MUNTAC: A Verified Certificate Checker for Timed Automata

Simon Wimmer

Abstract

MUNTA is a verified model checker for timed automata. This entry presents MUNTAC, an extension of MUNTA that can check certificates for Büechi and reachability properties generated by other model checkers. This work has been described in detail in a PhD thesis [2] and conference publications [4, 3]. This entry supersedes earlier versions of MUNTAC hosted on GitHub.

Contents

Introduction

Munta is a verified model checker for timed automata. This entry presents Muntac, an extension of Munta that can check certificates for Büechi and reachability properties generated by other model checkers. This work has been has been described in detail in a PhD thesis [2] and presented in peer-reviewed conference publications [4, 3]. This entry supersedes earlier versions of Muntac hosted on GitHub: https://github.com/wimmers/munta.

Usage Theory `Simple_Network_Language_Certificate_Code` (page 238) describes how to obtain executable code for and run Muntac from within Isabelle.

This entry also provides a build setup for the muntac executable using the MLton and the Poly/ML compilers for Standard ML. Instructions to build and run executables are given in theory `Munta_Certificate_Compile_MLton` (on page 259) and `Munta_Certificate_Compile_Poly` (on page 261), respectively. The MLton executable has much better performance on a single core, while the Poly/ML executable offers the ability for parallel certificate checking. A number of example model files can be found in the `benchmarks` directory.

This entry also bundles up the unverified model checker MLunta[1], which has previously been used to generate test certificates for Muntac. The aforementioned build setups also exercise this tool chain by first compiling MLunta using MLton and then generating certificates for some of the benchmarks, which are in turn checked by Muntac.

Munta itself can also produce reachability certificates for Muntac, as demonstrated on page 242. Büchi certificates from the artifact[2] associated with the FORMATS paper [3] remain compatible with Muntac. Note that the `-i 4` option must be specified explicitly, for example:

```
muntac -i 4 -m cc4/cc4.muntax -r cc4/cc4.renaming -c cc4/cc4-tchecker-ndfs13.cert
```

[1] Git commit `99a3b55ca7f56ebe0bf0e5f8384adf73fa074bad`
[2] https://doi.org/10.6084/m9.figshare.12620582.v1

1 Additions to Lars Noschinski's Graph Library

theory *More_Graph_Theory*
imports *Graph_Theory.Graph_Theory*
begin

lemma *vwalkE2*:
assumes *vwalk p G*
obtains *u* **where** $p = [u]$ $u \in verts\ G$
| *u v es* **where** $p = u \# v \# es$ $(u,v) \in arcs\_ends\ G$ *vwalk* $(v \# es)$ *G*
by (*metis assms list.sel*(*1*) *vwalkE vwalk_consE vwalk_singleton vwalk_to_vpath.cases*)

lemma (**in** *wf_digraph*) *reachable_vwalk_conv2*:
$u \rightarrow^*_G v \longleftrightarrow (u = v \wedge u \in verts\ G \vee (\exists vs.\ vwalk\ (u \# vs\ @\ [v])\ G))$
unfolding *reachable_vwalk_conv*
apply (*rule iffI*)
subgoal
apply (*elim exE conjE*)
apply (*erule vwalkE2*)
apply (*simp*; *fail*)
by (*metis append_butlast_last_id last_ConsR list.distinct*(*1*) *list.sel*(*1*) *vwalk_Cons_Cons*)
apply *force*
done

context *pre_digraph*
begin

definition
$scc\_graph = (\!|$
$verts = sccs\_verts,$
$arcs = \{(x, y).\ \exists a \in x.\ \exists b \in y.\ x \neq y \wedge x \in sccs\_verts \wedge y \in sccs\_verts \wedge a \rightarrow b\},$
$tail = fst,$
$head = snd$
$|\!)$

interpretation *scc_digraph*: *loopfree_digraph scc_graph*
by *standard* (*auto simp*: *scc_graph_def*)

lemmas *scc_digraphI* = *scc_digraph.loopfree_digraph_axioms*

end

context *wf_digraph*
begin

interpretation *scc_digraph*: *loopfree_digraph scc_graph*
by (*rule scc_digraphI*)

lemma *scc_graph_edgeE*:
assumes $\langle x \rightarrow_{scc\_graph} y \rangle$ **obtains** *a b* **where**
$a \in x\ b \in y\ a \rightarrow b\ x \in sccs\_verts\ y \in sccs\_verts\ x \neq y$
using *assms* **by** (*force simp*: *scc_graph_def dest*!: *in_arcs_imp_in_arcs_ends*)

lemma *same_sccI*:
assumes $S \in sccs\_verts\ x \in S\ x \to^* y\ y \to^* x$
shows $y \in S$
using *assms* **by** (*auto simp*: *in_sccs_verts_conv_reachable*)

lemma *scc_graph_reachable1E*:
assumes $\langle x \to^+_{scc\_graph} y \rangle$ **obtains** $a\ b$ **where**
$x \in sccs\_verts\ \ y \in sccs\_verts\ \ x \neq y\ a \in x\ b \in y\ a \to^+ b$
using *assms*
proof (*atomize_elim*, *induction*)
case (*base y*)
then show *?case*
by (*auto elim*!: *scc_graph_edgeE*)
next
case (*step y z*)
then obtain $a\ b$ **where** *IH*: $x \in sccs\_verts\ y \in sccs\_verts\ a \in x\ b \in y\ a \to^+ b\ x \neq y$
by *auto*
from $\langle y \to_{scc\_graph} z \rangle$ **obtain** $b'\ c$ **where** $*$:
$b' \in y\ c \in z\ b' \to c\ y \in sccs\_verts\ z \in sccs\_verts$
by (*elim scc_graph_edgeE*)
with $\langle b \in y \rangle$ **have** $b \to^* b'$
by (*simp add*: *in_sccs_verts_conv_reachable*)
with $\langle b' \to c \rangle\ \langle a \to^+ b \rangle$ **have** $a \to^+ c$
using *reachable1_reachable_trans* **by** *blast*
moreover have $x \neq z$
proof (*rule ccontr*, *unfold not_not*)
assume $x = z$
with $\langle a \in x \rangle\ \langle c \in z \rangle\ \langle x \in \_ \rangle$ **have** $c \to^* a$
by (*simp add*: *in_sccs_verts_conv_reachable*)
with $\langle b \to^* b' \rangle\ \langle b' \to c \rangle$ **have** $b \to^* a$ — XXX: This should really be *by simp*
by (*meson reachable_adj_trans reachable_trans*)
with *IH* **have** $b \in x$
by $-$ (*rule same_sccI*, *auto*)
with *sccs_verts_disjoint IH* **show** *False*
by *auto*
qed
ultimately show *?case*
using *IH* $*$ **by** *auto*
qed

end

locale *dag* = *wf_digraph* +
assumes *acyclic*: $x \to^+ x \Longrightarrow False$

locale *fin_dag* = *fin_digraph* + *dag*

context *wf_digraph*
begin

interpretation *scc_digraph*: *loopfree_digraph scc_graph*

by (*rule scc_digraphI*)

interpretation *scc_digraph*: *dag scc_graph*
by *standard* (*auto elim*: *scc_graph_reachable1E*)

lemmas *scc_digraphI* = *scc_digraph.dag_axioms*

end

context *fin_digraph*
begin

interpretation *scc_digraph*: *dag scc_graph*
by (*rule scc_digraphI*)

interpretation *scc_digraph*: *fin_dag scc_graph*
apply *standard*
unfolding *scc_graph_def*
subgoal *finite_verts*
by (*simp add*: *finite_sccs_verts*)
subgoal *finite_arcs*
using *finite_sccs_verts*
by − (*rule finite_subset*[**where** $B = \{(x, y).\ x \in sccs\_verts \wedge y \in sccs\_verts\}$], *auto*)
done

lemmas *scc_digraphI* = *scc_digraph.fin_dag_axioms*

end

end

2 Transferring Theorems Between Graph Libraries

theory *Graph_Library_Adaptor*
imports *Timed_Automata.Graphs More_Graph_Theory*
begin

no_notation *funcset* (**infixr** $\rightarrow$ *60*)

2.1 Miscellaneous

lemma (**in** −) *pair_in_pair_set_iff*:
$(a, b) \in \{(x, y).\ P\ x\ y\} \longleftrightarrow P\ a\ b$
by *auto*

2.2 From *Graph_Theory* to *Graphs*

context *pre_digraph*
begin

definition $E \equiv \lambda x\ y.\ (x, y) \in arcs\_ends\ G$

interpretation *Graph_Defs E* **.**

lemma *dominates_iff*:
u $\rightarrow_G$ *v* $\longleftrightarrow$ *E u v*
unfolding *E_def* **by** *simp*

lemma *reachable1_iff*:
u $\rightarrow^+_G$ *v* $\longleftrightarrow$ *reaches1 u v*
unfolding *tranclp_unfold E_def* **by** *simp*

end

context *wf_digraph*
begin

interpretation *Graph_Defs E* **.**

lemma *reachable_iff*:
u $\rightarrow^*_G$ *v* $\longleftrightarrow$ *reaches u v* **if** *u* $\in$ *verts G*
apply (*rule iffI*)
subgoal premises *prems*
using *prems* **unfolding** *reachable_def* **by** *induction* (*auto simp*: *dominates_iff*)
subgoal premises *prems*
using *prems* **by** *induction*
(*auto 4 3 simp*: *that dominates_iff*[*symmetric*] *reachable_def intro*: *rtrancl_on_into_rtrancl_on*)
done

lemma *vwalk_iff*:
vwalk (*u* # *xs*) *G* $\longleftrightarrow$ *steps* (*u* # *xs*) **if** *u* $\in$ *verts G*
apply (*rule iffI*)
apply (*induction rule*: *vwalk_induct*; *auto simp*: *dominates_iff*)
subgoal premises *prems*
using *prems that* **by** (*induction u # xs arbitrary*: *u xs*; *simp add*: *adj_in_verts*(*2*) *dominates_iff*)
done

lemmas *graph_convs1* = *reachable_iff reachable1_iff vwalk_iff*

end

context *dag*
begin

sublocale *graph*: *DAG E*
by *standard* (*auto simp*: *graph_convs1 intro*: *acyclic*)

— Transferring a theorem: every DAG has a topological numbering
lemma *topological_numbering*:
fixes *S* **assumes** *finite S*
shows $\exists f :: \_ \Rightarrow nat.\ inj\_on\ f\ S \wedge (\forall x \in S.\ \forall y \in S.\ x \rightarrow y \longrightarrow f\ x < f\ y)$
using *graph.topological_numbering*[*OF assms*] **unfolding** *dominates_iff* **.**

end

sublocale *fin_digraph* ⊆ *graph*: *Finite_Graph E*
by *standard* (*auto intro*!: *finite_subset*[*OF _ finite_verts*] *simp*: *E_def Graph_Defs.vertices_def*)

sublocale *fin_dag* ⊆ *graph*: *Finite_DAG E* **..**

2.3 From *Graphs* to *Graph_Theory*

context *Graph_Defs*
begin

definition $G = (\!|verts = UNIV,\ arcs = \{(a,\ b).\ E\ a\ b\},\ tail = fst,\ head = snd|\!)$

definition $G_p = (\!|pverts = UNIV,\ parcs = \{(a,\ b).\ E\ a\ b\}|\!)$

lemma *G_pair_conv*:
with_proj $G_p = G$
unfolding *G_p_def G_def with_proj_def* **by** *simp*

sublocale *digraph*: *nomulti_digraph G*
by *standard* (*auto simp*: *G_def arc_to_ends_def*)

sublocale *pdigraph*: *pair_wf_digraph* G_p
using *G_pair_conv digraph.wf_digraph_axioms wf_digraph_wp_iff* **by** *metis*

lemma *arc_to_ends_eq*[*simp*]:
arc_to_ends G = *id*
by (*auto simp add*: *G_def arc_to_ends_def*)

lemma *arcs_ends_eq*[*simp*]:
arcs_ends G = $\{(a,\ b).\ E\ a\ b\}$
unfolding *arcs_ends_def arc_to_ends_eq* **by** (*simp add*: *G_def*)

lemma *dominates_iff*[*simp*]:
$u \rightarrow_G v \longleftrightarrow E\ u\ v$
by *simp*

lemma *verts_eq*[*simp*]:
verts G = *UNIV*
unfolding *G_def* **by** *simp*

lemma *reachable_iff*:
$u \rightarrow^*_G v \longleftrightarrow u \rightarrow* v$
apply (*simp only*: *reachable_def arcs_ends_eq verts_eq*)
apply (*rule iffI*)
subgoal premises *prems*
using *prems* **by** *induction auto*
subgoal premises *prems*
using *prems* **by** *induction* (*auto intro*: *rtrancl_on_into_rtrancl_on*)
done

lemma *reachable1_iff*:

$u \rightarrow^+_G v \longleftrightarrow u \rightarrow^+ v$
by (*simp only*: *arcs_ends_eq pair_in_pair_set_iff tranclp_unfold*)

lemma *vwalk_iff*:
$vwalk\ xs\ G \longleftrightarrow steps\ xs$
apply (*rule iffI*)
apply (*induction rule*: *vwalk_induct*; *auto*)
apply (*induction rule*: *steps.induct*; *auto*)
done

lemmas *graph_convs2* = *reachable_iff reachable1_iff vwalk_iff*

— Transferring a theorem $((?u \rightarrow^*_G ?v) = (\exists p.\ vpath\ p\ G \wedge hd\ p = ?u \wedge last\ p = ?v))$:
lemma *reachable_path_conv*:
$u \rightarrow* v \longleftrightarrow (\exists p.\ steps\ p \wedge distinct\ p \wedge hd\ p = u \wedge last\ p = v)$
unfolding *graph_convs2*[*symmetric*] **by** (*simp add*: *digraph.reachable_vpath_conv vpath_def*)

end

context *Subgraph_Defs*
begin

definition $G' = (\!|verts = UNIV,\ arcs = \{(a,\ b).\ E'\ a\ b\},\ tail = fst,\ head = snd|\!)$

definition $G_p' = (\!|pverts = UNIV,\ parcs = \{(a,\ b).\ E'\ a\ b\}|\!)$

lemma *G'_pair_conv*:
$with\_proj\ G_p' = G'$
unfolding *G_p'_def G'_def with_proj_def* **by** *simp*

sublocale *digraph_sub*: *wf_digraph G'*
unfolding *wf_digraph_def G'_def* **by** *simp*

sublocale *pdigraph_sub*: *pair_wf_digraph* G_p'
using *G'_pair_conv digraph_sub.wf_digraph_axioms wf_digraph_wp_iff* **by** *metis*

lemma *verts_eq*[*simp*]:
$verts\ G' = UNIV$
unfolding *G'_def* **by** *simp*

end

context *Subgraph*
begin

lemma *subgraph*:
subgraph G' G.G
unfolding *subgraph_def*
apply (*intro conjI*)
subgoal
by *simp*
subgoal

by (*auto simp*: *G.G_def G'_def*)
subgoal
by (*simp add*: *G.digraph.wf_digraph_axioms*)
subgoal
by (*simp add*: *digraph_sub.wf_digraph_axioms*)
subgoal
by (*simp add*: *compatible_def G.G_def G'_def*)
done

lemma *spanning*:
spanning G' G.G
unfolding *spanning_def* **by** (*simp add*: *subgraph*)

lemma *G'_eq*:
G'.G = *G'*
by (*auto simp*: *Graph_Defs.G_def G'_def*)

lemmas *subgraph_convs* = *G'.graph_convs2*[*unfolded G'_eq*]

end

context *Subgraph_Node_Defs*
begin

— Node-induced subgraph. Compare with *G'*.
definition G_n = (|*verts* = {*x*. *V x*}, *arcs* = {(*a*, *b*). *E' a b*}, *tail* = *fst*, *head* = *snd*|)

sublocale *digraph_nodes*: *wf_digraph* G_n
unfolding *wf_digraph_def* G_n*_def E'_def* **by** *simp*

lemma *verts_eq*[*simp*]:
verts G_n = {*x*. *V x*}
unfolding G_n*_def* **by** *simp*

lemma *arcs_eq*[*simp*]:
arcs G_n = *arcs G'*
unfolding G_n*_def G'_def* **by** *simp*

lemma *subgraph_dominatesI*:
$a \rightarrow_{G_n} b$ **if** *a* → *b V a V b*
using *that* **by** (*intro digraph_nodes.dominatesI*) (*auto simp*: G_n*_def arc_to_ends_def E'_def*)

lemma *arcs_ends_eq*:
arcs_ends G_n = *arcs_ends G'*
unfolding G_n*_def G'_def E'_def arcs_ends_def arc_to_ends_def* **by** *simp*

lemma *subgraph_nodes'*:
subgraph G_n *G'*
unfolding *subgraph_def*
apply (*intro conjI*)
subgoal
by *simp*
subgoal

by *simp*
subgoal
by (*simp add*: *digraph_sub.wf_digraph_axioms*)
subgoal
by (*simp add*: *digraph_nodes.wf_digraph_axioms*)
subgoal
by (*simp add*: *compatible_def G'_def* G_n*_def*)
done

lemma *subgraph_nodes*:
subgraph G_n *G*
by (*rule subgraph_nodes' subgraph subgraph_trans*)+

lemma *induced_subgraph*:
induced_subgraph G_n *G*
unfolding *induced_subgraph_def* **by** (*intro conjI subgraph_nodes*) (*auto simp*: G_n*_def G_def E'_def*)

— The notions of walks in the two versions of node-induced subgraphs are not equivalent for empty paths, thus this is the only valid "standard" conversion theorem between paths:
lemma *reachable1_iff*:
$u \rightarrow^+_{G_n} v \longleftrightarrow u \rightarrow^+_{G'} v$
unfolding *arcs_ends_eq* **..**

lemma *dominates_iff*:
$u \rightarrow_{G_n} v \longleftrightarrow E'\ u\ v$
unfolding G_n*_def arcs_ends_def arc_to_ends_def E'_def* **by** *simp*

lemma *subgraph_vwalk_iff*:
vwalk $(v \# vs)$ $G_n \longleftrightarrow G'.steps\ (v \# vs)$ **if** $V\ v$
apply (*rule iffI*)
apply (*induction rule*: *vwalk_induct*; *auto simp*: *dominates_iff*; *fail*)
subgoal premises *prems*
using *prems that*
by (*induction v # vs arbitrary*: *v vs rule*: *G'.steps.induct*; *auto simp*: *dominates_iff E'_def*)
done

lemma *subgraph_reaches_iff*:
$u \rightarrow^*_{G_n} v \longleftrightarrow G'.reaches\ u\ v$ **if** $V\ u$
by (*simp add*: *that G'.reaches_steps_iff2 digraph_nodes.reachable_vwalk_conv2 subgraph_vwalk_iff*)

lemma *subgraph_reaches1_iff*:
$u \rightarrow^+_{G_n} v \longleftrightarrow G'.reaches1\ u\ v$
unfolding *reachable1_iff subgraph_convs* **..**

end

context *Graph_Invariant*
begin

lemma *reachable_iff*:
$u \rightarrow^*_{G_n} v \longleftrightarrow u \rightarrow^*_{G} v$ **if** $P\ u$
using *that* **by** (*simp add*: *graph_convs2 subgraph_reaches_iff invariant_reaches_iff*)

lemma *reachable1_iff*:
u $\rightarrow^+_{G_n}$ *v* $\longleftrightarrow$ *u* $\rightarrow^+_{G}$ *v* **if** *P u*
unfolding *subgraph_reaches1_iff invariant_reaches1_iff*[*OF that*] *graph_convs2* **..**

lemma *vwalk_iff*:
vwalk (*u* # *vs*) G_n $\longleftrightarrow$ *vwalk* (*u* # *vs*) *G* **if** *P u*
using *that* **by** (*simp add*: *subgraph_vwalk_iff invariant_steps_iff graph_convs2*)

end

2.4 Application: Topological Numberings on the SCCs of a Digraph

context *fin_digraph*
begin

interpretation *scc_digraph*: *fin_dag scc_graph*
by (*rule scc_digraphI*)

definition
scc_num ≡ *SOME f* :: (_ ⇒ *nat*).
inj_on f sccs_verts ∧ (∀ *x*∈*sccs_verts*. ∀ *y*∈*sccs_verts*. *x* $\rightarrow_{scc\_graph}$ *y* ⟶ *f x* < *f y*)

lemma
shows *scc_num_inj*: *inj_on scc_num sccs_verts* (**is** *?thesis1*)
and *scc_num_topological*:
∀ *x*∈*sccs_verts*. ∀ *y*∈*sccs_verts*. *x* $\rightarrow_{scc\_graph}$ *y* ⟶ *scc_num x* < *scc_num y* (**is** *?thesis2*)
proof −
from *scc_digraph.topological_numbering*[*OF finite_sccs_verts*] **have** *?thesis1* ∧ *?thesis2*
unfolding *scc_num_def* **by** (*rule someI_ex*)
then show *?thesis1* **and** *?thesis2*
by *auto*
qed

end

context *Finite_Graph*
begin

interpretation *Graph_Invariant*
where *E* = *E* **and** *P* = λ*x*. *x* ∈ *vertices*
by *standard* (*auto simp*: *vertices_def*)

interpretation *digraph_nodes*: *fin_digraph* G_n
apply *standard*
subgoal *finite_verts*
by (*simp add*: *Subgraph_Node_Defs.*G_n*_def finite_graph*)
subgoal
by (*rule finite_subset*[**where** *B* = *verts* G_n × *verts* G_n])
(*auto simp*: *G'_def E'_def* G_n*_def finite_graph*)
done

definition

is_max_scc S ≡
S ⊆ *vertices* ∧ *S* ≠ {} ∧ (∀ *u* ∈ *S*. ∀ *v* ∈ *S*. *u* →∗ *v*) ∧ (∀ *u* ∈ *S*. ∀ *v*. *v* ∉ *S* ⟶ ¬*u* →∗ *v* ∨ ¬*v* →∗ *u*)

lemma *is_max_scc_iff*:
is_max_scc S ⟷ *S* ∈ *digraph_nodes.sccs_verts*
proof (*rule iffI*)
assume *is_max_scc S*
then show *S* ∈ *digraph_nodes.sccs_verts*
unfolding *is_max_scc_def digraph_nodes.sccs_verts_def*
by (*clarsimp*, *safe*) (*auto simp*: *reachable_iff graph_convs2 invariant_reaches*)
next
assume *S* ∈ *digraph_nodes.sccs_verts*
then have *S* ⊆ *vertices*
using *digraph_nodes.sccs_verts_subsets* **by** *auto*
then show *is_max_scc S*
using ‹*S* ∈ _› **unfolding** *is_max_scc_def digraph_nodes.sccs_verts_def*
by (*auto simp*: *Graph_Defs.reachable_iff in_mono invariant_reaches reachable_iff*)
qed

lemma *max_sccI*:
assumes *a* ∈ *vertices*
obtains *A* **where** *is_max_scc A a* ∈ *A*
subgoal premises *that*
using *assms*
by (*intro that*[*of* {*b*. *a* →∗ *b* ∧ *b* →∗ *a*}])
(*auto intro*: *invariant_reaches simp*: *is_max_scc_def*, (*meson rtranclp_trans*)+)
— XXX: graph automation
done

lemma *is_max_scc_disjoint*:
assumes *is_max_scc V is_max_scc W V* ≠ *W*
shows *V* ∩ *W* = {}
using *assms* **unfolding** *is_max_scc_iff* **by** (*rule digraph_nodes.sccs_verts_disjoint*)

definition
edge V W ≡ ∃ *a* ∈ *V*. ∃ *b* ∈ *W*. *V* ≠ *W* ∧ *E a b*

definition
scc_num ≡ *SOME f* :: (_ ⇒ *nat*).
inj_on f {*V*. *is_max_scc V*} ∧ (∀ *V*. ∀ *W*. *is_max_scc V* ∧ *is_max_scc W* ∧ *edge V W* ⟶ *f V* < *f W*)

interpretation *scc_digraph*: *fin_dag digraph_nodes.scc_graph*
by (*rule digraph_nodes.scc_digraphI*)

lemma *edge_iff*:
edge x y ⟷ *x* →$_{digraph\_nodes.scc\_graph}$ *y*
if *x* ∈ *digraph_nodes.sccs_verts y* ∈ *digraph_nodes.sccs_verts*
unfolding *edge_def*
apply *rule*
subgoal
using *that* **unfolding** *digraph_nodes.scc_graph_def arcs_ends_def arc_to_ends_def*
unfolding *G*$_n$*_def G'_def E'_def* **by** (*auto simp add*: *vertices_def*)

using *subgraph_nodes*
by (*auto 4 3 dest*!: *digraph.adj_mono*[*rotated*] *elim*!: *digraph_nodes.scc_graph_edgeE*)

lemma
shows *scc_num_inj*: *inj_on scc_num* {*V*. *is_max_scc V*} (**is** *?thesis1*)
and *scc_num_topological*:
∀ *V*. ∀ *W*. *is_max_scc V* ∧ *is_max_scc W* ∧ *edge V W* ⟶ *scc_num V* < *scc_num W* (**is** *?thesis2*)
proof −
let *?P* = λ*f* :: (_ ⇒ *nat*).
inj_on f {*V*. *is_max_scc V*} ∧ (∀ *V*. ∀ *W*. *is_max_scc V* ∧ *is_max_scc W* ∧ *edge V W* ⟶ *f V* < *f W*)
from *digraph_nodes.scc_num_inj digraph_nodes.scc_num_topological* **have** *?thesis1* ∧ *?thesis2*
unfolding *scc_num_def* **by** − (*rule someI*[**where** *P* = *?P*], *auto simp*: *is_max_scc_iff edge_iff*)
then show *?thesis1* **and** *?thesis2*
by *auto*
qed

end

end

3 Certificates for the Absence of Lassos

theory *Lasso_Freeness_Certificates_Complete*
imports *Main HOL−Library.Countable Graph_Library_Adaptor*
begin

This theory gives two valid ways of generating a certificate that proves that the given graph does not contain an accepting lasso. It is not directly used by the Muntac formalization but summarizes the idea of how to get from certificates for unreachability to certificates for the absence of Büechi properties.

locale *Contract_Downward* =
fixes *E* :: 'a :: *countable* ⇒ 'a ⇒ *bool* **and** *F* :: 'a ⇒ *bool*
fixes *f* :: 'a ⇒ *nat*
assumes *f_topo*: *E a b* ⟹ *if F a then f a* > *f b else f a* ≥ *f b*
begin

lemma *f_downward*:
assumes *E a b*
shows *f a* ≥ *f b*
using *assms f_topo* **by** (*cases F a*) (*auto simp*: *less_imp_le*)

lemma *f_strict*:
assumes *E a b F a*
shows *f a* > *f b*
using *assms f_topo* **by** (*auto simp*: *less_imp_le*)

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes *F a E a a*
shows *False*

```
  using assms f_topo by (meson less_irrefl)

lemma reaches_f_mono:
  assumes E** a b
  shows f a ≥ f b
  using assms by induction (auto intro: f_downward order.trans)

lemma no_accepting_cycle:
  assumes E++ x x
  shows ¬ F x
proof (rule ccontr, unfold not_not)
  assume F x
  from ‹E++ x x› obtain y where E x y E** y x
    including graph_automation_aggressive by auto
  from ‹E x y› ‹F x› have f x > f y
    by (rule f_strict)
  moreover from ‹E** y x› have f y ≥ f x
    by (rule reaches_f_mono)
  ultimately show False
    by simp
qed

sublocale Graph_Defs .

lemma run_f_mono:
  assumes run (x ## xs) c ≥ f x
  shows pred_stream (λa. c ≥ f a) xs
  using assms
  by (coinduction arbitrary: x xs) (metis f_downward order.trans run.cases stream.inject)

lemma no_Buechi_run:
  assumes run xs infs F xs
  shows False
proof −
  let ?S = f ‘ sset xs let ?T = f ‘ sset (sfilter F xs)
  obtain x ys where xs = x ## ys
    by (cases xs)
  let ?ub = f x
  from ‹run xs› have run ys f (shd ys) ≤ ?ub
    unfolding ‹xs = _› by (force elim: run.cases dest: f_downward)+
  then have pred_stream (λx. f x ≤ ?ub) ys
    by (coinduction arbitrary: ys) (fastforce elim: run.cases dest: f_downward intro: order.trans)
  then have finite ?S
    unfolding stream.pred_set ‹xs = _› finite_nat_set_iff_bounded_le by auto
  from ‹run xs› ‹infs F xs› have sdistinct (smap f (sfilter F xs))
  proof (coinduction arbitrary: xs)
    case (sdistinct x xs ys)
    obtain y xs' where [simp]: xs = y ## xs'
      by (cases xs)
    obtain as bs x' y' ys' where eqs:
      F x' F y' x = f x' y = f y'
      ys = as @− x' ## bs @− y' ## ys' xs' = smap f (sfilter F ys')
      list_all (Not o F) as list_all (Not o F) bs
    proof (atomize_elim, goal_cases)
```

```
    case 1
    from sdistinct have ev (holds F) ys
      by auto
    from ‹x ## xs = _› have x ## y ## xs' = smap f (sfilter F ys)
      by simp
    then obtain x' y' ys' where x = f x' y = f y' sfilter F ys = x' ## y' ## ys'
      xs' = smap f ys'
      apply atomize_elim
      unfolding scons_eq_smap
      apply (elim conjE)
      apply (intro exI conjI)
       apply assumption+
      apply (subst stream.collapse)+
      ..
    with ‹ev (holds F) ys› show ?case
      apply −
      apply (drule (1) sfilter_SCons_decomp)
      apply (elim conjE exE)
      apply (drule sfilter_SCons_decomp, metis alw.cases alw_shift sdistinct(3) stream.sel(2))
      by force
  qed
  have run (y' ## ys')
    using ‹run ys› unfolding ‹ys = _›
    by (metis alw.cases alw_iff_sdrop alw_shift run_sdrop stream.sel(2))
  from eqs ‹run ys› have steps (as @ x' # bs @ [y'])
    by (simp add: run_decomp)
  then have steps (x' # bs @ [y'])
    using steps_appendD2 by blast
  then have E++ x' y'
    using steps_reaches1 by blast
  with ‹F x'› have f x' > f y'
    using f_strict reaches1_reaches_iff1 reaches_f_mono by fastforce
  moreover have pred_stream (λa. f y' ≥ f a) ys'
    by (rule run_f_mono[where c = f y']) (auto intro: ‹run (y' ## ys')›)
  ultimately have x ∉ f ‘ sset ys'
    unfolding stream.pred_set eqs by auto
  from ‹infs F ys› have infs F ys'
    unfolding eqs by auto
  then have sset (sfilter F ys') ⊆ sset ys'
    by (rule sset_sfilter)
  with ‹x ∉ _› ‹f x' > f y'› have x ∉ sset xs
    unfolding stream.pred_set eqs ‹xs = _› by auto
  with ‹run (y' ## ys')› ‹infs F ys'› eqs ‹xs' = _› show ?case
    by − (safe, rule exI[where x = y'##ys'], auto)
 qed
 then have infinite ?T
   by − (drule sdistinct_infinite_sset, simp)
 moreover have ?T ⊆ ?S
   by (rule image_mono sset_sfilter assms)+
 ultimately show ?thesis
   using ‹finite ?S› finite_subset by auto
qed

end
```

context
  fixes $E :: 'a :: countable \Rightarrow 'a \Rightarrow bool$ **and** $F :: 'a \Rightarrow bool$
begin

context
  fixes $f :: 'a \Rightarrow nat$
  assumes *f_topo*: $E\ a\ b \Longrightarrow$ *if* $F\ a$ *then* $f\ a < f\ b$ *else* $f\ a \leq f\ b$
begin

lemma *f_forward*:
  assumes $E\ a\ b$
  shows $f\ a \leq f\ b$
  using *assms f_topo* **by** (*cases* $F\ a$) (*auto simp*: *less_imp_le*)

lemma *f_strict*:
  assumes $E\ a\ b\ F\ a$
  shows $f\ a < f\ b$
  using *assms f_topo* **by** (*auto simp*: *less_imp_le*)

We do not even need this property any more.

