{deadlock } x_0$ **if** $\forall x_0 \in a_0. \neg \text{deadlock } x_0$

<proof>

context

fixes *P*

assumes *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

begin

lemma *reaches-all-1*:

fixes *b* :: 'a set **and** *y* :: 'a **and** *as* :: 'a set list

assumes *A*: $\forall y. (\exists x_0 \in \bigcup(\text{closure } (\text{hd as})). \exists xs. \text{hd } xs = x_0 \wedge \text{last } xs = y \wedge \text{steps } xs) \longrightarrow P y$

and $y \in \text{last as}$ **and** $a_0 = \text{hd as}$ **and** Steps *as*

shows $P y$

<proof>

lemma *reaches-all-2*:

fixes $x_0 a xs$

assumes *A*: $\forall b y. (\exists xs. \text{hd } xs = a_0 \wedge \text{last } xs = b \wedge \text{Steps } xs) \wedge y \in b \longrightarrow P y$

and $hd\ xs \in a$ **and** $a \in closure\ a_0$ **and** $steps\ xs$
shows P (*last xs*)
 ⟨*proof*⟩

lemma *reaches-all*:

$(\forall y. (\exists x_0 \in \bigcup (closure\ a_0). reaches\ x_0\ y) \longrightarrow P\ y) \longleftrightarrow (\forall b\ y. Steps.reaches\ a_0\ b \wedge y \in b \longrightarrow P\ y)$
 ⟨*proof*⟩

lemma *reaches-all'*:

$(\forall x_0 \in \bigcup (closure\ a_0). \forall y. reaches\ x_0\ y \longrightarrow P\ y) = (\forall y. Steps.reaches\ a_0\ y \longrightarrow (\forall x \in y. P\ x))$
 ⟨*proof*⟩

lemma *reaches-all''*:

$(\forall y. \forall x_0 \in a_0. reaches\ x_0\ y \longrightarrow P\ y) \longleftrightarrow (\forall b\ y. Steps.reaches\ a_0\ b \wedge y \in b \longrightarrow P\ y)$
 ⟨*proof*⟩

lemma *reaches-ex*:

$(\exists y. \exists x_0 \in \bigcup (closure\ a_0). reaches\ x_0\ y \wedge P\ y) = (\exists b\ y. Steps.reaches\ a_0\ b \wedge y \in b \wedge P\ y)$
 ⟨*proof*⟩

lemma *reaches-ex'*:

$(\exists y. \exists x_0 \in a_0. reaches\ x_0\ y \wedge P\ y) \longleftrightarrow (\exists b\ y. Steps.reaches\ a_0\ b \wedge y \in b \wedge P\ y)$
 ⟨*proof*⟩

end

lemma (*in Double-Simulation-Complete-Bisim*) *P1-deadlocked-compatible*:

deadlocked x = deadlocked y **if** $x \in a$ $y \in a$ *P1 a* **for** $x\ y\ a$

⟨*proof*⟩

lemma *steps-Steps-no-deadlock*:

$\neg Steps.deadlock\ a_0$

if *no-deadlock*: $\forall x_0 \in \bigcup (closure\ a_0). \neg deadlock\ x_0$

⟨*proof*⟩

lemma *steps-Steps-no-deadlock1*:

$\neg Steps.deadlock\ a_0$

if *no-deadlock*: $\forall x_0 \in a_0. \neg deadlock\ x_0$ **and** *closure-simp*: $\bigcup (closure\ a_0) = a_0$

$\langle \text{proof} \rangle$

lemma *Alw-alw-iff*:

$(\forall x_0 \in \bigcup (\text{closure } a_0). \text{Alw-alw } P x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P c) a_0$

if *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and *no-deadlock*: $\forall x_0 \in \bigcup (\text{closure } a_0). \neg \text{deadlock } x_0$

$\langle \text{proof} \rangle$

lemma *Alw-alw-iff1*:

$(\forall x_0 \in a_0. \text{Alw-alw } P x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P c) a_0$

if *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and *no-deadlock*: $\forall x_0 \in a_0. \neg \text{deadlock } x_0$ **and** *closure-simp*: $\bigcup (\text{closure } a_0) = a_0$

$\langle \text{proof} \rangle$

lemma *Alw-alw-iff2*:

$(\forall x_0 \in a_0. \text{Alw-alw } P x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P c) a_0$

if *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and *no-deadlock*: $\forall x_0 \in a_0. \neg \text{deadlock } x_0$

$\langle \text{proof} \rangle$

lemma *Steps-all-Alw-ev*:

$\forall x_0 \in a_0. \text{Alw-ev } P x_0$ **if** *Steps.Alw-ev* $(\lambda a. \forall c \in a. P c) a_0$

$\langle \text{proof} \rangle$

lemma *closure-compatible-Steps-all-ex-iff*:

$\text{Steps.Alw-ev } (\lambda a. \forall c \in a. P c) a_0 \longleftrightarrow \text{Steps.Alw-ev } (\lambda a. \exists c \in a. P c) a_0$

if *closure-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P2 a \implies P x \longleftrightarrow P y$

$\langle \text{proof} \rangle$

lemma *(in -) compatible-imp*:

assumes $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies Q x \longleftrightarrow Q y$

shows $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies (Q x \longrightarrow P x) \longleftrightarrow (Q y \longrightarrow P y)$

$\langle \text{proof} \rangle$

lemma *Leadsto-iff*:

$(\forall x_0 \in \bigcup (\text{closure } a_0). \text{leadsto } P Q x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P c \longrightarrow \text{Alw-ev } Q c) a_0$

if *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$

and *P1-Q*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies Q x \longleftrightarrow Q y$

and no-deadlock: $\forall x_0 \in \bigcup(\text{closure } a_0). \neg \text{deadlock } x_0$
 $\langle \text{proof} \rangle$

lemma *Leadsto-iff1:*

$(\forall x_0 \in a_0. \text{leadsto } P \ Q \ x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P \ c \longrightarrow \text{Alw-ev } Q \ c) \ a_0$
if $P1\text{-}P: \bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies P \ x \longleftrightarrow P \ y$
and $P1\text{-}Q: \bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies Q \ x \longleftrightarrow Q \ y$
and no-deadlock: $\forall x_0 \in a_0. \neg \text{deadlock } x_0$ **and** *closure-simp:* $\bigcup(\text{closure } a_0) = a_0$
 $\langle \text{proof} \rangle$

lemma *Leadsto-iff2:*

$(\forall x_0 \in a_0. \text{leadsto } P \ Q \ x_0) \longleftrightarrow \text{Steps.Alw-alw } (\lambda a. \forall c \in a. P \ c \longrightarrow \text{Alw-ev } Q \ c) \ a_0$
if $P1\text{-}P: \bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies P \ x \longleftrightarrow P \ y$
and $P1\text{-}Q: \bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies Q \ x \longleftrightarrow Q \ y$
and no-deadlock: $\forall x_0 \in a_0. \neg \text{deadlock } x_0$
 $\langle \text{proof} \rangle$

lemma (**in** $-$) *compatible-convert1:*

assumes $\bigwedge x \ y \ a. P \ x \implies x \in a \implies y \in a \implies P1 \ a \implies P \ y$
shows $\bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies P \ x \longleftrightarrow P \ y$
 $\langle \text{proof} \rangle$

lemma (**in** $-$) *compatible-convert2:*

assumes $\bigwedge a \ x \ y. x \in a \implies y \in a \implies P1 \ a \implies P \ x \longleftrightarrow P \ y$
shows $\bigwedge x \ y \ a. P \ x \implies x \in a \implies y \in a \implies P1 \ a \implies P \ y$
 $\langle \text{proof} \rangle$

lemma (**in** *Double-Simulation-Defs*)

assumes *compatible:* $\bigwedge x \ y \ a. P \ x \implies x \in a \implies y \in a \implies P1 \ a \implies P \ y$
and that: $\forall x \in a. P \ x$
shows $\forall x \in \bigcup(\text{closure } a). P \ x$
 $\langle \text{proof} \rangle$

end

context *Double-Simulation-Finite-Complete-Bisim-Cover*

begin

lemma *Alw-ev-Steps-ex:*

$(\forall x_0 \in \bigcup(\text{closure } a_0). \text{Alw-ev } P \ x_0) \longrightarrow \text{Steps.Alw-ev } (\lambda a. \exists c \in a. P$

c) a_0
if *closure-P*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies P2 a$
 $\implies P x \longleftrightarrow P y$
 $\langle \text{proof} \rangle$

lemma *Alw-ev-Steps-ex2*:
 $(\forall x_0 \in a_0. \text{Alw-ev } P x_0) \longrightarrow \text{Steps.Alw-ev } (\lambda a. \exists c \in a. P c) a_0$
if *closure-P*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies P2 a$
 $\implies P x \longleftrightarrow P y$
and *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$
 $\langle \text{proof} \rangle$

lemma *Alw-ev-Steps-ex1*:
 $(\forall x_0 \in a_0. \text{Alw-ev } P x_0) \longrightarrow \text{Steps.Alw-ev } (\lambda a. \exists c \in a. P c) a_0$ **if**
 $\bigcup(\text{closure } a_0) = a_0$
and *closure-P*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies P2 a$
 $\implies P x \longleftrightarrow P y$
 $\langle \text{proof} \rangle$

lemma *closure-compatible-Alw-ev-Steps-iff*:
 $(\forall x_0 \in a_0. \text{Alw-ev } P x_0) \longleftrightarrow \text{Steps.Alw-ev } (\lambda a. \forall c \in a. P c) a_0$
if *closure-P*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies P2 a$
 $\implies P x \longleftrightarrow P y$
and *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$
 $\langle \text{proof} \rangle$

lemma *Leadsto-iff'*:
 $(\forall x_0 \in a_0. \text{leadsto } P Q x_0)$
 $\longleftrightarrow \text{Steps.Alw-aw } (\lambda a. (\forall c \in a. P c) \longrightarrow \text{Steps.Alw-ev } (\lambda a. \forall c \in a. Q c) a) a_0$
if *P1-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies P x \longleftrightarrow P y$
and *P1-Q*: $\bigwedge a x y. x \in a \implies y \in a \implies P1 a \implies Q x \longleftrightarrow Q y$
and *closure-Q*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies$
 $P2 a \implies Q x \longleftrightarrow Q y$
and *closure-P*: $\bigwedge a x y. x \in a \implies y \in a \implies P2 a \implies P x \longleftrightarrow P y$
and *no-deadlock*: $\forall x_0 \in a_0. \neg \text{deadlock } x_0$ **and** *closure-simp*: $\bigcup(\text{closure } a_0) = a_0$
 $\langle \text{proof} \rangle$

context
fixes $P :: 'a \Rightarrow \text{bool}$ — The property we want to check
assumes *closure-P*: $\bigwedge a x y. x \in \bigcup(\text{closure } a) \implies y \in \bigcup(\text{closure } a) \implies$
 $P2 a \implies P x \longleftrightarrow P y$
and *P1-P*: $\bigwedge a x y. P x \implies x \in a \implies y \in a \implies P1 a \implies P y$

begin

lemma *run-aw-ev-bisim*:

$run (x \#\# xs) \implies x \in \bigcup (closure\ a_0) \implies alw (ev (holds\ P))\ xs$
 $\implies \exists y\ ys. y \in a_0 \wedge run (y \#\# ys) \wedge alw (ev (holds\ P))\ ys$
<proof>

lemma *φ -closure-compatible*:

$P\ a \implies x \in \bigcup (closure\ a) \implies P\ x \longleftrightarrow (\forall x \in \bigcup (closure\ a). P\ x)$
<proof>

theorem *infinite-buechi-run-cycle-iff*:

$(\exists x_0\ xs. x_0 \in \bigcup (closure\ a_0) \wedge run (x_0 \#\# xs) \wedge alw (ev (holds\ P)) (x_0$
 $\#\# xs))$
 $\longleftrightarrow (\exists as\ a\ bs. a \neq \{\}) \wedge post-defs.Steps (closure\ a_0 \# as @ a \# bs @$
 $[a]) \wedge (\forall x \in \bigcup a. P\ x)$
<proof>

end

end

Possible Solution

context *Graph-Invariant*

begin

definition *E-inv* $x\ y \equiv E\ x\ y \wedge P\ x \wedge P\ y$

lemma *bisim-E-inv*:

Bisimulation-Invariant $E\ E\text{-inv} (=) P\ P$
<proof>

interpretation *G-inv*: *Graph-Defs* *E-inv* *<proof>*

lemma *steps-G-inv-steps*:

$steps (x \# xs) \longleftrightarrow G\text{-inv.steps} (x \# xs)$ **if** $P\ x$
<proof>

end

R-of/from-R **definition** *R-of* $lR = snd\ 'lR$

definition *from-R* $lR = \{(l, u) \mid u. u \in R\}$

lemma *from-R-fst*:

$\forall x \in \text{from-R } l \ R. \text{fst } x = l$
 $\langle \text{proof} \rangle$

lemma *R-of-from-R* [simp]:
 $R\text{-of } (\text{from-R } l \ R) = R$
 $\langle \text{proof} \rangle$

lemma *from-R-loc*:
 $l' = l$ **if** $(l', u) \in \text{from-R } l \ Z$
 $\langle \text{proof} \rangle$

lemma *from-R-val*:
 $u \in Z$ **if** $(l', u) \in \text{from-R } l \ Z$
 $\langle \text{proof} \rangle$

lemma *from-R-R-of*:
 $\text{from-R } l \ (R\text{-of } S) = S$ **if** $\forall x \in S. \text{fst } x = l$
 $\langle \text{proof} \rangle$

lemma *R-of-I*[intro]:
 $Z \in R\text{-of } S$ **if** $(l, Z) \in S$
 $\langle \text{proof} \rangle$

lemma *from-R-I*[intro]:
 $(l', u') \in \text{from-R } l' \ Z'$ **if** $u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *R-of-non-emptyD*:
 $a \neq \{\}$ **if** $R\text{-of } a \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *R-of-empty*[simp]:
 $R\text{-of } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *fst-simp*:
 $x = l$ **if** $\forall x \in a. \text{fst } x = l \ (x, y) \in a$
 $\langle \text{proof} \rangle$

lemma *from-R-D*:
 $u \in Z$ **if** $(l', u) \in \text{from-R } l \ Z$
 $\langle \text{proof} \rangle$

locale *Double-Simulation-paired-Defs* =

fixes $C :: ('a \times 'b) \Rightarrow ('a \times 'b) \Rightarrow bool$ — Concrete step relation
and $A1 :: ('a \times 'b \text{ set}) \Rightarrow ('a \times 'b \text{ set}) \Rightarrow bool$
— Step relation for the first abstraction layer
and $P1 :: ('a \times 'b \text{ set}) \Rightarrow bool$ — Valid states of the first abstraction layer
layer
and $A2 :: ('a \times 'b \text{ set}) \Rightarrow ('a \times 'b \text{ set}) \Rightarrow bool$
— Step relation for the second abstraction layer
and $P2 :: ('a \times 'b \text{ set}) \Rightarrow bool$ — Valid states of the second abstraction layer
layer
begin

definition

$$\begin{aligned}
A1' = & (\lambda lR lR'. \exists l l'. (\forall x \in lR. \text{fst } x = l) \wedge (\forall x \in lR'. \text{fst } x = l') \\
& \wedge P1 (l, R\text{-of } lR) \wedge A1 (l, R\text{-of } lR) (l', R\text{-of } lR') \\
&)
\end{aligned}$$

definition

$$\begin{aligned}
A2' = & (\lambda lR lR'. \exists l l'. (\forall x \in lR. \text{fst } x = l) \wedge (\forall x \in lR'. \text{fst } x = l') \\
& \wedge P2 (l, R\text{-of } lR) \wedge A2 (l, R\text{-of } lR) (l', R\text{-of } lR') \\
&)
\end{aligned}$$

definition

$$P1' = (\lambda lR. \exists l. (\forall x \in lR. \text{fst } x = l) \wedge P1 (l, R\text{-of } lR))$$

definition

$$P2' = (\lambda lR. \exists l. (\forall x \in lR. \text{fst } x = l) \wedge P2 (l, R\text{-of } lR))$$

definition $\text{closure}' l a = \{x. P1 (l, x) \wedge a \cap x \neq \{\}\}$

sublocale $\text{sim: Double-Simulation-Defs } C A1' P1' A2' P2' \langle \text{proof} \rangle$

end

locale $\text{Double-Simulation-paired} = \text{Double-Simulation-paired-Defs} +$

assumes $\text{prestable: } P1 (l, S) \Longrightarrow A1 (l, S) (l', T) \Longrightarrow \forall s \in S. \exists s' \in T. C (l, s) (l', s')$

and $\text{closure-poststable:}$

$$s' \in \text{closure}' l' y \Longrightarrow P2 (l, x) \Longrightarrow A2 (l, x) (l', y)$$

$$\Longrightarrow \exists s \in \text{closure}' l x. A1 (l, s) (l', s')$$

and $P1\text{-distinct: } P1 (l, x) \Longrightarrow P1 (l, y) \Longrightarrow x \neq y \Longrightarrow x \cap y = \{\}$

and $P1\text{-finite: } \text{finite } \{(l, x). P1 (l, x)\}$

and $P2\text{-cover: } P2 (l, a) \Longrightarrow \exists x. P1 (l, x) \wedge x \cap a \neq \{\}$

begin

sublocale *sim: Double-Simulation* $C A1' P1' A2' P2'$
<proof>

context

assumes *P2-invariant*: $P2\ a \implies A2\ a\ a' \implies P2\ a'$

begin

lemma *A2-A2'-bisim: Bisimulation-Invariant* $A2\ A2' \lambda\ (l, Z)\ b.\ b =$
from-R l Z) P2 P2'

<proof>

end

end

locale *Double-Simulation-Complete-paired = Double-Simulation-paired +*

fixes $l_0\ a_0$

assumes *complete*: $C\ (l, x)\ (l', y) \implies x \in S \implies P2\ (l, S) \implies \exists\ T.\ A2$
 $(l, S)\ (l', T) \wedge y \in T$

assumes *P2-invariant*: $P2\ a \implies A2\ a\ a' \implies P2\ a'$

and *P2-a0'*: $P2\ (l_0, a_0)$

begin

interpretation *Bisimulation-Invariant* $A2\ A2' \lambda\ (l, Z)\ b.\ b =$ *from-R l Z*
 $P2\ P2'$

<proof>

sublocale *Double-Simulation-Complete* $C\ A1' P1' A2' P2'$ *from-R l0 a0*

<proof>

sublocale *P2-invariant'*: *Graph-Invariant-Start* $A2\ (l_0, a_0)\ P2$

<proof>

end

locale *Double-Simulation-Finite-Complete-paired = Double-Simulation-Complete-paired*

+

assumes *finite-abstract-reachable*: *finite* $\{(l, a).\ A2^{**}\ (l_0, a_0)\ (l, a) \wedge P2$
 $(l, a)\}$

begin

interpretation *Bisimulation-Invariant* $A2\ A2' \lambda\ (l, Z)\ b.\ b =$ *from-R l Z*
 $P2\ P2'$

$\langle \text{proof} \rangle$

sublocale *Double-Simulation-Finite-Complete* C $A1'$ $P1'$ $A2'$ $P2'$ *from-R*

l_0 a_0

$\langle \text{proof} \rangle$

end

locale *Double-Simulation-Complete-Bisim-paired* = *Double-Simulation-Complete-paired*

+

assumes *A1-complete*: $C(l, x) (l', y) \implies P1(l, S) \implies x \in S \implies \exists T.$

$A1(l, S) (l', T) \wedge y \in T$

and *P1-invariant*: $P1(l, S) \implies A1(l, S) (l', T) \implies P1(l', T)$

begin

sublocale *Double-Simulation-Complete-Bisim* C $A1'$ $P1'$ $A2'$ $P2'$ *from-R*

l_0 a_0

$\langle \text{proof} \rangle$

end

locale *Double-Simulation-Finite-Complete-Bisim-paired* = *Double-Simulation-Finite-Complete-paired*

+

Double-Simulation-Complete-Bisim-paired

begin

sublocale *Double-Simulation-Finite-Complete-Bisim* C $A1'$ $P1'$ $A2'$ $P2'$

from-R l_0 a_0 $\langle \text{proof} \rangle$

end

locale *Double-Simulation-Complete-Bisim-Cover-paired* =

Double-Simulation-Complete-Bisim-paired +

assumes *P2-P1-cover*: $P2(l, a) \implies x \in a \implies \exists a'. a \cap a' \neq \{\} \wedge P1$

$(l, a') \wedge x \in a'$

begin

sublocale *Double-Simulation-Complete-Bisim-Cover* C $A1'$ $P1'$ $A2'$ $P2'$

from-R l_0 a_0

$\langle \text{proof} \rangle$

end

locale *Double-Simulation-Finite-Complete-Bisim-Cover-paired* =

Double-Simulation-Complete-Bisim-Cover-paired +
Double-Simulation-Finite-Complete-Bisim-paired
begin

sublocale *Double-Simulation-Finite-Complete-Bisim-Cover* C $A1'$ $P1'$ $A2'$
 $P2'$ *from-R* l_0 a_0 \langle *proof* \rangle

end

locale *Double-Simulation-Complete-Abstraction-Prop-paired* =
Double-Simulation-Complete-paired +
fixes $P :: 'a \Rightarrow bool$ — The property we want to check
assumes *P2-non-empty*: $P2$ $(l, a) \Longrightarrow a \neq \{\}$
begin

definition $\varphi = P \circ fst$

lemma *P2-φ*:
 $a \cap Collect \varphi = a$ **if** $P2' a a \cap Collect \varphi \neq \{\}$
 \langle *proof* \rangle

sublocale *Double-Simulation-Complete-Abstraction-Prop* C $A1'$ $P1'$ $A2'$
 $P2'$ *from-R* l_0 a_0 φ
 \langle *proof* \rangle

end

locale *Double-Simulation-Finite-Complete-Abstraction-Prop-paired* =
Double-Simulation-Complete-Abstraction-Prop-paired +
Double-Simulation-Finite-Complete-paired
begin

sublocale *Double-Simulation-Finite-Complete-Abstraction-Prop* C $A1'$ $P1'$
 $A2'$ $P2'$ *from-R* l_0 a_0 φ \langle *proof* \rangle

end

locale *Double-Simulation-Complete-Abstraction-Prop-Bisim-paired* =
Double-Simulation-Complete-Abstraction-Prop-paired +
Double-Simulation-Complete-Bisim-paired
begin

interpretation *bisim*: *Bisimulation-Invariant* $A2$ $A2' \lambda (l, Z) b. b = \text{from-R}$
 $l Z P2 P2'$