lemma *no_trivial_lasso*:
  assumes $F\ a\ E\ a\ a$
  shows *False*
  using *assms f_topo* **by** (*meson less_irrefl*)

lemma *reaches_f_mono*:
  assumes $E^{**}\ a\ b$
  shows $f\ a \leq f\ b$
  using *assms* **by** *induction* (*auto intro*: *f_forward order.trans*)

lemma *no_accepting_cycle*:
  assumes $E^{++}\ x\ x$
  shows $\neg\ F\ x$
proof (*rule ccontr*, *unfold not_not*)
  assume $F\ x$
  from ‹$E^{++}\ x\ x$› **obtain** y **where** $E\ x\ y\ E^{**}\ y\ x$
    including *graph_automation_aggressive* **by** *auto*
  from ‹$E\ x\ y$› ‹$F\ x$› **have** $f\ x < f\ y$
    by (*rule f_strict*)
  moreover from ‹$E^{**}\ y\ x$› **have** $f\ y \leq f\ x$
    by (*rule reaches_f_mono*)
  ultimately show *False*
    by *simp*
qed

end

context
  assumes *no_accepting_cycle*: $E^{++}\ x\ x \Longrightarrow \neg\ F\ x$
begin

definition
  reach $x \equiv$ *card* $\{y.\ F\ y \wedge E^{**}\ x\ y\}$

context
  assumes *finite*: *finite* $\{y.\ E^{**}\ x\ y\}$
begin

lemma *reach_mono*:
  assumes $E\ x\ y$
  shows $reach\ x \geq reach\ y$
  using *assms* **unfolding** *reach_def* **by** (*intro card_mono finite_subset*[*OF* _ *finite*]) *auto*

lemma *reach_strict*:
  assumes $E\ x\ y\ F\ x$
  shows $reach\ x > reach\ y$
proof −
  have *False*
    if $E\ x\ y$
      and $F\ x$
      and $\{ya.\ F\ ya \wedge E^{**}\ y\ ya\} = \{y.\ F\ y \wedge E^{**}\ x\ y\}$
  proof −
    from *that* **have** $E^{**}\ y\ x$
      by *auto*
    with ‹$E\ x\ y$› **have** $E^{++}\ x\ x$
      by *auto*
    with ‹$F\ x$› *no_accepting_cycle* **show** *False*
      by *auto*
  qed
  then show *?thesis*
    using *assms* **unfolding** *reach_def*
    apply (*intro psubset_card_mono finite_subset*[*OF* _ *finite*])
     apply *force*
    by *auto*
qed

end

end

end

context *Finite_Graph*
begin

context
  assumes *no_accepting_cycle*: $E^{++}\ x\ x \Longrightarrow \neg\ F\ x$
begin

private definition
  $f\ x \equiv scc\_num\ (THE\ V.\ is\_max\_scc\ V \wedge x \in V)$

lemma *f_topo*:
  assumes $E\ a\ b$
  shows $if\ F\ a\ then\ f\ a < f\ b\ else\ f\ a \leq f\ b$
proof −
  from *assms* **obtain** A **where** A: $is\_max\_scc\ A\ a \in A$

```
    by − (rule max_sccI, auto simp: vertices_def)
  from assms obtain B where B: is_max_scc B b ∈ B
    by − (rule max_sccI[where a = b], auto simp: vertices_def)
  from A B have [simp]: f a = scc_num A f b = scc_num B
    unfolding f_def using is_max_scc_disjoint
    by (intro arg_cong[where f = scc_num] the_equality, auto)+
  have f a ≤ f b
  proof (cases A = B)
    case True
    then show ?thesis
      by simp
  next
    case False
    with ‹E a b› ‹a ∈ A› ‹b ∈ B› have edge A B
      unfolding edge_def by auto
    then have f a < f b
      using A B scc_num_topological by simp
    then show ?thesis
      by simp
  qed
  moreover have f a ≠ f b if F a
  proof (rule ccontr)
    assume ¬ f a ≠ f b
    with A B have A = B
      using scc_num_inj unfolding inj_on_def by simp
    with A ‹b ∈ B› have E** b a
      unfolding is_max_scc_def by auto
    with ‹E a b› have E++ a a
      by auto
    with ‹F a› no_accepting_cycle show False
      by auto
  qed
  ultimately show ?thesis
    by auto
qed

end

end

end
```

4 Certification Theorems Based on Subsumption and Simulation Graphs

```
theory Simulation_Graphs_Certification
 imports Munta_Base.Subsumption_Graphs Timed_Automata.Simulation_Graphs HOL−Eisbach.Eisbach
begin
```

4.1 Misc

```
lemma finite_ImageI:
  assumes finite S ∀ a ∈ S. finite {b. (a, b) ∈ R}
```

```
  shows finite (R `` S)
proof −
  have R `` S ⊆ (⋃ x ∈ S. {y. (x, y) ∈ R})
    unfolding Image_def by auto
  also have finite ...
    using assms by auto
  finally show ?thesis .
qed

lemma (in Graph_Defs) steps_two_iff[simp]:
  steps [a, b] ⟷ E a b
  by (auto elim!: steps.cases)

lemma (in Graph_Defs) steps_Cons_two_iff:
  steps (a # b # as) ⟷ E a b ∧ steps (b # as)
  by (auto elim: steps.cases)
```

4.2 Certifying Cycle-Freeness in Graphs

```
locale Contract1 =
  fixes A B :: 'a ⇒ 'a ⇒ bool and G :: 'a ⇒ bool
begin

definition E a b ≡ A a b ∧ G a ∨ ¬ G a ∧ B a b ∧ G b
definition E' a b ≡ G a ∧ G b ∧ (A a b ∨ (∃ c. A a c ∧ ¬ G c ∧ B c b))

sublocale E: Graph_Defs E .
sublocale E': Graph_Defs E' .

lemma steps_contract:
  assumes E.steps (a # xs @ [b]) G a G b
  shows ∃ as. E'.steps (a # as @ [b])
  using assms
proof (induction xs arbitrary: a rule: length_induct)
  case prems: (1 xs)
  show ?case
  proof (cases xs)
    case Nil
    with prems show ?thesis
      apply (inst_existentials [] :: 'a list)
      apply simp
      apply (auto simp: E_def E'_def )
      done
  next
    case (Cons y ys)
    with ‹E.steps (a # xs @ [b])› have E a y E.steps (y # ys @ [b])
      by (cases, auto)+
    from this(1) ‹G a› consider (graph) A a y G a G y | (non_graph) A a y G a ¬ G y
      unfolding E_def by auto
    then show ?thesis
    proof cases
      case graph
      with ‹E.steps (y # ys @ [b])› prems(1) ‹G b› obtain as where E'.steps (y # as @ [b])
        unfolding ‹xs = _› by auto
```

```
      also from graph have E' a y
        unfolding E'_def by auto
      finally (E'.steps.Cons[rotated]) show ?thesis
        by (inst_existentials y # as) auto
    next
      case non_graph
      show ?thesis
      proof (cases ys)
        case Nil
        with ‹E.steps (y # ys @ [b])› have E y b
          by auto
        with ‹¬ G y› have B y b G b
          unfolding E_def by auto
        with ‹A a y› ‹G a› ‹¬ G y› have E'.steps [a, b]
          by (auto simp: E'_def)
        then show ?thesis
          by (inst_existentials [] :: 'a list) auto
      next
        case (Cons z zs)
        with ‹E.steps (y # ys @ [b])› have E y z E.steps (z # zs @ [b])
          by (auto simp: E.steps_Cons_two_iff)
        with ‹¬ G y› have B y z G z
          unfolding E_def by auto
        with ‹A a y› ‹G a› ‹¬ G y› have E' a z
          by (auto simp: E'_def)
        also from ‹E.steps (z # zs @ [b])› prems(1) obtain as where E'.steps (z # as @ [b])
          unfolding ‹xs = _› ‹ys = _› using ‹G z› ‹G b› by force
        finally (E'.steps.Cons) show ?thesis
          by (inst_existentials z # as) auto
      qed
    qed
  qed
qed

lemma E'_G[dest, intro]:
  assumes E' x y
  shows G x G y
  using assms unfolding E'_def by auto

lemma E'_steps_G:
  assumes E'.steps (x # xs) xs ≠ []
  shows G x
proof −
  from ‹E'.steps _› ‹xs ≠ []› obtain y where E' x y
    by (auto elim: E'.steps.cases)
  then show ?thesis
    by auto
qed

lemma E_E'_cycle:
  assumes E^++ x x G x
  shows E'^++ x x
proof −
  from ‹E^++ x x› obtain xs where E.steps (x # xs @ [x])
```

including *graph_automation* **by** *auto*
with ‹*G x*› **obtain** *ys* **where** *E′.steps* (*x* # *ys* @ [*x*])
by (*auto dest*: *steps_contract*)
then show *?thesis*
using *E′.reaches1_steps_iff* **by** *blast*
qed

This can be proved by rotating the cycle if ¬ *G x*

lemma
assumes E^{**} *s x* E^{++} *x x G s*
shows E'^{**} *s x* ∧ E'^{++} *x x*
proof −
show *?thesis*
proof (*cases G x*)
case *True*
with ‹E^{++} *x x*› **have** E'^{++} *x x*
by (*auto dest*: *E_E′_cycle*)
from ‹E^{**} *s x*› **consider** (*refl*) *s* = *x* | (*steps*) E^{++} *s x*
by *cases* (*auto simp*: *E.reaches1_reaches_iff2*)
then show *?thesis*
proof *cases*
case *refl*
with ‹E'^{++} *x x*› **show** *?thesis*
by *simp*
next
case *steps*
then obtain *xs* **where** *E.steps* (*s* # *xs* @ [*x*])
including *graph_automation* **by** *auto*
with ‹*G s*› ‹*G x*› **obtain** *ys* **where** *E′.steps* (*s* # *ys* @ [*x*])
by (*auto dest*: *steps_contract*)
with ‹E'^{++} *x x*› **show** *?thesis*
including *graph_automation* **by** *auto*
next
oops

Certifying cycle-freeness with a topological ordering of graph components

context
fixes *f* :: *'a* ⇒ *nat* **and** *F* :: *'a* ⇒ *bool*
assumes *f_topo1*: ∀ *a b*. *G a* ∧ *A a b* ∧ *G b* ⟶ (*if F a then f a* < *f b else f a* ≤ *f b*)
and *f_topo2*:
∀ *a b c*. *G a* ∧ *A a b* ∧ ¬ *G b* ∧ *G c* ∧ *B b c* ⟶ (*if F a then f a* < *f c else f a* ≤ *f c*)
begin

lemma *f_forward*:
assumes *E′ a b*
shows *f a* ≤ *f b*
using *assms f_topo1 f_topo2* **unfolding** *E′_def* **by** (*cases F a*) (*auto simp*: *less_imp_le*)

lemma *f_strict*:
assumes *E′ a b F a*
shows *f a* < *f b*
using *assms f_topo1 f_topo2* **unfolding** *E′_def* **by** (*auto simp*: *less_imp_le*)

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes $F\ a\ G\ a\ E'\ a\ a$
shows *False*
using *assms f_topo1 f_topo2* **unfolding** *E'_def* **by** (*meson less_irrefl*)

lemma *reaches_f_mono*:
assumes $E'^{**}\ a\ b$
shows $f\ a \leq f\ b$
using *assms* **by** *induction* (*auto intro*: *f_forward order.trans*)

theorem *no_accepting_cycle*:
assumes $E'^{++}\ x\ x$
shows $\neg\ F\ x$
proof (*rule ccontr*, *unfold not_not*)
assume $F\ x$
from ‹$E'^{++}\ x\ x$› **obtain** y **where** $E'\ x\ y\ E'^{**}\ y\ x$
including *graph_automation_aggressive* **by** *auto*
from ‹$E'\ x\ y$› ‹$F\ x$› **have** $f\ x < f\ y$
by (*rule f_strict*)
moreover from ‹$E'^{**}\ y\ x$› **have** $f\ y \leq f\ x$
by (*rule reaches_f_mono*)
ultimately show *False*
by *simp*
qed

end

end

locale *Contract* =
fixes $E :: 'a \Rightarrow 'a \Rightarrow bool$
fixes $f :: 'a \Rightarrow nat$ **and** $F :: 'a \Rightarrow bool$
assumes *f_topo*:
$\forall\ a\ b.\ E\ a\ b \longrightarrow (if\ F\ a\ then\ f\ a < f\ b\ else\ f\ a \leq f\ b)$
begin

sublocale *E*: *Graph_Defs E* **.**

lemma *f_forward*:
assumes $E\ a\ b$
shows $f\ a \leq f\ b$
using *assms f_topo* **by** (*cases F a*) (*auto simp*: *less_imp_le*)

lemma *f_strict*:
assumes $E\ a\ b\ F\ a$
shows $f\ a < f\ b$
using *assms f_topo* **by** (*auto simp*: *less_imp_le*)

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes $F\ a\ G\ a\ E\ a\ a$
shows *False*
using *assms f_topo* **by** (*meson less_irrefl*)

lemma *reaches_f_mono*:
assumes $E^{**}\ a\ b$
shows $f\ a \leq f\ b$
using *assms* **by** *induction* (*auto intro*: *f_forward order.trans*)

theorem *no_accepting_cycle*:
assumes $E^{++}\ x\ x$
shows $\neg\ F\ x$
proof (*rule ccontr*, *unfold not_not*)
assume $F\ x$
from ‹$E^{++}\ x\ x$› **obtain** y **where** $E\ x\ y\ E^{**}\ y\ x$
including *graph_automation_aggressive* **by** *auto*
from ‹$E\ x\ y$› ‹$F\ x$› **have** $f\ x < f\ y$
by (*rule f_strict*)
moreover from ‹$E^{**}\ y\ x$› **have** $f\ y \leq f\ x$
by (*rule reaches_f_mono*)
ultimately show *False*
by *simp*
qed

end

locale *Contract2* =
fixes $A\ B :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $G :: 'a \Rightarrow bool$
fixes $f :: 'a \Rightarrow nat$ **and** $F :: 'a \Rightarrow bool$
assumes *f_topo*:
$\forall\ a\ b\ c.\ G\ a \wedge A\ a\ b \wedge G\ c \wedge B\ b\ c \longrightarrow (\mathit{if}\ F\ a\ \mathit{then}\ f\ a < f\ c\ \mathit{else}\ f\ a \leq f\ c)$
begin

definition $E\ a\ c \equiv \exists\ b.\ G\ a \wedge G\ c \wedge A\ a\ b \wedge B\ b\ c$

sublocale *Contract E f F*
by *standard* (*auto simp*: *E_def f_topo*)

theorem *no_accepting_cycle*:
assumes $E^{++}\ x\ x$
shows $\neg\ F\ x$
using *assms* **by** (*rule no_accepting_cycle*)

end

4.3 Reachability and Over-approximation

context
fixes $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $P :: 'a \Rightarrow bool$
and *less_eq* $:: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** $\preceq$ *50*) **and** *less* (**infix** $\prec$ *50*)
assumes *preorder*: *class.preorder less_eq less*
assumes *mono*: $a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow \exists\ b'.\ E\ b\ b' \wedge a' \preceq b'$
assumes *invariant*: $P\ a \Longrightarrow E\ a\ a' \Longrightarrow P\ b$
begin

interpretation *preorder less_eq less*
by (*rule preorder*)

interpretation *Simulation_Invariant*
  $E\ \lambda\ x\ y.\ \exists\ z.\ z \preceq y \wedge E\ x\ z \wedge P\ z\ (\preceq)\ P\ P$
  by *standard* (*auto 0 4 intro*: *invariant dest*: *mono*)

context
  fixes $F :: 'a \Rightarrow bool$ — Final states
  assumes *F_mono*[*intro*]: $F\ a \Longrightarrow a \preceq b \Longrightarrow F\ b$
begin

corollary *reachability_correct*:
  $\nexists\ s'.\ A.reaches\ a\ s' \wedge F\ s'$ **if** $\nexists\ s'.\ B.reaches\ b\ s' \wedge F\ s'\ a \preceq b\ P\ a\ P\ b$
  using *that* **by** (*auto dest*!: *simulation_reaches*)

end

end

context *Simulation_Invariant*
begin

context
  fixes $F :: 'a \Rightarrow bool$ **and** $F' :: 'b \Rightarrow bool$ — Final states
  assumes *F_mono*[*intro*]: $F\ a \Longrightarrow a \sim b \Longrightarrow F'\ b$
begin

corollary *reachability_correct*:
  $\nexists\ s'.\ A.reaches\ a\ s' \wedge F\ s'$ **if** $\nexists\ s'.\ B.reaches\ b\ s' \wedge F'\ s'\ a \sim b\ PA\ a\ PB\ b$
  using *that* **by** (*auto dest*!: *simulation_reaches*)

end

end

4.4 Invariants for Un-reachability

locale *Unreachability_Invariant0 = Subsumption_Graph_Pre_Defs + preorder less_eq less +*
  fixes $S :: 'a\ set$ **and** $SE :: 'a \Rightarrow 'a \Rightarrow bool$
  assumes *mono*:
    $s \preceq s' \Longrightarrow s \rightarrow t \Longrightarrow \exists\ t'.\ t \preceq t' \wedge s' \rightarrow t'$
  assumes *S_E_subsumed*: $s \in S \Longrightarrow s \rightarrow t \Longrightarrow \exists\ t' \in S.\ SE\ t\ t'$
  assumes *subsumptions_subsume*: $SE\ s\ t \Longrightarrow s \preceq t$
begin

definition E' **where**
  $E'\ s\ t \equiv \exists s'.\ E\ s\ s' \wedge SE\ s'\ t \wedge t \in S$

sublocale *G*: *Graph_Defs* E' **.**

interpretation *Simulation_Invariant E E'* $(\preceq)$ $\lambda s.\ True$ $\lambda x.\ x \in S$
proof *standard*
  fix $a\ b\ a' :: 'a$
  assume $\langle a \rightarrow b \rangle\ \langle a' \in S \rangle\ \langle a \preceq a' \rangle$
  with *mono*[*OF* $\langle a \preceq a' \rangle\ \langle a \rightarrow b \rangle$] **obtain** b' **where** $b \preceq b'\ a' \rightarrow b'$

```
    by auto
  with S_E_subsumed[OF ‹a' ∈ S›] obtain c where c ∈ S SE b' c
    by auto
  with ‹a' → b'› have E' a' c
    unfolding E'_def by auto
  with ‹b ⪯ b'› ‹SE b' c› subsumptions_subsume show ‹∃ b'. E' a' b' ∧ b ⪯ b'›
    by (blast intro: order_trans)
qed (auto simp: E'_def)

context
  fixes s₀ s₀'
  assumes s₀ ⪯ s₀' s₀' ∈ S
begin

lemma run_subsumed:
  assumes run (s₀ ## xs)
  obtains ys where G.run (s₀' ## ys) stream_all2 (⪯) xs ys pred_stream (λx. x ∈ S) ys
proof −
  from ‹s₀ ⪯ _› ‹s₀' ∈ S› have equiv' s₀ s₀'
    unfolding equiv'_def by auto
  with assms show ?thesis
    by − (drule simulation_run, auto
        dest: PB_invariant.invariant_run elim: stream_all2_weaken intro!: that simp: equiv'_def)
qed

context
  fixes F :: 'a ⇒ bool — Final states
  assumes F_mono[intro]: reaches s₀ a ⟹ F a ⟹ a ⪯ b ⟹ b ∈ S ⟹ F b
begin

corollary final_unreachable:
  ∄ s'. reaches s₀ s' ∧ F s' if ∀ s' ∈ S. ¬ F s'
  using ‹s₀ ⪯ s₀'› ‹s₀' ∈ S› simulation_reaches that by blast

lemma buechi_run_lasso:
  assumes finite S run (s₀ ## xs) alw (ev (holds F)) (s₀ ## xs)
  obtains s where G.reaches s₀' s G.reaches1 s s F s
proof −
  interpret Finite_Graph E' s₀'
    by (standard, rule finite_subset[OF _ assms(1)])
      (auto intro: PB_invariant.invariant_reaches ‹s₀' ∈ S›)
  from run_subsumed[OF assms(2)] obtain ys where ys:
    G.run (s₀' ## ys) stream_all2 (⪯) xs ys pred_stream (λx. x ∈ S) ys .
  moreover from ‹run _› have pred_stream (reaches s₀) xs
    using run_reachable by blast
  ultimately have stream_all2 (λx y. x ⪯ y ∧ reaches s₀ x ∧ y ∈ S) xs ys
    by (smt stream.pred_set stream.rel_mono_strong)
  with assms(3) ‹s₀ ⪯ _› ‹s₀' ∈ S› have alw (ev (holds F)) (s₀' ## ys)
    by (elim alw_ev_lockstep) (erule stream.rel_intros[rotated], auto)
  from buechi_run_lasso[OF ys(1) this] show ?thesis
    using that .
qed

end
```

end

end

locale *Unreachability_Invariant1* = *Subsumption_Graph_Pre_Defs* + *preorder less_eq less* +
  fixes *I* **and** *SE* :: *'a* ⇒ *'a* ⇒ *bool* **and** *P*
  assumes *mono*:
    $s \preceq s' \Longrightarrow s \to t \Longrightarrow P\ s \Longrightarrow P\ s' \Longrightarrow \exists\ t'.\ t \preceq t' \wedge s' \to t'$
  assumes *S_E_subsumed*: $I\ s \Longrightarrow s \to t \Longrightarrow \exists\ t'.\ I\ t' \wedge SE\ t\ t'$
  assumes *subsumptions_subsume*: $SE\ s\ t \Longrightarrow s \preceq t$
  assumes *I_P*[*intro*]: $I\ s \Longrightarrow P\ s$
  assumes *P_invariant*: $P\ s \Longrightarrow s \to s' \Longrightarrow P\ s'$
begin

context
  assumes *subsumes_P*: $\bigwedge s\ s'.\ s \preceq s' \Longrightarrow P\ s'$
begin

interpretation *E*: *Unreachability_Invariant0*
  where $E = \lambda\ x\ y.\ E\ x\ y \wedge P\ y$
    and $S = \{s.\ I\ s\}$
  apply *standard*
  subgoal for *s s' t*
    using *subsumes_P mono* **by** *blast*
  using *S_E_subsumed subsumptions_subsume* **by** *auto*

end

Alternative for defining the simulating transition system. Does not need the invariant at the tip of the subsumption but requires us to use *Simulation* instead of *Simulation_Invariant*.

definition *E'* **where**
  $E'\ s\ t \equiv \exists s'.\ E\ s\ s' \wedge SE\ s'\ t \wedge I\ t$

sublocale *G*: *Graph_Defs E'* **.**

interpretation *Simulation_Invariant E E'* (⪯) *P I*
proof *standard*
  fix *a b a'* :: ‹*'a*›
  assume ‹$a \to b$› ‹*P a*› ‹*I a'*› ‹$a \preceq a'$›
  with *mono*[*OF* ‹$a \preceq a'$› ‹$a \to b$›] **obtain** *b'* **where** $b \preceq b'\ a' \to b'$
    by *auto*
  with *S_E_subsumed*[*OF* ‹*I a'*›] **obtain** *c* **where** *I c SE b' c*
    by *auto*
  with ‹$a' \to b'$› **have** *E' a' c*
    unfolding *E'_def* **by** *auto*
  with ‹$b \preceq b'$› ‹*SE b' c*› *subsumptions_subsume* **show** ‹$\exists\, b'.\ E'\ a'\ b' \wedge b \preceq b'$›
    by (*blast intro*: *order_trans*)
qed (*auto simp*: *E'_def P_invariant*)

context
  fixes $s_0\ {s_0}'$
  assumes $s_0 \preceq {s_0}'\ P\ s_0\ I\ {s_0}'$

begin

lemma *run_subsumed*:
assumes *run* (s_0 ## *xs*)
obtains *ys* **where** *G.run* (s_0' ## *ys*) *stream_all2* ($\preceq$) *xs ys pred_stream I ys*
proof −
from ‹$s_0 \preceq$ _› ‹*P* s_0› ‹*I* s_0'› **have** *equiv'* s_0 s_0'
unfolding *equiv'_def* **by** *auto*
with *assms* **show** *?thesis*
by − (*drule simulation_run*, *auto*
dest: *PB_invariant.invariant_run elim*: *stream_all2_weaken intro*!: *that simp*: *equiv'_def*)
qed

context
fixes *F* :: *'a* ⇒ *bool* — Final states
assumes *F_mono*[*intro*]: *P a* ⟹ *F a* ⟹ *a* $\preceq$ *b* ⟹ *P b* ⟹ *F b*
begin

corollary *final_unreachable*:
∄ *s'*. *reaches* s_0 *s'* ∧ *F s'* **if** ∀ *s'*. *I s'* ⟶ ¬ *F s'*
using ‹$s_0 \preceq s_0'$› ‹*P* s_0› ‹*I* s_0'› *simulation_reaches that I_P* **by** *blast*

lemma *buechi_run_lasso*:
assumes *finite* {*s*. *I s*} *run* (s_0 ## *xs*) *alw* (*ev* (*holds F*)) (s_0 ## *xs*)
obtains *s* **where** *G.reaches* s_0' *s G.reaches1 s s F s*
proof −
interpret *Finite_Graph E'* s_0'
by (*standard*, *rule finite_subset*[*OF* _ *assms*(*1*)])
(*auto intro*: *PB_invariant.invariant_reaches* ‹*I* s_0'›)
from *run_subsumed*[*OF assms*(*2*)] **obtain** *ys* **where** *ys*:
G.run (s_0' ## *ys*) *stream_all2* ($\preceq$) *xs ys pred_stream I ys* .
moreover from ‹*run* _› **have** *pred_stream P xs*
using *PA_invariant.invariant_run* ‹*P* s_0› **by** *auto*
ultimately have *stream_all2* (λ*x y*. *x* $\preceq$ *y* ∧ *P x* ∧ *I y*) *xs ys*
by (*smt stream.pred_set stream.rel_mono_strong*)
with *assms*(*3*) ‹$s_0 \preceq$ _› ‹*P* s_0› ‹*I* s_0'› **have** *alw* (*ev* (*holds F*)) (s_0' ## *ys*)
by (*elim alw_ev_lockstep*) (*erule stream.rel_intros*[*rotated*], *auto*)
from *buechi_run_lasso*[*OF ys*(*1*) *this*] **show** *?thesis*
using *that* .
qed

end

end

end

locale *Unreachability_Invariant* = *Subsumption_Graph_Pre_Defs* + *preorder less_eq less* +
fixes s_0
fixes *S* :: *'a set* **and** *SE* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *mono*:
s $\preceq$ *s'* ⟹ *s* → *t* ⟹ *s* ∈ *S* ∨ *reaches* s_0 *s* ⟹ *s'* ∈ *S* ∨ *reaches* s_0 *s'*
⟹ ∃ *t'*. *t* $\preceq$ *t'* ∧ *s'* → *t'*

assumes *S_E_subsumed*: $s \in S \Longrightarrow s \to t \Longrightarrow \exists\ t' \in S.\ SE\ t\ t'$
assumes *subsumptions_subsume*: $SE\ s\ t \Longrightarrow s \preceq t$
begin

interpretation *Unreachability_Invariant1*
where $P = \lambda s.\ s \in S \vee reaches\ s_0\ s$
and $I = \lambda s.\ s \in S$
apply *standard*
using *mono S_E_subsumed subsumptions_subsume*
apply (*solves auto*)+

oops

lemma *reachable_S_subsumed*:
$\exists\ s'.\ s' \in S \wedge s \preceq s'$ **if** $s' \in S\ s_0 \preceq s'\ reaches\ s_0\ s$
supply [*intro*] = *order_trans less_trans less_imp_le*
using *that*(*3*) **apply** *induction*
subgoal
using *that*(*1*,*2*) **by** *blast*
apply *clarify*
apply (*drule mono*)
using *S_E_subsumed subsumptions_subsume* **by** *blast*+

definition E' **where**
$E'\ s\ t \equiv \exists\, s'.\ E\ s\ s' \wedge SE\ s'\ t \wedge t \in S$

sublocale *G*: *Graph_Defs* E' **.**

context
fixes $s_0{}'$
assumes $s_0 \preceq s_0{}'\ s_0{}' \in S$
begin

interpretation *Simulation_Invariant E* E' $(\preceq)$ $\lambda s.\ reaches\ s_0\ s\ \lambda x.\ x \in S$
proof *standard*
fix $a\ b\ a' :: \langle'a\rangle$
assume $\langle a \to b\rangle\ \langle s_0 \to* a\rangle\ \langle a' \in S\rangle\ \langle a \preceq a'\rangle$
with *mono*[*OF* $\langle a \preceq a'\rangle\ \langle a \to b\rangle$] **obtain** b' **where** $b \preceq b'\ a' \to b'$
by *auto*
with *S_E_subsumed*[*OF* $\langle a' \in S\rangle$] **obtain** c **where** $c \in S\ SE\ b'\ c$
by *auto*
with $\langle a' \to b'\rangle$ **have** $E'\ a'\ c$
unfolding *E'_def* **by** *auto*
with $\langle b \preceq b'\rangle\ \langle SE\ b'\ c\rangle$ *subsumptions_subsume* **show** $\langle\exists\, b'.\ E'\ a'\ b' \wedge b \preceq b'\rangle$
by (*blast intro*: *order_trans*)
qed (*auto simp*: *E'_def*)

lemma *run_subsumed*:
assumes *run* $(s_0\ \#\#\ xs)$
obtains *ys* **where** *G.run* $(s_0{}'\ \#\#\ ys)$ *stream_all2* $(\preceq)$ *xs ys pred_stream* $(\lambda x.\ x \in S)$ *ys*
proof −
from $\langle s_0 \preceq \_\rangle\ \langle s_0{}' \in S\rangle$ **have** *equiv'* $s_0\ s_0{}'$
unfolding *equiv'_def* **by** *auto*
with *assms* **show** *?thesis*

by − (*drule simulation_run*, *auto*
dest: *PB_invariant.invariant_run elim*: *stream_all2_weaken intro*!: *that simp*: *equiv'_def*)
qed

context
fixes $F :: 'a \Rightarrow bool$ — Final states
assumes *F_mono*[*intro*]: *reaches* s_0 $a \Longrightarrow F\ a \Longrightarrow a \preceq b \Longrightarrow b \in S \Longrightarrow F\ b$
begin

corollary *final_unreachable*:
$\nexists\ s'$. *reaches* s_0 $s' \wedge F\ s'$ **if** $\forall\ s' \in S. \neg F\ s'$
using *that* ‹$s_0 \preceq {s_0}'$› ‹${s_0}' \in S$› **by** (*auto 4 3 dest*: *reachable_S_subsumed*)

lemma *buechi_run_lasso*:
assumes *finite S run* (s_0 ## *xs*) *alw* (*ev* (*holds F*)) (s_0 ## *xs*)
obtains *s* **where** *G.reaches* ${s_0}'$ *s G.reaches1 s s F s*
proof −
interpret *Finite_Graph E'* ${s_0}'$
by (*standard*, *rule finite_subset*[*OF _ assms*(*1*)])
(*auto intro*: *PB_invariant.invariant_reaches* ‹${s_0}' \in S$›)
from *run_subsumed*[*OF assms*(*2*)] **obtain** *ys* **where** *ys*:
G.run (${s_0}'$ ## *ys*) *stream_all2* ($\preceq$) *xs ys pred_stream* ($\lambda x.\ x \in S$) *ys* **.**
moreover from ‹*run* _› **have** *pred_stream* (*reaches* s_0) *xs*
using *run_reachable* **by** *blast*
ultimately have *stream_all2* ($\lambda x\ y.\ x \preceq y \wedge reaches\ s_0\ x \wedge y \in S$) *xs ys*
by (*smt stream.pred_set stream.rel_mono_strong*)
with *assms*(*3*) ‹$s_0 \preceq$ _› ‹${s_0}' \in S$› **have** *alw* (*ev* (*holds F*)) (${s_0}'$ ## *ys*)
by (*elim alw_ev_lockstep*) (*erule stream.rel_intros*[*rotated*], *auto*)
from *buechi_run_lasso*[*OF ys*(*1*) *this*] **show** *?thesis*
using *that* **.**
qed

end

end

end

context *Unreachability_Invariant1*
begin

interpretation *inv*: *Graph_Invariant*
by *standard* (*rule P_invariant*)

context
fixes $s_0 :: 'a$
assumes $P\ s_0$
begin

interpretation *E*: *Unreachability_Invariant*
where $S = \{s.\ I\ s\}$
by *standard* (*auto intro*: ‹$P\ s_0$› *inv.invariant_reaches mono S_E_subsumed subsumptions_subsume*)

end

end

locale *Unreachability_Invariant_Contract1* =
  Unreachability_Invariant1 +
  fixes $f :: 'a \Rightarrow nat$ **and** $F :: 'a \Rightarrow bool$
  assumes *f_topo*:
    $I\ a \Longrightarrow E\ a\ b \Longrightarrow SE\ b\ c \Longrightarrow (\mathit{if}\ F\ a\ \mathit{then}\ f\ a < f\ c\ \mathit{else}\ f\ a \leq f\ c)$
  assumes *finite_invariant*: *finite* $\{s.\ I\ s\}$
  assumes *F_mono*[*intro*]: $P\ a \Longrightarrow F\ a \Longrightarrow a \preceq b \Longrightarrow P\ b \Longrightarrow F\ b$
begin

interpretation *c*: *Contract* E'
  using *f_topo*
  apply −
  apply *standard*
  apply (*auto simp*: *E'_def*)
  oops

definition
  $E1 \equiv \lambda a\ c.\ \exists b.\ I\ a \wedge E\ a\ b \wedge SE\ b\ c$

interpretation *c*: *Contract E1*
  using *f_topo* **by** − (*standard*, *auto simp*: *E1_def*)

lemma *no_buechi_run*:
  assumes [*intro*, *simp*]: $P\ s_0\ I\ {s_0}'\ s_0 \preceq {s_0}'$
  assumes *Graph_Defs.run E* $(s_0 \#\# xs)$ *alw* (*ev* (*holds F*)) $(s_0 \#\# xs)$
  shows *False*
proof −
  interpret *sim*: *Simulation* E' *E1* $\lambda s\ s'.\ s' = s \wedge I\ s$
    unfolding *E'_def E1_def* **by** *standard auto*
  obtain *s* **where**
    G.reaches ${s_0}'\ s$ *G.reaches1 s s F s*
    using *finite_invariant assms* **by** − (*rule buechi_run_lasso*[**where** *F*= *F*], *rule assms*, *auto*)
  then have *c.E.reaches1 s s*
    using *E'_def G.reaches1_reaches_iff2* **by** − (*frule sim.simulation_reaches1*, *auto*)
  then have $\neg\ F\ s$
    by (*rule c.no_accepting_cycle*)
  from *this* ‹*F s*› **show** *?thesis* **..**
qed

end

locale *Reachability_Invariant_paired_pre_defs* =
  ord less_eq less **for** *less_eq* :: $'s \Rightarrow 's \Rightarrow bool$ (**infix** $\preceq$ *50*) **and** *less* (**infix** $\prec$ *50*) +
  fixes $E :: ('l \times 's) \Rightarrow ('l \times 's) \Rightarrow bool$

locale *Reachability_Invariant_paired_defs* =
  Reachability_Invariant_paired_pre_defs **where** $E = E$ **for** $E :: ('l \times 's) \Rightarrow ('l \times 's) \Rightarrow bool$ +
  fixes $M :: 'l \Rightarrow 's\ set$
    and $L :: 'l\ set$
begin

definition *covered* $\equiv \lambda\ (l,\ s).\ \exists\ s' \in M\ l.\ s \prec s'$

definition $RE = (\lambda(l,\ s)\ ab.\ s \in M\ l \wedge \neg\ covered\ (l,\ s) \wedge E\ (l,\ s)\ ab)$

end

locale *Unreachability_Invariant_paired_pre_defs* =
Reachability_Invariant_paired_pre_defs _ _ E +
preorder less_eq less **for** $E :: ('l \times 's) \Rightarrow$ _ +
fixes $P :: ('l \times 's) \Rightarrow bool$

locale *Unreachability_Invariant_paired_defs* =
Unreachability_Invariant_paired_pre_defs **where** $E = E$ +
Reachability_Invariant_paired_defs **where** $E = E$
for $E :: ('l \times 's) \Rightarrow$ _

locale *Unreachability_Invariant_paired_pre* =
Unreachability_Invariant_paired_pre_defs +
assumes *mono*:
$s \preceq s' \Longrightarrow E\ (l,\ s)\ (l',\ t) \Longrightarrow P\ (l,\ s) \Longrightarrow P\ (l,\ s') \Longrightarrow \exists\ t'.\ t \preceq t' \wedge E\ (l,\ s')\ (l',\ t')$
assumes *P_invariant*: $P\ (l,\ s) \Longrightarrow E\ (l,\ s)\ (l',\ s') \Longrightarrow P\ (l',\ s')$

locale *Unreachability_Invariant_paired* =
Unreachability_Invariant_paired_pre **where** $E = E$ **and** $P = P$ +
Unreachability_Invariant_paired_defs **where** $M = M$ **and** $L = L$ **and** $E = E$ **and** $P = P$
for $M :: 'l \Rightarrow 's\ set$ **and** $L :: 'l\ set$ **and** $E :: 'l \times 's \Rightarrow$ _ **and** $P :: 'l \times 's \Rightarrow$ _ +
fixes $l_0 :: 'l$ **and** $s_0 :: 's$
fixes $SE :: 'l \times 's \Rightarrow 'l \times 's \Rightarrow bool$
assumes *subsumption_edges_subsume*: $SE\ (l,\ s)\ (l',\ s') \Longrightarrow l' = l \wedge s \preceq s'$
assumes *M_invariant*: $l \in L \Longrightarrow s \in M\ l \Longrightarrow P\ (l,\ s)$
assumes *start*: $l_0 \in L\ \exists s' \in M\ l_0.\ s_0 \preceq s'\ P\ (l_0,\ s_0)$
assumes *closed*:
$\forall\ l \in L.\ \forall\ s \in M\ l.\ \forall\ l'\ s'.\ E\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\ s'' \in M\ l'.\ SE\ (l',\ s')\ (l',\ s''))$
begin

interpretation *Graph_Defs* E **.**

interpretation *inv*: *Graph_Invariant* $E\ P$
by *standard* (*auto intro*: *P_invariant*)

interpretation *unreach1*: *Unreachability_Invariant1*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'\ E$
$\lambda(l,\ s).\ l \in L \wedge s \in M\ l$
supply [*intro*] = *order_trans less_trans less_imp_le*
apply *standard*
using *subsumption_edges_subsume closed*
apply (*all* ‹(*auto intro*: *M_invariant P_invariant dest*: *mono simp*: *less_le_not_le*; *fail*)?›)
using *closed* **by** (*metis* (*mono_tags*, *lifting*) *old.prod.case surj_pair*)

interpretation *Unreachability_Invariant*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'\ E\ (l_0,\ s_0)$
$\{(l,\ s) \mid l\ s.\ l \in L \wedge s \in M\ l\}$
supply [*intro*] = *order_trans less_trans less_imp_le*

apply *standard*
apply (*fastforce intro*: *inv.invariant_reaches*[*OF* _ *start*(*3*)] *M_invariant dest*: *mono*)
subgoal for *s t*
using *closed* **by** (*cases s*; *cases t*) *fastforce*
subgoal
using *subsumption_edges_subsume* **by** *auto*
done

lemma *E_L*:
assumes $l \in L$ $s \in M\ l\ E\ (l, s)\ (l', s')$
shows $l' \in L$
using *assms closed* **by** *simp*

context
fixes *F*
assumes *F_mono*: $\bigwedge a\ b.\ P\ a \Longrightarrow F\ a \Longrightarrow (\lambda(l, s)\ (l', s').\ l' = l \wedge s \preceq s')\ a\ b \Longrightarrow P\ b \Longrightarrow F\ b$
begin

private definition
$s' \equiv SOME\ s.\ s \in M\ l_0 \wedge s_0 \preceq s$

private lemma *s'_correct*:
$s' \in M\ l_0 \wedge s_0 \preceq s'$
using *start*(*2*) **unfolding** *s'_def* **by** − (*rule someI_ex*, *auto*)

private lemma *s'_1*:
$(l_0, s') \in \{(l, s)\ |l\ s.\ l \in L \wedge s \in M\ l\}$
using *s'_correct start* **by** *auto*

private lemma *s'_2*:
(*case* (l_0, s_0) *of* $(l, s) \Rightarrow \lambda(l', s').\ l' = l \wedge s \preceq s')\ (l_0, s')$
using *s'_correct start* **by** *auto*

theorem *final_unreachable*:
assumes $\forall s' \in \{(l, s)\ |l\ s.\ l \in L \wedge s \in M\ l\}.\ \neg\ F\ s'$
shows $\nexists s'.\ (l_0, s_0) \rightarrow* s' \wedge F\ s'$
using *inv.invariant_reaches start*(*3*) *M_invariant F_mono assms*
by − (*simp*, *rule final_unreachable*[*OF s'_2 s'_1*, *simplified*], *blast*+)

lemma *buechi_run_lasso*:
assumes *finite* $\{(l, s)\ |l\ s.\ l \in L \wedge s \in M\ l\}$
assumes *run* $((l_0, s_0)\ \#\#\ xs)$ *alw* (*ev* (*holds F*)) $((l_0, s_0)\ \#\#\ xs)$
obtains $s_0'\ l\ s$ **where**
G.reaches $(l_0, s_0')\ (l, s)$ *G.reaches1* $(l, s)\ (l, s)\ F\ (l, s)\ s_0' \in M\ l_0\ s_0 \preceq s_0'$
using *F_mono assms M_invariant inv.invariant_reaches start*(*3*) *s'_correct*
by − (*rule buechi_run_lasso*[*OF s'_2 s'_1*, *of F*]; *clarsimp*; *blast*)

context
fixes $f :: 'l \times 's \Rightarrow nat$
assumes *finite*: *finite L* $\forall\ l \in L.$ *finite* $(M\ l)$
assumes *f_topo*: $\bigwedge l\ s\ l1\ s1\ l2\ s2.$
$l \in L \Longrightarrow s \in M\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M\ l2 \Longrightarrow E\ (l, s)\ (l1, s1) \Longrightarrow SE\ (l1, s1)\ (l2, s2) \Longrightarrow$

if F (l, s) then f (l, s) < f (l2, s2) else f (l, s) ≤ f (l2, s2)
begin

definition E1 ≡ λ(l, s) (l', s'). ∃l'' s''. l ∈ L ∧ s ∈ M l ∧ E (l, s) (l'', s'') ∧ SE (l'', s'') (l', s') ∧ l' ∈ L ∧ s' ∈ M l' interpretation c: Contract E1 using f_topo by ‹standard, auto›
simp: E1_def lemma no_buechi_run: assumes Graph_Defs.run E ((l₀, s₀) ## xs) alw (ev (holds F)) ((l₀, s₀) ## xs) shows False proof ‹have finite: finite {(l, s). l ∈ L ∧ s ∈ M l}› (is finite ?S) proof ‹have ?S ⊆ L × ⋃ (M ' L)› by auto ‹also from finite have finite› by auto ‹finally show ?thesis› qed interpret sim: Simulation E' E1 λ(l, s) (l', s'). l = l' ∧ s' = s ∧ l ∈ L ∧ s ∈ M l unfolding E'_def E1_def by standard auto obtain s₀' l s where ‹s₀ ⪯ s₀' s₀' ∈ M l₀ G.reaches (l₀, s₀) (l, s) G.reaches1 (l, s) (l, s) F (l, s)› using finite assms by ‹rule buechi_run_lasso, auto› then have c.E.reaches1 (l, s) (l, s) using E'_def G.reaches1_reaches_iff2 by ‹frule sim.simulation_reaches1, auto› then have ‹F (l, s)› by ‹rule c.no_accepting_cycle› from this ‹F (l, s)› show ?thesis .. qed

interpretation *c*: *Contract2*
  where $A = \lambda(l, s)\ (l', s').\ l \in L \wedge s \in M\ l \wedge E\ (l, s)\ (l', s')$
    and $B = SE$
    and $G = \lambda(l, s).\ l \in L \wedge s \in M\ l$
  using *f_topo* **by** − (*standard*, *auto*)

lemma *no_buechi_run*:
  assumes *Graph_Defs.run E* $((l_0, s_0)$ *##* $xs)$ *alw* (*ev* (*holds F*)) $((l_0, s_0)$ *##* $xs)$
  shows *False*
proof −
  have *finite*: *finite* $\{(l, s).\ l \in L \wedge s \in M\ l\}$ (**is** *finite ?S*)
  proof −
    have $?S \subseteq L \times \bigcup\ (M\ `\ L)$
      by *auto*
    also from *finite* **have** *finite* ...
      by *auto*
    finally show *?thesis* **.**
  qed
  interpret *sim*: *Simulation E' c.E* $\lambda(l, s)\ (l', s').\ l = l' \wedge s' = s \wedge l \in L \wedge s \in M\ l$
    unfolding *E'_def c.E_def* **by** *standard auto*
  obtain $s_0'\ l\ s$ **where**
    $s_0 \preceq s_0'\ s_0' \in M\ l_0$ *G.reaches* (l_0, s_0') (l, s) *G.reaches1* (l, s) (l, s) *F* (l, s)
    using *finite assms* **by** − (*rule buechi_run_lasso*, *auto*)
  then have *c.E.reaches1* (l, s) (l, s)
    using *E'_def G.reaches1_reaches_iff2* **by** − (*frule sim.simulation_reaches1*, *auto*)
  then have $\neg\ F\ (l, s)$
    by (*rule c.no_accepting_cycle*)
  from *this* ‹*F* (l, s)› **show** *?thesis* **..**
qed

end

end

end

4.5 Unused

lemma (**in** *Reachability_Compatible_Subsumption_Graph_Final*) *no_accepting_cycleI*:

assumes $\nexists\ x.\ G'.reachable\ x \wedge x \rightarrow_G{}^{+\prime}\ x \wedge F\ x$
shows $\nexists\ x.\ reachable\ x \wedge x \rightarrow^{+}\ x \wedge F\ x$
using *cycle_G'_cycle assms F_mono* **by** *auto*

locale *Reachability_Compatible_Subsumption_Graph_Final2_pre* =
Subsumption_Graph_Defs + *preorder less_eq less* +
fixes $P :: 'a \Rightarrow bool$ **and** $G :: 'a \Rightarrow bool$
assumes *mono*:
$a \preceq b \Longrightarrow E\ a\ a' \Longrightarrow P\ a \Longrightarrow P\ b \Longrightarrow \exists\, b'.\ E\ b\ b' \wedge a' \preceq b'$
assumes *P_invariant*: $P\ a \Longrightarrow E\ a\ b \Longrightarrow P\ b$
assumes *G_P*[*intro*]: $G\ a \Longrightarrow P\ a$
assumes *subgraph*: $\forall\ s\ s'.\ RE\ s\ s' \longrightarrow E\ s\ s'$
assumes *G_finite*: *finite* $\{a.\ G\ a\}$
and *finitely_branching*: $\bigwedge a.\ G\ a \Longrightarrow$ *finite* (*Collect* (*E a*))
assumes *G_start*: $G\ s_0$
fixes $F :: 'a \Rightarrow bool$ — Final states
assumes *F_mono*[*intro*]: $F\ a \Longrightarrow a \preceq b \Longrightarrow F\ b$
assumes *G_anti_symmetric*[*rule_format*]: $\forall\ a\ b.\ G\ a \wedge G\ b \wedge a \preceq b \wedge b \preceq a \longrightarrow a = b$
begin

abbreviation $E\_mix \equiv \lambda\ x\ y.\ RE\ x\ y \wedge G\ x \wedge G\ y \vee E\ x\ y \wedge G\ x \wedge \neg\ G\ y \vee \neg\ G\ x \wedge x \prec y \wedge G\ y$

sublocale *G_mix*: *Graph_Start_Defs E_mix* s_0 **.**

sublocale *P_invariant*: *Graph_Invariant E P*
by *standard* (*auto intro*: *P_invariant*)

sublocale *G_invariant*: *Graph_Invariant E_mix P*
by *standard* (*auto simp*: *subgraph intro*: *P_invariant*)

lemma *mix_reachable_G*[*intro*]:
$P\ x$ **if** *G_mix.reachable x*
proof −
from *G_start* **have** $P\ s_0$
by *auto*
with *that* **show** *?thesis*
by *induction* (*auto simp*: *subgraph intro*: *P_invariant*)
qed

lemma *P_reachable*[*intro*]:
$P\ a$ **if** *reachable a*
using *that G_start* **by** (*auto simp*: *reachable_def intro*: *P_invariant.invariant_reaches*)

lemma *mix_finite_reachable*: *finite* $\{a.\ G\_mix.reachable\ a\}$
proof −
have $G\ x \vee (\exists\ a.\ G\ a \wedge E\ a\ x)$ **if** *G_mix.reachable x* **for** x
using *that G_start* **by** *induction auto*
then have $\{a.\ G\_mix.reachable\ a\} \subseteq Collect\ G \cup \{(a,\ b).\ E\ a\ b\}$ ‘‘ *Collect G*
by *auto*
also have *finite* ...
using *G_finite* **by** (*auto intro*: *finitely_branching*)
finally show *?thesis* **.**

qed

end

locale *Reachability_Compatible_Subsumption_Graph_Final2* =
Reachability_Compatible_Subsumption_Graph_Final2_pre +
assumes *liveness_compatible*:
$\forall\, a\ a'\ b.\ P\ a \wedge G\ a' \wedge a \preceq a' \wedge E\ a\ b$
$\longrightarrow (\exists\, b'.\ b \preceq b' \wedge a' \rightarrow_G b' \wedge G\ b')$
$\vee\ (\exists\, b'\ b''.\ b \preceq b' \wedge b' \prec b'' \wedge E\ a'\ b' \wedge \neg\ G\ b' \wedge G\ b'')$
begin

Setup for automation

context
includes *graph_automation*
begin

lemma *subsumption_step*:
$(\exists\, b'.\ b \preceq b' \wedge a' \rightarrow_G b' \wedge G\ b') \vee (\exists\, b'\ b''.\ b \preceq b' \wedge b' \prec b'' \wedge E\ a'\ b' \wedge \neg\ G\ b' \wedge G\ b'')$
if $P\ a\ E\ a\ b\ G\ a'\ a \preceq a'$
using *liveness_compatible that* **by** *auto*

lemma *subsumption_step'*:
$\exists\ b'.\ b \preceq b' \wedge$ *G_mix.reaches1* $a'\ b' \wedge G\ b'$ **if** $P\ a\ E\ a\ b\ G\ a'\ a \preceq a'$
proof −
from *subsumption_step*[*OF that*] **show** *?thesis*
proof (*safe*, *goal_cases*)
case (*1* b')
then show *?case*
using *that*(*3*) **by** *auto*
next
case (*2* $b'\ b''$)
with ‹$G\ a'$› **have** *E_mix* $b'\ b''$ *E_mix* $a'\ b'$
by *auto*
with ‹$b \preceq b'$› ‹$b' \prec b''$› ‹$G\ b''$› **show** *?case*
including *graph_automation_aggressive* **using** *le_less_trans less_imp_le* **by** *blast*
qed
qed

theorem *reachability_complete'*:
$\exists\ s'.\ s \preceq s' \wedge$ *G_mix.reachable* $s' \wedge G\ s'$ **if** $a \rightarrow* s$ *G_mix.reachable* $a\ G\ a$
using *that*
proof (*induction*)
case *base*
then show *?case* **by** *auto*
next
case (*step* $s\ t$)
then obtain s' **where** $s \preceq s'$ *G_mix.reachable* $s'\ G\ s'$
by *auto*
from ‹$G\ a$› ‹$a \rightarrow* s$› **have** $P\ s$
by (*auto intro*: *P_invariant.invariant_reaches*)
from *subsumption_step*[*OF this* ‹$E\ s\ t$› ‹$G\ s'$› ‹$s \preceq s'$›] **show** *?case*
proof *safe*
fix b' :: ‹$'a$›

```
    assume ‹t ⪯ b'› and ‹s' →_G b'› and ‹G b'›
    with ‹G s'› have E_mix s' b'
      by auto
    with ‹t ⪯ b'› ‹G b'› ‹G_mix.reachable s'› show ‹∃ s'. t ⪯ s' ∧ G_mix.reachable s' ∧ G s'›
      by auto
  next
    fix b' b'' :: ‹'a›
    assume  ‹t ⪯ b'› and ‹b' ≺ b''› and ‹s' → b'› and ‹¬ G b'› and ‹G b''›
    with ‹G s'› have E_mix s' b' E_mix b' b''
      by auto
    with ‹G b''› ‹t ⪯ _› ‹_ ≺ b''› ‹G_mix.reachable s'› show
      ‹∃ s'. t ⪯ s' ∧ G_mix.reachable s' ∧ G s'›
      apply (inst_existentials b'')
      subgoal
        using le_less_trans less_imp_le by auto
      by auto
  qed
qed

lemma steps_G_mix_steps:
  ∃ ys ns. list_all2 (⪯) xs (nths ys ns) ∧ G_mix.steps (b # ys) if
  steps (a # xs) P a a ⪯ b G b
  using that
proof (induction a # xs arbitrary: a b xs)
  case (Single)
  then show ?case by force
next
  case (Cons x y xs)
  from subsumption_step'[OF ‹P x› ‹E x y› _ ‹x ⪯ b›] ‹G b› obtain b' where
    y ⪯ b' G_mix.reaches1 b b' G b'
    by auto
  with ‹P x› Cons.hyps(1) Cons.prems(3) obtain ys ns where
    list_all2 (⪯) xs (nths ys ns) G_mix.steps (b' # ys)
    by atomize_elim
      (blast intro: Cons.hyps(3)[OF _ ‹y ⪯ b'›]
        P_invariant graphI_aggressive G_invariant.invariant_reaches
      )
  from  ‹G_mix.reaches1 b b'› this(2) obtain as where
    G_mix.steps (b # as @ b' # ys)
    by (fastforce intro: G_mix.graphI_aggressive1)
  with ‹y ⪯ b'› show ?case
    apply (inst_existentials as @ b' # ys {length as} ∪ {n + length as + 1 | n. n ∈ ns})
    subgoal
      apply (subst nths_split, force)
      apply (subst nths_nth, (simp; fail))
      apply simp
      apply (subst nths_shift, force)
      subgoal premises prems
      proof −
        have
          {x − length as |x. x ∈ {Suc (n + length as) |n. n ∈ ns}} = {n + 1 | n. n ∈ ns}
          by force
        with ‹list_all2 _ _ _› show ?thesis
          by (simp add: nths_Cons)
```

```
    qed
    done
  by assumption
qed

lemma cycle_G_mix_cycle'':
  assumes steps (s₀ # ws @ x # xs @ [x])
  shows ∃ x' xs' ys'. x ⪯ x' ∧ G_mix.steps (s₀ # xs' @ x' # ys' @ [x'])
proof −
  let ?n = card {x. G_mix.reachable x} + 1
  let ?xs = x # concat (replicate ?n (xs @ [x]))
  from assms(1) have steps (x # xs @ [x])
    by (auto intro: graphI_aggressive2)
  with steps_replicate[of x # xs @ [x] ?n] have steps ?xs
    by auto
  have steps (s₀ # ws @ ?xs)
  proof −
    from assms have steps (s₀ # ws @ [x])
      by (auto intro: graphI_aggressive2)
    with ‹steps ?xs› show ?thesis
      by (fastforce intro: graphI_aggressive1)
  qed
  from steps_G_mix_steps[OF this, of s₀] G_start obtain ys ns where ys:
    list_all2 (⪯) (ws @ x # concat (replicate ?n (xs @ [x]))) (nths ys ns)
    G_mix.steps (s₀ # ys)
    by auto
  then obtain x' ys' ns' ws' where ys':
    G_mix.steps (x' # ys') G_mix.steps (s₀ # ws' @ [x'])
    list_all2 (⪯) (concat (replicate ?n (xs @ [x]))) (nths ys' ns')
    apply atomize_elim
    apply simp
    apply (subst (asm) list_all2_append1)
    apply safe
    apply (subst (asm) list_all2_Cons1)
    apply safe
    apply (drule nths_eq_appendD)
    apply safe
    apply (drule nths_eq_ConsD)
    apply safe
    subgoal for ys1 ys2 z ys3 ys4 ys5 ys6 ys7 i
      apply (inst_existentials z ys7)
      subgoal premises prems
        using prems(1) by (auto intro: G_mix.graphI_aggressive2)
      subgoal premises prems
      proof −
        from prems have G_mix.steps ((s₀ # ys4 @ ys6 @ [z]) @ ys7)
          by auto
        moreover then have G_mix.steps (s₀ # ys4 @ ys6 @ [z])
          by (auto intro: G_mix.graphI_aggressive2)
        ultimately show ?thesis
          by (inst_existentials ys4 @ ys6) auto
      qed
      by force
    done
```

```
  let ?ys = filter ((≼) x) ys'
  have length ?ys ≥ ?n
    using list_all2_replicate_elem_filter[OF ys'(3), of x]
    using filter_nths_length[of ((≼) x) ys' ns']
    by auto
  from ‹G_mix.steps (s₀ # ws' @ [x'])› have G_mix.reachable x'
    by − (rule G_mix.reachable_reaches, auto)
  have set ?ys ⊆ set ys'
    by auto
  also have ... ⊆ {x. G_mix.reachable x}
    using ‹G_mix.steps (x' # _)› ‹G_mix.reachable x'›
    by clarsimp (rule G_mix.reachable_steps_elem[rotated], assumption, auto)
  finally have ¬ distinct ?ys
    using distinct_card[of ?ys] ‹_ >= ?n›
    by − (rule ccontr; drule distinct_length_le[OF mix_finite_reachable]; simp)
  from not_distinct_decomp[OF this] obtain as y bs cs where ?ys = as @ [y] @ bs @ [y] @ cs
    by auto
  then obtain as' bs' cs' where
    ys' = as' @ [y] @ bs' @ [y] @ cs'
    apply atomize_elim
    apply simp
    apply (drule filter_eq_appendD filter_eq_ConsD filter_eq_appendD[OF sym], clarify)+
    apply clarsimp
    subgoal for as1 as2 bs1 bs2 cs'
      by (inst_existentials as1 @ as2 bs1 @ bs2) simp
    done
  have G_mix.steps (y # bs' @ [y])
  proof −
    from ‹G_mix.steps (x' # _)› ‹ys' = _› show ?thesis
      by (force intro: G_mix.graphI_aggressive2)
  qed
  moreover have G_mix.steps (s₀ # ws' @ x' # as' @ [y])
  proof −
    from ‹G_mix.steps (x' # ys')› ‹ys' = _› have G_mix.steps (x' # as' @ [y])
      by (force intro: G_mix.graphI_aggressive2)
    with ‹G_mix.steps (s₀ # ws' @ [x'])› show ?thesis
      by (fastforce intro: G_mix.graphI_aggressive1)
  qed
  moreover from ‹?ys = _› have x ≼ y
  proof −
    from ‹?ys = _› have y ∈ set ?ys by auto
    then show ?thesis by auto
  qed
  ultimately show ?thesis
    by (inst_existentials y ws' @ x' # as' bs'; fastforce intro: G_mix.graphI_aggressive1)
qed

lemma cycle_G_mix_cycle':
  assumes steps (s₀ # ws @ x # xs @ [x])
  shows ∃ y ys. x ≼ y ∧ G_mix.steps (y # ys @ [y]) ∧ G_mix.reachable y
proof −
  from cycle_G_mix_cycle''[OF assms] obtain x' xs' ys' where
    x ≼ x' G_mix.steps (s₀ # xs' @ x' # ys' @ [x'])
    by auto
```

then show *?thesis*
apply (*inst_existentials* x' ys')
subgoal by *assumption*
subgoal by (*auto intro*: *G_mix.graphI_aggressive2*)
by (*rule G_mix.reachable_reaches*, *auto*)
qed

lemma *cycle_G_mix_cycle*:
assumes *reachable x x* $\rightarrow^+$ *x*
shows $\exists$ *y ys. x* $\preceq$ *y* $\wedge$ *G_mix.reachable y* $\wedge$ *G_mix.reaches1 y y*
proof −
from *assms*(*2*) **obtain** *xs* **where** $*$: *steps* (*x* # *xs* @ *x* # *xs* @ [*x*])
by (*fastforce intro*: *graphI_aggressive1*)
from *reachable_steps*[*of x*] *assms*(*1*) **obtain** *ws* **where** *steps ws hd ws* = s_0 *last ws* = *x*
by *auto*
with $*$ **obtain** *us* **where** *steps* (s_0 # (*us* @ *xs*) @ *x* # *xs* @ [*x*])
by (*cases ws*; *force intro*: *graphI_aggressive1*)
from *cycle_G_mix_cycle'*[*OF this*] **show** *?thesis*
by (*auto simp*: *G_mix.steps_reaches1*)
qed

theorem *no_accepting_cycleI*:
assumes $\nexists$ *x. G_mix.reachable x* $\wedge$ *G_mix.reaches1 x x* $\wedge$ *F x*
shows $\nexists$ *x. reachable x* $\wedge$ *x* $\rightarrow^+$ *x* $\wedge$ *F x*
using *cycle_G_mix_cycle assms F_mono* **by** *auto*

end

end

locale *Reachability_Compatible_Subsumption_Graph_Final'_pre* =
Subsumption_Graph_Defs + *preorder less_eq less* +
fixes *P* :: $'a \Rightarrow$ *bool* **and** *G* :: $'a \Rightarrow$ *bool*
assumes *mono*:
a $\preceq$ *b* $\Longrightarrow$ *E a a'* $\Longrightarrow$ *P a* $\Longrightarrow$ *P b* $\Longrightarrow$ $\exists$ *b'. E b b'* $\wedge$ *a'* $\preceq$ *b'*
assumes *P_invariant*: *P a* $\Longrightarrow$ *E a b* $\Longrightarrow$ *P b*
assumes *G_P*[*intro*]: *G a* $\Longrightarrow$ *P a*
assumes *subgraph*: $\forall$ *s s'. RE s s'* $\longrightarrow$ *E s s'*
assumes *G_finite*: *finite* {*a. G a*}
assumes *G_start*: *G* s_0
fixes *F* :: $'a \Rightarrow$ *bool* — Final states
assumes *F_mono*[*intro*]: *F a* $\Longrightarrow$ *a* $\preceq$ *b* $\Longrightarrow$ *F b*
assumes *G_anti_symmetric*[*rule_format*]: $\forall$ *a b. G a* $\wedge$ *G b* $\wedge$ *a* $\preceq$ *b* $\wedge$ *b* $\preceq$ *a* $\longrightarrow$ *a* = *b*
begin

abbreviation *E_mix* $\equiv \lambda$ *x y. RE x y* $\wedge$ *G y* $\vee$ *x* $\prec$ *y* $\wedge$ *G y*

sublocale *G_mix*: *Graph_Start_Defs E_mix* s_0 **.**

sublocale *P_invariant*: *Graph_Invariant E P*
by *standard* (*auto intro*: *P_invariant*)

sublocale *G_invariant*: *Graph_Invariant E_mix G*

by *standard auto*

lemma *mix_reachable_G*[*intro*]:
G x **if** *G_mix.reachable x*
using *that G_start* **by** *induction auto*

lemma *P_reachable*[*intro*]:
P a **if** *reachable a*
using *that G_start* **by** (*auto simp*: *reachable_def intro*: *P_invariant.invariant_reaches*)

lemma *mix_finite_reachable*: *finite* {*a*. *G_mix.reachable a*}
by (*blast intro*: *finite_subset*[*OF _ G_finite*])

end

locale *Reachability_Compatible_Subsumption_Graph_Final''* =
Reachability_Compatible_Subsumption_Graph_Final'_pre +
assumes *liveness_compatible*:
∀ *a a' b*. *P a* ∧ *G a'* ∧ *a* ⪯ *a'* ∧ *E a b*
⟶ (∃ *a'' b'*. *a'* ⪯ *a''* ∧ *b* ⪯ *b'* ∧ *a''* →$_G$ *b'* ∧ *G a''* ∧ *G b'*)
begin