$\langle \text{proof} \rangle$

sublocale *Double-Simulation-Complete-Abstraction-Prop-Bisim*
 $C A1' P1' A2' P2'$ *from-R* $l_0 a_0 \varphi$ $\langle \text{proof} \rangle$

lemma *P2'-non-empty*:

$P2' a \implies a \neq \{\}$

$\langle \text{proof} \rangle$

lemma *from-R-int-φ[simp]*:

from-R $l R \cap \text{Collect } \varphi = \text{from-R } l R$ **if** $P l$

$\langle \text{proof} \rangle$

interpretation G_φ : *Graph-Start-Defs*

$\lambda (l, Z) (l', Z'). A2 (l, Z) (l', Z') \wedge P l' (l_0, a_0)$ $\langle \text{proof} \rangle$

interpretation *Bisimulation-Invariant* $\lambda (l, Z) (l', Z'). A2 (l, Z) (l', Z')$
 $\wedge P l'$

$A2\text{-}\varphi \lambda (l, Z) b. b = \text{from-R } l Z P2 P2'$

$\langle \text{proof} \rangle$

lemma *from-R-subst-φ*:

from-R $l a \subseteq \text{Collect } \varphi$ **if** $P l$

$\langle \text{proof} \rangle$

lemma *P2'-from-R*:

$\exists l' Z'. x = \text{from-R } l' Z'$ **if** $P2' x$

$\langle \text{proof} \rangle$

lemma *P2-from-R-list'*:

$\exists as'. \text{map } (\lambda(x, y). \text{from-R } x y) as' = as$ **if** *list-all* $P2' as$

$\langle \text{proof} \rangle$

end

locale *Double-Simulation-Finite-Complete-Abstraction-Prop-Bisim-paired* =
Double-Simulation-Complete-Abstraction-Prop-Bisim-paired +
Double-Simulation-Finite-Complete-Bisim-paired

begin

interpretation *bisim*: *Bisimulation-Invariant* $A2 A2' \lambda (l, Z) b. b = \text{from-R}$
 $l Z P2 P2'$

$\langle \text{proof} \rangle$

sublocale *Double-Simulation-Finite-Complete-Abstraction-Prop-Bisim*
 $C A1' P1' A2' P2'$ from-R $l_0 a_0 \varphi$ $\langle proof \rangle$

interpretation G_φ : *Graph-Start-Defs*
 $\lambda (l, Z) (l', Z'). A2 (l, Z) (l', Z') \wedge P l' (l_0, a_0)$ $\langle proof \rangle$

interpretation *Bisimulation-Invariant* $\lambda (l, Z) (l', Z'). A2 (l, Z) (l', Z')$
 $\wedge P l'$
 $A2\text{-}\varphi \lambda (l, Z) b. b = \text{from-R } l Z P2 P2'$
 $\langle proof \rangle$

theorem *Alw-ev-mc*:
 $(\forall x_0 \in a_0. \text{sim.Alw-ev } (\text{Not} \circ \varphi) (l_0, x_0)) \longleftrightarrow$
 $\neg P l_0 \vee (\nexists as \ a \ bs. G_\varphi.steps ((l_0, a_0) \# as @ a \# bs @ [a]))$
 $\langle proof \rangle$

theorem *Alw-ev-mc1*:
 $(\forall x_0 \in a_0. \text{sim.Alw-ev } (\text{Not} \circ \varphi) (l_0, x_0)) \longleftrightarrow \neg (P l_0 \wedge (\exists a. G_\varphi.reachable$
 $a \wedge G_\varphi.reaches1 a a))$
 $\langle proof \rangle$

end

context *Double-Simulation-Complete-Bisim-Cover-paired*
begin

interpretation *bisim*: *Bisimulation-Invariant* $A2 A2' \lambda (l, Z) b. b = \text{from-R}$
 $l Z P2 P2'$
 $\langle proof \rangle$

interpretation *Start*: *Double-Simulation-Complete-Abstraction-Prop-Bisim-paired*
 $C A1 P1 A2 P2 l_0 a_0 \lambda \neg. \text{True}$
 $\langle proof \rangle$

lemma *sim-reaches-equiv*:
 $P2\text{-invariant}.reaches (l, Z) (l', Z') \longleftrightarrow \text{sim.Steps}.reaches (\text{from-R } l Z)$
 $(\text{from-R } l' Z')$
if $P2 (l, Z)$
 $\langle proof \rangle$

lemma *reaches-all*:
assumes
 $\bigwedge u u' R l. u \in R \implies u' \in R \implies P1 (l, R) \implies P l u \longleftrightarrow P l u'$
shows

$$\begin{aligned}
& (\forall u. (\exists x_0 \in \bigcup (\text{sim.closure } (\text{from-R } l_0 \ a_0)). \text{sim.reaches } x_0 \ (l, u)) \longrightarrow \\
& P \ l \ u) \longleftrightarrow \\
& (\forall Z \ u. P2\text{-invariant}'.\text{reaches } (l_0, a_0) \ (l, Z) \wedge u \in Z \longrightarrow P \ l \ u) \\
& \langle \text{proof} \rangle
\end{aligned}$$

context

fixes $P \ Q :: 'a \Rightarrow \text{bool}$ — The state properties we want to check

begin

definition $\varphi' = P \circ \text{fst}$

definition $\psi = Q \circ \text{fst}$

lemma ψ -closure-compatible:

$$\psi \ (l, x) \Longrightarrow x \in a \Longrightarrow y \in a \Longrightarrow P1 \ (l, a) \Longrightarrow \psi \ (l, y)$$

$\langle \text{proof} \rangle$

lemma ψ -closure-compatible':

$$(\text{Not } \circ \psi) \ (l, x) \Longrightarrow x \in a \Longrightarrow y \in a \Longrightarrow P1 \ (l, a) \Longrightarrow (\text{Not } \circ \psi) \ (l, y)$$

$\langle \text{proof} \rangle$

lemma $P1$ - $P1'$:

$$R \neq \{\} \Longrightarrow P1 \ (l, R) \Longrightarrow P1' \ (\text{from-R } l \ R)$$

$\langle \text{proof} \rangle$

lemma ψ -Alw-ev-compatible:

assumes $u \in R \ u' \in R \ P1 \ (l, R)$

shows $\text{sim.Alw-ev } (\text{Not } \circ \psi) \ (l, u) = \text{sim.Alw-ev } (\text{Not } \circ \psi) \ (l, u')$

$\langle \text{proof} \rangle$

interpretation $\text{Graph-Start-Defs } A2 \ (l_0, a_0) \langle \text{proof} \rangle$

interpretation G_ψ : Graph-Start-Defs

$$\lambda \ (l, Z) \ (l', Z'). A2 \ (l, Z) \ (l', Z') \wedge Q \ l' \ (l_0, a_0) \langle \text{proof} \rangle$$

end

end

context $\text{Double-Simulation-Finite-Complete-Bisim-Cover-paired}$

begin

interpretation bisim : $\text{Bisimulation-Invariant } A2 \ A2' \ \lambda \ (l, Z) \ b. b = \text{from-R } l \ Z \ P2 \ P2'$

$\langle proof \rangle$

context

fixes $P Q :: 'a \Rightarrow bool$ — The state properties we want to check
begin

interpretation $Graph-Start-Defs A2 (l_0, a_0) \langle proof \rangle$

interpretation $G_\psi: Graph-Start-Defs$

$\lambda (l, Z) (l', Z'). A2 (l, Z) (l', Z') \wedge Q l' (l_0, a_0) \langle proof \rangle$

lemma $Alw-ev-mc1:$

$(\forall x_0 \in from-R\ l\ Z. sim.Alw-ev (Not \circ \psi\ Q) x_0) \longleftrightarrow$
 $\neg (Q\ l \wedge (\exists a. G_\psi.reaches (l, Z) a \wedge G_\psi.reaches1 a a))$

if $P2-invariant'.reachable (l, Z)$ **for** $l\ Z$

$\langle proof \rangle$

theorem $leadsto-mc1:$

$(\forall x_0 \in a_0. sim.leadsto (\varphi' P) (Not \circ \psi\ Q) (l_0, x_0)) \longleftrightarrow$
 $(\nexists x. P2-invariant'.reaches (l_0, a_0) x \wedge P (fst x) \wedge Q (fst x)$
 $\wedge (\exists a. G_\psi.reaches x a \wedge G_\psi.reaches1 a a)$
 $)$

if $no-deadlock: \forall x_0 \in a_0. \neg sim.deadlock (l_0, x_0)$

$\langle proof \rangle$

end

end

The second bisimulation property in prestable and complete simulation graphs. **context** $Simulation-Graph-Complete-Prestable$

begin

lemma $C-A-bisim:$

$Bisimulation-Invariant\ C\ A\ (\lambda x a. x \in a)\ (\lambda -. True)\ P$

$\langle proof \rangle$

interpretation $Bisimulation-Invariant\ C\ A\ \lambda x a. x \in a\ \lambda -. True\ P$

$\langle proof \rangle$

lemma $C-A-Leadsto-iff:$

fixes $\varphi\ \psi :: 'a \Rightarrow bool$

assumes $\varphi-compatible: \bigwedge x\ y\ a. \varphi\ x \implies x \in a \implies y \in a \implies P\ a \implies$

φy
and ψ -compatible: $\bigwedge x y a. \psi x \implies x \in a \implies y \in a \implies P a \implies \psi y$
and $x \in a P a$
shows $leadsto \varphi \psi x = Steps.leadsto (\lambda a. \forall x \in a. \varphi x) (\lambda a. \forall x \in a. \psi x) a$
 $\langle proof \rangle$
end

Comments

- Pre-stability can easily be extended to infinite runs (see construction with *sscan* above)
- Post-stability can not
- Pre-stability + Completeness means that for every two concrete states in the same abstract class, there are equivalent runs
- For Büchi properties, the predicate has to be compatible with whole closures instead of single *P1*-states. This is because for a finite graph where every node has at least indegree one, we cannot necessarily conclude that there is a cycle through *every* node.