Setup for automation

context
includes *graph_automation*
begin

lemma *subsumption_step*:
∃ *a'' b'*. *a'* ⪯ *a''* ∧ *b* ⪯ *b'* ∧ *a''* →$_G$ *b'* ∧ *G a''* ∧ *G b'* **if**
P a E a b G a' a ⪯ *a'*
using *liveness_compatible that* **by** *auto*

lemma *G_mix_reaches*:
assumes *G a G b a* ⪯ *b*
shows *G_mix.reaches a b*
using *assms* **by** (*cases a* ≺ *b*) (*auto simp*: *G_anti_symmetric less_le_not_le*)

lemma *subsumption_step'*:
∃ *b'*. *b* ⪯ *b'* ∧ *G_mix.reaches1 a' b'* **if** *P a E a b G a' a* ⪯ *a'*
proof −
from *subsumption_step*[*OF that*] **obtain** *a'' b'* **where**
a' ⪯ *a''* *b* ⪯ *b' a''* →$_G$ *b' G a'' G b'*
by *atomize_elim*
with ‹*G a'*› **have** *G_mix.reaches a' a'' E_mix a'' b'*
by (*auto intro*: *G_mix_reaches*)
with ‹*b* ⪯ *b'*› **show** *?thesis*
unfolding *G_mix.reaches1_reaches_iff2* **by** *auto*
qed

theorem *reachability_complete'*:
∃ *s'*. *s* ⪯ *s'* ∧ *G_mix.reachable s'* ∧ *G s'* **if** *a* →∗ *s G_mix.reachable a G a*
using *that*
proof (*induction*)

```
  case base
  then show ?case by auto
next
  case (step s t)
  then obtain s' where s ⪯ s' G_mix.reachable s' G s'
    by auto
  from ‹G a› ‹a →* s› have P s
    by (auto intro: P_invariant.invariant_reaches)
  from subsumption_step[OF this ‹E s t› ‹G s'› ‹s ⪯ s'›] obtain s'' t' where
    s' ⪯ s'' t ⪯ t' s'' →_G t' G s'' G t' by atomize_elim
  with ‹G_mix.reachable s'› show ?case
    using G_mix_reaches by (inst_existentials t') blast+
qed

lemma steps_G_mix_steps:
  ∃ ys ns. list_all2 (⪯) xs (nths ys ns) ∧ G_mix.steps (b # ys) if
  steps (a # xs) P a a ⪯ b G b
  using that
proof (induction a # xs arbitrary: a b xs)
  case (Single)
  then show ?case by force
next
  case (Cons x y xs)
  from subsumption_step'[OF ‹P x› ‹E x y› _ ‹x ⪯ b›] ‹G b› obtain b' where
    y ⪯ b' G_mix.reaches1 b b'
    by auto
  with ‹P x› Cons.hyps(1) Cons.prems(3) obtain ys ns where
    list_all2 (⪯) xs (nths ys ns) G_mix.steps (b' # ys)
    by atomize_elim
      (blast intro: Cons.hyps(3)[OF _ ‹y ⪯ b'›]
        P_invariant graphI_aggressive G_invariant.invariant_reaches
      )
  from ‹G_mix.reaches1 b b'› this(2) obtain as where
    G_mix.steps (b # as @ b' # ys)
    by (fastforce intro: G_mix.graphI_aggressive1)
  with ‹y ⪯ b'› show ?case
    apply (inst_existentials as @ b' # ys {length as} ∪ {n + length as + 1 | n. n ∈ ns})
    subgoal
      apply (subst nths_split, force)
      apply (subst nths_nth, (simp; fail))
      apply simp
      apply (subst nths_shift, force)
      subgoal premises prems
      proof −
        have
          {x − length as |x. x ∈ {Suc (n + length as) |n. n ∈ ns}} = {n + 1 | n. n ∈ ns}
          by force
        with ‹list_all2 _ _ _› show ?thesis
          by (simp add: nths_Cons)
      qed
      done
    by assumption
qed
```

lemma *cycle_G_mix_cycle''*:
assumes *steps* $(s_0 \# ws @ x \# xs @ [x])$
shows $\exists\ x'\ xs'\ ys'.\ x \preceq x' \wedge$ *G_mix.steps* $(s_0 \# xs' @ x' \# ys' @ [x'])$
proof −
let *?n* = *card* $\{x.$ *G_mix.reachable* $x\}$ + *1*
let *?xs* = $x \#$ *concat* (*replicate ?n* $(xs @ [x])$)
from *assms*(*1*) **have** *steps* $(x \# xs @ [x])$
by (*auto intro*: *graphI_aggressive2*)
with *steps_replicate*[*of* $x \# xs @ [x]$ *?n*] **have** *steps ?xs*
by *auto*
have *steps* $(s_0 \# ws @$ *?xs*)
proof −
from *assms* **have** *steps* $(s_0 \# ws @ [x])$
by (*auto intro*: *graphI_aggressive2*)
with ‹*steps ?xs*› **show** *?thesis*
by (*fastforce intro*: *graphI_aggressive1*)
qed
from *steps_G_mix_steps*[*OF this*, *of* s_0] **obtain** *ys ns* **where** *ys*:
list_all2 $(\preceq)$ $(ws @ x \#$ *concat* (*replicate ?n* $(xs @ [x]))) (nths\ ys\ ns)$
G_mix.steps $(s_0 \# ys)$
by *auto*
then obtain $x'\ ys'\ ns'\ ws'$ **where** ys':
G_mix.steps $(x' \# ys')$ *G_mix.steps* $(s_0 \# ws' @ [x'])$
list_all2 $(\preceq)$ (*concat* (*replicate ?n* $(xs @ [x]))) (nths\ ys'\ ns')$
apply *atomize_elim*
apply *simp*
apply (*subst* (*asm*) *list_all2_append1*)
apply *safe*
apply (*subst* (*asm*) *list_all2_Cons1*)
apply *safe*
apply (*drule nths_eq_appendD*)
apply *safe*
apply (*drule nths_eq_ConsD*)
apply *safe*
subgoal for *ys1 ys2 z ys3 ys4 ys5 ys6 ys7 i*
apply (*inst_existentials z ys7*)
subgoal premises *prems*
using *prems*(*1*) **by** (*auto intro*: *G_mix.graphI_aggressive2*)
subgoal premises *prems*
proof −
from *prems* **have** *G_mix.steps* $((s_0 \#$ *ys4* $@$ *ys6* $@ [z]) @$ *ys7*)
by *auto*
moreover then have *G_mix.steps* $(s_0 \#$ *ys4* $@$ *ys6* $@ [z])$
by (*auto intro*: *G_mix.graphI_aggressive2*)
ultimately show *?thesis*
by (*inst_existentials ys4* $@$ *ys6*) *auto*
qed
by *force*
done
let *?ys* = *filter* $((\preceq)\ x)\ ys'$
have *length ?ys* $\geq$ *?n*
using *list_all2_replicate_elem_filter*[*OF ys'*(*3*), *of x*]
using *filter_nths_length*[*of* $((\preceq)\ x)\ ys'\ ns'$]
by *auto*

```
  from ‹G_mix.steps (s₀ # ws' @ [x'])› have G_mix.reachable x'
    by − (rule G_mix.reachable_reaches, auto)
  have set ?ys ⊆ set ys'
    by auto
  also have ... ⊆ {x. G_mix.reachable x}
    using ‹G_mix.steps (x' # _)› ‹G_mix.reachable x'›
    by clarsimp (rule G_mix.reachable_steps_elem[rotated], assumption, auto)
  finally have ¬ distinct ?ys
    using distinct_card[of ?ys] ‹_ >= ?n›
    by − (rule ccontr; drule distinct_length_le[OF mix_finite_reachable]; simp)
  from not_distinct_decomp[OF this] obtain as y bs cs where ?ys = as @ [y] @ bs @ [y] @ cs
    by auto
  then obtain as' bs' cs' where
    ys' = as' @ [y] @ bs' @ [y] @ cs'
    apply atomize_elim
    apply simp
    apply (drule filter_eq_appendD filter_eq_ConsD filter_eq_appendD[OF sym], clarify)+
    apply clarsimp
    subgoal for as1 as2 bs1 bs2 cs'
      by (inst_existentials as1 @ as2 bs1 @ bs2) simp
    done
  have G_mix.steps (y # bs' @ [y])
  proof −
    from ‹G_mix.steps (x' # _)› ‹ys' = _› show ?thesis
      by (force intro: G_mix.graphI_aggressive2)
  qed
  moreover have G_mix.steps (s₀ # ws' @ x' # as' @ [y])
  proof −
    from ‹G_mix.steps (x' # ys')› ‹ys' = _› have G_mix.steps (x' # as' @ [y])
      by (force intro: G_mix.graphI_aggressive2)
    with ‹G_mix.steps (s₀ # ws' @ [x'])› show ?thesis
      by (fastforce intro: G_mix.graphI_aggressive1)
  qed
  moreover from ‹?ys = _› have x ≼ y
  proof −
    from ‹?ys = _› have y ∈ set ?ys by auto
    then show ?thesis by auto
  qed
  ultimately show ?thesis
    by (inst_existentials y ws' @ x' # as' bs'; fastforce intro: G_mix.graphI_aggressive1)
qed

lemma cycle_G_mix_cycle':
  assumes steps (s₀ # ws @ x # xs @ [x])
  shows ∃ y ys. x ≼ y ∧ G_mix.steps (y # ys @ [y]) ∧ G_mix.reachable y
proof −
  from cycle_G_mix_cycle''[OF assms] obtain x' xs' ys' where
    x ≼ x' G_mix.steps (s₀ # xs' @ x' # ys' @ [x'])
    by auto
  then show ?thesis
    apply (inst_existentials x' ys')
    subgoal by assumption
    subgoal by (auto intro: G_mix.graphI_aggressive2)
    by (rule G_mix.reachable_reaches, auto)
```

qed

lemma *cycle_G_mix_cycle*:
assumes *reachable x x* $\rightarrow^+$ *x*
shows $\exists\, y.\ x \preceq y \wedge$ *G_mix.reachable y* $\wedge$ *G_mix.reaches1 y y*
proof −
from *assms*(*2*) **obtain** *xs* **where** $*$: *steps* (*x* # *xs* @ *x* # *xs* @ [*x*])
by (*fastforce intro*: *graphI_aggressive1*)
from *reachable_steps*[*of x*] *assms*(*1*) **obtain** *ws* **where** *steps ws hd ws* = s_0 *last ws* = *x*
by *auto*
with $*$ **obtain** *us* **where** *steps* (s_0 # (*us* @ *xs*) @ *x* # *xs* @ [*x*])
by (*cases ws*; *force intro*: *graphI_aggressive1*)
from *cycle_G_mix_cycle*′[*OF this*] **show** *?thesis*
by (*auto simp*: *G_mix.steps_reaches1*)
qed

end

end

locale *Reachability_Compatible_Subsumption_Graph_Final*′ =
Reachability_Compatible_Subsumption_Graph_Final′*_pre* +
assumes *liveness_compatible*:
$\forall\ s.\ G\ s \longrightarrow (\forall\ s'.\ E\ s\ s' \longrightarrow RE\ s\ s' \wedge G\ s') \vee (\exists\ t.\ s \prec t \wedge G\ t)$
begin

Setup for automation

context
includes *graph_automation*
begin

lemmas *preorder_intros* = *order_trans less_trans less_imp_le*

lemma *reachable_has_surrogate*:
$\exists\ t.\ G\ t \wedge s \preceq t \wedge (\forall\ s'.\ E\ t\ s' \longrightarrow RE\ t\ s' \wedge G\ s')$ **if** *G s*
proof −
note [*intro*] = *preorder_intros*
from *G_finite* ‹*G s*› **obtain** *x* **where**
$\forall\, s'.\ E\ x\ s' \longrightarrow RE\ x\ s' \wedge G\ s'\ G\ x\ ((\prec)^{**})\ s\ x$
by *atomize_elim* (*induction rule*: *rtranclp_ev_induct2*, *auto simp*: *liveness_compatible*)
moreover from ‹$((\prec)^{**})$ *s x*› **have** $s \prec x \vee s = x$
by *induction auto*
ultimately show *?thesis*
by *auto*
qed

lemma *subsumption_step*:
$\exists\ a''\ b'.\ a' \preceq a'' \wedge b \preceq b' \wedge a'' \rightarrow_G b' \wedge G\ a'' \wedge G\ b'$ **if**
P a E a b G a′ *a* $\preceq$ *a*′
proof −
note [*intro*] = *preorder_intros*
from *mono*[*OF* ‹*a* $\preceq$ *a*′› ‹*E a b*›] ‹*P a*› ‹*G a*′› **obtain** *b*′ **where** *E a*′ *b*′ *b* $\preceq$ *b*′
by *auto*

from *reachable_has_surrogate*[*OF* ‹*G a'*›] **obtain** *a''*
where $a' \preceq a''$ *G a''* **and** $*$: $\forall$ *s'*. *E a'' s'* $\longrightarrow$ *RE a'' s'* $\wedge$ *G s'*
by *auto*
from *mono*[*OF* ‹$a' \preceq a''$› ‹*E a' b'*›] ‹*G a'*› ‹*G a''*› **obtain** *b''* **where**
E a'' b'' b' $\preceq$ *b''*
by *auto*
with $*$ ‹$a' \preceq a''$› ‹$b \preceq b'$› ‹*G a''*› **show** *?thesis*
by *auto*
qed

end

sublocale *Reachability_Compatible_Subsumption_Graph_Final''*
using *subsumption_step* **by** − (*standard*, *auto*)

theorem *no_accepting_cycleI*:
assumes $\nexists$ *x*. *G_mix.reachable x* $\wedge$ *G_mix.reaches1 x x* $\wedge$ *F x*
shows $\nexists$ *x*. *reachable x* $\wedge$ *x* $\rightarrow^+$ *x* $\wedge$ *F x*
using *cycle_G_mix_cycle assms F_mono* **by** *auto*

end

locale *Reachability_Compatible_Subsumption_Graph_Final1* =
Reachability_Compatible_Subsumption_Graph_Final'_pre +
assumes *reachable_has_surrogate*:
$\forall$ *s*. *G s* $\longrightarrow$ ($\exists$ *t*. *G t* $\wedge$ *s* $\preceq$ *t* $\wedge$ ($\forall$ *s'*. *E t s'* $\longrightarrow$ *RE t s'* $\wedge$ *G s'*))
begin

Setup for automation

context
includes *graph_automation*
begin

lemmas *preorder_intros* = *order_trans less_trans less_imp_le*

lemma *subsumption_step*:
$\exists$ *a'' b'*. $a' \preceq a'' \wedge b \preceq b' \wedge a'' \rightarrow_G b' \wedge$ *G a'* $\wedge$ *G b'* **if**
P a E a b G a' $a \preceq a'$
proof −
note [*intro*] = *preorder_intros*
from *mono*[*OF* ‹$a \preceq a'$› ‹*E a b*›] ‹*P a*› ‹*G a'*› **obtain** *b'* **where** *E a' b' b* $\preceq$ *b'*
by *auto*
from *reachable_has_surrogate* ‹*G a'*› **obtain** *a''*
where $a' \preceq a''$ *G a''* **and** $*$: $\forall$ *s'*. *E a'' s'* $\longrightarrow$ *RE a'' s'* $\wedge$ *G s'*
by *auto*
from *mono*[*OF* ‹$a' \preceq a''$› ‹*E a' b'*›] ‹*G a'*› ‹*G a''*› **obtain** *b''* **where**
E a'' b'' b' $\preceq$ *b''*
by *auto*
with $*$ ‹$a' \preceq a''$› ‹$b \preceq b'$› ‹*G a''*› **show** *?thesis*
by *auto*
qed

end

sublocale *Reachability_Compatible_Subsumption_Graph_Final''*
using *subsumption_step* **by** − (*standard*, *auto*)

theorem *no_accepting_cycleI*:
assumes $\nexists\ x.\ G\_mix.reachable\ x \wedge G\_mix.reaches1\ x\ x \wedge F\ x$
shows $\nexists\ x.\ reachable\ x \wedge x \rightarrow^{+} x \wedge F\ x$
using *cycle_G_mix_cycle assms F_mono* **by** *auto*

end

4.6 Instantiating Reachability Compatible Subsumption Graphs

locale *Reachability_Invariant_paired* =
Reachability_Invariant_paired_defs **where** $M = M$ +
preorder: *preorder less_eq less* **for** $M :: 'l \Rightarrow 's\ set$ +
fixes $l_0 :: 'l$ **and** $s_0 :: 's$
assumes E_T: $\forall\ l\ s\ l'\ s'.\ E\ (l,\ s)\ (l',\ s') \longleftrightarrow (\exists\ f.\ (l',\ f) \in T\ l \wedge s' = f\ s)$
assumes *mono*:
$s \preceq s' \Longrightarrow E\ (l,\ s)\ (l',\ t) \Longrightarrow E^{**}\ (l_0,\ s_0)\ (l,\ s) \Longrightarrow E^{**}\ (l_0,\ s_0)\ (l,\ s')$
$\Longrightarrow \exists\ t'.\ t \preceq t' \wedge E\ (l,\ s')\ (l',\ t')$
assumes *finitely_branching*: *finite* (*Collect* ($E\ (a,\ b)$))
assumes *start*:$l_0 \in L\ s_0 \in S\ s_0 \in M(l_0)$
assumes *closed*:
$\forall\ l \in L.\ \forall\ (l',\ f) \in T\ l.\ l' \in L \wedge (\forall\ s \in M\ l.\ (\exists\ s' \in M\ l'.\ f\ s \prec s') \vee f\ s \in M\ l')$
assumes *reachable*:
$\forall\ l \in L.\ \forall\ s \in M\ l.\ RE^{**}\ (l_0,\ s_0)\ (l,\ s)$
assumes *finite*:
finite $L\ \forall\ l \in L.$ *finite* ($M\ l$)
begin

lemma *invariant*:
$((\exists\ s' \in M\ l.\ s \prec s') \vee s \in M\ l) \wedge l \in L$ **if** $RE^{**}\ (l_0,\ s_0)\ (l,\ s)$
using *that start closed* **by** (*induction rule*: *rtranclp_induct2*) (*auto 4 3 simp*: *RE_def*)

interpretation *Reachability_Compatible_Subsumption_Graph_View*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'$
$E\ (l_0,\ s_0)\ RE$
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'$ *covered*
supply [*intro*] = *order_trans less_trans less_imp_le*
apply *standard*
apply (*solves* ‹*auto simp*: *preorder.less_le_not_le*›)
apply (*solves* ‹*auto simp*: *preorder.less_le_not_le*›)
apply (*solves* ‹*auto simp*: *preorder.order.trans*›)
apply *clarsimp*
apply (*drule mono*, *assumption*+, (*simp add*: *Graph_Start_Defs.reachable_def*; *fail*)+)
apply *blast*
apply (*clarsimp simp*: *Graph_Start_Defs.reachable_def*; *safe*)
subgoal for $l\ s$
apply (*drule invariant*)
using *reachable* **unfolding** *covered_def RE_def* **by** *fastforce*
subgoal for $l\ s\ l'\ s'$
apply (*drule invariant*)
unfolding *RE_def covered_def* **using** E_T **by** *force*

```
  subgoal
    using E_T by force
  subgoal
    unfolding RE_def using E_T by force
  subgoal
    unfolding Graph_Start_Defs.reachable_def
  proof −
    have 1: finite {(l, s). l ∈ L ∧ s ∈ M l}
    proof −
      let ?S = {(l, s). l ∈ L ∧ s ∈ M l}
      have ?S ⊆ (⋃l∈L. ((λ s. (l, s)) ‘ M l))
        by auto
      also have finite ...
        using finite by auto
      finally show finite ?S .
    qed
    have 2: finite (Collect (E (l, s))) for l s
      by (rule finitely_branching)
    let ?S = {a. RE** (l0, s0) a}
    have ?S ⊆ {(l0, s0)} ∪ (⋃ ((λ a. {b. E a b}) ‘ {(l, s). l ∈ L ∧ s ∈ M l}))
      using invariant by (auto simp: RE_def elim: rtranclp.cases)
    also have finite ...
      using 1 2 by auto
    finally show finite ?S .
  qed
  done

context
  assumes no_subsumption_cycle: G'.reachable x ⟹ x →G+' x ⟹ x →G+ x
  fixes F :: 'l × 's ⇒ bool — Final states
  assumes F_mono[intro]: F (l, a) ⟹ a ⪯ b ⟹ F (l, b)
begin

interpretation Liveness_Compatible_Subsumption_Graph
  λ (l, s) (l', s'). l' = l ∧ s ⪯ s' λ (l, s) (l', s'). l' = l ∧ s ≺ s'
  E (l0, s0) RE F
  by standard (auto intro: no_subsumption_cycle)

lemma
  ∄ x. reachable x ∧ F x ∧ x →+ x if ∄ x. G.reachable x ∧ F x ∧ x →G+ x
  using cycle_iff that by auto

lemma no_reachable_cycleI:
  ∄ x. reachable x ∧ F x ∧ x →+ x if ∄ x. G'.reachable x ∧ F x ∧ x →G+ x
  using that cycle_iff unfolding G_G'_reachable_iff[symmetric] by auto
```

We can certify this by accepting a set of components and checking that:

- there are no cycles in the component graph (including subsumption edges)
- no non-trivial component contains a final state
- no component contains an internal subsumption edge

```
end
```

end

4.7 Certifying Cycle-Freeness with Graph Components

context
  fixes *E1 E2* :: $'a \Rightarrow 'a \Rightarrow bool$
  fixes S :: $'a\ set\ set$
  fixes s_0 :: $'a$ **and** a_0 :: $'a\ set$
  assumes *start*: $s_0 \in a_0\ a_0 \in S$
  assumes *closed*:
    $\forall\ a \in S.\ \forall\ x \in a.\ \forall\ y.\ E1\ x\ y \longrightarrow (\exists\ b \in S.\ y \in b)$
    $\forall\ a \in S.\ \forall\ x \in a.\ \forall\ y.\ E2\ x\ y \longrightarrow (\exists\ b \in S.\ y \in b)$
  assumes *finite*: $\forall\ a \in S.\ finite\ a$
begin

definition $E \equiv \lambda\ x\ y.\ E1\ x\ y \vee E2\ x\ y$

definition $C \equiv \lambda\ a\ b.\ \exists\ x \in a.\ \exists\ y \in b.\ E\ x\ y \wedge b \in S$

interpretation *E1*: *Graph_Start_Defs E1* s_0 .
interpretation *E2*: *Graph_Start_Defs E2* s_0 .
interpretation *E*: *Graph_Start_Defs E* s_0 .
interpretation *C*: *Graph_Start_Defs C* a_0 .

lemma *E_closed*:
  $\forall\ a \in S.\ \forall\ x \in a.\ \forall\ y.\ E\ x\ y \longrightarrow (\exists\ b \in S.\ y \in b)$
  using *closed* **unfolding** *E_def* **by** *auto*

interpretation *E_invariant*: *Graph_Invariant E* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
  using *E_closed* **by** − (*standard*, *auto*)

interpretation *E1_invariant*: *Graph_Invariant E1* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
  using *closed* **by** − (*standard*, *auto*)

interpretation *E2_invariant*: *Graph_Invariant E2* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
  using *closed* **by** − (*standard*, *auto*)

interpretation *C_invariant*: *Graph_Invariant C* $\lambda\ a.\ a \in S \wedge a \neq \{\}$
  unfolding *C_def* **by** *standard auto*

interpretation *Simulation_Invariant E C* $(\in)$ $\lambda\ x.\ \exists\ a \in S.\ x \in a\ \lambda\ a.\ a \in S \wedge a \neq \{\}$
  unfolding *C_def* **by** (*standard*; *blast dest*: *E_invariant.invariant*[*rotated*])+

interpretation *Subgraph E E1*
  unfolding *E_def* **by** *standard auto*

context
  assumes *no_internal_E2*: $\forall\ a \in S.\ \forall\ x \in a.\ \forall\ y \in a.\ \neg\ E2\ x\ y$
    and *no_component_cycle*: $\forall\ a \in S.\ \nexists\ b.\ (C.reachable\ a \wedge C\ a\ b \wedge C.reaches\ b\ a \wedge a \neq b)$
begin

lemma *E_C_reaches1*:
  C.reaches1 a b **if** *E.reaches1 x y x* $\in$ *a a* $\in$ *S y* $\in$ *b b* $\in$ *S*

```
  using that
proof (induction arbitrary: b)
  case (base y)
  then have C a b
    unfolding C_def by auto
  then show ?case
    by auto
next
  case (step y z c)
  then obtain b where y ∈ b b ∈ S
    by (meson E_invariant.invariant_reaches tranclp_into_rtranclp)
  from step.IH[OF ‹x ∈ a› ‹a ∈ S› this] have C.reaches1 a b .
  moreover from step.prems ‹E _ _› ‹y ∈ b› ‹b ∈ S› have C b c
    unfolding C_def by auto
  finally show ?case .
qed

lemma certify_no_E1_cycle:
  assumes E.reachable x E.reaches1 x x
    shows E1.reaches1 x x
proof (rule ccontr)
  assume A: ¬ E1.reaches1 x x
  with ‹E.reaches1 x x› obtain y z where E.reaches x y E2 y z E.reaches z x
    by (fastforce dest!: non_subgraph_cycle_decomp simp: E_def)
  from start ‹E.reachable x› obtain a where [intro]: C.reachable a x ∈ a a ∈ S
    unfolding E.reachable_def C.reachable_def by (auto dest: simulation_reaches)
  with ‹E.reaches x y› obtain b where C.reaches a b y ∈ b b ∈ S
    by (auto dest: simulation_reaches)
  from ‹C.reaches a b› have C.reachable b
    by (blast intro: C.reachable_reaches)
  from ‹E.reaches z x› have ‹E.reaches1 z x ∨ z = x›
    by (meson rtranclpD)
  then show False
  proof
    assume E.reaches1 z x
    from ‹y ∈ b› ‹b ∈ S› ‹E2 y z› obtain c where C b c ‹z ∈ c› ‹c ∈ S›
      using A_B_step[of y z b] unfolding E_def by (auto dest: C_invariant.invariant[rotated])
    with no_internal_E2 ‹y ∈ _› ‹E2 y z› have b ≠ c
      by auto
    with ‹E2 y z› ‹y ∈ b› ‹z ∈ c› ‹c ∈ S› have C b c
      unfolding C_def E_def by auto
    from ‹E.reaches1 z x› ‹z ∈ c› ‹c ∈ S› ‹x ∈ a› have C.reaches1 c a
      by (auto intro: E_C_reaches1)
    with ‹C.reaches a b› have C.reaches1 c b
      by auto
    then have C.reaches c b
      by auto
    with ‹C b c› ‹b ∈ S› ‹C.reachable b› ‹b ≠ c› no_component_cycle show False
      by auto
  next
    assume [simp]: z = x
    with ‹y ∈ b› ‹E2 y z› ‹a ∈ S› ‹x ∈ a› have C b a
      unfolding C_def E_def by auto
    with no_internal_E2 ‹y ∈ _› ‹E2 y z› have b ≠ a
```

```
    by auto
  with ‹C.reaches a b› ‹C b a› ‹b ∈ S› ‹C.reachable b› no_component_cycle show False
    by auto
 qed
qed

lemma certify_no_accepting_cycle:
 assumes
  ∀ a ∈ S. card a > 1 ⟶ (∀ x ∈ a. ¬ F x)
  ∀ a ∈ S. ∀ x. a = {x} ⟶ (¬ F x ∨ ¬ E1 x x)
 assumes E.reachable x E1.reaches1 x x
 shows ¬ F x
proof (rule ccontr, unfold not_not)
 assume F x
 from start ‹E.reachable x› obtain a where x ∈ a a ∈ S C.reachable a
  unfolding E.reachable_def C.reachable_def by (auto dest: simulation_reaches)
 consider card a = 0 | card a = 1 | card a > 1
  by force
 then show False
 proof cases
  assume card a = 0
  with finite ‹x ∈ a› ‹a ∈ S› show False
   by auto
 next
  assume card a > 1
  with ‹x ∈ a› ‹a ∈ S› assms(1) ‹F x› show False
   by auto
 next
  assume card a = 1
  with finite ‹x ∈ a› ‹a ∈ S› have a = {x}
   using card_1_singletonE by blast
  with ‹a ∈ S› assms(2) ‹F x› have ¬ E1 x x
   by auto
  from assms(4) show False
  proof (cases rule: converse_tranclpE, assumption, goal_cases)
   case 1
   with ‹¬ E1 x x› show ?thesis
    by auto
  next
   case (2 y)
   interpret sim: Simulation E1 E (=)
    by (rule Subgraph_Simulation)
   from ‹E1.reaches1 y x› have E.reaches1 y x
    by (auto dest: sim.simulation_reaches1)
   from ‹E1 x y› ‹¬ E1 x x› ‹a = _› ‹x ∈ a› ‹a ∈ S› obtain b where y ∈ b b ∈ S C a b
    by (meson C_def E_def closed(1))
   with ‹a = _› ‹¬ E1 x x› ‹E1 x y› have a ≠ b
    by auto
   from ‹y ∈ b› ‹b ∈ S› ‹E.reaches1 y x› ‹x ∈ a› ‹a ∈ S› have C.reaches1 b a
    by (auto intro: E_C_reaches1)
   with ‹x ∈ a› ‹a ∈ S› ‹C a b› ‹C.reachable a› ‹a ≠ b› show ?thesis
    including graph_automation_aggressive using no_component_cycle by auto
  qed
 qed
```

qed

end

end

end

5 Certificates for (Un)Reachability Properties

theory *Unreachability_Misc*
imports
Simulation_Graphs_Certification
Worklist_Algorithms.Leadsto_Impl
Munta_Base.Printing
Difference_Bound_Matrices.DBM_Imperative_Loops
Munta_Base.Trace_Timing
begin

Misc **theorem** (**in** −) *arg_max_nat_lemma2*:
fixes $f :: 'a \Rightarrow nat$
assumes $P\ k$
and *finite* (*Collect P*)
shows $P\ (arg\_max\ f\ P) \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ (arg\_max\ f\ P))$
proof −
let *?b* = *Max* (*f* ‘ *Collect P*) + *1*
from *assms*(*2*) **have** $\forall y.\ P\ y \longrightarrow f\ y < ?b$
by (*auto intro*: *Max_ge le_imp_less_Suc*)
with *assms*(*1*) **show** *?thesis*
by (*rule arg_max_nat_lemma*)
qed

Misc *heap* **lemma** *hoare_triple_success*:
assumes <*P*> *c* <*Q*> **and** $(h,\ as) \models P$
shows *success c h*
using *assms* **unfolding** *hoare_triple_def Let_def success_def*
by (*cases execute c h*) (*force simp*: *run.simps*)+

lemma *run_return*: *run* (*return x*) (*Some h*) (*Some h*) *x* **for** *h*
by (*auto simp*: *execute_simps intro*: *run.regular*)

lemma *return_htD*:
assumes <*Q x*> *return x* <*PP*>
shows $Q\ x \Longrightarrow_A PP\ x$
using *assms* **unfolding** *hoare_triple_def Let_def* **by** (*force intro*: *run_return entailsI*)

definition *run_heap* :: *'a Heap* ⇒ *'a* **where**
run_heap h = *fst* (*the* (*execute h Heap.empty*))

code_printing constant *run_heap* ⇀ (*SML*) (*fn f => f* ()) _
code_printing constant *run_heap* ⇀ (*OCaml*) (*fun f -> f* ()) _

definition *run_map_heap* :: (*'a* ⇒ *'b Heap*) ⇒ *'a list* ⇒ *'b list* **where**

run_map_heap f xs = map (run_heap o f) xs

lemma *hoare_triple_executeD*:
assumes <*emp*> *c* <λ*r*. ↑(*P r*)>$_t$
shows *P* (*fst* (*the* (*execute c h*)))
proof −
have (*h*, {}) ⊨ *emp*
by *simp*
with *assms*(*1*) **have** *success c h*
by (*rule hoare_triple_success*)
then obtain *r h′* **where** *execute c h* = *Some* (*r*, *h′*)
unfolding *success_def* **by** *auto*
then have *run c* (*Some h*) (*Some h′*) *r*
by (*intro regular*) *auto*
with ‹*execute c h* = _› **show** *?thesis*
using *assms* **unfolding** *hoare_triple_def* **by** (*force intro*: *mod_emp_simp*)
qed

lemma *hoare_triple_run_heapD*:
assumes <*emp*> *c* <λ*r*. ↑(*P r*)>$_t$
shows *P* (*run_heap c*)
using *hoare_triple_executeD*[*OF assms*] **unfolding** *run_heap_def* **.**

lemma *list_all2_conjI*:
assumes *list_all2 P as bs list_all2 Q as bs*
shows *list_all2* (λ*a b*. *P a b* ∧ *Q a b*) *as bs*
using *assms* **unfolding** *list_all2_conv_all_nth* **by** *auto*

lemma *hoare_triple_run_map_heapD*:
assumes *list_all* (λ*x*. <*emp*> *c x* <λ*r*. ↑(*P x r*)>$_t$) *xs*
shows *list_all2 P xs* (*run_map_heap c xs*)
using *assms* **unfolding** *run_map_heap_def list_all2_map2 list.pred_rel*
by (*elim list_all2_mono*) (*auto simp*: *eq_onp_def intro*: *hoare_triple_run_heapD*)

lemma *hoare_triple_run_map_heapD′*:
assumes *list_all2* (λ*x xi*. <*emp*> *c xi* <λ*r*. ↑(*P x r*)>$_t$) *xs xsi*
shows *list_all2 P xs* (*run_map_heap c xsi*)
using *assms* **unfolding** *run_map_heap_def list_all2_map2 list.pred_rel*
by (*elim list_all2_mono*) (*auto simp*: *eq_onp_def intro*: *hoare_triple_run_heapD*)

definition
parallel_fold_map = *Heap_Monad.fold_map*

Misc *nres* **lemma** *no_fail_RES_bindI*:
assumes ⋀*x*. *x* ∈ *S* ⟹ *nofail* (*f x*)
shows *nofail* (*RES S* ⤜ *f*)
using *assms pw_RES_bind_choose*(*1*) **by** *blast*

lemma *nfoldli_ub_RES_rule*:
assumes ⋀*x s*. *x* ∈ *set xs* ⟹ *s* ∈ *S* ⟹ *f x s* ≤ *RES S s* ∈ *S*
shows *nfoldli xs c f s* ≤ *RES S*
using *assms*
by (*induction xs arbitrary*: *s*; *simp*; *metis* (*no_types*) *inres_simps*(*2*) *pw_bind_le_iff pw_le_iff*)

lemma *nfoldli_ub_rule*:
assumes $\bigwedge x\ s.\ x \in set\ xs \Longrightarrow inres\ ub\ s \Longrightarrow f\ x\ s \leq ub\ inres\ ub\ s$
shows *nfoldli xs c f s* $\leq$ *ub*
using *nfoldli_ub_RES_rule assms* **by** (*metis inres_def nofail_RES_conv nres_order_simps*(*21*) *pw_leI'*)

lemma *nfoldli_nofail_rule*:
assumes $\bigwedge x\ s.\ x \in set\ xs \Longrightarrow inres\ ub\ s \Longrightarrow f\ x\ s \leq ub\ inres\ ub\ s\ nofail\ ub$
shows *nofail* (*nfoldli xs c f s*)
using *assms* **by** − (*erule pwD1*[*rotated*], *rule nfoldli_ub_rule*)

lemma *SUCCEED_lt_RES_iff*[*simp*]:
SUCCEED $<$ *RES S* $\longleftrightarrow$ $S \neq \{\}$
unfolding *bot_nres_def* **by** (*subst less_nres.simps*) *auto*

lemma *SUCCEED_lt_RETURN*[*intro*, *simp*]:
SUCCEED $<$ *RETURN x*
unfolding *RETURN_def* **by** *simp*

lemma *SUCCEED_lt_FAIL*[*intro*, *simp*]:
SUCCEED $<$ *FAIL*
unfolding *bot_nres_def top_nres_def* **by** (*subst less_nres.simps*) *auto*

lemma *bind_RES_gt_SUCCEED_I*:
assumes $\bigwedge s.\ f\ s > SUCCEED\ S \neq \{\}$
shows *do* $\{x \leftarrow RES\ S; f\ x\} > SUCCEED$
by (*metis RES_bind_choose assms*(*1*) *assms*(*2*) *le_less preorder_class.less_le_not_le set_notEmptyE*)

Monadic *list_all* **and** *list_ex* **definition**
monadic_list_all P xs ≡ *nfoldli xs id* (*λx _. P x*) *True*