locale *Graph-Abstraction* =
Graph-Defs **for** $A :: 'a set \Rightarrow 'a set \Rightarrow bool +$
fixes $\alpha :: 'a set \Rightarrow 'a set$
assumes *idempotent*: $\alpha(\alpha(x)) = \alpha(x)$
assumes *enlarging*: $x \subseteq \alpha(x)$
assumes *α -mono*: $x \subseteq y \implies \alpha(x) \subseteq \alpha(y)$
assumes *mono*: $a \subseteq a' \implies A a b \implies \exists b'. b \subseteq b' \wedge A a' b'$
assumes *finite-abstraction*: *finite* (α ' *UNIV*)
begin

definition *E* **where** $E a b \equiv \exists b'. A a b' \wedge b = \alpha(b')$

interpretation *sim1*: *Simulation-Invariant* $A E \lambda a b. \alpha(a) \subseteq b \lambda-. True$
 $\lambda-. True$
 $\langle proof \rangle$

interpretation *sim2*: *Simulation-Invariant* $A E \lambda a b. a \subseteq b \lambda-. True \lambda x.$
 $\alpha(x) = x$
 $\langle proof \rangle$

This variant needs the least assumptions.

interpretation *sim3*: *Simulation-Invariant A E λ a b. a ⊆ b λ-. True λ-. True*
True
⟨proof⟩

interpretation *sim4*: *Simulation-Invariant A E λ a b. a ⊆ b λ-. True λ a. ∃ a'. α a' = a*
True λ a.
∃ a'. α a' = a
⟨proof⟩

end

lemmas [*simp del*] = *holds.simps*

end

theory *Simulation-Graphs-TA*

imports *Simulation-Graphs DBM-Zone-Semantics Approx-Beta*

begin

7.9 Instantiation of Simulation Locales

inductive *step-trans* ::

$(\ 'a, \ 'c, \ 't, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ ('t::time)) \ cval \Rightarrow \ (('c, \ 't) \ cconstraint \times \ 'a \times \ 'c \ list)$
 $\Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ cval \Rightarrow \ bool$
 $(\ \langle - \vdash_t \langle -, - \rangle \rightarrow_- \langle -, - \rangle \ [61,61,61] \ 61)$

where

$\llbracket A \vdash l \xrightarrow{g,a,r} l'; u \vdash g; u' \vdash \text{inv-of } A \ l'; u' = [r \rightarrow 0]u \rrbracket$
 $\Longrightarrow (A \vdash_t \langle l, u \rangle \rightarrow_{(g,a,r)} \langle l', u' \rangle)$

inductive *step-trans'* ::

$(\ 'a, \ 'c, \ 't, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ ('t::time)) \ cval \Rightarrow \ ('c, \ 't) \ cconstraint \times \ 'a \times \ 'c \ list$
 $\Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ cval \Rightarrow \ bool$
 $(\ \langle - \vdash'' \langle -, - \rangle \rightarrow^{\cdot} \langle -, - \rangle \ [61,61,61,61] \ 61)$

where

step': $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow A \vdash_t \langle l', u' \rangle \rightarrow_t \langle l'', u'' \rangle \Longrightarrow A \vdash' \langle l, u \rangle \rightarrow^t \langle l'', u'' \rangle$

inductive *step-trans-z* ::

$(\ 'a, \ 'c, \ 't, \ 's) \ ta \Rightarrow \ 's \Rightarrow \ ('c, \ ('t::time)) \ zone$
 $\Rightarrow \ (('c, \ 't) \ cconstraint \times \ 'a \times \ 'c \ list) \ action \Rightarrow \ 's \Rightarrow \ ('c, \ 't) \ zone \Rightarrow \ bool$
 $(\ \langle - \vdash \langle -, - \rangle \rightsquigarrow^{\cdot} \langle -, - \rangle \ [61,61,61,61] \ 61)$

where

step-trans-t-z:

$A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l, Z^{\uparrow} \cap \{u. u \vdash \text{inv-of } A \ l\} \rangle \mid$

step-trans-a-z:
 $A \vdash \langle l, Z \rangle \rightsquigarrow^{1(g,a,r)} \langle l', \text{zone-set } (Z \cap \{u. u \vdash g\}) \ r \cap \{u. u \vdash \text{inv-of } A \ l'\} \rangle$
if $A \vdash l \longrightarrow^{g,a,r} l'$

inductive *step-trans-z'* ::

$(\ 'a, 'c, 't, 's) \text{ ta} \Rightarrow 's \Rightarrow ('c, ('t::\text{time})) \text{ zone} \Rightarrow (('c, 't) \text{ cconstraint} \times 'a \times 'c \text{ list})$
 $\Rightarrow 's \Rightarrow ('c, 't) \text{ zone} \Rightarrow \text{bool}$
 $(\langle - \vdash'' \langle -, - \rangle \rightsquigarrow^{\tau} \langle -, - \rangle \rangle [61,61,61,61] \ 61)$

where

step-trans-z':
 $A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l, Z' \rangle \Longrightarrow A \vdash \langle l, Z' \rangle \rightsquigarrow^{1t} \langle l', Z'' \rangle \Longrightarrow A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z'' \rangle$

lemmas [*intro*] =

step-trans.intros
step-trans'.intros
step-trans-z.intros
step-trans-z'.intros

context

notes [*elim!*] =

step.cases step-t.cases
step-trans.cases step-trans'.cases step-trans-z.cases step-trans-z'.cases

begin

lemma *step-trans-t-z-sound:*

$A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l', Z' \rangle \Longrightarrow \forall u' \in Z'. \exists u \in Z. \exists d. A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-trans-a-z-sound:*

$A \vdash \langle l, Z \rangle \rightsquigarrow^{1t} \langle l', Z' \rangle \Longrightarrow \forall u' \in Z'. \exists u \in Z. \exists d. A \vdash_t \langle l, u \rangle \rightarrow_t \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-trans-a-z-complete:*

$A \vdash_t \langle l, u \rangle \rightarrow_t \langle l', u' \rangle \Longrightarrow u \in Z \Longrightarrow \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow^{1t} \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *step-trans-t-z-complete:*

$A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \Longrightarrow u \in Z \Longrightarrow \exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l', Z' \rangle \wedge u' \in Z'$
 $\langle \text{proof} \rangle$

lemma *step-trans-t-z-iff*:

$$A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l', Z' \rangle = A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l', Z' \rangle$$

<proof>

lemma *step-z-complete*:

$$A \vdash \langle l, u \rangle \rightarrow \langle l', u' \rangle \implies u \in Z \implies \exists Z' t. A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle \wedge u' \in Z'$$

<proof>

lemma *step-trans-a-z-exact*:

$$u' \in Z' \text{ if } A \vdash_t \langle l, u \rangle \rightarrow_t \langle l', u' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow^{1t} \langle l', Z' \rangle \text{ if } u \in Z$$

<proof>

lemma *step-trans-t-z-exact*:

$$u' \in Z' \text{ if } A \vdash \langle l, u \rangle \rightarrow^d \langle l', u' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow^{\tau} \langle l', Z' \rangle \text{ if } u \in Z$$

<proof>

lemma *step-trans-z'-exact*:

$$u' \in Z' \text{ if } A \vdash' \langle l, u \rangle \rightarrow^t \langle l', u' \rangle \text{ if } A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle \text{ if } u \in Z$$

<proof>

lemma *step-trans-z-step-z-action*:

$$A \vdash \langle l, Z \rangle \rightsquigarrow_{1a} \langle l', Z' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow^{1(g,a,r)} \langle l', Z' \rangle$$

<proof>

lemma *step-trans-z-step-z*:

$$\exists a. A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle$$

<proof>

lemma *step-z-step-trans-z-action*:

$$\exists g r. A \vdash \langle l, Z \rangle \rightsquigarrow^{1(g,a,r)} \langle l', Z' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow_{1a} \langle l', Z' \rangle$$

<proof>

lemma *step-z-step-trans-z*:

$$\exists t. A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle$$

<proof>

end

lemma *step-z'-step-trans-z'*:

$$\exists t. A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle$$

<proof>

lemma *step-trans-z'-step-z'*:

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z' \rangle$ **if** $A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z' \rangle$
 ⟨proof⟩

lemma *step-trans-z-determ*:

$Z1 = Z2$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z1 \rangle$ $A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z2 \rangle$
 ⟨proof⟩

lemma *step-trans-z'-determ*:

$Z1 = Z2$ **if** $A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z1 \rangle$ $A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z2 \rangle$
 ⟨proof⟩

lemma (in *Alpha-defs*) *step-trans-z-V*: $A \vdash \langle l, Z \rangle \rightsquigarrow^t \langle l', Z \rangle \implies Z \subseteq V$
 $\implies Z' \subseteq V$

⟨proof⟩

7.9.1 Additional Lemmas on Regions

context *AlphaClosure*

begin

inductive *step-trans-r* ::

$(\text{'a}, \text{'c}, t, \text{'s}) \text{ta} \Rightarrow - \Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow ((\text{'c}, t) \text{cconstraint} \times \text{'a} \times \text{'c}$
list) *action*

$\Rightarrow \text{'s} \Rightarrow (\text{'c}, t) \text{zone} \Rightarrow \text{bool}$

$(\text{<-}, \vdash \text{<-}, -) \rightsquigarrow^{\tau} \text{<-}, - \rangle [61, 61, 61, 61, 61] \text{ 61}$

where

step-trans-t-r:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{\tau} \langle l', R' \rangle$ **if**

valid-abstraction $A \ X \ (\lambda x. \text{real } o \ k \ x) \ R \in \mathcal{R} \ l \ R' \in \text{Succ } (\mathcal{R} \ l) \ R \ R' \subseteq$
 $\{\text{inv-of } A \ l\} \mid$

step-trans-a-r:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{1(g, a, r)} \langle l', R' \rangle$ **if**

valid-abstraction $A \ X \ (\lambda x. \text{real } o \ k \ x) \ A \vdash l \longrightarrow^{g, a, r} l' \ R \in \mathcal{R} \ l$
 $R \subseteq \{\text{g}\} \ \text{region-set}' \ R \ r \ \emptyset \subseteq R' \ R' \subseteq \{\text{inv-of } A \ l'\} \ R' \in \mathcal{R} \ l'$

lemmas [*intro*] = *step-trans-r.intros*

lemma *step-trans-t-r-iff[simp]*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{\tau} \langle l', R' \rangle = A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{\tau} \langle l', R' \rangle$
 ⟨proof⟩

lemma *step-trans-r-step-r-action*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{1a} \langle l', R' \rangle$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{1(g, a, r)} \langle l', R' \rangle$

$\langle \text{proof} \rangle$

lemma *step-r-step-trans-r-action*:

$\exists g r. A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{1(g,a,r)} \langle l', R' \rangle$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{1a} \langle l', R' \rangle$
 $\langle \text{proof} \rangle$

inductive *step-trans-r'* ::