Debug version

definition
monadic_list_all_fail P xs ≡
nfoldli xs (*λx. x = None*) (*λx _. do* {*b* ← *P x*; *RETURN* (*if b then None else Some x*)})
None

lemma *monadic_list_all_fail_alt_def*:
monadic_list_all_fail P xs =
nfoldli xs (*λx. x = None*) (*λx _. do* {
b ← *P* (*COPY x*); *if b then RETURN None else RETURN* (*Some x*)}) *None*
unfolding *monadic_list_all_fail_def*
apply (*intro arg_cong2*[**where** *f* = *nfoldli xs* (*λx. x = None*)] *ext*)
apply *simp*
apply (*rule bind_cong*)
apply *auto*
done

definition
monadic_list_all_fail′ P xs ≡
nfoldli xs (*λx. x = None*) (*λx _. do* {
r ← *P x*; *RETURN* (*case r of None* ⇒ *None* | *Some r* ⇒ *Some r*)})
None

lemma *monadic_list_all_fail'_alt_def*:
monadic_list_all_fail' P xs =
nfoldli xs (λ*x*. *x* = *None*) (λ*x* _. *do* {
r ← *P x*; *case r of None* ⇒ *RETURN None* | *Some r* ⇒ *RETURN* (*Some r*)})
None
unfolding *monadic_list_all_fail'_def*
apply (*intro arg_cong2*[**where** *f* = *nfoldli xs* (λ*x*. *x* = *None*)] *ext*)
apply *simp*
apply (*rule bind_cong*)
apply (*auto split*: *option.splits*)
done

lemma *monadic_list_all_fail_monadic_list_all_fail'*:
monadic_list_all_fail P xs =
monadic_list_all_fail' (λ*x*. *do* {*b* ← *P x*; *RETURN* (*if b then None else Some x*)}) *xs*
unfolding *monadic_list_all_fail_def monadic_list_all_fail'_def*
apply (*intro arg_cong2*[**where** *f* = *nfoldli xs* (λ*x*. *x* = *None*)] *ext*)
apply *simp*
apply (*rule bind_cong*)
apply *auto*
done

lemma *monadic_list_all_rule*:
assumes ⋀*x*. *Pi x* ≤ *SPEC* (λ*r*. *r* = *P x*)
shows *monadic_list_all Pi xs* ≤ *SPEC* (λ*r*. *r* ⟷ *list_all P xs*)
using *assms* **unfolding** *monadic_list_all_def*
by (*intro nfoldli_rule*[**where** *I* = λ*as bs b*. *b* = *list_all P as* ∧ *set* (*as* @ *bs*) = *set xs*]) *auto*

definition
monadic_list_ex P xs ≡ *nfoldli xs Not* (λ*x* _. *P x*) *False*

lemma *monadic_list_ex_rule*:
assumes ⋀*x*. *Pi x* ≤ *SPEC* (λ*r*. *r* = *P x*)
shows *monadic_list_ex Pi xs* ≤ *SPEC* (λ*r*. *r* ⟷ *list_ex P xs*)
using *assms* **unfolding** *monadic_list_ex_def*
by (*intro nfoldli_rule*[**where** *I* = λ*as bs b*. *b* = *list_ex P as* ∧ *set* (*as* @ *bs*) = *set xs*]) *auto*

lemma *monadic_list_ex_empty*[*simp*]:
monadic_list_ex P [] = *RETURN False*
unfolding *monadic_list_ex_def* **by** *simp*

lemma *monadic_list_all_empty*[*simp*]:
monadic_list_all P [] = *RETURN True*
unfolding *monadic_list_all_def* **by** *simp*

lemma *monadic_list_all_False*: *monadic_list_all* (λ*x*. *RETURN False*) *xs* = *RETURN* (*xs* = [])
by (*cases xs*) (*auto simp*: *monadic_list_all_def*)

lemma *monadic_list_all_RETURN*:
monadic_list_all (λ*x*. *RETURN* (*P x*)) *xs* = *RETURN* (*list_all P xs*)
proof (*induction xs*)
case *Nil*

```
  then show ?case
    by auto
next
  case (Cons x xs)
  then show ?case
    by (cases P x) (auto simp: monadic_list_all_def)
qed

lemma monadic_list_ex_RETURN:
  monadic_list_ex (λx. RETURN (P x)) xs = RETURN (list_ex P xs)
proof (induction xs)
  case Nil
  then show ?case
    by auto
next
  case (Cons x xs)
  then show ?case
    by (cases P x) (auto simp: monadic_list_ex_def)
qed

lemma monadic_list_ex_RETURN_mono:
  assumes set xs = set ys
  shows monadic_list_ex (λs. RETURN (P s)) xs ≤ monadic_list_ex (λs. RETURN (P s)) ys
  using assms by (simp add: monadic_list_ex_RETURN list_ex_iff)

lemma monadic_list_ex_nofailI:
  assumes ⋀ x. x ∈ set xs ⟹ nofail (f x)
  shows nofail (monadic_list_ex f xs)
  using assms unfolding monadic_list_ex_def
  by − (rule nfoldli_nofail_rule[where ub = RES UNIV]; simp add: pw_le_iff)

lemma monadic_list_all_nofailI:
  assumes ⋀ x. x ∈ set xs ⟹ nofail (f x)
  shows nofail (monadic_list_all f xs)
  using assms unfolding monadic_list_all_def
  by − (rule nfoldli_nofail_rule[where ub = RES UNIV]; simp add: pw_le_iff)

context
  fixes xs and g :: _ ⇒ bool nres
  assumes g_gt_SUCCEED: ⋀x. x ∈ set xs ⟹ g x > SUCCEED
begin

private lemma nfoldli_gt_SUCCEED: nfoldli xs c (λx _. g x) a > SUCCEED for a c
  using g_gt_SUCCEED
proof (induction xs arbitrary: a)
  case (Cons x xs)
  then show ?case
    by (cases g x; force intro: bind_RES_gt_SUCCEED_I simp: monadic_list_all_def)
qed simp

lemma monadic_list_all_gt_SUCCEED:
  monadic_list_all g xs > SUCCEED
  using nfoldli_gt_SUCCEED unfolding monadic_list_all_def .
```

lemma *monadic_list_ex_gt_SUCCEED*:
monadic_list_ex g xs > *SUCCEED*
using *nfoldli_gt_SUCCEED* **unfolding** *monadic_list_ex_def* **.**

end

lemma *monadic_list_ex_is_RETURN*:
∃ *r*. *monadic_list_ex* (λ*x*. *RETURN* (*P x*)) *xs* = *RETURN r*
proof (*induction xs*)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons x xs*)
then show *?case*
by (*cases P x*) (*auto simp*: *monadic_list_ex_def*)
qed

lemma *monadic_list_all_list_ex_is_RETURN*:
∃*r*. *monadic_list_all* (λ*x*. *monadic_list_ex* (λ*y*. *RETURN* (*P x y*)) *ys*) *xs* = *RETURN r*
proof −
let *?f* = λ*x*. *SOME r*. *monadic_list_ex* (λ*y*. *RETURN* (*P x y*)) *ys* = *RETURN r*
have *monadic_list_all* (λ*x*. *monadic_list_ex* (λ*y*. *RETURN* (*P x y*)) *ys*) *xs*
= *monadic_list_all* (λ*x*. *RETURN* (*?f x*)) *xs*
by (*fo_rule arg_cong2*; *intro HOL.refl monadic_list_ex_is_RETURN ext someI_ex*)
then show *?thesis*
by (*simp add*: *monadic_list_all_RETURN*)
qed

lemma *monadic_list_all_mono*[*refine_mono*]:
monadic_list_all P xs ≤ *monadic_list_all Q xs* **if** ∀ *x* ∈ *set xs*. *P x* ≤ *Q x*
proof −
have *nfoldli xs id* (λ*x* _. *P x*) *a* ≤ *nfoldli xs id* (λ*x* _. *Q x*) *a* **for** *a*
using *that* **by** (*induction xs arbitrary*: *a*; *clarsimp*; *refine_mono*)
then show *?thesis*
unfolding *monadic_list_all_def* **.**
qed

lemma *monadic_list_ex_mono*[*refine_mono*]:
monadic_list_ex P xs ≤ *monadic_list_ex Q xs* **if** ∀ *x* ∈ *set xs*. *P x* ≤ *Q x*
proof −
have *nfoldli xs Not* (λ*x* _. *P x*) *a* ≤ *nfoldli xs Not* (λ*x* _. *Q x*) *a* **for** *a*
using *that* **by** (*induction xs arbitrary*: *a*; *clarsimp*; *refine_mono*)
then show *?thesis*
unfolding *monadic_list_ex_def* **.**
qed

Abstract *nres* **algorithm** **context** *Reachability_Invariant_paired_defs*
begin

context
fixes *P* :: (′*l* × ′*s*) ⇒ *bool*
begin

definition *check_prop* ≡
do {
  xs ← *SPEC* (λ*xs*. *set xs* = *PR_CONST L*);
  monadic_list_all (λ*l*. *do* {
    xs ← *SPEC* (λ*xs*. *set xs* = *PR_CONST M l*);
    monadic_list_all (λ*s*. *RETURN* (*PR_CONST P* (*l*, *s*))) *xs*
  }
) *xs*
}

lemma *check_prop_correct*:
  check_prop ≤ *SPEC* (λ*r*. *r* ⟷ (∀ *l* ∈ *L*. ∀ *s* ∈ *M l*. *P* (*l*, *s*)))
  unfolding *check_prop_def*
  by (*refine_vcg monadic_list_all_rule monadic_list_ex_rule*) (*auto simp*: *list_all_iff*)

end

end

locale *Reachability_Impl_base* =
  Unreachability_Invariant_paired_pre_defs **where** *E* = *E* **for** *E* :: *'l* × *'s* ⇒ _ +
  fixes *succs* :: *'l* ⇒ *'s set* ⇒ (*'l* × *'s set*) *list*
  assumes *succs_correct*:
    ⋀*l*. ∀ *s* ∈ *xs*. *P* (*l*, *s*)
  ⟹ {(*l'*, *s'*)| *l' ys s'*. (*l'*, *ys*) ∈ *set* (*succs l xs*) ∧ *s'* ∈ *ys*}
    = (⋃ *s* ∈ *xs*. *Collect* (*E* (*l*, *s*)))

locale *Reachability_Impl_invariant* =
  Reachability_Impl_base **where** *E* = *E* +
  Unreachability_Invariant_paired_defs **where** *E* = *E* **for** *E* :: *'l* × *'s* ⇒ _
begin

definition *check_invariant L'* ≡
do {
  monadic_list_all (λ*l*.
  do {
    let as = *M l*;
    let succs = *succs l as*;
    monadic_list_all (λ(*l'*, *xs*).
    do {
      xs ← *SPEC* (λ*xs'*. *set xs'* = *xs*);
      if xs = [] *then RETURN True else do* {
        b1 ← *RETURN* (*l'* ∈ *L*);
        ys ← *SPEC* (λ*xs*. *set xs* = *M l'*);
        b2 ← *monadic_list_all* (λ*x*.
          monadic_list_ex (λ*y*. *RETURN* (*x* ≼ *y*)) *ys*
) *xs*;
        RETURN (*b1* ∧ *b2*)
      }
    }
) *succs*
  }

```
  ) L'
}


definition
  check_invariant_spec L' ≡
  ∀ l ∈ L'. ∀ s ∈ M l. ∀ l' s'. E (l, s) (l', s') ⟶ l' ∈ L ∧ (∃ s'' ∈ M l'. s' ≼ s'')

definition
  check_invariant_spec_pre L' ≡
  ∀ l ∈ set L'. ∀ (l',ys) ∈ set (succs l (M l)). ∀ s' ∈ ys. l' ∈ L ∧ (∃ s'' ∈ M l'. s' ≼ s'')

lemma check_invariant_correct_pre:
  check_invariant L' ≤ RETURN (check_invariant_spec_pre L')
  unfolding check_invariant_def check_invariant_spec_pre_def
 by (refine_vcg monadic_list_all_rule monadic_list_ex_rule) (auto simp: list_all_iff list_ex_iff)

context
  fixes L'
  assumes pre: ∀ l ∈ L. ∀ s ∈ M l. P (l, s) set L' ⊆ L
begin

lemma check_invariant_spec_pre_eq_check_invariant_spec:
  check_invariant_spec_pre L' = check_invariant_spec (set L')
proof −
  have ∗: (∀ (l',ys) ∈ set (succs l xs). ∀ s' ∈ ys. l' ∈ L ∧ (∃ s'' ∈ M l'. s' ≼ s'')) =
      (∀ s∈M l.
          ∀ l' s'.
             E (l, s) (l', s') ⟶ l' ∈ L ∧ (∃ s''∈M l'. s' ≼ s''))
    if xs = M l l ∈ L
     for l xs
    using succs_correct[of xs l] pre(1) that
    apply clarsimp
    apply safe
      apply clarsimp_all
      apply fastforce
     apply fastforce

    subgoal premises prems for a b s'
    proof −
      from prems have ∀ s∈xs. P (l, s)
        by auto
      from succs_correct[OF this] prems(3,6,7) obtain s where s ∈ M l E (l, s) (a, s')
        by fastforce
      with prems show ?thesis
        by auto
    qed
    apply fastforce
    done
  then show ?thesis
    unfolding check_invariant_spec_def check_invariant_spec_pre_def
    by (simp add: ∗ pre(2)[THEN subsetD])
qed
```

lemma *check_invariant_correct_strong*:
check_invariant $L' \leq$ *RETURN* (*check_invariant_spec* (*set* L'))
using *check_invariant_correct_pre*
unfolding *check_invariant_spec_pre_eq_check_invariant_spec*[*symmetric*] **.**

lemma *check_invariant_correct*:
check_invariant $L' \leq$ *SPEC* ($\lambda r.\ r \longrightarrow$ *check_invariant_spec* (*set* L'))
using *check_invariant_correct_strong* **by** (*rule Orderings.order.trans*) *simp*

end

end

locale *Reachability_Impl_base2* =
Reachability_Impl_base **where** $E = E$ +
Unreachability_Invariant_paired_pre_defs **where** $E = E$
for $E :: 'l \times 's \Rightarrow \_$ +
fixes P' **and** F
assumes P'_P: $\bigwedge\ l\ s.\ P'\ (l,\ s) \Longrightarrow P\ (l,\ s)$
assumes F_mono: $\bigwedge a\ b.\ P\ a \Longrightarrow F\ a \Longrightarrow (\lambda(l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s')\ a\ b \Longrightarrow P\ b \Longrightarrow F\ b$

locale Reachability_Impl_base2 = Reachability_Impl_base where E = E + Unreachability_Invariant_paired_pre_defs where E = E for E :: 'l × 's ⇒ _ + fixes P' and F assumes P'_P: ⋀ l s. P' (l, s) ⟹ P (l, s) assumes F_mono: ⋀a b. P a ⟹ F a ⟹ (λ(l, s) (l', s'). l' = l ∧ s ⪯ s') a b ⟹ P b ⟹ F b

locale Reachability_Impl_pre = Reachability_Impl_invariant where E = E + Reachability_Impl_base2 where E = E for E :: 'l × 's ⇒ _ begin

locale *Reachability_Impl_pre* =
Reachability_Impl_invariant **where** $E = E$ +
Reachability_Impl_base2 **where** $E = E$ **for** $E :: 'l \times 's \Rightarrow \_$
begin

definition
check_final $\equiv$ *do* {
$l \leftarrow$ *SPEC* ($\lambda xs.$ *set* $xs = L$);
monadic_list_all ($\lambda l.$ *do* {
$xs \leftarrow$ *SPEC* ($\lambda xs.$ *set* $xs = M\ l$);
monadic_list_all ($\lambda s.$
RETURN ($\neg\ F\ (l,\ s)$)
) *xs*
}
) l
}

definition
check_final_spec = ($\forall\ s' \in \{(l,\ s)\ |l\ s.\ l \in L \wedge s \in M\ l\}.\ \neg\ F\ s'$)

lemma *check_final_correct*:
check_final $\leq$ *SPEC* ($\lambda r.\ r \longleftrightarrow$ *check_final_spec*)
unfolding *check_final_def check_final_spec_def*

by (*refine_vcg monadic_list_all_rule*) (*auto simp*: *list_all_iff list_ex_iff*)

lemma *check_final_nofail*:
nofail check_final
by (*metis check_final_correct nofail_simps*(2) *pwD1*)

definition
check_init l_0 $s_0 \equiv$ *do* {
b1 $\leftarrow$ *RETURN* $(l_0 \in L)$;
b2 $\leftarrow$ *RETURN* $(P'\ (l_0,\ s_0))$;
xs $\leftarrow$ *SPEC* $(\lambda xs.\ set\ xs = M\ l_0)$;
b3 $\leftarrow$ *monadic_list_ex* $(\lambda s.\ RETURN\ (s_0 \preceq s))\ xs$;
RETURN $(b1 \wedge b2 \wedge b3)$
}

definition *check_all_pre_alt_def*:
check_all_pre l_0 $s_0 \equiv$ *do* {
b1 $\leftarrow$ *check_init* l_0 s_0;
b2 $\leftarrow$ *check_prop* P';
RETURN $(b1 \wedge b2)$
}

lemma *check_all_pre_def*:
check_all_pre l_0 $s_0 =$ *do* {
b1 $\leftarrow$ *RETURN* $(l_0 \in L)$;
b2 $\leftarrow$ *RETURN* $(P'\ (l_0,\ s_0))$;
xs $\leftarrow$ *SPEC* $(\lambda xs.\ set\ xs = M\ l_0)$;
b3 $\leftarrow$ *monadic_list_ex* $(\lambda s.\ RETURN\ (s_0 \preceq s))\ xs$;
b4 $\leftarrow$ *check_prop* P';
RETURN $(b1 \wedge b2 \wedge b3 \wedge b4)$
}
unfolding *check_all_pre_alt_def check_init_def* **by** *simp*

definition
check_init_spec l_0 $s_0 \equiv l_0 \in L \wedge (\exists\ s' \in M\ l_0.\ s_0 \preceq s') \wedge P'\ (l_0,\ s_0)$

definition
check_all_pre_spec l_0 $s_0 \equiv$
$(\forall\, l \in L.\ \forall\, s \in M\ l.\ P'\ (l,\ s)) \wedge l_0 \in L \wedge (\exists\ s' \in M\ l_0.\ s_0 \preceq s') \wedge P'\ (l_0,\ s_0)$

lemma *check_all_pre_correct*:
check_all_pre l_0 $s_0 \leq$ *RETURN* (*check_all_pre_spec* l_0 s_0)
unfolding *check_all_pre_def check_all_pre_spec_def*
by (*refine_vcg check_prop_correct monadic_list_ex_rule*; *standard*; *auto simp*: *list_ex_iff*)

lemma *check_init_correct*:
check_init l_0 $s_0 \leq$ *RETURN* (*check_init_spec* l_0 s_0)
unfolding *check_init_def check_init_spec_def*
by (*refine_vcg monadic_list_ex_rule*; *standard*; *auto simp*: *list_ex_iff*)

end

locale *Reachability_Impl_pre_start* =

```
  Reachability_Impl_pre where E = E for E :: 'l × 's ⇒ _ +
  fixes l₀ :: 'l and s₀ :: 's
begin

definition
  check_all ≡ do {
  b ← check_all_pre l₀ s₀;
  if b then SPEC (λr. r ⟶ check_invariant_spec L) else RETURN False
  }

definition
  certify_unreachable = do {
    b1 ← check_all;
    b2 ← check_final;
    RETURN (b1 ∧ b2)
  }

lemma certify_unreachable_alt_def:
  certify_unreachable = do {
    b1 ← check_all_pre l₀ s₀;
    b2 ← SPEC (λr. r ⟶ check_invariant_spec L);
    b3 ← check_final;
    RETURN (b1 ∧ b2 ∧ b3)
  }
  unfolding certify_unreachable_def check_all_def
  apply simp
  apply (fo_rule arg_cong2, auto)
  apply (rule ext)
  apply simp
   by (metis RETURN_rule Refine_Basic.bind_mono(1) dual_order.eq_iff let_to_bind_conv
lhs_step_bind nofail_simps(2))

definition
  check_all_spec ≡ check_all_pre_spec l₀ s₀ ∧ check_invariant_spec L

lemma check_all_correct:
  check_all ≤ SPEC (λr. r ⟶ check_all_spec)
  unfolding check_all_def check_all_spec_def check_all_pre_def check_all_pre_spec_def
  by (refine_vcg check_prop_correct check_invariant_correct monadic_list_ex_rule)
     (auto simp: list_ex_iff dest: P'_P)

end


locale Reachability_Impl_correct =
  Reachability_Impl_pre_start where E = E +
  Unreachability_Invariant_paired_pre where E = E for E :: 'l × 's ⇒ _
begin

lemma Unreachability_Invariant_pairedI[rule_format]:
  check_all_spec
  ⟶ Unreachability_Invariant_paired (⪯) (≺) M L E P l₀ s₀ (λ(l, u) (l', u'). l' = l ∧ u ⪯ u')
  unfolding check_all_spec_def check_all_pre_spec_def check_invariant_spec_def
  by clarsimp (standard, auto dest: P'_P)
```

lemma *check_all_correct'*:
check_all ≤ *SPEC* (λ*r*. *r* ⟶
Unreachability_Invariant_paired (⪯) (≺) *M L E P* l_0 s_0 (λ(*l*, *u*) (*l'*, *u'*). *l'* = *l* ∧ *u* ⪯ *u'*))
by (*refine_vcg Unreachability_Invariant_pairedI check_all_correct*) *fast*

lemma *certify_unreachableI*:
check_all_spec ∧ *check_final_spec* ⟶ (∄ *s'*. *E*** (l_0, s_0) *s'* ∧ *F s'*)
by (*rule impI Unreachability_Invariant_paired.final_unreachable Unreachability_Invariant_pairedI*)+
(*auto intro*: *F_mono simp*: *check_final_spec_def*)

lemma *certify_unreachable_correct*:
certify_unreachable ≤ *SPEC* (λ*r*. *r* ⟶ *check_all_spec* ∧ *check_final_spec*)
unfolding *certify_unreachable_def* **by** (*refine_vcg check_all_correct check_final_correct*; *fast*)

lemma *certify_unreachable_correct'*:
certify_unreachable ≤ *SPEC* (λ*r*. *r* ⟶ (∄ *s'*. *E*** (l_0, s_0) *s'* ∧ *F s'*))
by (*refine_vcg certify_unreachableI*[*rule_format*] *certify_unreachable_correct*; *fast*)

end

locale *Buechi_Impl_invariant* =
Reachability_Impl_base **where** *E* = *E* **for** *E* :: *'l* × *'s* ⇒ _ +
fixes *L* :: *'l set* **and** *M* :: *'l* ⇒ (*'s* × *nat*) *set*
begin

definition *check_invariant_buechi R L'* ≡
monadic_list_all (λ*l*.
do {
let as = *M l*;
as ← *SPEC* (λ*xs'*. *set xs'* = *as*);
monadic_list_all (λ(*x*, *i*). *do* {
let succs = *succs l* {*x*};
monadic_list_all (λ(*l'*, *xs*). *do* {
xs ← *SPEC* (λ*xs'*. *set xs'* = *xs*);
b1 ← *RETURN* (*l'* ∈ *L*);
if xs = [] *then RETURN True else do* {
ys ← *SPEC* (λ*xs*. *set xs* = *M l'*);
b2 ← *monadic_list_all* (λ*y*.
monadic_list_ex (λ(*z*, *j*). *RETURN* (*R l l' i j x y z*)) *ys*
) *xs*;
RETURN (*b1* ∧ *b2*)
}
}) *succs*
}) *as*
}) *L'*

definition
check_invariant_buechi_spec R L' ≡
∀ *l* ∈ *L'*. ∀ (*s*, *i*) ∈ *M l*. ∀ *l' s'*.
E (*l*, *s*) (*l'*, *s'*) ⟶ *l'* ∈ *L* ∧ (∃ (*s''*, *j*) ∈ *M l'*. *R l l' i j s s' s''*)

lemma *check_invariant_buechi_correct*:

check_invariant_buechi R L′ ≤ SPEC (λr. r ⟶ check_invariant_buechi_spec R (set L′))
(**is** _ ≤ *?rhs*)
if *assms*: ∀ *l* ∈ *L*. ∀ (*s*, _) ∈ *M l*. *P* (*l*, *s*) *set L′* ⊆ *L*
proof −
have ∗:
(*case x of* (*s*, *i*) ⇒ ∀ *x*∈*set* (*succs l* {*s*}). *case x of* (*l′*, *ys*) ⇒
∀ *s′* ∈ *ys*. *l′* ∈ *L* ∧ (∃ (*s″*, *j*) ∈ *M l′*. *R l l′ i j s s′ s″*)) =
(*case x of* (*s*, *i*) ⇒
∀ *l′ s′*. *E* (*l*, *s*) (*l′*, *s′*) ⟶ *l′* ∈ *L* ∧ (∃ (*s″*, *j*) ∈ *M l′*. *R l l′ i j s s′ s″*))
if *x* ∈ *M l l* ∈ *L* **for** *x l* **using** *succs_correct*[*of* {*fst x*} *l*] *assms*(*1*) *that* **by** *fastforce*
let *?R* = λ*l s i l′ s′*. (∃ (*s″*, *j*) ∈ *M l′*. *R l l′ i j s s′ s″*)
let *?Q* = λ*l s i*. λ(*l′*,*ys*). (∀ *s′* ∈ *ys*. *l′* ∈ *L* ∧ *?R l s i l′ s′*)
let *?P* = λ*l* (*s*, *i*). ∀ (*l′*,*ys*) ∈ *set* (*succs l* {*s*}). *?Q l s i* (*l′*, *ys*)
have *check_invariant_buechi R L′ ≤ SPEC* (λ*r*. *r* ⟷
(∀ *l* ∈ *set L′*. ∀ (*s*, *i*) ∈ *M l*. ∀ (*l′*,*ys*) ∈ *set* (*succs l* {*s*}). (∀ *s′* ∈ *ys*. *l′* ∈ *L* ∧
(∃ (*s″*, *j*) ∈ *M l′*. *R l l′ i j s s′ s″*))))
unfolding *check_invariant_buechi_def*
apply (*rule monadic_list_all_rule*[*unfolded list_all_iff*])
apply *refine_vcg*
subgoal for *l xs*
apply (*refine_vcg monadic_list_all_rule*[*unfolded list_all_iff*, **where** *P* = *?P l*])
subgoal for _ *s i*
apply (*refine_vcg monadic_list_all_rule*[*unfolded list_all_iff*, **where** *P* = *?Q l s i*])
apply (*solves auto*)
subgoal for _ *l′*
apply (*refine_vcg monadic_list_all_rule*[*unfolded list_all_iff*, **where** *P* = *?R l s i l′*])
subgoal for *s′*
apply (*refine_vcg monadic_list_ex_rule*[*unfolded list_ex_iff*])
by *auto*
by *auto*
by *auto*
by *auto*
done
also have ... ≤ *?rhs*
unfolding *check_invariant_buechi_spec_def* **by** (*auto simp*: ∗ *assms*(*2*)[*THEN subsetD*])
finally show *?thesis* **.**
qed

end

locale *Buechi_Impl_pre* =
Buechi_Impl_invariant **where** *M* = *M* +
Reachability_Impl_base2 **for** *M* :: *'l* ⇒ (*'s* × *nat*) *set* +
assumes *finite*: *finite L* ∀ *l* ∈ *L*. *finite* (*M l*)
begin

definition
buechi_prop l l′ i j s s′ s″ ≡ *l′* ∈ *L* ∧ *s′* ≼ *s″* ∧
(*if F* (*l*, *s*) *then i* < *j else i* ≤ *j*)

Old alternative definition. Slightly easier to work with but subsumptions are not deterministic.

definition
has_SE ≡ λ*s l*. ∃ *s′ j*. *s* ≼ *s′* ∧ (*s′*, *j*) ∈ *M l*

definition
SE ≡ λ(*l*, *s*) (*l'*, *s'*). *l'* = *l* ∧ *l* ∈ *L* ∧ *has_SE s l* ∧
(∃*j*. (*s'*, *j*) = *arg_max* (λ(*s*, *i*). *i*) (λ(*s'*, *j*). *s* ≼ *s'* ∧ (*s'*, *j*) ∈ *M l*))

lemma
assumes *SE* (*l*, *s*) (*l'*, *s'*)
shows *SE_same_loc*: *l'* = *l* **and** *SE_subsumes*: *s* ≼ *s'*
and *SE_is_arg_max*: ∃*j*. *is_arg_max* (λ(*s*, *i*). *i*) (λ(*s'*, *j*). *s* ≼ *s'* ∧ (*s'*, *j*) ∈ *M l*) (*s'*, *j*)
(**is** ∃*j*. *is_arg_max ?f ?P* (*s'*, *j*))
proof −
from *assms* **have** *has_SE s l' l'* ∈ *L* **and** [*simp*]: *l'* = *l*
unfolding *SE_def* **by** *auto*
then obtain *s1 j* **where** *?P* (*s1*, *j*)
unfolding *has_SE_def* **by** *auto*
moreover have *finite* (*Collect ?P*)
using *finite* ‹*l'* ∈ *L*› **by** (*auto intro*: *finite_subset*)
moreover note *arg_max_rule* = *arg_max_nat_lemma2*[*of ?P*, *OF calculation*, *of* λ(*s*, *i*). *i*]
then show *l'* = *l s* ≼ *s'* ∃*j*. *is_arg_max ?f ?P* (*s'*, *j*)
using *assms* **unfolding** *is_arg_max_linorder SE_def is_arg_max_linorder* **by** *auto*
qed

lemma *SE_deterministic*:
assumes ⋀*s*. *s1* ≼ *s* ⟷ *s2* ≼ *s*
assumes *SE* (*l*, *s1*) (*l'*, *s1'*) *SE* (*l*, *s2*) (*l'*, *s2'*)
shows *s2'* = *s1'*
using *assms*(*2*,*3*) **unfolding** *SE_def* **by** (*clarsimp simp*: *assms*(*1*)) (*metis prod.inject*)

lemma *SE_I*:
assumes (*s''*, *j*) ∈ *M l' buechi_prop l l' i j s s' s''*
shows ∃(*s''*, *j*) ∈ *M l'*. *SE* (*l'*, *s'*) (*l'*, *s''*)
proof −
let *?P* = λ(*s1*, *j*). *s'* ≼ *s1* ∧ (*s1*, *j*) ∈ *M l'*
let *?f* = λ(*s*, *i*). *i*
from *assms* **have** *?P* (*s''*, *j*) *l'* ∈ *L*
unfolding *buechi_prop_def* **by** *auto*
have *finite* (*Collect ?P*)
using *finite* ‹*l'* ∈ *L*› **by** (*auto intro*: *finite_subset*)
from *arg_max_nat_lemma2*[*OF* ‹*?P* (*s''*, *j*) › *this*, *of ?f*] ‹*l'* ∈ *L*› **show** *?thesis*
unfolding *has_SE_def SE_def is_arg_max_linorder* **by** (*auto 4 3*)
qed

definition
check_all_pre_spec1 inits ≡
(∀ *l* ∈ *L*. ∀ *s* ∈ *fst* ‘ *M l*. *P'* (*l*, *s*)) ∧
(∀ (*l*$_0$, *s*$_0$) ∈ *inits*. *l*$_0$ ∈ *L* ∧ (∃ (*s'*, _) ∈ *M l*$_0$. *s*$_0$ ≼ *s'*) ∧ *P'* (*l*$_0$, *s*$_0$))

definition
check_buechi inits ≡ *do* {
b ← *SPEC* (λ*r*. *r* ⟶ *check_all_pre_spec1 inits*);
if b then do {
ASSERT (*check_all_pre_spec1 inits*);
SPEC (λ*r*. *r* ⟶ *check_invariant_buechi_spec* (*buechi_prop*) *L*)
} *else RETURN False*