$('a, 'c, t, 's) ta \Rightarrow - \Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow ('c, t) cconstraint \times 'a \times 'c$
list
 $\Rightarrow 's \Rightarrow ('c, t) zone \Rightarrow bool$
 $(\langle -, - \vdash'' \langle -, - \rangle \rightsquigarrow^{\cdot} \langle -, - \rangle) [61, 61, 61, 61, 61] 61$

where

$A, \mathcal{R} \vdash' \langle l, R \rangle \rightsquigarrow^t \langle l', R' \rangle$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{\tau} \langle l, R' \rangle$ $A, \mathcal{R} \vdash \langle l, R' \rangle \rightsquigarrow^{1t} \langle l', R'' \rangle$

lemma *step-trans-r'-step-r'*:

$A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$ **if** $A, \mathcal{R} \vdash' \langle l, R \rangle \rightsquigarrow^{(g,a,r)} \langle l', R' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-r'-step-trans-r'*:

$\exists g r. A, \mathcal{R} \vdash' \langle l, R \rangle \rightsquigarrow^{(g,a,r)} \langle l', R' \rangle$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-trans-a-r-sound*:

assumes $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^{1a} \langle l', R' \rangle$
shows $\forall u \in R. \exists u' \in R'. A \vdash_t \langle l, u \rangle \rightarrow_a \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

lemma *step-trans-r'-sound*:

assumes $A, \mathcal{R} \vdash' \langle l, R \rangle \rightsquigarrow^t \langle l', R' \rangle$
shows $\forall u \in R. \exists u' \in R'. A \vdash' \langle l, u \rangle \rightarrow^t \langle l', u' \rangle$
 $\langle \text{proof} \rangle$

end

context *AlphaClosure*

begin

context

fixes $l l' :: 's$ **and** $A :: ('a, 'c, t, 's) ta$
assumes *valid-abstraction*: *valid-abstraction* $A X k$

begin

interpretation *alpha*: *AlphaClosure-global* - $k l \mathcal{R} l \langle \text{proof} \rangle$

lemma [*simp*]: $\text{alpha.cla} = \text{cla } l \langle \text{proof} \rangle$

interpretation alpha' : *AlphaClosure-global - k l' R l' <proof>*

lemma [*simp*]: $\text{alpha}'.\text{cla} = \text{cla } l' \langle \text{proof} \rangle$

lemma *regions-poststable1*:

assumes

$A \vdash \langle l, Z \rangle \rightsquigarrow^a \langle l', Z' \rangle \ Z \subseteq V \ R' \in \mathcal{R} \ l' \ R' \cap Z' \neq \{\}$

shows $\exists R \in \mathcal{R} \ l. \ A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow^a \langle l', R' \rangle \wedge R \cap Z \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *regions-poststable'*:

assumes

$A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \ Z \subseteq V \ R' \in \mathcal{R} \ l' \ R' \cap Z' \neq \{\}$

shows $\exists R \in \mathcal{R} \ l. \ A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle \wedge R \cap Z \neq \{\}$
 $\langle \text{proof} \rangle$

end

lemma *regions-poststable2*:

assumes *valid-abstraction*: *valid-abstraction A X k*

and prems: $A \vdash' \langle l, Z \rangle \rightsquigarrow^a \langle l', Z' \rangle \ Z \subseteq V \ R' \in \mathcal{R} \ l' \ R' \cap Z' \neq \{\}$

shows $\exists R \in \mathcal{R} \ l. \ A, \mathcal{R} \vdash' \langle l, R \rangle \rightsquigarrow^a \langle l', R' \rangle \wedge R \cap Z \neq \{\}$
 $\langle \text{proof} \rangle$

Poststability of Closures: For every transition in the zone graph and each region in the closure of the resulting zone, there exists a similar transition in the region graph.

lemma *regions-poststable*:

assumes *valid-abstraction*: *valid-abstraction A X k*

and A:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\tau} \langle l', Z' \rangle \ A \vdash \langle l', Z' \rangle \rightsquigarrow_{1a} \langle l'', Z'' \rangle$

$Z \subseteq V \ R'' \in \mathcal{R} \ l'' \ R'' \cap Z'' \neq \{\}$

shows $\exists R \in \mathcal{R} \ l. \ A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l'', R'' \rangle \wedge R \cap Z \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-t-r-loc*:

$l' = l$ **if** $A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_{\tau} \langle l', R' \rangle$

$\langle \text{proof} \rangle$

lemma \mathcal{R} -V:

$u \in V$ **if** $R \in \mathcal{R} \ l \ u \in R$

$\langle \text{proof} \rangle$

lemma *step-r'-complete:*

assumes $A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ *valid-abstraction* $A \ X \ (\lambda x. \text{real } o \ k \ x) \ u \in V$

shows $\exists a \ R'. \ u' \in R' \wedge A, \mathcal{R} \vdash \langle l, [u]_l \rangle \rightsquigarrow_a \langle l', R' \rangle$
<proof>

lemma *step-r- \mathcal{R} :*

$R' \in \mathcal{R} \ l' \text{ if } A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$
<proof>

lemma *step-r'- \mathcal{R} :*

$R' \in \mathcal{R} \ l' \text{ if } A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$
<proof>

end

context *Regions*

begin

lemma *closure-parts-mono:*

$\{R \in \mathcal{R} \ l. \ R \cap Z \neq \{\}\} \subseteq \{R \in \mathcal{R} \ l. \ R \cap Z' \neq \{\}\}$ **if** $\text{Closure}_{\alpha, l} \ Z \subseteq \text{Closure}_{\alpha, l} \ Z'$
<proof>

lemma *closure-parts-id:*

$\{R \in \mathcal{R} \ l. \ R \cap Z \neq \{\}\} = \{R \in \mathcal{R} \ l. \ R \cap Z' \neq \{\}\}$ **if** $\text{Closure}_{\alpha, l} \ Z = \text{Closure}_{\alpha, l} \ Z'$
<proof>

More lemmas on regions **context**

fixes $l' :: 's$

begin

interpretation *regions: Regions-global - - - k l'*

<proof>

context

fixes $A :: ('a, 'c, t, 's) \text{ ta}$

assumes *valid-abstraction: valid-abstraction* $A \ X \ k$

begin

lemmas *regions-poststable = regions-poststable[OF valid-abstraction]*

lemma *clkp-set-clkp-set1*:

$\exists l. (c, x) \in \text{clkp-set } A \text{ } l$ **if** $(c, x) \in \text{Timed-Automata.clkp-set } A$
 $\langle \text{proof} \rangle$

lemma *clkp-set-clkp-set2*:

$(c, x) \in \text{Timed-Automata.clkp-set } A$ **if** $(c, x) \in \text{clkp-set } A \text{ } l$ **for** l
 $\langle \text{proof} \rangle$

lemma *clock-numbering-le*: $\forall c \in \text{clk-set } A. v \ c \leq n$

$\langle \text{proof} \rangle$

lemma *beta-alpha-step*:

$A \vdash \langle l, Z \rangle \rightsquigarrow_{\alpha(a)} \langle l', \text{Closure}_{\alpha, l'} Z \rangle$ **if** $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta(a)} \langle l', Z \rangle$ $Z \in V'$
 $\langle \text{proof} \rangle$

lemma *beta-alpha-region-step*:

$\exists a. \exists R \in \mathcal{R} \ l. R \cap Z \neq \{\}$ $\wedge A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R \rangle$ **if**
 $A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z \rangle$ $Z \in V'$ $R' \in \mathcal{R}$ $l' R' \cap Z' \neq \{\}$
 $\langle \text{proof} \rangle$

lemmas *step-z-beta'-V' = step-z-beta'-V'* [OF valid-abstraction clock-numbering-le]

lemma *step-trans-z'-closure-subs*:

assumes

$A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z \rangle$ $Z \subseteq V$ $\forall R \in \mathcal{R} \ l. R \cap Z \neq \{\}$ $\longrightarrow R \cap W \neq \{\}$

shows

$\exists W'. A \vdash' \langle l, W \rangle \rightsquigarrow^t \langle l', W \rangle \wedge (\forall R \in \mathcal{R} \ l'. R \cap Z' \neq \{\}) \longrightarrow R \cap W' \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-trans-z'-closure-eq*:

assumes

$A \vdash' \langle l, Z \rangle \rightsquigarrow^t \langle l', Z \rangle$ $Z \subseteq V$ $W \subseteq V$ $\forall R \in \mathcal{R} \ l. R \cap Z \neq \{\}$ $\longleftrightarrow R \cap W \neq \{\}$

shows

$\exists W'. A \vdash' \langle l, W \rangle \rightsquigarrow^t \langle l', W \rangle \wedge (\forall R \in \mathcal{R} \ l'. R \cap Z' \neq \{\}) \longleftrightarrow R \cap W' \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *step-z'-closure-subs*:

assumes

$A \vdash \langle l, Z \rangle \rightsquigarrow \langle l', Z \rangle$ $Z \subseteq V$ $\forall R \in \mathcal{R} \ l. R \cap Z \neq \{\}$ $\longrightarrow R \cap W \neq \{\}$

shows

$\exists W'. A \vdash \langle l, W \rangle \rightsquigarrow \langle l', W' \rangle \wedge (\forall R \in \mathcal{R} l'. R \cap Z' \neq \{\} \longrightarrow R \cap W' \neq \{\})$
 <proof>

end

lemma *apx-finite*:

finite {*Approx* _{β} $l' Z \mid Z. Z \subseteq V$ } (**is finite** ?*S*)
 <proof>

lemmas *apx-subset = regions.beta-interp.apx-subset*

lemma *step-z-beta'-empty*:

$Z' = \{\}$ **if** $A \vdash \langle l, \{\} \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle$
 <proof>

end

lemma *step-z-beta'-complete*:

assumes $A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle \ u \in Z \ Z \subseteq V$
shows $\exists Z'. A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle \wedge u' \in Z'$
 <proof>

end

7.9.2 Instantiation of Double Simulation

7.9.3 Auxiliary Definitions

definition *state-set* :: (*'a, 'c, 'time, 's*) *ta* \Rightarrow *'s set* **where**

state-set $A \equiv \text{fst } \text{' (fst } A) \cup (\text{snd } o \text{ snd } o \text{ snd } o \text{ snd}) \text{' (fst } A)$

lemma *finite-trans-of-finite-state-set*:

finite (*state-set* A) **if** *finite* (*trans-of* A)
 <proof>

lemma *state-setI1*:

$l \in \text{state-set } A$ **if** $A \vdash l \longrightarrow^{g,a,r} l'$
 <proof>

lemma *state-setI2*:

$l' \in \text{state-set } A$ **if** $A \vdash l \longrightarrow^{g,a,r} l'$
 <proof>

lemma (in *AlphaClosure*) *step-r'-state-set*:
 $l' \in \text{state-set } A \text{ if } A, \mathcal{R} \vdash \langle l, R \rangle \rightsquigarrow_a \langle l', R' \rangle$
 ⟨proof⟩

lemma (in *Regions*) *step-z-beta'-state-set2*:
 $l' \in \text{state-set } A \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle$
 ⟨proof⟩

7.9.4 Instantiation

locale *Regions-TA* = *Regions* X - - k for $X :: 'c \text{ set}$ and $k :: 's \Rightarrow 'c \Rightarrow \text{nat} +$
 fixes $A :: ('a, 'c, t, 's) \text{ ta}$
 assumes *valid-abstraction*: *valid-abstraction* A X k
 and *finite-state-set*: *finite* (*state-set* A)
begin

no-notation *Regions-Beta.part* ($\langle [-] \cdot \rangle$ [*61,61*] *61*)

notation *part''* ($\langle [-] \cdot \rangle$ [*61,61*] *61*)

lemma *step-z-beta'-state-set1*:
 $l \in \text{state-set } A \text{ if } A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle$
 ⟨proof⟩

sublocale *sim*: *Double-Simulation-paired*

$\lambda (l, u) (l', u'). A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ — Concrete step relation
 $\lambda (l, Z) (l', Z'). \exists a. A, \mathcal{R} \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \wedge Z' \neq \{\}$
 — Step relation for the first abstraction layer
 $\lambda (l, R). l \in \text{state-set } A \wedge R \in \mathcal{R} \text{ } l$ — Valid states of the first abstraction layer
 layer
 $\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\}$
 — Step relation for the second abstraction layer
 $\lambda (l, Z). l \in \text{state-set } A \wedge Z \in V' \wedge Z \neq \{\}$ — Valid states of the second abstraction layer
 abstraction layer
 ⟨proof⟩

sublocale *Graph-Defs*

$\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\}$ ⟨proof⟩

lemmas *step-z-beta'-V'* = *step-z-beta'-V'*[*OF valid-abstraction*]

lemma *step-r'-complete-spec*:

assumes $A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle \ u \in V$
shows $\exists a \ R'. \ u' \in R' \wedge A, \mathcal{R} \vdash \langle l, [u]_l \rangle \rightsquigarrow_a \langle l', R' \rangle$
 $\langle proof \rangle$

end

7.9.5 Büchi Runs

locale *Regions-TA-Start-State* = *Regions-TA* - - - - *A* **for** $A :: ('a, 'c, t, 's) \ ta +$

fixes $l_0 :: 's$ **and** $Z_0 :: ('c, t)$ *zone*

assumes *start-state*: $l_0 \in \text{state-set } A \ Z_0 \in V' \ Z_0 \neq \{\}$

begin

definition $a_0 = \text{from-}R \ l_0 \ Z_0$

sublocale *sim-complete'*: *Double-Simulation-Finite-Complete-paired*

$\lambda (l, u) (l', u'). \ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ — Concrete step relation

$\lambda (l, Z) (l', Z'). \ \exists a. \ A, \mathcal{R} \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \wedge Z' \neq \{\}$

— Step relation for the first abstraction layer

$\lambda (l, R). \ l \in \text{state-set } A \wedge R \in \mathcal{R} \ l$ — Valid states of the first abstraction layer

$\lambda (l, Z) (l', Z'). \ A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\}$

— Step relation for the second abstraction layer

$\lambda (l, Z). \ l \in \text{state-set } A \wedge Z \in V' \wedge Z \neq \{\}$ — Valid states of the second abstraction layer

$l_0 \ Z_0$

$\langle proof \rangle$

sublocale *sim-complete-bisim'*: *Double-Simulation-Finite-Complete-Bisim-Cover-paired*

$\lambda (l, u) (l', u'). \ A \vdash' \langle l, u \rangle \rightarrow \langle l', u' \rangle$ — Concrete step relation

$\lambda (l, Z) (l', Z'). \ \exists a. \ A, \mathcal{R} \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle \wedge Z' \neq \{\}$

— Step relation for the first abstraction layer

$\lambda (l, R). \ l \in \text{state-set } A \wedge R \in \mathcal{R} \ l$ — Valid states of the first abstraction layer

$\lambda (l, Z) (l', Z'). \ A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\}$

— Step relation for the second abstraction layer

$\lambda (l, Z). \ l \in \text{state-set } A \wedge Z \in V' \wedge Z \neq \{\}$ — Valid states of the second abstraction layer

$l_0 \ Z_0$

$\langle proof \rangle$

7.9.6 State Formulas

context

fixes $P :: 's \Rightarrow bool$ — The state property we want to check
begin

definition $\varphi = P \circ fst$

State formulas are compatible with closures.

Runs satisfying a formula all the way long interpretation G_φ :
Graph-Start-Defs

$\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\} \wedge P l' (l_0, Z_0) \langle proof \rangle$

theorem *Alw-ev-mc1*:

$(\forall x_0 \in a_0. \text{sim.sim.Alw-ev } (Not \circ \varphi) x_0) \longleftrightarrow \neg (P l_0 \wedge (\exists a. G_\varphi.\text{reachable } a \wedge G_\varphi.\text{reaches1 } a a))$
 $\langle proof \rangle$

end

7.9.7 Leads-To Properties

context

fixes $P Q :: 's \Rightarrow bool$ — The state properties we want to check
begin

definition $\psi = Q \circ fst$

interpretation G_ψ : *Graph-Defs*

$\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_\beta \langle l', Z' \rangle \wedge Z' \neq \{\} \wedge Q l' \langle proof \rangle$

theorem *leadsto-mc1*:

$(\forall x_0 \in a_0. \text{sim.sim.leadsto } (\varphi P) (Not \circ \psi) x_0) \longleftrightarrow$
 $(\nexists x. \text{reaches } (l_0, Z_0) x \wedge P (fst x) \wedge Q (fst x) \wedge (\exists a. G_\psi.\text{reaches } x a \wedge G_\psi.\text{reaches1 } a a))$
if $\forall x_0 \in a_0. \neg \text{sim.sim.deadlock } x_0$
 $\langle proof \rangle$

end

lemma *from-R-reaches*:

assumes $\text{sim.sim.Steps.reaches } (\text{from-R } l_0 Z_0) b$
obtains $l Z$ **where** $b = \text{from-R } l Z$

<proof>

lemma *ta-reaches-ex-iff*:

assumes *compatible*:

$\bigwedge l u u' R.$

$u \in R \implies u' \in R \implies R \in \mathcal{R} \ l \implies l \in \text{state-set } A \implies P(l, u) = P(l, u')$

shows

$(\exists x_0 \in a_0. \exists l u. \text{sim.sim.reaches } x_0(l, u) \wedge P(l, u)) \longleftrightarrow$
 $(\exists l Z. \exists u \in Z. \text{reaches}(l_0, Z_0)(l, Z) \wedge P(l, u))$

<proof>

lemma *ta-reaches-all-iff*:

assumes *compatible*:

$\bigwedge l u u' R.$

$u \in R \implies u' \in R \implies R \in \mathcal{R} \ l \implies l \in \text{state-set } A \implies P(l, u) = P(l, u')$

shows

$(\forall x_0 \in a_0. \forall l u. \text{sim.sim.reaches } x_0(l, u) \longrightarrow P(l, u)) \longleftrightarrow$
 $(\forall l Z. \text{reaches}(l_0, Z_0)(l, Z) \longrightarrow (\forall u \in Z. P(l, u)))$

<proof>

end

end

8 Forward Analysis with DBMs and Widening

theory *Normalized-Zone-Semantics*

imports *DBM-Zone-Semantics Approx-Beta Simulation-Graphs-TA*

begin

hide-const (**open**) *D*

no-notation *infinity* ($\langle \infty \rangle$)

lemma *rtranclp-backwards-invariant-iff*:

assumes *invariant*: $\bigwedge y z. E^{**} x y \implies P z \implies E y z \implies P y$

and *E'*: $E' = (\lambda x y. E x y \wedge P y)$

shows $E'^{**} x y \wedge P x \longleftrightarrow E^{**} x y \wedge P y$

<proof>

context *Bisimulation-Invariant*
begin

context

fixes $\varphi :: 'a \Rightarrow \text{bool}$ and $\psi :: 'b \Rightarrow \text{bool}$

assumes compatible: $a \sim b \Longrightarrow PA\ a \Longrightarrow PB\ b \Longrightarrow \varphi\ a \longleftrightarrow \psi\ b$

begin

lemma *reaches-ex-iff*:

$(\exists b. A.\text{reaches}\ a\ b \wedge \varphi\ b) \longleftrightarrow (\exists b. B.\text{reaches}\ a'\ b \wedge \psi\ b)$ if $a \sim a'$ $PA\ a\ PB\ a'$

<proof>

lemma *reaches-all-iff*:

$(\forall b. A.\text{reaches}\ a\ b \longrightarrow \varphi\ b) \longleftrightarrow (\forall b. B.\text{reaches}\ a'\ b \longrightarrow \psi\ b)$ if $a \sim a'$ $PA\ a\ PB\ a'$

<proof>

end

end

lemma *step-z-dbm-delay-loc*:

$l' = l$ if $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,\tau} \langle l', D \rangle$

<proof>

lemma *step-z-dbm-action-state-set1*:

$l \in \text{state-set}\ A$ if $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,1a} \langle l', D \rangle$

<proof>

lemma *step-z-dbm-action-state-set2*:

$l' \in \text{state-set}\ A$ if $A \vdash \langle l, D \rangle \rightsquigarrow_{v,n,1a} \langle l', D \rangle$

<proof>

lemma *step-delay-loc*:

$l' = l$ if $A \vdash \langle l, u \rangle \rightarrow^d \langle l', u \rangle$

<proof>

lemma *step-a-state-set1*:

$l \in \text{state-set}\ A$ if $A \vdash \langle l, u \rangle \rightarrow_a \langle l', u \rangle$

<proof>

lemma *step'-state-set1*:

$l \in \text{state-set } A \text{ if } A \vdash' \langle l, u \rangle \rightarrow \langle l', u \wedge \langle \text{proof} \rangle$

8.1 DBM-based Semantics with Normalization

8.1.1 Single Step

inductive *step-z-norm* ::

$(\text{'a}, \text{'c}, t, \text{'s}) \text{ ta}$
 $\Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow (\text{'s} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow (\text{'c} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{'a} \text{ action}$
 $\Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow \text{bool}$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{-, -, -} \langle -, - \rangle \rangle [61, 61, 61, 61, 61, 61] \ 61)$

where *step-z-norm*:

$A \vdash \langle l, D \rangle \rightsquigarrow_{v, n, a} \langle l', D \wedge \rangle \Longrightarrow A \vdash \langle l, D \rangle \rightsquigarrow_{k, v, n, a} \langle l', \text{norm } (FW \ D' \ n) \ (k \ l') \ n \rangle$

inductive *step-z-norm'* ::