}

definition
check_buechi_spec inits ≡
check_all_pre_spec1 inits
∧ (∀ *l* ∈ *L*. ∀ (*s*, *i*) ∈ *M l*. ∀ *l′ s′*. *E* (*l*, *s*) (*l′*, *s′*)
⟶ (∃ (*s″*, *j*) ∈ *M l′*. *buechi_prop l l′ i j s s′ s″*))

definition
check_buechi_spec′ inits ≡
(∀ (*l*$_0$, *s*$_0$) ∈ *inits*. *Unreachability_Invariant_paired* (⪯) (≺) (λ*l*. *fst* ‘ *M l*) *L E P l*$_0$ *s*$_0$ *SE*)
∧ (∀ *l* ∈ *L*. ∀ (*s*, *i*) ∈ *M l*. ∀ *l′ s′*. *E* (*l*, *s*) (*l′*, *s′*)
⟶ (∃ (*s″*, *j*) ∈ *M l′*. *buechi_prop l l′ i j s s′ s″*))

lemma *check_buechi_correct*:
check_buechi inits ≤ *SPEC* (λ*r*. *r* ⟶ *check_buechi_spec inits*)
unfolding *check_buechi_def check_invariant_buechi_spec_def check_buechi_spec_def*
by (*refine_vcg*; *blast*)

end

locale *Buechi_Impl_correct* =
Buechi_Impl_pre **where** *M* = *M* **and** *E* = *E*+
Unreachability_Invariant_paired_pre **where** *E* = *E* **for** *E* **and** *M* :: *′l* ⇒ (*′s* × *nat*) *set*
begin

lemma *check_buechi_correct′*:
check_buechi inits ≤ *SPEC* (λ*r*. *r* ⟶ *check_buechi_spec′ inits*)
proof −
have *Unreachability_Invariant_paired* (⪯) (≺) (λ*l*. *fst* ‘ *M l*) *L E P l*$_0$ *s*$_0$ *SE*
if (*l*$_0$, *s*$_0$) ∈ *inits check_all_pre_spec1 inits check_invariant_buechi_spec* (*buechi_prop*) *L*
for *l*$_0$ *s*$_0$
using *that* **unfolding** *check_invariant_buechi_spec_def check_all_pre_spec1_def*
apply −
apply *standard*
apply (*use SE_I SE_same_loc SE_subsumes* **in**
‹*auto 4 3 dest*!: *P′_P simp*: *list_ex_iff Ball_def_raw Bex_def_raw*›)
apply (*smt case_prodE fst_conv*)
done
then show *?thesis*
unfolding *check_buechi_def check_invariant_buechi_spec_def check_buechi_spec′_def*
by (*refine_vcg*; *blast*)
qed

definition *f_topo* **where**
f_topo ≡ λ(*l*, *s*). *Max* {*i*. (*s*, *i*) ∈ *M l*}

lemma
assumes *l* ∈ *L* (*s*, *i*) ∈ *M l*
shows *f_in*: (*s*, *f_topo* (*l*, *s*)) ∈ *M l* (**is** *?P1*)
and *f_ge*: ∀ *j*. (*s*, *j*) ∈ *M l* ⟶ *j* ≤ *f_topo* (*l*, *s*) (**is** *?P2*)
proof −
have *finite* {*i*. (*s*, *i*) ∈ *M l*}

```
    using finite ‹l ∈ L› [[simproc add: finite_Collect]] by auto
  with assms(2) show ?P1 ?P2
    unfolding f_topo_def by (auto intro: Max_ge dest: Max_in)
qed

lemma f_topo:
  fixes l :: ‹'l› and s :: ‹'s› and l1 :: ‹'l› and s1 :: ‹'s› and l2 :: ‹'l› and s2 :: ‹'s›
  assumes
    check_buechi_spec' inits
    ‹l ∈ L› and
    ‹s ∈ fst ‘ M l› and
    ‹l2 ∈ L› and
    ‹s2 ∈ fst ‘ M l2› and
    ‹E (l, s) (l1, s1)› and
    ‹SE (l1, s1) (l2, s2)›
  shows ‹if F (l, s) then f_topo (l, s) < f_topo (l2, s2) else f_topo (l, s) ≤ f_topo (l2, s2)›
proof −
  let ?P = λs l' (s', j). s ≼ s' ∧ (s', j) ∈ M l'
  let ?f = λ(s, i). i
  let ?le = λl s i j. if F(l, s) then i < j else i ≤ j
  from ‹SE _ _› obtain j where (s2, j) ∈ M l2 and [simp]: l2 = l1
    and is_max: is_arg_max ?f (?P s1 l1) (s2, j)
    using SE_is_arg_max SE_same_loc SE_subsumes
    by atomize_elim (fastforce simp: Lattices_Big.ord_class.is_arg_max_def)
  from f_in assms have (s, f_topo (l, s)) ∈ M l
    by auto
  with assms obtain s' i where (s', i) ∈ M l2 buechi_prop l l2 (f_topo (l, s)) i s s1 s'
    unfolding check_buechi_spec'_def by fastforce
  then have (s', i) ∈ M l2 s1 ≼ s' ?le l s (f_topo (l, s)) i
    unfolding buechi_prop_def by auto
  from is_max ‹(s', i) ∈ _› ‹s1 ≼ s'› have i ≤ j
    unfolding is_arg_max_linorder by simp
  also from f_ge ‹(s2, j) ∈ M l2› have j ≤ f_topo (l2, s2)
    using assms by auto
  finally show ?thesis
    using ‹?le l s _ _› by auto
qed

lemma no_buechi_run:
  assumes check: check_buechi_spec' inits
  assumes accepting_run:
    (l₀, s₀) ∈ inits Graph_Defs.run E ((l₀, s₀) ## xs) alw (ev (holds F)) ((l₀, s₀) ## xs)
  shows False
proof −
  interpret Unreachability_Invariant_paired (≼) (≺) λl. fst ‘ M l L E P l₀ s₀ SE
    using check ‹_ ∈ inits› unfolding check_buechi_spec'_def by blast
  show ?thesis
    apply (rule no_buechi_run[where F = F and f = f_topo])
        apply (rule F_mono; assumption)
    using finite apply blast+
      apply (rule f_topo, rule check, assumption+)
     apply (rule accepting_run)+
    done
qed
```

end

locale *Reachability_Impl_common* =
Reachability_Impl_pre **where** *less_eq* = *less_eq* **and** $M = \lambda x.$ *case* $M\ x$ *of None* $\Rightarrow \{\} \mid$ *Some* $S \Rightarrow S$
for *less_eq* :: $'b \Rightarrow 'b \Rightarrow$ *bool* (**infix** $\preceq$ 50) **and** $M :: 'k \Rightarrow 'b$ *set option* +
assumes *L_finite*: *finite L*
and *M_ran_finite*: $\forall S \in$ *ran M*. *finite S*
and *succs_finite*: $\forall l\ S.\ \forall (l', S') \in$ *set* (*succs l S*). *finite S* $\longrightarrow$ *finite* S'
and *succs_empty*: $\bigwedge l.$ *succs l* $\{\} = []$

begin

lemma *M_listD*:
assumes *M l* = *Some S*
shows $\exists$ *xs*. *set xs* = *S*
using *M_ran_finite assms* **unfolding** *ran_def* **by** (*auto intro*: *finite_list*)

lemma *L_listD*:
shows $\exists$ *xs*. *set xs* = *L*
using *L_finite* **by** (*rule finite_list*)

lemma *check_prop_gt_SUCCEED*:
check_prop $P' >$ *SUCCEED*
unfolding *check_prop_def* **using** *L_listD*
by (*fastforce split*: *option.split dest*: *M_listD*
intro: *monadic_list_all_gt_SUCCEED bind_RES_gt_SUCCEED_I*
)

definition
check_final' $L'\ M' =$ *do* {
$l \leftarrow$ *SPEC* ($\lambda xs.$ *set xs* = L');
monadic_list_all ($\lambda l.$ *do* {
let S = *op_map_lookup l M'*;
case S of None $\Rightarrow$ *RETURN True* | *Some S* $\Rightarrow$ *do* {
xs $\leftarrow$ *SPEC* ($\lambda xs.$ *set xs* = *S*);
monadic_list_all ($\lambda s.$
RETURN ($\neg$ *PR_CONST F* (*l*, *s*))
) *xs*
}
}
) *l*
}

lemma *check_final_alt_def*:
check_final' $L\ M =$ *check_final*
unfolding *check_final'_def check_final_def*
by (*fo_rule arg_cong2*, *simp*, *fo_rule arg_cong*) (*auto split*: *option.split simp*: *bind_RES*)

definition *check_prop'* **where**
check_prop' $L'\ M' =$ *do* {
$l \leftarrow$ *SPEC* ($\lambda xs.$ *set xs* = L');

```
  monadic_list_all (λl. do {
    let S = op_map_lookup l M';
    case S of None ⇒ RETURN True | Some S ⇒ do {
      xs ← SPEC (λxs. set xs = S);
      r ← monadic_list_all (λs.
        RETURN (PR_CONST P' (l, s))
      ) xs;
      RETURN r
    }
    }
  ) l
  }

lemma check_prop_alt_def:
  check_prop' L M = check_prop P'
  unfolding check_prop_def check_prop'_def
  by (fo_rule arg_cong2, simp, fo_rule arg_cong) (auto split: option.split simp: bind_RES)

lemma check_prop'_alt_def:
  check_prop' L' M' = do {
  l ← SPEC (λxs. set xs = L');
  monadic_list_all (λl. do {
    let (S, M) = op_map_extract l M';
    case S of None ⇒ RETURN True | Some S ⇒ do {
      xs ← SPEC (λxs. set xs = S);
      r ← monadic_list_all (λs.
        RETURN (PR_CONST P' (l, s))
      ) xs;
      RETURN r
    }
    }
  ) l
  }
  unfolding check_prop'_def
  by (fo_rule arg_cong2, simp, fo_rule arg_cong) (auto split: option.split simp: bind_RES)

end


locale Certification_Impl_base = Reachability_Impl_base2 where less = less
  for less :: 's ⇒ 's ⇒ bool (infix ≺ 50) +
  fixes A :: 's ⇒ ('si :: heap) ⇒ assn
    and K :: 'k ⇒ ('ki :: {hashable,heap}) ⇒ assn
    and Fi and keyi and Pi and copyi and Lei and succsi
  assumes [sepref_fr_rules]: (keyi,RETURN o PR_CONST fst) ∈ (prod_assn K A)^k →_a K
  assumes copyi[sepref_fr_rules]: (copyi, RETURN o COPY) ∈ A^k →_a A
  assumes Pi_P'[sepref_fr_rules]: (Pi,RETURN o PR_CONST P') ∈ (prod_assn K A)^k →_a
bool_assn
  assumes Fi_F[sepref_fr_rules]: (Fi,RETURN o PR_CONST F) ∈ (prod_assn K A)^d →_a
bool_assn
  assumes succsi[sepref_fr_rules]:
    (uncurry succsi,uncurry (RETURN oo PR_CONST succs))
    ∈ K^k *_a (lso_assn A)^d →_a list_assn (K ×_a lso_assn A)
  assumes Lei[sepref_fr_rules]:
```

(*uncurry Lei*,*uncurry* (*RETURN oo PR_CONST less_eq*)) ∈ A^k ∗$_a$ A^k →$_a$ *bool_assn*
assumes *pure_K*: *is_pure K*
assumes *left_unique_K*: *IS_LEFT_UNIQUE* (*the_pure K*)
assumes *right_unique_K*: *IS_RIGHT_UNIQUE* (*the_pure K*)

locale *Reachability_Impl* =
Reachability_Impl_common **where** *M* = *M* +
Certification_Impl_base **where** *K* = *K* **and** *A* = *A* +
Reachability_Impl_correct **where** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S*
for *M* :: *'k* ⇒ *'a set option*
and *K* :: *'k* ⇒ *'ki* :: {*hashable*,*heap*} ⇒ *assn* **and** *A* :: *'a* ⇒ *'ai* :: *heap* ⇒ *assn* +
fixes l_0i :: *'ki Heap* **and** s_0i :: *'ai Heap*
assumes $l_0i\_l_0$[*sepref_fr_rules*]:
(*uncurry0* l_0i, *uncurry0* (*RETURN* (*PR_CONST* l_0))) ∈ *unit_assn*^{k} →$_a$ *K*
assumes $s_0i\_s_0$[*sepref_fr_rules*]:
(*uncurry0* s_0i, *uncurry0* (*RETURN* (*PR_CONST* s_0))) ∈ *unit_assn*^{k} →$_a$ *A*

definition
print_check s b = *println* (*s* + *STR ''*: *''* + (*if b then STR ''passed'' else STR ''failed''*))

definition
PRINT_CHECK = *RETURN oo print_check*

lemma [*sepref_import_param*]:
(*print_check*, *print_check*) ∈ *Id* → *Id* → *Id*
by *simp*

sepref_definition *print_check_impl* **is**
uncurry PRINT_CHECK :: *id_assn*^{k} ∗$_a$ *id_assn*^{k} →$_a$ *id_assn*
unfolding *PRINT_CHECK_def* **by** *sepref*

sepref_register *PRINT_CHECK*

lemmas [*sepref_fr_rules*] = *print_check_impl.refine*

Misc implementation **sepref_decl_op** *map_lookup_copy*: λ*k* (*m* :: _ ⇀ _). (*m k*, *m*)
:: *K* → ⟨*K*,*V*⟩*map_rel* → ⟨*V*⟩*option_rel* ×$_r$ ⟨*K*,*V*⟩*map_rel*
where *single_valued K single_valued* (K^{-1})
apply (*rule fref_ncI*)
apply *parametricity*
unfolding *map_rel_def*
apply (*elim IntE*)
apply *parametricity*
done

definition
heap_map copy xs ≡ *do* {
xs ← *imp_nfoldli xs* (λ_. *return True*) (λ*x xs*. *do* {*x* ← *copy x*; *return* (*x* # *xs*)}) [];
return (*rev xs*)
}

definition
monadic_map copy xs ≡ *do* {
xs ← *monadic_nfoldli xs* (λ_. *RETURN True*) (λ*x xs*. *do* {*x* ← *copy x*; *RETURN* (*x* # *xs*)})
[];
RETURN (*rev xs*)
}

context
begin

private lemma *monadic_nfoldli_rev*:
monadic_nfoldli x (λ_. *RETURN True*) (λ*x xs*. *RETURN* (*x* # *xs*)) [] ≤ *SPEC* (λ*r*. *r* = *rev x*)
unfolding *nfoldli_to_monadic*[**where** *c* = λ_.*True*, *symmetric*]
by (*rule nfoldli_rule*[**where** *I* = λ *xs ys zs*. *rev zs* @ *ys* = *x*]) *auto*

private lemma *frame2*:
hn_ctxt (*list_assn A*) *x xi* ∗ *hn_invalid* (*list_assn A*) [] [] ∗ *hn_ctxt* (*list_assn A*) *xa x′*
⟹$_t$ *hn_ctxt* (*list_assn A*) *x xi* ∗ *hn_ctxt* (*list_assn A*) *xa x′*
by (*simp add*: *frame_rem2 frame_rem3*)

private lemma *frame3*:
hn_ctxt (*list_assn A*) *x xi* ∗ *hn_invalid* (*list_assn A*) [] []
⟹$_t$ *hn_ctxt* (*list_assn A*) *x xi* ∗ *hn_ctxt* (*pure UNIV*) *xa x′*
by (*simp add*: *frame_rem2 frame_rem3 pure_def entt_fr_drop hn_ctxt_def*)

lemma *list_rev_aux*: *list_assn A a c* ⟹$_A$ *list_assn A* (*rev a*) (*rev c*)
apply (*subst list_assn_aux_len*, *clarsimp*)
apply (*induction rule*: *list_induct2*)
apply (*sep_auto*; *fail*)
apply (*sep_auto*, *erule ent_frame_fwd*, *frame_inference*, *sep_auto*)
done

theorem *copy_list_refine*:
assumes
copy: (*copy*, *RETURN o COPY*) ∈ A^k →$_a$ *A*
shows
hn_refine
(*hn_ctxt* (*list_assn A*) *x xi*)
(*heap_map copy* $ *xi*)
(*hn_ctxt* (*list_assn A*) *x xi*)
(*list_assn A*)
(*monadic_map* (*RETURN* ∘ *COPY*) $ *x*)
unfolding *monadic_map_def heap_map_def*
apply *sep_auto*
apply (*rule hnr_bind*)
apply (*rule monadic_nfoldli_refine_aux′*[
where *S* = *set x* **and** *Rs* = *list_assn A* **and** *Rl* = *A* **and** *Rl′* = *A* **and** *Rl″* = *A* **and** Γ = *emp*,
THEN hn_refine_cons_pre[*rotated*]])
apply *sep_auto*
subgoal
by *standard* (*sep_auto simp*: *pure_def*)
subgoal

```
    supply [sep_heap_rules]  = copy[to_hnr, unfolded hn_refine_def, simplified]
    apply standard
    apply sep_auto

   by (smt assn_times_comm ent_refl ent_star_mono hn_ctxt_def invalidate_clone star_aci(3))

     apply (sep_auto; fail)
    apply (sep_auto simp: pure_def; fail)
   prefer 2
   apply (rule frame3; fail)
  apply standard
  apply sep_auto
  apply (drule order_trans, rule monadic_nfoldli_rev)
  apply (rule ent_true_drop(2))
  apply (rule ent_star_mono)
   apply sep_auto
  unfolding hn_ctxt_def
  apply (rule list_rev_aux)
  done

end

lemma monadic_map_refine':
  (heap_map copy, monadic_map (RETURN o COPY)) ∈ (list_assn A)^k →_a list_assn A
  if (copy, RETURN o COPY) ∈ A^k →_a A
  using that by (rule copy_list_refine[to_hfref])

lemma copy_list_COPY:
  monadic_map (RETURN o COPY) = RETURN o COPY (is ?l = ?r)
proof (rule ext)
  fix xs :: 'a list
  have *:
    monadic_nfoldli xs (λ_. RETURN True)
     (λx xs. (RETURN ∘ (λx. x)) x ⤜ (λx. RETURN (x # xs))) as
    = RETURN (rev xs @ as) for as
    by (induction xs arbitrary: as) auto
  show ?l xs = ?r xs
    unfolding monadic_map_def COPY_def by (subst *) simp
qed

lemma copy_list_lso_assn_refine:
  (heap_map copy, RETURN o COPY) ∈ (lso_assn A)^k →_a lso_assn A
  if (copy, RETURN o COPY) ∈ A^k →_a A
  supply [sep_heap_rules] =
   monadic_map_refine'[OF that, to_hnr, unfolded copy_list_COPY hn_refine_def hn_ctxt_def,
simplified]
  unfolding lso_assn_def hr_comp_def by sepref_to_hoare sep_auto

end
theory Unreachability_Certification
  imports
    Simulation_Graphs_Certification
    Worklist_Algorithms.Leadsto_Impl
    Munta_Base.Printing
```

Difference_Bound_Matrices.DBM_Imperative_Loops
Munta_Base.Trace_Timing
Unreachability_Misc
begin

context
fixes $K :: 'k \Rightarrow ('ki :: \{heap\}) \Rightarrow assn$
assumes *pure_K*: *is_pure K*
assumes *left_unique_K*: *IS_LEFT_UNIQUE* (*the_pure K*)
assumes *right_unique_K*: *IS_RIGHT_UNIQUE* (*the_pure K*)
begin

lemma *pure_equality_impl*:
(*uncurry* (*return oo* (=)), *uncurry* (*RETURN oo* (=))) ∈ $(K^k *_a K^k) \rightarrow_a$ *bool_assn*
proof −
have *1*: *K* = *pure* (*the_pure K*)
using *pure_K* **by** *auto*
have [*dest*]: *a* = *b* **if** (*bi*, *b*) ∈ *the_pure K* (*bi*, *a*) ∈ *the_pure K* **for** *bi a b*
using *that right_unique_K* **by** (*elim single_valuedD*) *auto*
have [*dest*]: *a* = *b* **if** (*a*, *bi*) ∈ *the_pure K* (*b*, *bi*) ∈ *the_pure K* **for** *bi a b*
using *that left_unique_K* **unfolding** *IS_LEFT_UNIQUE_def* **by** (*elim single_valuedD*)
auto
show *?thesis*
by (*subst 1*, *subst* (*2*) *1*, *sepref_to_hoare*, *sep_auto*)
qed

definition
is_member x L ≡ *do* {
xs ← *SPEC* (λ*xs*. *set xs* = *L*);
monadic_list_ex (λ*y*. *RETURN* (*y* = *x*)) *xs*
}

lemma *is_member_refine*:
is_member x L ≤ *mop_set_member x L*
unfolding *mop_set_member_alt is_member_def*
by (*refine_vcg monadic_list_ex_rule*) (*auto simp*: *list_ex_iff*)

lemma *is_member_correct*:
(*uncurry is_member*, *uncurry* (*RETURN* ∘∘ *op_set_member*)) ∈ *Id* ×$_r$ *Id* → ⟨*bool_rel*⟩*nres_rel*
using *is_member_refine* **by** (*force simp*: *pw_le_iff pw_nres_rel_iff*)

lemmas [*sepref_fr_rules*] = *lso_id_hnr*

sepref_definition *is_member_impl* **is**
uncurry is_member :: $K^k *_a (lso\_assn\ K)^k \rightarrow_a$ *bool_assn*
supply [*sepref_fr_rules*] = *pure_equality_impl*
supply [*safe_constraint_rules*] = *pure_K left_unique_K right_unique_K*
unfolding *is_member_def monadic_list_ex_def list_of_set_def*[*symmetric*] **by** *sepref*

lemmas *op_set_member_lso_hnr* = *is_member_impl.refine*[*FCOMP is_member_correct*]

end

Concrete Implementations **context** *Reachability_Impl*
begin

definition
check_invariant′ L′ M′ ≡ *do* {
monadic_list_all (λ*l*.
do {
case op_map_lookup l M′ of None ⇒ *RETURN True* | *Some xs* ⇒ *do* {
let succs = *PR_CONST succs l xs*;
monadic_list_all (λ(*l′*, *xs*). *do* {
xs ← *SPEC* (λ*xs′*. *set xs′* = *xs*);
if xs = [] *then RETURN True*
else do {
case op_map_lookup l′ M′ of None ⇒ *RETURN False* | *Some ys* ⇒ *do* {
ys ← *SPEC* (λ*xs*. *set xs* = *ys*);
monadic_list_all (λ*x′*.
monadic_list_ex (λ*y*. *RETURN* (*PR_CONST less_eq x′ y*)) *ys*
) *xs*
}
}
}) *succs*
}
}) *L′*
}

lemma *succs_listD*:
assumes (*l′*, *S′*) ∈ *set* (*succs l S*) *finite S*
shows ∃ *xs*. *set xs* = *S′*
using *assms succs_finite* **by** (*force intro*!: *finite_list*)

lemma *check_invariant′_refine*:
check_invariant′ L_part M ≤ *check_invariant L_part* **if** *L* = *dom M set L_part* ⊆ *L*
unfolding *check_invariant_def check_invariant′_def*
unfolding *PR_CONST_def*
apply *refine_mono*
apply (*clarsimp split*: *option.splits simp*: *succs_empty*)
apply *refine_mono*
apply (*clarsimp split*: *option.splits*)
apply *safe*
subgoal
by *refine_mono* (*auto simp*: *bind_RES monadic_list_all_False dest*: *M_listD*)
subgoal
apply *refine_mono*
apply *clarsimp*
apply *refine_mono*
apply *clarsimp*
subgoal for *xs1 xs2*
using *monadic_list_all_list_ex_is_RETURN*[*of* λ *x y*. *less_eq x y xs2 xs1*]
by (*auto simp*: *bind_RES* ‹*L* = *dom M*›)
done
done

lemma *check_invariant′_no_fail*:
nofail (*check_invariant′ L′ M′*)

unfolding *check_invariant'_def*
by (*intro monadic_list_all_nofailI monadic_list_ex_nofailI no_fail_RES_bindI*
| *clarsimp split*: *option.splits*)+

lemma *check_invariant'_correct_pre*:
check_invariant' L_part M ≤ *RETURN* (*check_invariant_spec_pre L_part*)
if *L* = *dom M set L_part* ⊆ *L*
by (*rule check_invariant_correct_pre check_invariant'_refine*[*OF that*] *Orderings.order.trans*)+

definition *check_invariant_spec_pre1* **where**
check_invariant_spec_pre1 L' M' ≡
∀ *l* ∈ *set L'*. ∀ (*l'*,*ys*) ∈ *set* (*succs l* (*M' l*)). ∀ *s'* ∈ *ys*. (∃ *s''* ∈ *M' l'*. *s'* ≼ *s''*)

lemma *check_invariant'_correct_pre1*:
check_invariant' L' M'
≤ *RETURN* (*check_invariant_spec_pre1 L'* (λ*l*. *case M' l of None* ⇒ {} | *Some S* ⇒ *S*))
unfolding *check_invariant'_def check_invariant_spec_pre1_def*
by (*refine_vcg monadic_list_all_rule monadic_list_ex_rule*)
(*auto simp*: *list_all_iff list_ex_iff succs_empty*)

lemmas [*safe_constraint_rules*] = *pure_K left_unique_K right_unique_K*

lemmas [*sepref_fr_rules*] = *lso_id_hnr*

sepref_register
PR_CONST L list_of_set PR_CONST P' PR_CONST F PR_CONST succs PR_CONST less_eq
PR_CONST M :: ('*k*, '*b set*) *i_map*

lemma [*sepref_import_param*]: (*id*, *id*) ∈ (*Id* :: (*bool* × *bool*) *set*) → *Id*
by *simp*

sepref_definition *check_prop_impl_wrong* **is**
uncurry check_prop' :: (*lso_assn K*)^k ∗$_a$ (*hm.hms_assn' K* (*lso_assn A*))^k →$_a$ *id_assn*
unfolding *check_prop'_alt_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def*
apply *sepref_dbg_keep*
oops

sepref_decl_impl *map_lookup*:
copy_list_lso_assn_refine[*OF copyi*, *THEN hms_hm.hms_lookup_hnr*]
uses *op_map_lookup.fref*[**where** *V* = *Id*] **.**

abbreviation *table_assn* ≡ *hm.hms_assn' K* (*lso_assn A*)

sepref_thm *check_prop_impl* **is**
uncurry (*PR_CONST check_prop'*) :: (*lso_assn K*)^k ∗$_a$ *table_assn*^{k} →$_a$ *id_assn*
unfolding *PR_CONST_def*
unfolding *check_prop'_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def*
by *sepref*

concrete_definition (**in** −) *check_prop_impl*
uses *Reachability_Impl.check_prop_impl.refine_raw* **is** (*uncurry ?f*,_)∈_

sepref_thm *check_final_impl* **is**
uncurry (*PR_CONST check_final'*) :: $(lso\_assn\ K)^k *_a table\_assn^k \to_a id\_assn$
unfolding *PR_CONST_def*
unfolding *check_final'_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def*
by *sepref*

concrete_definition (**in** −) *check_final_impl*
uses *Reachability_Impl.check_final_impl.refine_raw* **is** (*uncurry ?f*,_)∈_

lemma *K_equality*[*sepref_fr_rules*]:
(*uncurry* (*return oo* (=)), *uncurry* (*RETURN oo* (=))) ∈ $(K^k *_a K^k) \to_a bool\_assn$
proof −
have *1*: *K* = *pure* (*the_pure K*)
using *pure_K* **by** *auto*
have [*dest*]: *a* = *b* **if** (*bi*, *b*) ∈ *the_pure K* (*bi*, *a*) ∈ *the_pure K* **for** *bi a b*
using *that right_unique_K* **by** (*elim single_valuedD*) *auto*
have [*dest*]: *a* = *b* **if** (*a*, *bi*) ∈ *the_pure K* (*b*, *bi*) ∈ *the_pure K* **for** *bi a b*
using *that left_unique_K* **unfolding** *IS_LEFT_UNIQUE_def* **by** (*elim single_valuedD*)
auto
show *?thesis*
by (*subst 1*, *subst* (*2*) *1*, *sepref_to_hoare*, *sep_auto*)
qed

sepref_definition *is_K_eq_impl* **is**
uncurry (*RETURN oo* (=)) :: $(K^k *_a K^k) \to_a bool\_assn$
unfolding *is_member_def monadic_list_ex_def list_of_set_def*[*symmetric*] **by** *sepref*

lemmas [*sepref_fr_rules*] = *op_set_member_lso_hnr*[*OF pure_K left_unique_K right_unique_K*]

sepref_definition *check_invariant_impl* **is**
uncurry (*PR_CONST check_invariant'*) :: $(list\_assn\ K)^k *_a table\_assn^k \to_a id\_assn$
unfolding *PR_CONST_def*
unfolding *check_invariant'_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def monadic_list_ex_def*
by *sepref*

lemma *check_invariant_impl_ht*:
<*table_assn M bi* ∗ *list_assn K a ai*>
check_invariant_impl ai bi
<λ*r*. *table_assn M bi* ∗ *list_assn K a ai* ∗ ↑(*r* ⟶ *check_invariant_spec* (*set a*))>$_t$
if *nofail* (*check_invariant' a M*) *L* = *dom M set a* ⊆ *L* ∀ *l* ∈ *L*. ∀ *s* ∈ *the* (*M l*). *P* (*l*, *s*)
apply (*rule cons_post_rule*)
apply (*rule check_invariant_impl.refine*[
to_hnr, *unfolded hn_refine_def hn_ctxt_def*, *simplified*, *rule_format*])
subgoal
by (*rule that*)
using *that*
apply *sep_auto*
apply (*drule Orderings.order.trans*,

```
    rule Orderings.order.trans[OF check_invariant'_refine check_invariant_correct])
  apply (sep_auto simp: pure_def split: option.splits)+
  done


definition
  check_all_pre' L' M' ≡ do {
  b1 ← RETURN (PR_CONST l₀ ∈ L');
  b2 ← RETURN (PR_CONST P' (PR_CONST l₀, PR_CONST s₀));
  let S = op_map_lookup (PR_CONST l₀) M';
  case S of None ⇒ RETURN False | Some S ⇒ do {
    xs ← SPEC (λxs. set xs = S);
    b3 ← monadic_list_ex (λs. RETURN (PR_CONST less_eq (PR_CONST s₀) s)) xs;
    b4 ← PR_CONST check_prop' L' M';
    PRINT_CHECK STR ''Start state is in state list'' b1;
    PRINT_CHECK STR ''Start state fulfills property'' b2;
    PRINT_CHECK STR ''Start state is subsumed'' b3;
    PRINT_CHECK STR ''Check property'' b4;
    RETURN (b1 ∧ b2 ∧ b3 ∧ b4)
    }
  }


definition
  check_all' L' M' ≡ do {
  b ← check_all_pre' L' M';
  if b
  then do {
    r ← SPEC (λr. r⟶ check_invariant_spec L');
    PRINT_CHECK STR ''State set invariant check'' r;
    RETURN r
  }
  else RETURN False
  }


definition check_init' where
  check_init' S ≡
  case S of None ⇒ RETURN False | Some S ⇒ do {
    xs ← SPEC (λxs. set xs = S);
    monadic_list_ex (λs. RETURN (PR_CONST less_eq (PR_CONST s₀) s)) xs
  }


lemma check_all_pre'_refine:
  check_all_pre' L M ≤ check_all_pre l₀ s₀
  unfolding check_all_pre_def check_all_pre'_def PR_CONST_def Let_def
  using check_prop_gt_SUCCEED
  apply (cases op_map_lookup l₀ M; simp add: bind_RES)
   apply (cases check_prop P')
    apply (clarsimp_all intro: less_eq_Sup simp: bind_RES check_prop_alt_def)
  apply (rule less_eq_Sup)
  subgoal for a v
```

apply *clarsimp*
apply (*rule Sup_least*)
apply *clarsimp*
supply [*refine_mono*] = *monadic_list_ex_mono monadic_list_ex_RETURN_mono*
apply *refine_mono*
apply (*simp add*: *PRINT_CHECK_def*)
done
subgoal
by (*auto dest*: *M_listD*)
done

lemma *check_all'_refine*:
check_all' L M ≤ *check_all*
supply [*refine_mono*] = *check_all_pre'_refine*
unfolding *check_all_def check_all'_def PRINT_CHECK_def* **by** *refine_mono auto*

sepref_register
PR_CONST check_invariant' :: *'k list* ⇒ (*'k*, *'b set*) *i_map* ⇒ *bool nres*
PR_CONST check_all_pre' :: *'k set* ⇒ (*'k*, *'b set*) *i_map* ⇒ *bool nres*

PR_CONST check_prop' :: *'k set* ⇒ (*'k*, *'b set*) *i_map* ⇒ *bool nres*
PR_CONST check_all' :: *'k set* ⇒ (*'k*, *'b set*) *i_map* ⇒ *bool nres*
PR_CONST check_final' :: *'k set* ⇒ (*'k*, *'b set*) *i_map* ⇒ *bool nres*
PR_CONST check_init
PR_CONST l_0 *PR_CONST* s_0

sepref_definition *check_init_impl* **is**
check_init' :: (*option_assn* (*lso_assn A*))^d →$_a$ *bool_assn*
unfolding *check_init'_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def monadic_list_ex_def*
by *sepref*

definition
L_member L' ≡ *PR_CONST* l_0 ∈ *L'*

sepref_thm *L_member_impl* **is**
RETURN o PR_CONST L_member :: (*lso_assn K*)^k →$_a$ *id_assn*
unfolding *PR_CONST_def* **unfolding** *L_member_def* **by** *sepref*

lemma *L_member_fold*:
PR_CONST l_0 ∈ *L'* ≡ *PR_CONST L_member L'*
unfolding *L_member_def PR_CONST_def* **.**

definition
lookup (*M'* :: *'k* ⇒ *'a set option*) = *op_map_lookup* (*PR_CONST* l_0) *M'*

lemma *loookp_fold*:
op_map_lookup (*PR_CONST* l_0) = *PR_CONST lookup*
unfolding *lookup_def PR_CONST_def* **..**

sepref_register *L_member lookup* :: (*'k*, *'b set*) *i_map* ⇒ *'b set option*

definition *check2* **where**
check2 b = PR_CONST check_init′ (PR_CONST lookup b)

lemma *check2_fold*:
PR_CONST check_init′ (PR_CONST lookup b) = PR_CONST check2 b
unfolding *check2_def PR_CONST_def* **..**

sepref_register *check2 :: ('k, 'b set) i_map ⇒ bool nres*

definition *check1* **where**
check1 = PR_CONST P′ (PR_CONST l_0, PR_CONST s_0)

lemma *check1_fold*:
PR_CONST check1 = PR_CONST P′ (PR_CONST l_0, PR_CONST s_0)
unfolding *check1_def PR_CONST_def* **..**

sepref_register *check1*

sepref_thm *check1_impl* **is**
uncurry0 (RETURN (PR_CONST check1)) :: $id\_assn^k \to_a id\_assn$
unfolding *PR_CONST_def* **unfolding** *check1_def* **by** *sepref*

sepref_thm *check2_impl* **is**
PR_CONST check2 :: $table\_assn^k \to_a id\_assn$
unfolding *PR_CONST_def*
unfolding *check2_def*
unfolding *PR_CONST_def*
unfolding *check_init′_def lookup_def*
unfolding *list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def monadic_list_ex_def*
by *sepref*

lemmas [*sepref_fr_rules*] =
L_member_impl.refine_raw
check1_impl.refine_raw
check2_impl.refine_raw
check_prop_impl.refine_raw

context
fixes *splitter :: 'k list ⇒ 'k list list* **and** *splitteri :: 'ki list ⇒ 'ki list list*

assumes *full_split*: *set xs = ($\bigcup$ xs ∈ set (splitter xs). set xs)*

and *same_split*:
⋀L Li. list_assn (list_assn K) (splitter L) (splitteri Li) = list_assn K L Li
begin

definition
check_invariant_all_impl L′ M′ ≡ do {
bs ← parallel_fold_map (λL. check_invariant_impl L M′) (splitteri L′);
return (list_all id bs)
}

definition
split_spec ≡ λ*L M*. *RETURN* (*list_all* (λ*x*. *RETURN True* ≤ *check_invariant′ x M*) (*splitter L*))

lemma *check_invariant_all_impl_refine*:
(*uncurry check_invariant_all_impl*, *uncurry* (*PR_CONST split_spec*))
∈ (*list_assn K*)^k ∗$_a$ *table_assn*^{k} →$_a$ *bool_assn*
unfolding *split_spec_def PR_CONST_def*
apply *sepref_to_hoare*
apply *sep_auto*
subgoal for *M Mi L Li*
unfolding *check_invariant_all_impl_def parallel_fold_map_def*
apply *sep_auto*
apply (*rule Hoare_Triple.cons_pre_rule*[*rotated*])
apply (*rule fold_map_ht2*[
where *Q* = λ*L r*. *RETURN r* ≤ *check_invariant′ L M*
and *R* = *list_assn K* **and** *A* = *table_assn M Mi* **and** *xs* = *splitter L*
])
apply (*rule Hoare_Triple.cons_post_rule*)

apply (*rule frame_rule*)
apply (*rule check_invariant_impl.refine*[*to_hnr*, *unfolded hn_refine_def hn_ctxt_def*, *simplified*, *rule_format*])
subgoal
by (*rule check_invariant′_no_fail*)
apply (*sep_auto simp*: *pure_def*; *fail*)
subgoal
unfolding *same_split* **by** *solve_entails*
apply *sep_auto*
subgoal
unfolding *list_all_length list_all2_conv_all_nth* **by** *auto*
subgoal for *x*
unfolding *list_all_length list_all2_conv_all_nth*
using *check_invariant′_correct_pre1 nres_order_simps*(*20*) *Orderings.order.trans*
by *fastforce*
subgoal
unfolding *same_split* **by** *solve_entails*
done
done

sepref_definition *check_all_pre_impl* **is**
uncurry (*PR_CONST check_all_pre′*) :: (*lso_assn K*)^k ∗$_a$ *table_assn*^{k} →$_a$ *id_assn*
unfolding *PR_CONST_def*
unfolding *check_all_pre′_def list_of_set_def*[*symmetric*]
unfolding *monadic_list_all_def monadic_list_ex_def*
by *sepref*

lemma *check_all_pre_impl_ht*:
<*table_assn b bi* ∗ *lso_assn K a ai*>
check_all_pre_impl ai bi
<λ*r*. *table_assn b bi* ∗ *lso_assn K a ai* ∗ ↑ (*RETURN r* ≤ *check_all_pre′ a b*)>$_t$
if *nofail* (*check_all_pre′ a b*)
apply (*rule cons_post_rule*)

apply (*rule check_all_pre_impl.refine*[*to_hnr*, *unfolded hn_refine_def hn_ctxt_def*, *simplified*, *rule_format*])
subgoal
by (*rule that*)
apply (*sep_auto simp*: *pure_def*)
done

definition
check_all'' L' M' ≡ *do* {
b ← *PR_CONST check_all_pre' L' M'*;
if b
then do {
xs ← *SPEC* (λ*xs*. *set xs* = *L'*);
r ← *PR_CONST split_spec xs M'*;
PRINT_CHECK STR ''State set invariant check'' r;
RETURN r
}
else RETURN False
}

sepref_register *split_spec* :: *'k list* ⇒ ((*'k*, *'b set*) *i_map*) ⇒ *bool nres*

lemmas [*sepref_fr_rules*] =
check_all_pre_impl.refine_raw
check_invariant_all_impl_refine

sepref_thm *check_all_impl* **is**
uncurry (*PR_CONST check_all''*) :: (*lso_assn K*)^k *∗$_a$ table_assn*^{k} →$_a$ *id_assn*
unfolding *PR_CONST_def*
unfolding *check_all''_def list_of_set_def*[*symmetric*]
by *sepref*

lemma *split_spec_correct*:
split_spec L' M ≤ *SPEC* (λ*r*. *r* ⟶ *check_invariant_spec_pre L'*)
if *assms*: *dom M* = *L set L'* ⊆ *L*
proof −
let *?M* = (λ*l*. *case M l of None* ⇒ {} | *Some S* ⇒ *S*)
have *RETURN True* ≤ *check_invariant' L_part M* ⟶ *check_invariant_spec_pre L_part*
if *L_part* ∈ *set* (*splitter L'*) **for** *L_part*
proof −
have *check_invariant' L_part M* ≤ *RETURN* (*check_invariant_spec_pre L_part*)
using *full_split*[*of L'*] *that assms* **by** − (*rule check_invariant'_correct_pre*, *auto*)
then show *?thesis*
using *Orderings.order.trans nres_order_simps*(*20*) **by** *metis*
qed
then have *list_all* (λ*x*. *RETURN True* ≤ *check_invariant' x M*) (*splitter L'*)
⟶ *list_all* (λ*xs*. *check_invariant_spec_pre xs*) (*splitter L'*)
using *list.pred_mono_strong* **by** *force*
moreover have ... ⟶ *check_invariant_spec_pre L'*
using *full_split*[*of L'*] **unfolding** *list_all_def check_invariant_spec_pre_def* **by** *auto*
ultimately show *?thesis*
unfolding *split_spec_def nres_order_simps* **by** *simp*
qed

lemma
  assumes *dom M = L*
  shows *check_all''_refine1*: *check_all'' L M ≤ check_all* (**is** *?A*)
    and *check_all''_refine2*: *check_all'' L M ≤ check_all' L M* (**is** *?B*)
proof −
  have ∗[*refine_mono*]: *check_invariant_spec L*
    if *RETURN True ≤ check_all_pre' L M check_invariant_spec_pre xs set xs = L* **for** *xs*
  proof −
    note *that*(*1*)
    also note *check_all_pre'_refine*
    also note *check_all_pre_correct*
    finally show *?thesis*
      using *that P'_P*
    by (*auto simp*: *check_all_pre_spec_def check_invariant_spec_pre_eq_check_invariant_spec*)
  qed
  note [*refine_mono*] = *check_all_pre'_refine split_spec_correct*
  show *?A*
    using *assms*
    unfolding *check_all''_def check_all_def PR_CONST_def*
    apply −
    apply (*rule Refine_Basic.bind_mono*, *refine_mono*)
    apply *refine_mono*
    apply (*rule specify_left*, *refine_mono*)
    apply (*rule specify_left*, *refine_mono*, (*simp*; *fail*))
    apply (*simp add*: *PRINT_CHECK_def*)
    using ∗ **by** *clarsimp*
  show *?B*
    using *assms*
    unfolding *check_all''_def check_all'_def PR_CONST_def*
    apply −
    apply (*rule Refine_Basic.bind_mono*, *refine_mono*)
    apply *refine_mono*
    apply (*rule specify_left*, *refine_mono*)
    apply *refine_mono*
    apply (*rule Orderings.order.trans*, *refine_mono*)
    using ∗ **by** *clarsimp*+
qed

sepref_thm *check_all_impl* **is**
  uncurry (*PR_CONST check_all'*) :: $(lso\_assn\ K)^k *_a table\_assn^k \to_a id\_assn$
  unfolding *PR_CONST_def*
  unfolding *check_all'_def list_of_set_def*[*symmetric*]
  unfolding *monadic_list_all_def monadic_list_ex_def*
  oops

sepref_thm *check_all_impl* **is**
  uncurry (*PR_CONST check_all'*) :: $(lso\_assn\ K)^k *_a table\_assn^k \to_a id\_assn$
  unfolding *PR_CONST_def*
  unfolding *check_all'_def check_all_pre'_def list_of_set_def*[*symmetric*]
  unfolding *monadic_list_all_def monadic_list_ex_def*
  oops

lemma *certify_unreachable_alt_def*:
certify_unreachable ≡ *do* {
b1 ← *PR_CONST check_all*;
b2 ← *PR_CONST check_final*;
RETURN (*b1* ∧ *b2*)
}
unfolding *certify_unreachable_def PR_CONST_def* .

definition *certify_unreachable′* **where**
certify_unreachable′ L′ M′ ≡ *do* {
START_TIMER ();
b1 ← *PR_CONST check_all″ L′ M′*;
SAVE_TIME STR ″Time for state space invariant check″;
START_TIMER ();
b2 ← *PR_CONST check_final′ L′ M′*;
SAVE_TIME STR ″Time to check final state predicate″;
PRINT_CHECK STR ″All check: *″ b1*;
PRINT_CHECK STR ″Target property check: *″ b2*;
RETURN (*b1* ∧ *b2*)
}

lemma *certify_unreachable′_refine*:
certify_unreachable′ L M ≤ *certify_unreachable* **if** *L* = *dom M*
unfolding *certify_unreachable′_def certify_unreachable_def PR_CONST_def check_final_alt_def*
unfolding *PRINT_CHECK_def START_TIMER_def SAVE_TIME_def*
using *check_all″_refine1 that* **by** *simp refine_mono*

sepref_register
PR_CONST check_all″ :: *′k set* ⇒ ((*′k*, *′b set*) *i_map*) ⇒ *bool nres*
PR_CONST check_final

lemmas [*sepref_fr_rules*] =
check_all_impl.refine_raw
check_final_impl.refine_raw

sepref_thm *certify_unreachable_impl′* **is**
uncurry (*PR_CONST certify_unreachable′*) :: (*lso_assn K*)^k $*_a$ *table_assn*^{k} $\rightarrow_a$ *id_assn*
unfolding *PR_CONST_def* **unfolding** *certify_unreachable′_def*
by *sepref*

lemma *certify_unreachable_correct′*:
(*uncurry0* (*certify_unreachable′ L M*), *uncurry0* (*SPEC* (λ*r*. *r* ⟶ (∄*s′*. *E*** (*l*$_0$, *s*$_0$) *s′* ∧ *F s′*))))
∈ *Id* → ⟨*bool_rel*⟩*nres_rel* **if** *L* = *dom M*
using *certify_unreachable_correct′ certify_unreachable′_refine*[*OF that*]
by (*clarsimp simp*: *pw_le_iff pw_nres_rel_iff*) *fast*

lemmas *certify_unreachable_impl′_refine* =
certify_unreachable_impl′.refine_raw[

```
    unfolded is_member_impl_def[OF pure_K left_unique_K right_unique_K]
      check_invariant_all_impl_def check_invariant_impl_def
  ]


definition certify_unreachable_new where
  certify_unreachable_new L_list M_table ≡ let
  _ = start_timer ();
  b1 = run_heap (do {M' ← M_table; check_all_pre_impl L_list M'});
  _ = save_time STR ''Time for checking basic preconditions'';
  _ = start_timer ();
  xs = run_map_heap (λLi. do {M' ← M_table; check_invariant_impl Li M'}) (splitteri L_list);
  b2 = list_all id xs;
  _ = save_time STR ''Time for state space invariant check'';
  _ = print_check STR ''State set invariant check'' b2;
  _ = start_timer ();
  b3 = run_heap (do {M' ← M_table; check_final_impl Fi copyi L_list M'});
  _ = save_time STR ''Time to check final state predicate'';
  _ = print_check STR ''All check: '' (b1 ∧ b2);
  _ = print_check STR ''Target property check: '' b3
  in b1 ∧ b2 ∧ b3

lemmas certify_unreachable_new_alt_def =
  certify_unreachable_new_def[unfolded
    check_invariant_impl_def
    check_all_pre_impl_def
    is_member_impl_def[OF pure_K left_unique_K right_unique_K]
    ]

end

concrete_definition (in −) check_all_impl
  uses Reachability_Impl.check_all_impl.refine_raw is (uncurry ?f,_)∈_

concrete_definition (in −) certify_unreachable_impl_inner
  uses Reachability_Impl.certify_unreachable_impl'_refine is (uncurry ?f,_)∈_

lemmas certify_unreachable'_impl_hnr =
  certify_unreachable_impl_inner.refine[OF Reachability_Impl_axioms]



concrete_definition (in −) certify_unreachable_impl2
  uses Reachability_Impl.certify_unreachable_new_alt_def

context
  fixes L_list and M_table
  assumes L_impl[sepref_fr_rules]:
    (uncurry0 (return L_list), uncurry0 (RETURN (PR_CONST L))) ∈ id_assn^k →_a lso_assn
K
  assumes M_impl[sepref_fr_rules]:
    (uncurry0 M_table, uncurry0 (RETURN (PR_CONST M))) ∈ id_assn^k →_a hm.hms_assn'
K (lso_assn A)
  fixes splitter :: 'k list ⇒ 'k list list and splitteri :: 'ki list ⇒ 'ki list list
```

```
  assumes full_split: set xs = (⋃ xs ∈ set (splitter xs). set xs)
      and same_split:
    ⋀L Li. list_assn (list_assn K) (splitter L) (splitteri Li) = list_assn K L Li
begin

lemmas [sepref_fr_rules] = certify_unreachable'_impl_hnr[OF full_split same_split]

sepref_register
  certify_unreachable' splitter :: 'k set ⇒ ('k, 'b set) i_map ⇒ bool nres

sepref_thm certify_unreachable_impl is
  uncurry0 (PR_CONST (certify_unreachable' splitter) (PR_CONST L) (PR_CONST M))
  :: id_assn^k →_a id_assn
  by sepref_dbg_keep

lemmas certify_unreachable_impl_refine =
  certify_unreachable_impl.refine_raw[
   unfolded PR_CONST_def is_member_impl_def[OF pure_K left_unique_K right_unique_K],
    FCOMP certify_unreachable_correct'[OF full_split same_split]
  ]

lemma certify_unreachable_new_correct:
  assumes dom M = L
  shows certify_unreachable_new splitteri L_list M_table ⟶ (∄ s'. E** (l_0, s_0) s' ∧ F s')
proof −
  have [sep_heap_rules]: <emp> M_table <λr. hm.hms_assn' K (lso_assn A) M r>_t
    by (sep_auto simp: pure_def heap: M_impl[to_hnr, unfolded hn_refine_def, simplified])
  have true_emp: true = emp * true
    by simp
  have *: emp ⟹_A lso_assn K L L_list * true
    using L_impl[to_hnr, unfolded hn_refine_def, simplified]
    apply −
    apply (drule return_htD)
    apply (erule ent_trans)
    apply (sep_auto simp: pure_def)
    done
  then have *: true ⟹_A lso_assn K L L_list * true
    by (subst true_emp) (erule ent_true_drop)
  have check_all_pre'_refine: check_all_pre' L M ≤ RETURN (check_all_pre_spec l_0 s_0)
    by (blast intro: check_all_pre_correct check_all_pre'_refine Orderings.order.trans)
  have list_assn_K_eq: list_assn K = pure (⟨the_pure K⟩list_rel)
    using pure_K by (simp add: list_assn_pure_conv[symmetric])
  have 1:
    <emp>
      do {M' ← M_table; check_all_pre_impl L_list M'}
    <λr. ↑ (r ⟶ check_all_pre_spec l_0 s_0)>_t
    apply sep_auto
    subgoal for Mi
      apply (rule cons_rule[rotated 2])
        apply (rule frame_rule[where R = true])
       apply (rule check_all_pre_impl_ht[OF full_split same_split, where a = L and b = M])
      subgoal
        using check_all_pre'_refine unfolding RETURN_def by (rule le_RES_nofailI)
      subgoal
```

```
      using ∗ by (elim ent_frame_fwd) solve_entails+
    subgoal
      apply sep_auto
      apply (drule Orderings.order.trans, rule check_all_pre'_refine)
      apply auto
      done
    done
  done
 have 3:
  <emp>
    do {M' ← M_table; check_final_impl Fi copyi L_list M'}
  <λr. ↑(r ⟶ check_final_spec)>t
  apply (sep_auto)
  subgoal for Mi
    apply (rule cons_rule[rotated 2])
      apply (rule frame_rule[where R = true])
      apply (rule check_final_impl.refine[
        OF Reachability_Impl_axioms, to_hnr, unfolded hn_refine_def hn_ctxt_def, simplified,
          rule_format, where a = L and b = M])
    subgoal
      using check_final_nofail unfolding check_final_alt_def .
    subgoal
      using ∗ by (elim ent_frame_fwd) solve_entails+
    subgoal
      apply sep_auto
      unfolding check_final_alt_def
      apply (drule Orderings.order.trans, rule check_final_correct)
      apply (sep_auto simp: pure_def)
      done
    done
  done
 have 2:
  <emp>
    do {M' ← M_table; check_invariant_impl Li M'}
 <λr. ↑(r ⟶ check_invariant_spec (set L'))>t
 if
  ∀ l∈L. ∀ s∈the (M l). P (l, s)
  Li ∈ set (splitteri L_list) (Li, L') ∈ ⟨(the_pure K)⟩list_rel set L' ⊆ L
 for Li L'
  apply sep_auto
  subgoal for Mi
    apply (rule cons_rule[rotated 2])
      apply (rule frame_rule[where R = true])
      apply (rule check_invariant_impl_ht[where bi = Mi and a = L'])
       using that check_invariant'_no_fail ‹dom M = L› by (auto simp: list_assn_K_eq
pure_def)
  done
 have
  emp ⟹A ∃A LL. ↑(list_all2 (λLi Lp. list_all2 (λxi x. (xi, x) ∈ the_pure K) Li Lp)
                  (splitteri L_list) (splitter LL)
                 ∧ set LL = L) ∗ true
  using ‹emp ⟹A lso_assn K L L_list ∗ true›
  apply (rule ent_trans)
  apply (sep_auto simp: br_def lso_assn_def hr_comp_def same_split[symmetric])
```

```
    apply (sep_auto simp: list_assn_K_eq list_assn_pure_conv)
    apply (sep_auto simp: pure_def list_rel_def; fail)
   apply simp
   done
 then obtain LL where LL: set LL = L length (splitter LL) = length (splitteri L_list)
  ∀ n < length (splitter LL). (splitteri L_list ! n, splitter LL ! n) ∈ ⟨the_pure K⟩list_rel
  by (simp add: list_rel_def list_all2_conv_all_nth)
   (smt (verit) ent_ex_preI ent_iffI ent_pure_pre_iff ent_pure_pre_iff_sng pure_assn_eq_emp_iff)
 have 2: check_invariant_spec L
  if check_all_pre_spec l₀ s₀
    list_all id (run_map_heap (λLi. M_table ⤜ check_invariant_impl Li) (splitteri L_list))
 proof −
  let ?c = (λLi. M_table ⤜ check_invariant_impl Li)
  have A: ∀ l∈L. ∀ s∈the (M l). P (l, s)
   using ‹check_all_pre_spec l₀ s₀› ‹dom M = L›
   unfolding check_all_pre_spec_def by (force intro: P'_P)
  have
   list_all2 (λL' r. r ⟶ check_invariant_spec (set L'))
    (splitter LL) (run_map_heap ?c (splitteri L_list))
   using LL
   apply −
   apply (rule hoare_triple_run_map_heapD')
   apply (rule list_all2_all_nthI, assumption)
   apply (rule 2[OF A]; simp)
   unfolding ‹set LL = L›[symmetric] by (subst (2) full_split) force
  then have list_all (λL'. check_invariant_spec (set L')) (splitter LL)
   using that(2) unfolding list_all2_conv_all_nth list_all_length by simp
  with full_split[of LL] show ?thesis
   unfolding list_all_iff check_invariant_spec_def ‹set LL = L› by auto
 qed
 show ?thesis
  apply standard
  apply (rule certify_unreachableI[rule_format])
  using hoare_triple_run_heapD[OF 1] hoare_triple_run_heapD[OF 3]
  unfolding certify_unreachable_new_def[OF full_split same_split] check_all_spec_def
  by (auto intro: 2)
qed

lemmas certify_unreachable_impl2_refine =
 certify_unreachable_new_correct[
  unfolded certify_unreachable_impl2.refine[OF Reachability_Impl_axioms full_split same_split]
  ]

end

concrete_definition (in −) certify_unreachable_impl
 uses Reachability_Impl.certify_unreachable_impl_refine is (uncurry0 ?f,_)∈_

Debugging  definition (in −)
 mk_st_string s1 s2 ≡ STR ''<'' + s1 + STR '', '' + s2 + STR ''>''


lemma [sepref_import_param]: (mk_st_string, mk_st_string) ∈ Id → Id → Id
```

by *simp*

definition (**in** −)
PRINTLN = *RETURN o println*

lemma (**in** −) [*sepref_import_param*]:
(*println*, *println*) ∈ *Id* → *Id*
by *simp*

sepref_definition (**in** −) *print_line_impl* **is**
PRINTLN :: *id_assn*^{k} →$_a$ *id_assn*
unfolding *PRINTLN_def* **by** *sepref*

sepref_register (**in** −) *PRINTLN*

lemmas [*sepref_fr_rules*] = *print_line_impl.refine*

context
fixes *show_loc* :: *'k* ⇒ *String.literal nres* **and** *show_dbm* :: *'a* ⇒ *String.literal nres*
and *show_dbm_impl* **and** *show_loc_impl*
assumes *show_dbm_impl*: (*show_dbm_impl*, *show_dbm*) ∈ A^d →$_a$ *id_assn*
assumes *show_loc_impl*: (*show_loc_impl*, *show_loc*) ∈ K^d →$_a$ *id_assn*
begin

lemma [*sepref_fr_rules*]: (*show_dbm_impl*, *PR_CONST show_dbm*) ∈ A^d →$_a$ *id_assn*
using *show_dbm_impl* **unfolding** *PR_CONST_def* **.**

lemma [*sepref_fr_rules*]: (*show_loc_impl*, *PR_CONST show_loc*) ∈ K^d →$_a$ *id_assn*
using *show_loc_impl* **unfolding** *PR_CONST_def* **.**

definition
show_st ≡ λ (*l*, *M*). *do* {
s1 ← *PR_CONST show_loc l*;
s2 ← *PR_CONST show_dbm M*;
RETURN (*mk_st_string s1 s2*)
}

sepref_register *PR_CONST show_st PR_CONST show_loc PR_CONST show_dbm*

sepref_thm *show_st_impl* **is**
PR_CONST show_st :: (*K* ×$_a$ *A*)^d →$_a$ *id_assn*
unfolding *PR_CONST_def* **unfolding** *show_st_def* **by** *sepref*

lemmas [*sepref_fr_rules*] = *show_st_impl.refine_raw*

definition *check_prop_fail* **where**
check_prop_fail L' M' = *do* {
l ← *SPEC* (λ*xs*. *set xs* = *L'*);
r ← *monadic_list_all_fail'* (λ*l*. *do* {
let S = *op_map_lookup l M'*;
case S of None ⇒ *RETURN None* | *Some S* ⇒ *do* {
xs ← *SPEC* (λ*xs*. *set xs* = *S*);
r ← *monadic_list_all_fail* (λ*s*.
RETURN (*PR_CONST P'* (*l*, *s*))

```
    ) xs;
    RETURN (case r of None ⇒ None | Some r ⇒ Some (l, r))

    //case r of None ⇒ RETURN None | Some r ⇒ RETURN (Some (l, r))
  }
  }
 ) l;
 case r of None ⇒ RETURN None |
  Some (l, M) ⇒ do {
  s ← PR_CONST show_st (l, COPY M);
  PRINTLN (STR ''Prop failed for: '');
  PRINTLN s;
  RETURN (Some (l, M))
 }
 }

sepref_thm check_prop_fail_impl is
 uncurry check_prop_fail :: (lso_assn K)^k *_a table_assn^d →_a option_assn (K ×_a A)
 unfolding check_prop_fail_def list_of_set_def[symmetric]
 unfolding monadic_list_all_fail'_alt_def monadic_list_all_fail_alt_def
 by sepref

end

definition
 check_invariant_fail L' M' ≡ do {
 l ← SPEC (λxs. set xs = L');
 monadic_list_all_fail' (λl.
 do {
  case op_map_lookup l M' of None ⇒ RETURN None  | Some as ⇒ do {
  let succs = PR_CONST succs l as;
  monadic_list_all_fail' (λ(l', xs). do {
   xs ← SPEC (λxs'. set xs' = xs);
   if xs = [] then RETURN None
   else do {
    b1 ← RETURN (l' ∈ L'); — XXX Optimize this
    if b1 then do {
     case op_map_lookup l' M' of None ⇒ RETURN (Some (Inl (Inr (l, l', xs)))) | Some ys
⇒ do {
      ys ← SPEC (λxs.  set xs  = ys);
      b2 ← monadic_list_all_fail (λx'.
       monadic_list_ex (λy. RETURN (PR_CONST less_eq x' y)) ys
      ) xs;
      case b2 of None ⇒ RETURN None | Some M ⇒ do {
       case op_map_lookup l M' of
        None ⇒ RETURN (Some (Inl (Inr (l, l', ys)))) — never used
       | Some as ⇒ do {
          as ← SPEC (λxs'. set xs' = as);
          RETURN (Some (Inr (l, as, l', M, ys)))}
        }
      }
    }
    else RETURN (Some (Inl (Inl (l, l', xs))))
```

```
    }
    }) succs
  }
  }
  ) l
}

sepref_thm check_invariant_fail_impl is
  uncurry check_invariant_fail
  :: (lso_assn K)^k *_a table_assn^k →_a
      option_assn ((K ×_a K ×_a list_assn A +_a K ×_a K ×_a list_assn A) +_a K ×_a list_assn A
×_a K ×_a A ×_a list_assn A)
  unfolding PR_CONST_def
  unfolding check_invariant_fail_def list_of_set_def[symmetric]
  unfolding monadic_list_all_fail'_alt_def monadic_list_all_fail_alt_def
  unfolding monadic_list_all_def monadic_list_ex_def
  by sepref

lemmas check_invariant_fail_impl_refine = check_invariant_fail_impl.refine_raw[
  unfolded is_member_impl_def[OF pure_K left_unique_K right_unique_K]
]

end

concrete_definition (in −) check_prop_fail_impl
  uses Reachability_Impl.check_prop_fail_impl.refine_raw is (uncurry ?f,_)∈_

concrete_definition (in −) check_invariant_fail_impl
  uses Reachability_Impl.check_invariant_fail_impl_refine is (uncurry ?f,_)∈_

export_code
  certify_unreachable_impl certify_unreachable_impl2 check_prop_fail_impl check_invariant_fail_impl
in SML module_name Test

end
theory Unreachability_Certification2
  imports
    Unreachability_Misc
    Munta_Base.Abstract_Term
    HOL−Library.Parallel
begin

hide_const (open) list_set_rel

lemma monadic_list_all_mono:
  monadic_list_all P xs ≤ monadic_list_all Q ys if list_all2 (λ x y. P x ≤ Q y) xs ys
proof −
  have nfoldli xs id (λx _. P x) a ≤ nfoldli ys id (λx _. Q x) a for a
    using that by (induction xs ys arbitrary: a; clarsimp; refine_mono)
  then show ?thesis
    unfolding monadic_list_all_def .
qed

lemma monadic_list_all_mono':
```

monadic_list_all P xs ≤ *monadic_list_all Q ys*
if (*xs*, *ys*) ∈ ⟨*R*⟩*list_rel* ⋀ *x y*. (*x*, *y*) ∈ *R* ⟹ *P x* ≤ *Q y*
using *that* **by** (*intro monadic_list_all_mono*) (*auto simp*: *list_all2_iff list_rel_def*)

lemma *monadic_list_ex_mono*:
monadic_list_ex P xs ≤ *monadic_list_ex Q ys* **if** *list_all2* (λ *x y*. *P x* ≤ *Q y*) *xs ys*
proof −
have *nfoldli xs Not* (λ*x* _. *P x*) *a* ≤ *nfoldli ys Not* (λ*x* _. *Q x*) *a* **for** *a*
using *that* **by** (*induction xs ys arbitrary*: *a*; *clarsimp*; *refine_mono*)
then show *?thesis*
unfolding *monadic_list_ex_def* **.**
qed

lemma *monadic_list_ex_mono′*:
monadic_list_ex P xs ≤ *monadic_list_ex Q ys*
if (*xs*, *ys*) ∈ ⟨*R*⟩*list_rel* ⋀ *x y*. (*x*, *y*) ∈ *R* ⟹ *P x* ≤ *Q y*
using *that* **by** (*intro monadic_list_ex_mono*) (*auto simp*: *list_all2_iff list_rel_def*)

lemma *monadic_list_all_rule′*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *Pi x* ≤ *SPEC* (λ*r*. *r* ⟷ *P x*)
shows *monadic_list_all Pi xs* ≤ *SPEC* (λ*r*. *r* ⟷ *list_all P xs*)
using *assms* **unfolding** *monadic_list_all_def*
by (*intro nfoldli_rule*[**where** *I* = λ*as bs b*. *b* = *list_all P as* ∧ *set* (*as* @ *bs*) = *set xs*]) *auto*

lemma *case_prod_mono*:
(*case x of* (*a*, *b*) ⇒ *f a b*) ≤ (*case y of* (*a*, *b*) ⇒ *g a b*)
if (*x*, *y*) ∈ *K* ×*r* *A* ⋀*ai bi a b*. (*ai*, *a*) ∈ *K* ⟹ (*bi*, *b*) ∈ *A* ⟹ *f ai bi* ≤ *g a b* **for** *x y f g*
using *that* **unfolding** *prod_rel_def* **by** *auto*

definition *list_set_rel* **where** [*to_relAPP*]:
list_set_rel R ≡ ⟨*R*⟩*list_rel O* {(*xs*, *S*). *set xs* = *S*}

lemma *list_set_relE*:
assumes (*xs*, *zs*) ∈ ⟨*R*⟩*list_set_rel*
obtains *ys* **where** (*xs*, *ys*) ∈ ⟨*R*⟩*list_rel set ys* = *zs*
using *assms* **unfolding** *list_set_rel_def* **by** *auto*

lemma *list_set_rel_Nil*[*simp*, *intro*]:
([], {}) ∈ ⟨*Id*⟩*list_set_rel*
unfolding *list_set_rel_def* **by** *auto*

lemma *specify_right*:
c ≤ *SPEC P* ⤜ *c′* **if** *P x c* ≤ *c′ x*
using *that* **by** (*auto intro*: *SUP_upper2*[**where** *i* = *x*] *simp*: *bind_RES*)

lemma *res_right*:
c ≤ *RES S* ⤜ *c′* **if** *x* ∈ *S c* ≤ *c′ x*
using *that* **by** (*auto intro*: *SUP_upper2*[**where** *i* = *x*] *simp*: *bind_RES*)

lemma *nres_relD*:
c ≤ ⇓*R a* **if** (*c*, *a*) ∈ ⟨*R*⟩*nres_rel*
using *that* **unfolding** *nres_rel_def* **by** *simp*

lemma *list_rel_setE1*:

assumes $x \in \mathit{set}\ \mathit{xs}$ $(\mathit{xs}, \mathit{ys}) \in \langle R \rangle \mathit{list\_rel}$
obtains y **where** $y \in \mathit{set}\ \mathit{ys}$ $(x, y) \in R$
using *assms* **unfolding** *list_rel_def* **by** (*auto dest*!: *list_all2_set1*)

lemma *list_rel_setE2*:
assumes $y \in \mathit{set}\ \mathit{ys}$ $(\mathit{xs}, \mathit{ys}) \in \langle R \rangle \mathit{list\_rel}$
obtains x **where** $x \in \mathit{set}\ \mathit{xs}$ $(x, y) \in R$
using *assms* **unfolding** *list_rel_def* **by** (*auto dest*!: *list_all2_set2*)

lemma *list_of_set_impl*[*autoref_rules*]:
$(\lambda \mathit{xs}.\ \mathit{RETURN}\ \mathit{xs}, \mathit{list\_of\_set}) \in \langle R \rangle \mathit{list\_set\_rel} \to \langle \langle R \rangle \mathit{list\_rel} \rangle \mathit{nres\_rel}$
unfolding *list_of_set_def* **by** *refine_rcg* (*auto elim*!: *list_set_relE intro*: *RETURN_SPEC_refine*)

lemma *case_option_mono*:
(*case x of None* $\Rightarrow$ *a* | *Some x'* $\Rightarrow$ *f x'*, *case y of None* $\Rightarrow$ *b* | *Some x'* $\Rightarrow$ *g x'*) $\in R$
if $(x, y) \in \langle S \rangle \mathit{option\_rel}$ $(a, b) \in R$ $(f, g) \in S \to R$
by (*metis fun_relD2 param_case_option' that*)

lemmas *case_option_mono'* =
case_option_mono[**where** $R = \langle R \rangle \mathit{nres\_rel}$ **for** R, *THEN nres_relD*, *THEN refine_IdD*]

lemma *bind_mono*:
assumes $m \leq \Downarrow R\ m'$
and $\bigwedge x\ y.\ (x, y) \in R \Longrightarrow f\ x \leq f'\ y$
shows *Refine_Basic.bind m f* $\leq m' \gg\!= f'$
using *assms* **by** (*force simp*: *refine_pw_simps pw_le_iff*)

lemma *list_all_split*:
assumes $\mathit{set}\ \mathit{xs} = (\bigcup \mathit{xs} \in \mathit{set}\ \mathit{split}.\ \mathit{set}\ \mathit{xs})$
shows *list_all P xs* = *list_all id* (*map* (*list_all P*) *split*)
unfolding *list_all_iff* **using** *assms* **by** *auto*

lemma *list_all_default_split*:
list_all P xs = *list_all id* (*map P xs*)
unfolding *list_all_iff* **by** *auto*

locale *Reachability_Impl_pure_base* =
Reachability_Impl_base **where** *less_eq* = *less_eq*
for *less_eq* :: $'a \Rightarrow 'a \Rightarrow \mathit{bool}$ (**infix** $\preceq$ *50*) +
fixes *get_succs* **and** *K* **and** *A* **and** *L* **and** *Li* **and** *lei*
and *Li_split* :: *'ki list list*
assumes *K_right_unique*: *single_valued K*
assumes *K_left_unique*: *single_valued* (K^{-1})
assumes *Li_L*: $(\mathit{Li}, L) \in \langle K \rangle \mathit{list\_set\_rel}$
assumes *lei_less_eq*: $(\mathit{lei}, \mathit{less\_eq}) \in A \to A \to \mathit{bool\_rel}$
assumes *get_succs_succs*[*param*]:
$(\mathit{get\_succs}, \mathit{succs}) \in K \to \langle A \rangle \mathit{list\_set\_rel} \to \langle K \times_r \langle A \rangle \mathit{list\_set\_rel} \rangle \mathit{list\_rel}$
assumes *full_split*: *set Li* = $(\bigcup \mathit{xs} \in \mathit{set}\ \mathit{Li\_split}.\ \mathit{set}\ \mathit{xs})$
begin

lemma *lei_refine*[*refine_mono*]:
$\langle \mathit{RETURN}\ (\mathit{lei}\ a\ b) \leq \mathit{RETURN}\ (a' \preceq b') \rangle$ **if** $\langle (a, a') \in A \rangle$ $\langle (b, b') \in A \rangle$
using *that* **using** *lei_less_eq* **by** *simp* (*metis pair_in_Id_conv tagged_fun_relD_both*)

definition *list_all_split* :: _ ⇒ *'ki list* ⇒ *bool* **where** [*simp*]:
list_all_split = *list_all*

definition *monadic_list_all_split* :: _ ⇒ *'ki list* ⇒ *bool nres* **where** [*simp*]:
monadic_list_all_split = *monadic_list_all*

lemmas *pure_unfolds* =
monadic_list_all_RETURN[**where** *'a* = *'ki*, *folded monadic_list_all_split_def list_all_split_def*]
monadic_list_ex_RETURN monadic_list_all_RETURN monadic_list_ex_RETURN
nres_monad1 option.case_distrib[**where** *h* = *RETURN*, *symmetric*]
if_distrib[**where** *f* = *RETURN*, *symmetric*] *prod.case_distrib*[**where** *h* = *RETURN*, *symmetric*]

lemma *list_all_split*:
list_all_split Q Li = *list_all id* (*Parallel.map* (*list_all Q*) *Li_split*)
unfolding *list_all_split_def list_all_split*[*OF full_split*, *symmetric*] *Parallel.map_def* **..**

end

locale *Reachability_Impl_pure_invariant* =
Reachability_Impl_invariant **where** *M* = *M* +
Reachability_Impl_pure_base **where** *Li_split* = *Li_split*
for *M* :: *'k* ⇒ *'a set* **and** *Li_split* :: *'ki list list*

locale *Reachability_Impl_pure_base2* =
Reachability_Impl_pure_base **where** *less_eq* = *less_eq* **and** *Li_split* = *Li_split* +
Reachability_Impl_base2 **where** *less_eq* = *less_eq*
for *less_eq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ≼ *50*) **and** *Li_split* :: *'ki list list* +
fixes *Pi* **and** *Fi*
assumes *Pi_P'*[*refine*,*param*]: (*Pi*, *P'*) ∈ *K* ×$_r$ *A* → *bool_rel*
assumes *Fi_F*[*refine*]: (*Fi*, *F*) ∈ *K* ×$_r$ *A* → *bool_rel*

locale Reachability_Impl_pure_base = Reachability_Impl_pure_invariant where less_eq = less_eq and M = M and Li_split = Li_split + Reachability_Impl_pre where less_eq = less_eq and M = M for less_eq :: 'a ⇒ 'a ⇒ bool (infix ≼ 50) and M :: 'k ⇒ 'a set and Li_split :: 'ki list list + fixes Pi and Fi assumes Pi_P'[refine,param]: (Pi, P') ∈ K ×r A → bool_rel assumes Fi_F[refine]: (Fi, F) ∈ K ×r A → bool_rel

locale *Reachability_Impl_pure* =
Reachability_Impl_common **where** *M* = *M* +
Reachability_Impl_pure_base2 +
Reachability_Impl_pre_start **where** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S*
for *M* :: *'k* ⇒ *'a set option* +
fixes *Mi l*$_0$*i s*$_0$*i*
assumes *Mi_M*[*param*]: (*Mi*, *M*) ∈ *K* → ⟨⟨*A*⟩*list_set_rel*⟩*option_rel*
assumes *l*$_0$*i_l*$_0$[*refine*,*param*]: (*l*$_0$*i*, *l*$_0$) ∈ *K*
and *s*$_0$*i_s*$_0$[*refine*,*param*,*refine_mono*]: (*s*$_0$*i*, *s*$_0$) ∈ *A*
begin