$(\text{'a}, \text{'c}, t, \text{'s}) \text{ ta} \Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow (\text{'s} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow (\text{'c} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow \text{bool}$
 $(\langle - \vdash' \langle -, - \rangle \rightsquigarrow_{-, -, -} \langle -, - \rangle \rangle [61, 61, 61, 61, 61] \ 61)$

where

step: $A \vdash \langle l', Z \wedge \rangle \rightsquigarrow_{v, n, \tau} \langle l'', Z'' \wedge \rangle$
 $\Longrightarrow A \vdash \langle l'', Z'' \wedge \rangle \rightsquigarrow_{k, v, n, \uparrow(a)} \langle l''', Z''' \wedge \rangle$
 $\Longrightarrow A \vdash' \langle l', Z \wedge \rangle \rightsquigarrow_{k, v, n} \langle l''', Z''' \wedge \rangle$

abbreviation *steps-z-norm* ::

$(\text{'a}, \text{'c}, t, \text{'s}) \text{ ta} \Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow (\text{'s} \Rightarrow \text{nat} \Rightarrow \text{nat}) \Rightarrow (\text{'c} \Rightarrow \text{nat}) \Rightarrow \text{nat} \Rightarrow \text{'s} \Rightarrow t \text{ DBM} \Rightarrow \text{bool}$
 $(\langle - \vdash \langle -, - \rangle \rightsquigarrow_{-, -, *} \langle -, - \rangle \rangle [61, 61, 61, 61, 61] \ 61) \text{ where}$

$A \vdash \langle l, D \rangle \rightsquigarrow_{k, v, n} \langle l', D \wedge \rangle \equiv (\lambda (l, Z) (l', Z'). A \vdash' \langle l, Z \rangle \rightsquigarrow_{k, v, n} \langle l', Z' \rangle)^{**}$
 $(l, D) (l', D')$

lemma *norm-empty-diag-preservation-real*:

fixes $k :: \text{nat} \Rightarrow \text{nat}$

assumes $i \leq n$

assumes $M \ i \ i < Le \ 0$

shows $\text{norm } M \ (\text{real } o \ k) \ n \ i \ i < Le \ 0$

$\langle \text{proof} \rangle$

context *Regions-defs*

begin

inductive *valid-dbm* **where**

$[M]_{v, n} \subseteq V \Longrightarrow \text{dbm-int } M \ n \Longrightarrow \text{valid-dbm } M$

```

inductive-cases valid-dbm-cases[elim]: valid-dbm M

declare valid-dbm.intros[intro]

end

locale Regions-common =
  Regions-defs X v n for  $X :: 'c \text{ set}$  and  $v \ n \ +$ 
  fixes not-in-X
  assumes finite: finite X
  assumes clock-numbering: clock-numbering' v n  $\forall k \leq n. k > 0 \longrightarrow (\exists c \in$ 
 $X. v \ c = k)$ 
   $\forall c \in X. v \ c \leq n$ 
  assumes not-in-X: not-in-X  $\notin X$ 
  assumes non-empty:  $X \neq \{\}$ 
begin

lemma FW-zone-equiv-spec:
  shows  $[M]_{v,n} = [FW \ M \ n]_{v,n}$ 
   $\langle proof \rangle$ 

lemma dbm-non-empty-diag:
  assumes  $[M]_{v,n} \neq \{\}$ 
  shows  $\forall k \leq n. M \ k \ k \geq 0$ 
   $\langle proof \rangle$ 

lemma cn-weak:  $\forall k \leq n. 0 < k \longrightarrow (\exists c. v \ c = k)$   $\langle proof \rangle$ 

lemma negative-diag-empty:
  assumes  $\exists k \leq n. M \ k \ k < 0$ 
  shows  $[M]_{v,n} = \{\}$ 
   $\langle proof \rangle$ 

lemma non-empty-cyc-free:
  assumes  $[M]_{v,n} \neq \{\}$ 
  shows cyc-free M n
   $\langle proof \rangle$ 

lemma FW-valid-preservation:
  assumes valid-dbm M
  shows valid-dbm (FW M n)
   $\langle proof \rangle$ 

```

end

context *Regions-global*

begin

sublocale *Regions-common* \langle *proof* \rangle

abbreviation $v' \equiv \text{beta-interp.v}'$

lemma *apx-empty-iff''*:

assumes *canonical* $M1\ n\ [M1]_{v,n} \subseteq V\ \text{dbm-int}\ M1\ n$

shows $[M1]_{v,n} = \{\} \longleftrightarrow [\text{norm}\ M1\ (k\ o\ v')\ n]_{v,n} = \{\}$

\langle *proof* \rangle

lemma *norm-FW-empty*:

assumes *valid-dbm* M

assumes $[M]_{v,n} = \{\}$

shows $[\text{norm}\ (FW\ M\ n)\ (k\ o\ v')\ n]_{v,n} = \{\}$ (**is** $[?M]_{v,n} = \{\}$)

\langle *proof* \rangle

lemma *apx-norm-eq-spec*:

assumes *valid-dbm* M

and $[M]_{v,n} \neq \{\}$

shows $\text{beta-interp.Approx}_\beta ([M]_{v,n}) = [\text{norm}\ (FW\ M\ n)\ (k\ o\ v')\ n]_{v,n}$

\langle *proof* \rangle

lemma *norm-FW-valid-preservation-non-empty*:

assumes *valid-dbm* $M\ [M]_{v,n} \neq \{\}$

shows *valid-dbm* $(\text{norm}\ (FW\ M\ n)\ (k\ o\ v')\ n)$ (**is** *valid-dbm* $?M$)

\langle *proof* \rangle

lemma *norm-int-all-preservation*:

fixes $M :: \text{real DBM}$

assumes *dbm-int-all* M

shows *dbm-int-all* $(\text{norm}\ M\ (k\ o\ v')\ n)$

\langle *proof* \rangle

lemma *norm-FW-valid-preservation-empty*:

assumes *valid-dbm* $M\ [M]_{v,n} = \{\}$

shows *valid-dbm* $(\text{norm}\ (FW\ M\ n)\ (k\ o\ v')\ n)$ (**is** *valid-dbm* $?M$)

\langle *proof* \rangle

lemma *norm-FW-valid-preservation*:

assumes *valid-dbm* M

shows *valid-dbm* (*norm* (*FW M n*) (*k o v*[^]) *n*)
 ⟨*proof*⟩

lemma *norm-FW-equiv*:

assumes *valid*: *dbm-int D n dbm-int M n [D]_{v,n} ⊆ V*

and *equiv*: $[D]_{v,n} = [M]_{v,n}$

shows $[norm (FW D n) (k o v') n]_{v,n} = [norm (FW M n) (k o v') n]_{v,n}$
 ⟨*proof*⟩

end

context *Regions*

begin

sublocale *Regions-common* ⟨*proof*⟩

definition $v' \equiv \lambda i. \text{if } 0 < i \wedge i \leq n \text{ then } (THE c. c \in X \wedge v c = i) \text{ else not-in-}X$

abbreviation *step-z-norm'* ($\langle \cdot \vdash \langle -, - \rangle \rightsquigarrow_{\mathcal{N}(\cdot)} \langle -, - \rangle \rangle$ [*61,61,61,61*] *61*)

where

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle \equiv A \vdash \langle l, D \rangle \rightsquigarrow_{(\lambda l. k l o v'), v, n, a} \langle l', D' \rangle$

definition *step-z-norm''* ($\langle \cdot \vdash'' \langle -, - \rangle \rightsquigarrow_{\mathcal{N}(\cdot)} \langle -, - \rangle \rangle$ [*61,61,61,61*] *61*)

where

$A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l'', D'' \rangle \equiv$

$\exists l' D'. A \vdash \langle l, D \rangle \rightsquigarrow_{v, n, \tau} \langle l', D' \rangle \wedge A \vdash \langle l', D' \rangle \rightsquigarrow_{\mathcal{N}(1a)} \langle l'', D'' \rangle$

abbreviation *steps-z-norm'* ($\langle \cdot \vdash \langle -, - \rangle \rightsquigarrow_{\mathcal{N}^*} \langle -, - \rangle \rangle$ [*61,61,61*] *61*)

where

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}^*} \langle l', D' \rangle \equiv (\lambda (l, D) (l', D'). \exists a. A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle)^{**} (l, D) (l', D')$

inductive-cases *step-z-norm'-elims*[*elim!*]: $A \vdash \langle l, u \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', u' \rangle$

declare *step-z-norm.intros*[*intro*]

lemma *step-z-valid-dbm*:

assumes $A \vdash \langle l, D \rangle \rightsquigarrow_{v, n, a} \langle l', D' \rangle$

and *global-clock-numbering A v n valid-abstraction A X k valid-dbm D*

shows *valid-dbm D'*

⟨*proof*⟩

lemma *step-z-norm-induct*[*case-names - step-z-norm step-z-refl*]:

assumes $x1 \vdash \langle x2, x3 \rangle \rightsquigarrow_{(\lambda l. k l o v'), v, n, a} \langle x7, x8 \rangle$

and *step-z-norm*:

$\bigwedge A l D l' D'$

$A \vdash \langle l, D \rangle \rightsquigarrow_{v, n, a} \langle l', D' \rangle \implies$

$P A l D l' (norm (FW D' n) (k l' o v') n)$

shows $P x1 x2 x3 x7 x8$

<proof>

context

fixes $l' :: 's$

begin

interpretation *regions*: *Regions-global - - - k l'*

<proof>

lemma *regions-v'-eq*[*simp*]:

regions.v' = v'

<proof>

lemma *step-z-norm-int-all-preservation*:

assumes

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ *global-clock-numbering A v n*

$\forall (x, m) \in \text{Timed-Automata.clkp-set } A. m \in \mathbb{N}$ *dbm-int-all D*

shows *dbm-int-all D'*

<proof>

lemma *step-z-norm-valid-dbm-preservation*:

assumes

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ *global-clock-numbering A v n valid-abstraction*

A X k valid-dbm D

shows *valid-dbm D'*

<proof>

lemma *norm-beta-sound*:

assumes $A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ *global-clock-numbering A v n valid-abstraction*

A X k

and *valid-dbm D*

shows $A \vdash \langle l, [D]_{v, n} \rangle \rightsquigarrow_{\beta(a)} \langle l', [D']_{v, n} \rangle$ *<proof>*

lemma *step-z-norm-valid-dbm*:

assumes

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ *global-clock-numbering* A v n
valid-abstraction A X k *valid-dbm* D
shows *valid-dbm* D' \langle *proof* \rangle

lemma *norm-beta-complete:*

assumes $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_{\beta(a)} \langle l', Z \rangle$ *global-clock-numbering* A v n *valid-abstraction*
 A X k
and *valid-dbm* D
obtains D' **where** $A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ $[D']_{v,n} = Z$ *valid-dbm* D'
 \langle *proof* \rangle

lemma *step-z-norm-mono:*

assumes $A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$ *global-clock-numbering* A v n *valid-abstraction*
 A X k
and *valid-dbm* D *valid-dbm* M
and $[D]_{v,n} \subseteq [M]_{v,n}$
shows $\exists M'. A \vdash \langle l, M \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', M' \rangle \wedge [D']_{v,n} \subseteq [M']_{v,n}$
 \langle *proof* \rangle

lemma *step-z-norm-equiv:*

assumes *step:* $A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle$
and *prems:* *global-clock-numbering* A v n *valid-abstraction* A X k
and *valid:* *valid-dbm* D *valid-dbm* M
and *equiv:* $[D]_{v,n} = [M]_{v,n}$
shows $\exists M'. A \vdash \langle l, M \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', M' \rangle \wedge [D']_{v,n} = [M']_{v,n}$
 \langle *proof* \rangle

end

8.1.2 Multi Step

lemma *valid-dbm-V':*

assumes *valid-dbm* M
shows $[M]_{v,n} \in V'$
 \langle *proof* \rangle

lemma *step-z-empty:*

assumes $A \vdash \langle l, Z \rangle \rightsquigarrow_a \langle l', Z' \rangle$ $Z = \{\}$
shows $Z' = \{\}$
 \langle *proof* \rangle

8.1.3 Connecting with Correctness Results for Approximating Semantics

context

fixes $A :: ('a, 'c, \text{real}, 's) \text{ta}$
assumes $\text{gcn}: \text{global-clock-numbering } A \ v \ n$
and $\text{va}: \text{valid-abstraction } A \ X \ k$

begin

context

notes $[\text{intro}] = \text{step-z-valid-dbm}[OF - \text{gcn } \text{va}]$

begin

lemma *valid-dbm-step-z-norm''*:

valid-dbm $D' \text{ if } A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle \text{ valid-dbm } D$
 $\langle \text{proof} \rangle$

lemma *steps-z-norm'-valid-dbm-invariant*:

valid-dbm $D' \text{ if } A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}^*} \langle l', D' \rangle \text{ valid-dbm } D$
 $\langle \text{proof} \rangle$

lemma *norm-beta-sound''*:

assumes $A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l'', D'' \rangle$
and *valid-dbm* D
shows $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_{\beta} \langle l'', [D'']_{v,n} \rangle$
 $\langle \text{proof} \rangle$

lemma *norm-beta-complete1*:

assumes $A \vdash \langle l, [D]_{v,n} \rangle \rightsquigarrow_{\beta} \langle l'', Z'' \rangle$
and *valid-dbm* D
obtains $a \ D'' \text{ where } A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l'', D'' \rangle \ [D'']_{v,n} = Z'' \text{ valid-dbm } D''$
 $\langle \text{proof} \rangle$

lemma *bisim*:

Bisimulation-Invariant
 $(\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle \wedge Z' \neq \{\})$
 $(\lambda (l, D) (l', D'). \exists a. A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\})$
 $(\lambda (l, Z) (l', D). l = l' \wedge Z = [D]_{v,n})$
 $(\lambda -. \text{True}) (\lambda (l, D). \text{valid-dbm } D)$
 $\langle \text{proof} \rangle$

end

interpretation *Bisimulation-Invariant*

$\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle \wedge Z' \neq \{\}$
 $\lambda (l, D) (l', D'). \exists a. A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\}$
 $\lambda (l, Z) (l', D). l = l' \wedge Z = [D]_{v,n}$
 $\lambda -. \text{True } \lambda (l, D). \text{valid-dbm } D$
 $\langle \text{proof} \rangle$

lemma *step-z-norm''-non-empty:*

$[D]_{v,n} \neq \{\}$ **if** $A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle [D']_{v,n} \neq \{\}$ *valid-dbm* D
 $\langle \text{proof} \rangle$

lemma *norm-steps-empty:*

$A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}^*} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\} \iff B.\text{reaches } (l, D) (l', D') \wedge [D]_{v,n} \neq \{\}$
if *valid-dbm* D
 $\langle \text{proof} \rangle$

context

fixes $P Q :: 's \Rightarrow \text{bool}$ — The state property we want to check

begin

interpretation *bisim-ψ: Bisimulation-Invariant*

$\lambda (l, Z) (l', Z'). A \vdash \langle l, Z \rangle \rightsquigarrow_{\beta} \langle l', Z' \rangle \wedge Z' \neq \{\} \wedge Q l'$
 $\lambda (l, D) (l', D'). \exists a. A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}(a)} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\} \wedge Q l'$
 $\lambda (l, Z) (l', D). l = l' \wedge Z = [D]_{v,n}$
 $\lambda -. \text{True } \lambda (l, D). \text{valid-dbm } D$
 $\langle \text{proof} \rangle$

end

context

assumes *finite-state-set: finite* (*state-set* A)

begin

interpretation *R: Regions-TA*

$\langle \text{proof} \rangle$

lemma *A-reaches-non-empty:*

$Z' \neq \{\}$ **if** $A.\text{reaches } (l, Z) (l', Z') Z \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *A-reaches-start-non-empty-iff:*

$(\exists Z'. (\exists u. u \in Z') \wedge A.\text{reaches } (l, Z) (l', Z')) \iff (\exists Z'. A.\text{reaches } (l,$

$Z) (l', Z') \wedge Z \neq \{\}$
 $\langle proof \rangle$

lemma *step-z-norm''-state-set1*:

$l \in \text{state-set } A$ **if** $A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}_a} \langle l', D' \rangle$
 $\langle proof \rangle$

lemma *step-z-norm''-state-set2*:

$l' \in \text{state-set } A$ **if** $A \vdash' \langle l, D \rangle \rightsquigarrow_{\mathcal{N}_a} \langle l', D' \rangle$
 $\langle proof \rangle$

theorem *steps-z-norm-decides-emptiness*:

assumes *valid-dbm* D

shows $(\exists D'. A \vdash \langle l, D \rangle \rightsquigarrow_{\mathcal{N}^*} \langle l', D' \rangle \wedge [D']_{v,n} \neq \{\})$
 $\longleftrightarrow (\exists u \in [D]_{v,n}. (\exists u'. A \vdash' \langle l, u \rangle \rightarrow^* \langle l', u' \rangle))$

$\langle proof \rangle$

end

end

context

fixes $A :: ('a, 'c, \text{real}, 's) \text{ ta}$

assumes *gcn*: *global-clock-numbering* $A \ v \ n$

and *va*: *valid-abstraction* $A \ X \ k$

begin

lemmas

$\text{step-z-norm-valid-dbm}' = \text{step-z-norm-valid-dbm}[OF - \text{gcn } va]$

lemmas

$\text{step-z-valid-dbm}' = \text{step-z-valid-dbm}[OF - \text{gcn } va]$

lemmas $\text{norm-beta-sound}' = \text{norm-beta-sound}[OF - \text{gcn } va]$

lemma *v-bound*:

$\forall c \in \text{clk-set } A. v \ c \leq n$

$\langle proof \rangle$

lemmas $\text{alpha-beta-step}'' = \text{alpha-beta-step}'[OF - va \ v\text{-bound}]$

lemmas $\text{step-z-dbm-sound}' = \text{step-z-dbm-sound}[OF - \text{gcn}]$

lemmas $step\text{-}z\text{-}V'' = step\text{-}z\text{-}V'[OF - va\ v\text{-}bound]$

end

end

8.2 Additional Useful Properties of the Normalized Semantics

Obsolete

lemma *norm-diag-alt-def*:

norm-diag $e = (if\ e < 0\ then\ Lt\ 0\ else\ if\ e = 0\ then\ e\ else\ \infty)$
<proof>

lemma *norm-diag-preservation*:

assumes $\forall l \leq n. M1\ ll \leq 0$
shows $\forall l \leq n. (norm\ M1\ (k :: nat \Rightarrow nat)\ n)\ ll \leq 0$
<proof>

8.3 Appendix: Standard Clock Numberings for Concrete Models

locale *Regions'* =

fixes X **and** $k :: 'c \Rightarrow nat$ **and** $v :: 'c \Rightarrow nat$ **and** $n :: nat$ **and** *not-in-X*
assumes *finite*: $finite\ X$
assumes *clock-numbering'*: $\forall c \in X. v\ c > 0\ \forall c. c \notin X \longrightarrow v\ c > n$
assumes *bij*: $bij\ betw\ v\ X\ \{1..n\}$
assumes *non-empty*: $X \neq \{\}$
assumes *not-in-X*: $not\text{-}in\text{-}X \notin X$

begin

lemma *inj*: $inj\text{-}on\ v\ X$ *<proof>*

lemma *cn-weak*: $\forall c. v\ c > 0$ *<proof>*

lemma *in-X*: **assumes** $v\ x \leq n$ **shows** $x \in X$ *<proof>*

end

sublocale *Regions'* \subseteq *Regions-global*

<proof>

lemma *standard-abstraction*:

assumes
 $finite (Timed-Automata.clkp-set A) \text{ finite } (Timed-Automata.collect-clkvt$
 $(trans-of A))$
 $\forall (-, m :: real) \in Timed-Automata.clkp-set A. m \in \mathbf{N}$
obtains $k :: 'c \Rightarrow nat$ **where** $Timed-Automata.valid-abstraction A (clk-set$
 $A) k$
 $\langle proof \rangle$

definition

$finite-ta A \equiv$
 $finite (Timed-Automata.clkp-set A) \wedge finite (Timed-Automata.collect-clkvt$
 $(trans-of A))$
 $\wedge (\forall (-, m) \in Timed-Automata.clkp-set A. m \in \mathbf{N}) \wedge clk-set A \neq \{\}$
 $- clk-set A \neq \{\}$

lemma *finite-ta-Regions'*:

fixes $A :: ('a, 'c, real, 's) ta$
assumes $finite-ta A$
obtains $v n x$ **where** $Regions' (clk-set A) v n x$
 $\langle proof \rangle$

lemma *finite-ta-RegionsD*:

fixes $A :: ('a, 'c, t, 's) ta$
assumes $finite-ta A$
obtains $k :: 'c \Rightarrow nat$ **and** $v n x$ **where**
 $Regions' (clk-set A) v n x \text{ Timed-Automata.valid-abstraction } A (clk-set$
 $A) k$
 $global-clock-numbering A v n$
 $\langle proof \rangle$

definition *valid-dbm* **where** $valid-dbm M n \equiv dbm-int M n \wedge (\forall i \leq n. M$
 $0 i \leq 0)$

lemma *dbm-positive*:

assumes $M 0 (v c) \leq 0 \text{ } v c \leq n \text{ DBM-val-bounded } v u M n$
shows $u c \geq 0$
 $\langle proof \rangle$

lemma *valid-dbm-pos*:

assumes $valid-dbm M n$
shows $[M]_{v,n} \subseteq \{u. \forall c. v c \leq n \longrightarrow u c \geq 0\}$
 $\langle proof \rangle$

lemma (in *Regions'*) *V-alt-def*:
shows $\{u. \forall c. v\ c > 0 \wedge v\ c \leq n \longrightarrow u\ c \geq 0\} = V$
<proof>
end

References

- [AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
- [AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
- [Bou04] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.
- [BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*, pages 87–124, 2003.
- [HHWt97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.
- [LPY97] G. Kim Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.
- [Yov97] Sergio Yovine. KRONOS: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.