Refinement **definition** *check_invariant1 L'* ≡

```
do {
  monadic_list_all_split (λl.
    case Mi l of
      None ⇒ RETURN True
    | Some as ⇒ do {
        let succs = get_succs l as;
        monadic_list_all (λ(l', xs).
        do {
          if xs = [] then RETURN True
          else do {
            case Mi l' of
              None ⇒ RETURN False
            | Some ys ⇒ monadic_list_all (λx.
              monadic_list_ex (λy. RETURN (lei x y)) ys
            ) xs
          }
        }
        ) succs
      }
  ) L'
}
```

```
lemma Mi_M_None_iff[simp]:
  M l = None ⟷ Mi li = None if (li, l) ∈ K
proof −
  from that have (Mi li, M l) ∈ ⟨⟨A⟩list_set_rel⟩option_rel
    by parametricity
  then show ?thesis
    by auto
qed

lemma check_invariant1_refine[refine]:
  check_invariant1 L1 ≤ check_invariant L'
  if (L1, L') ∈ ⟨K⟩list_rel L = dom M set L' ⊆ L
  unfolding check_invariant1_def check_invariant_def Let_def monadic_list_all_split_def
  apply (refine_rcg monadic_list_all_mono' refine_IdI that)
  apply (clarsimp split: option.splits simp: succs_empty; safe)
   apply (simp flip: Mi_M_None_iff; fail)
  apply (refine_rcg monadic_list_all_mono')
   apply parametricity
  subgoal
    using Mi_M by (auto dest!: fun_relD1)
  apply (clarsimp split: option.splits; safe)
       apply (solves ‹
      elim list_set_relE specify_right;
      auto simp: monadic_list_all_False intro!: res_right simp flip: Mi_M_None_iff›)+
  subgoal premises prems for _ _ _ _ li _ l _ S xsi
  proof −
    have ∗: ‹li ∈ set Li ⟷ l ∈ L› if ‹(li, l) ∈ K›
      using that using Li_L K_left_unique K_right_unique
      by (auto dest: single_valuedD elim: list_rel_setE1 list_rel_setE2 elim!: list_set_relE)
    have [simp]: l ∈ L
      using prems(6) that(2) by blast
```

```
    from prems have (Mi li, M l) ∈ ⟨⟨A⟩list_set_rel⟩option_rel
      by parametricity
    with prems have (xsi, S) ∈ ⟨A⟩list_set_rel
      by auto
    then obtain xs where (xsi, xs) ∈ ⟨A⟩list_rel set xs = S
      by (elim list_set_relE)
    with prems show ?thesis
      apply (elim list_set_relE, elim specify_right)
      apply (clarsimp, safe)
       apply (solves auto)
      apply (rule specify_right[where x = xs], rule HOL.refl)
      supply [refine_mono] = monadic_list_all_mono' monadic_list_ex_mono'
      apply refine_mono
      done
  qed
  done

definition check_prop1 where
  check_prop1 L' M' = do {
  l ← RETURN L';
  monadic_list_all (λl. do {
    let S = op_map_lookup l M';
    case S of None ⇒ RETURN True | Some S ⇒ do {
      xs ← RETURN S;
      r ← monadic_list_all (λs.
        RETURN (Pi (l, s))
      ) xs;
      RETURN r
    }
    }
  ) l
  }

lemma check_prop1_refine:
  (check_prop1, check_prop') ∈ ⟨K⟩list_set_rel → (K → ⟨⟨A⟩list_set_rel⟩option_rel) → ⟨Id⟩nres_rel
  supply [refine] =
   list_of_set_impl[THEN fun_relD, THEN nres_relD] monadic_list_all_mono' case_option_mono'
  supply [refine_mono] =
    monadic_list_all_mono' list_of_set_impl[THEN fun_relD, THEN nres_relD]
  unfolding check_prop1_def check_prop'_def list_of_set_def[symmetric] Let_def op_map_lookup_def
  apply refine_rcg
   apply assumption
  apply (refine_rcg refine_IdI, assumption)
   apply parametricity
  apply (rule bind_mono)
   apply refine_mono+
  using Pi_P'
  apply (auto simp add: prod_rel_def dest!: fun_relD)
  done

lemma [refine]:
  (Mi l₀i ≠ None, l₀ ∈ L) ∈ bool_rel if L = dom M
  using that by (auto simp: Mi_M_None_iff[symmetric, OF l₀i_l₀])
```

lemma [*refine*]:
(*Pi* ($l_0 i$, $s_0 i$), P' (l_0, s_0)) ∈ *bool_rel*
by *parametricity*

definition
check_all_pre1 ≡ *do* {
b1 ← *RETURN* (*Mi* $l_0 i$ ≠ *None*);
b2 ← *RETURN* (*Pi* ($l_0 i$, $s_0 i$));
case Mi $l_0 i$ *of*
None ⇒ *RETURN False*
| *Some xs* ⇒ *do* {
b3 ← *monadic_list_ex* (λ*s*. *RETURN* (*lei* $s_0 i$ *s*)) *xs*;
b4 ← *check_prop1 Li Mi*;
RETURN (*b1* ∧ *b2* ∧ *b3* ∧ *b4*)
}
}

lemma *check_all_pre1_refine*[*refine*]:
(*check_all_pre1*, *check_all_pre* l_0 s_0) ∈ ⟨*bool_rel*⟩*nres_rel* **if** *L* = *dom M*
proof (*cases M* l_0)
case *None*
then show *?thesis*
using *that check_prop_gt_SUCCEED* $l_0 i\_l_0$ **unfolding** *check_all_pre1_def check_all_pre_def*
by (*refine_rcg*) (*simp flip*: *Mi_M_None_iff add*: *bind_RES*; *cases check_prop P'*; *simp*)
next
case (*Some S*)
have (*Mi* $l_0 i$, *M* l_0) ∈ ⟨⟨*A*⟩*list_set_rel*⟩*option_rel*
by *parametricity*
with ‹*M* l_0 = _› **obtain** *xs ys* **where** *Mi* $l_0 i$ = *Some xs* (*xs*, *ys*) ∈ ⟨*A*⟩*list_rel set ys* = *S*
unfolding *option_rel_def* **by** (*auto elim*: *list_set_relE*)
with ‹*M* l_0 = _› **show** *?thesis*
unfolding *check_all_pre1_def check_all_pre_def*
apply (*refine_rcg that*; *simp*)
supply [*refine_mono*] =
monadic_list_ex_mono' monadic_list_ex_RETURN_mono specify_right HOL.refl
check_prop1_refine[*THEN fun_relD*, *THEN fun_relD*, *THEN nres_relD*, *THEN refine_IdD*,
of Li L Mi M, *unfolded check_prop_alt_def*]
Li_L Mi_M
apply *refine_mono*
done
qed

definition
check_all1 ≡ *do* {
b ← *check_all_pre1*;
if b
then do {
r ← *check_invariant1 Li*;
PRINT_CHECK STR ''State set invariant check'' *r*;
RETURN r
}
else RETURN False
}

lemma *check_all1_correct*[*refine*]:
check_all1 ≤ *SPEC* (λ*r*. *r* ⟶ *check_all_pre_spec* l_0 s_0 ∧ *check_invariant_spec L*) **if** *L* = *dom M*
proof −
from *check_all_pre1_refine*[*THEN nres_relD*, *OF that*] **have** *check_all_pre1* ≤ *check_all_pre* l_0 s_0
by (*rule refine_IdD*)
also note *check_all_pre_correct*
finally have [*refine*]: *check_all_pre1* ≤ *SPEC* (λ*r*. *r* ⟶ *check_all_pre_spec* l_0 s_0)
by (*rule Orderings.order.trans*) *simp*
obtain *L′* **where** (*Li*, *L′*) ∈ ⟨*K*⟩*list_rel set L′* = *L*
using *Li_L* **by** (*elim list_set_relE*)
note *check_invariant1_refine*
also note *check_invariant_correct*
also have [*refine*]: *check_invariant1 Li* ≤ *SPEC* (λ*r*. *r* ⟶ *check_invariant_spec L*)
if *check_all_pre_spec* l_0 s_0
apply (*subst* (*2*) ‹_ = *L*›[*symmetric*])
apply (*rule calculation*)
using *Li_L* ‹(*Li*, *L′*) ∈ _› *that* **unfolding** *check_all_pre_spec_def*
by (*auto simp*: ‹_ = *L*› ‹*L* = _› *dest*: *P′_P*)
show *?thesis*
unfolding *check_all1_def PRINT_CHECK_def comp_def* **by** (*refine_vcg*; *simp*)
qed

definition
check_final1 L′ = *do* {
monadic_list_all (λ*l*. *do* {
case op_map_lookup l Mi of None ⇒ *RETURN True* | *Some xs* ⇒ *do* {
monadic_list_all (λ*s*.
RETURN (¬ *PR_CONST Fi* (*l*, *s*))
) *xs*
}
}
) *L′*
}

lemma *check_final1_refine*:
check_final1 Li ≤ *check_final′ L M*
using *Li_L* **apply** (*elim list_set_relE*)
unfolding *check_final1_def check_final′_def*
apply (*erule specify_right*)
supply [*refine_mono*] = *monadic_list_all_mono′*
apply *refine_mono*
apply (*clarsimp split*: *option.splits*)
apply (*refine_rcg monadic_list_all_mono′*)
apply (*simp flip*: *Mi_M_None_iff*; *fail*)
subgoal premises *prems* **for** _ *li l xsi S*
proof −
from ‹(*li*, *l*) ∈ *K*› **have** (*Mi li*, *M l*) ∈ ⟨⟨*A*⟩*list_set_rel*⟩*option_rel*
by *parametricity*
with *prems* **have** (*xsi*, *S*) ∈ ⟨*A*⟩*list_set_rel*
by *auto*
then obtain *xs* **where** (*xsi*, *xs*) ∈ ⟨*A*⟩*list_rel set xs* = *S*

```
    by (elim list_set_relE)
  then show ?thesis
    using Fi_F ‹(li, l) ∈ K›
    by (refine_rcg monadic_list_all_mono' specify_right) (auto dest!: fun_relD)
 qed
 done

definition
 certify_unreachable1 = do {
  b1 ← check_all1;
  b2 ← check_final1 Li;
  RETURN (b1 ∧ b2)
 }

lemma certify_unreachable1_correct:
 certify_unreachable1 ≤ SPEC (λr. r ⟶ check_all_spec ∧ check_final_spec) if L = dom M
proof −
 note check_final1_refine[unfolded check_final_alt_def]
 also note check_final_correct
 finally have [refine]: check_final1 Li ≤ SPEC (λr. r = check_final_spec) .
 show ?thesis
  unfolding certify_unreachable1_def check_all_spec_def by (refine_vcg that; fast)
qed

Synthesizing a pure program via rewriting  lemma check_final1_alt_def:
 check_final1 L' = RETURN (list_all_split
  (λl. case op_map_lookup l Mi of None ⇒ True | Some xs ⇒ list_all (λs. ¬ Fi (l, s)) xs) L')
 unfolding check_final1_def
 unfolding monadic_list_all_RETURN[symmetric] list_all_split_def
 by (fo_rule arg_cong2, intro ext)
  (auto split: option.splits simp: monadic_list_all_RETURN[symmetric])

concrete_definition check_final_impl
 uses check_final1_alt_def is _ = RETURN ?f

schematic_goal check_prop1_alt_def:
 check_prop1 L' M' ≡ RETURN ?f
 unfolding check_prop1_def pure_unfolds Let_def .

concrete_definition check_prop_impl uses check_prop1_alt_def is _ ≡ RETURN ?f

schematic_goal check_all_pre1_alt_def:
 check_all_pre1 ≡ RETURN ?f
 unfolding check_all_pre1_def check_prop_impl.refine pure_unfolds .

concrete_definition check_all_pre_impl uses check_all_pre1_alt_def is _ ≡ RETURN ?f

schematic_goal check_invariant1_alt_def:
 check_invariant1 Li ≡ RETURN ?f
 unfolding check_invariant1_def Let_def pure_unfolds .

concrete_definition check_invariant_impl uses check_invariant1_alt_def is _ ≡ RETURN
?f
```

schematic_goal *certify_unreachable1_alt_def*:
certify_unreachable1 ≡ *RETURN ?f*
unfolding
certify_unreachable1_def check_all1_def check_final_impl.refine check_all_pre_impl.refine
check_invariant_impl.refine
pure_unfolds PRINT_CHECK_def comp_def short_circuit_conv **.**

concrete_definition *certify_unreachable_impl_pure1*
uses *certify_unreachable1_alt_def* **is** _ ≡ *RETURN ?f*

This is where we add parallel execution:

schematic_goal *certify_unreachable_impl_pure1_alt_def*:
certify_unreachable_impl_pure1 ≡ *?f*
unfolding *certify_unreachable_impl_pure1_def*
apply (*abstract_let check_invariant_impl check_invariant*)
apply (*abstract_let check_final_impl Li check_final*)
apply (*abstract_let check_all_pre_impl check_all_pre_impl*)
apply (*time_it STR ''Time for state set preconditions check'' check_all_pre_impl*)
apply (*time_it STR ''Time for state space invariant check'' check_invariant_impl*)
apply (*time_it STR ''Time to check final state predicate'' check_final_impl Li*)
unfolding
check_invariant_impl_def check_all_pre_impl_def
check_prop_impl_def check_final_impl_def
list_all_split
apply (*subst list_all_default_split*[**where** *xs* = *Li*, *folded Parallel.map_def*])
.

concrete_definition (**in** −) *certify_unreachable_impl_pure*
uses *Reachability_Impl_pure.certify_unreachable_impl_pure1_alt_def* **is** _ ≡ *?f*

sublocale *correct*: *Reachability_Impl_correct* **where**
M = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *abs_s* ‘ *S*
apply *standard*
oops

end

locale *Reachability_Impl_pure_correct* =
Reachability_Impl_pure **where** *M* = *M* +
Reachability_Impl_correct **where** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S* **for** *M*
begin

theorem *certify_unreachable_impl_pure_correct*:
certify_unreachable_impl_pure get_succs Li lei Li_split Pi Fi Mi $l_0 i$ $s_0 i$
⟶ (∄ *s'*. E^{**} (l_0, s_0) *s'* ∧ *F s'*)
if *L* = *dom M*
using *certify_unreachable1_correct that*
unfolding
certify_unreachable_impl_pure1.refine
certify_unreachable_impl_pure.refine[*OF Reachability_Impl_pure_axioms*]
using *certify_unreachableI* **by** *simp*

end

```
  }) L'

lemma Mi_M_None_iff[simp]:
  M l = None ⟷ Mi li = None if (li, l) ∈ K
proof −
  from that have (Mi li, M l) ∈ ⟨⟨A ×r Id⟩list_set_rel⟩option_rel
    by parametricity
  then show ?thesis
    by auto
qed

lemma (in −) list_set_rel_singletonI[param]:
  assumes (ai, a) ∈ A
  shows ([ai], {a}) ∈ ⟨A⟩list_set_rel
  unfolding list_set_rel_def
  using assms by (auto intro: relcompI[where b = [a]])

lemma check_invariant1_refine[refine]:
  check_invariant_buechi' L1 ≤ check_invariant_buechi buechi_prop L'
  if (L1, L') ∈ ⟨K⟩list_rel L = dom M set L' ⊆ L
 unfolding check_invariant_buechi'_def check_invariant_buechi_def Let_def monadic_list_all_split_def
  apply (refine_rcg monadic_list_all_mono' refine_IdI that)
  apply (clarsimp split: option.splits; safe)

   apply (simp add: RES_sng_eq_RETURN; fail)
  apply (simp flip: Mi_M_None_iff; fail)

  subgoal premises prems for ki k xs xsi
  proof −
   from prems fun_relD1[OF Mi_M] have (Mi ki, M k) ∈ ⟨⟨A ×r nat_rel⟩list_set_rel⟩option_rel
     by force
   with prems have (xsi, xs) ∈ ⟨A ×r nat_rel⟩list_set_rel
     by force
   from list_set_relE[OF this] obtain ys where
     (xsi, ys) ∈ ⟨A ×r nat_rel⟩list_rel set ys = xs .
   then show ?thesis
     using ‹(ki, k) ∈ _›
     apply −
     apply (erule specify_right)

     apply (refine_rcg monadic_list_all_mono')
      apply assumption
     apply (solves ‹parametricity, auto›)

     apply (clarsimp split: option.splits split del: if_split; safe)
      apply (solves ‹
        elim list_set_relE specify_right;
        auto simp: monadic_list_all_False intro!: res_right simp flip: Mi_M_None_iff›)+
     subgoal premises prems for si i s ki' xsi' k' xs' zs zsi
     proof −
       have k' ∈ L
         using prems(6) that(2) by blast
       from ‹(ki, k) ∈ K› ‹(si, s) ∈ A› have Fi_F: Fi (ki, si) ⟷ F (k, s)
         using Fi_F by (force dest: fun_relD1)
```

```
      from prems fun_relD1[OF Mi_M] have (Mi ki', M k') ∈ ⟨⟨A ×r nat_rel⟩list_set_rel⟩option_rel
          by force
        with prems have (zsi, zs) ∈ ⟨A ×r nat_rel⟩list_set_rel
          by force
        from list_set_relE[OF this] obtain ws where
          (zsi, ws) ∈ ⟨A ×r nat_rel⟩list_rel set ws = zs .
        moreover from list_set_relE[OF ‹_ ∈ ⟨A⟩list_set_rel›] obtain ys' where
          (xsi', ys') ∈ ⟨A⟩list_rel set ys' = xs' .
        ultimately show ?thesis
          using ‹k' ∈ L› lei_refine Fi_F
          unfolding buechi_prop_def
         supply [refine_mono] = monadic_list_all_mono' monadic_list_ex_mono' specify_right
HOL.refl
          by (refine_mono | auto)+
      qed
      done
  qed
  done

definition check_prop1 where
  check_prop1 L' M' = do {
  l ← RETURN L';
  monadic_list_all (λl. do {
    let S = op_map_lookup l M';
    case S of None ⇒ RETURN True | Some S ⇒ do {
      xs ← RETURN S;
      r ← monadic_list_all (λ(s, _).
        RETURN (Pi (l, s))
      ) xs;
      RETURN r
    }
    }
  ) l
  }

definition
  check_init1 l0i s0i ≡ do {
  b1 ← RETURN (Mi l0i ≠ None);
  b2 ← RETURN (Pi (l0i, s0i));
  case Mi l0i of
    None ⇒ RETURN False
  | Some xs ⇒ do {
      b3 ← monadic_list_ex (λ(s, _). RETURN (lei s0i s)) xs;
      RETURN (b1 ∧ b2 ∧ b3)
    }
  }

definition check_prop' where
  check_prop' L' M' = do {
  l ← SPEC (λxs. set xs = L');
  monadic_list_all (λl. do {
    let S = op_map_lookup l M';
    case S of None ⇒ RETURN True | Some S ⇒ do {
      xs ← SPEC (λxs. set xs = S);
```

```
    r ← monadic_list_all (λ(s, _).
      RETURN (PR_CONST P' (l, s))
    ) xs;
    RETURN r
  }
  }
) l
}

definition
  check_all_pre1 ≡ do {
    b1 ← monadic_list_all (λ(l₀, s₀). check_init1 l₀ s₀) initsi;
    b2 ← check_prop1 Li Mi;
    RETURN (b1 ∧ b2)
  }

lemma check_prop1_refine:
  (check_prop1, check_prop')
    ∈ ⟨K⟩list_set_rel → (K → ⟨⟨A ×ᵣ Id⟩list_set_rel⟩option_rel) → ⟨Id⟩nres_rel
  supply [refine] =
   list_of_set_impl[THEN fun_relD, THEN nres_relD] monadic_list_all_mono' case_option_mono'
  supply [refine_mono] =
    monadic_list_all_mono' list_of_set_impl[THEN fun_relD, THEN nres_relD]
 unfolding check_prop1_def check_prop'_def list_of_set_def[symmetric] Let_def op_map_lookup_def
  apply refine_rcg
   apply assumption
  apply (refine_rcg refine_IdI, assumption)
   apply parametricity
  apply (rule bind_mono)
   apply refine_mono+
  using Pi_P' by (auto simp add: prod_rel_def dest!: fun_relD)

sublocale reachability:
  Reachability_Impl_pre where M = image fst o (λx. case M x of None ⇒ {} | Some S ⇒ S) ..

lemma check_prop_gt_SUCCEED:
  reachability.check_prop P' > SUCCEED if L = dom M
  using finite that unfolding reachability.check_prop_def
  by (intro monadic_list_all_gt_SUCCEED bind_RES_gt_SUCCEED_I)
     (auto split: option.split simp: dom_def intro!: finite_list)

lemma check_prop_alt_def:
  check_prop' L M = reachability.check_prop P' if L = dom M
  unfolding reachability.check_prop_def check_prop'_def
  apply (fo_rule arg_cong2, simp, fo_rule arg_cong)
  apply (clarsimp split: option.split simp: bind_RES)
  apply (rule ext)
  apply (clarsimp split: option.split simp: bind_RES)
  apply (fo_rule arg_cong)
  unfolding pure_unfolds image_def
  subgoal premises prems
proof −
  {
    fix l :: ‹'k› and xs:: ‹('a × nat) list›
```

```
    assume ‹M l = Some (set xs)›
    have ‹∃x. set x = {y. ∃x∈set xs. y = fst x} ∧
      list_all (λ(s, _). P' (l, s)) xs = list_all (λs. P' (l, s)) x›
      by (rule exI[where x = map fst xs]) (auto simp: list_all_iff)
  }
  moreover
  {
    fix l :: ‹'k› and S :: ‹('a × nat) set› and ys :: ‹'a list›
    assume ‹M l = Some S› and ‹set ys = {y. ∃x∈S. y = fst x}›
    with ‹L = _› finite have finite S
      by force
    then obtain xs where set xs = S
      by atomize_elim (rule finite_list)
    with ‹set ys = _› have
      ‹∃x. set x = S ∧ list_all (λs. P' (l, s)) ys = list_all (λ(s, _). P' (l, s)) x›
      by (intro exI[where x = xs]) (auto simp: list_all_iff)
  }
  ultimately show ?thesis
    using prems by (safe; simp)
qed
done

lemma check_init1_refine:
  (check_init1, reachability.check_init) ∈ K → A → ⟨bool_rel⟩nres_rel if L = dom M
proof (intro fun_relI)
  fix l_0 l_0i s_0 s_0i
  assume l_0i_l_0: (l_0i, l_0) ∈ K and s_0i_s_0: (s_0i, s_0) ∈ A
  then have [refine]:
    (Pi (l_0i, s_0i), P' (l_0, s_0)) ∈ bool_rel
    by parametricity
  have [refine]:
    (Mi l_0i ≠ None, l_0 ∈ L) ∈ bool_rel
    using that by (auto simp: Mi_M_None_iff[symmetric, OF ‹_ ∈ K›])
  show (check_init1 l_0i s_0i, reachability.check_init l_0 s_0) ∈ ⟨bool_rel⟩nres_rel
  proof (cases M l_0)
    case None
    then show ?thesis
      using that l_0i_l_0 unfolding check_init1_def reachability.check_init_def
      by refine_rcg (simp flip: Mi_M_None_iff add: bind_RES; simp)
  next
    case (Some S)
    from ‹_ ∈ K› have (Mi l_0i, M l_0) ∈ ⟨⟨A ×_r Id⟩list_set_rel⟩option_rel
      by parametricity
    with ‹M l_0 = _› obtain xs ys where ∗:
      Mi l_0i = Some xs (xs, ys) ∈ ⟨A ×_r Id⟩list_rel set (map fst ys) = fst ‘ S
      unfolding option_rel_def by (auto elim: list_set_relE)
    then have (xs, map fst ys) ∈ ⟨{((x, i), y). (x, y) ∈ A}⟩list_rel
      unfolding list_rel_def by (auto elim: list_all2_induct)
    with ∗ ‹M l_0 = _› show ?thesis
      unfolding check_init1_def reachability.check_init_def
      apply (refine_rcg that; simp)
      supply [refine_mono] = monadic_list_ex_mono' specify_right Li_L Mi_M
      apply refine_mono
      using lei_refine s_0i_s_0 apply auto
```

```
    done
  qed
qed

lemma check_all_pre1_correct[refine]:
  check_all_pre1 ≤ SPEC (λr. r ⟶ check_all_pre_spec1 inits)  if L = dom M
proof −
  note check_init1_refine[THEN fun_relD, THEN fun_relD, THEN nres_relD, OF that]
  also note reachability.check_init_correct
  finally have [refine_mono, refine]:
    check_init1 l₀i s₀i ≤ SPEC (λr. r = reachability.check_init_spec l₀ s₀)
    if l₀i_l₀: (l₀i, l₀) ∈ K and s₀i_s₀: (s₀i, s₀) ∈ A for l₀ l₀i s₀ s₀i
    using that by (force intro: Orderings.order.trans)
  from initsi_inits obtain inits' where
    inits = set inits' and [refine_mono]: (initsi, inits') ∈ ⟨K ×ᵣ A⟩list_rel
    unfolding list_set_rel_def by auto
  note [refine_mono] = case_prod_mono monadic_list_all_mono'

  have [refine]: monadic_list_all (λ(l₀, s₀). check_init1 l₀ s₀) initsi
    ≤ SPEC (λr. r = list_all (λ(l₀, s₀). reachability.check_init_spec l₀ s₀) inits')
    by (rule Orderings.order.trans, refine_mono) (refine_vcg monadic_list_all_rule; auto)
   note check_prop1_refine[THEN fun_relD, THEN fun_relD, THEN nres_relD, THEN re-
fine_IdD,
      OF Li_L Mi_M, unfolded check_prop_alt_def[OF ‹L = _›]]
  also note reachability.check_prop_correct
  also note [refine] = calculation
  show ?thesis
    unfolding check_all_pre1_def
    by refine_vcg
      (fastforce simp:
       check_all_pre_spec1_def reachability.check_init_spec_def list_all_iff ‹inits = set inits'›)
qed

definition
  PRINT_CHECK' s b ≡ RETURN (let b = b; x = print_check s b in b)

definition
  certify_no_buechi_run ≡ do {
    b ← check_all_pre1;
    if b
    then do {
      r ← check_invariant_buechi' Li;
      PRINT_CHECK' STR ''State space invariant check'' r
    }
    else RETURN False
  }

lemma check_all1_refine:
  certify_no_buechi_run ≤ check_buechi inits if L = dom M
proof −
  obtain L' where aux: (Li, L') ∈ ⟨K⟩list_rel set L' = L
    using Li_L by (elim list_set_relE)
  note check_invariant1_refine
  also note check_invariant_buechi_correct
```

also note [*refine*] = *calculation*
have [*refine*]:
check_invariant_buechi' Li ≤ *SPEC* (λ*r*. *r* ⟶ *check_invariant_buechi_spec buechi_prop L*)
if ∀ *l*∈*L*. ∀ (*s*, _)∈*case M l of None* ⇒ {} | *Some S* ⇒ *S*. *P* (*l*, *s*)
using *aux that* ‹*L* = *dom M*› **by** *refine_vcg simp+*
show *?thesis*
using *P'_P that* **unfolding** *certify_no_buechi_run_def check_buechi_def PRINT_CHECK'_def comp_def*
by (*refine_mono*; *refine_vcg*;
unfold check_all_pre_spec1_def; *fastforce split*: *prod.splits simp*: *check_all_pre_spec1_def*)
qed

Synthesizing a pure program via rewriting **schematic_goal** *check_prop1_alt_def*:
check_prop1 L' M' ≡ *RETURN ?f*
unfolding *check_prop1_def pure_unfolds Let_def* **.**

concrete_definition *check_prop_impl* **uses** *check_prop1_alt_def* **is** _ ≡ *RETURN ?f*

lemma *check_all_pre1_printing*:
check_all_pre1 ≡ *do* {
b1 ← *monadic_list_all* (λ(*l*$_0$, *s*$_0$). *check_init1 l*$_0$ *s*$_0$) *initsi*;
b1 ← *PRINT_CHECK' STR ''Initial state check'' b1*;
b2 ← *check_prop1 Li Mi*;
b2 ← *PRINT_CHECK' STR ''State set preconditions check'' b2*;
RETURN (*b1* ∧ *b2*)
}
unfolding *check_all_pre1_def PRINT_CHECK'_def Let_def pure_unfolds* **.**

schematic_goal *check_all_pre1_alt_def*:
check_all_pre1 ≡ *RETURN ?f*
unfolding
check_all_pre1_printing check_init1_def check_prop_impl.refine pure_unfolds PRINT_CHECK'_def
.

concrete_definition *check_all_pre_impl* **uses** *check_all_pre1_alt_def* **is** _ ≡ *RETURN ?f*

schematic_goal *check_invariant_buechi'_alt_def*:
check_invariant_buechi' Li ≡ *RETURN ?f*
unfolding *check_invariant_buechi'_def Let_def pure_unfolds* **.**

lemma *PRINT_CHECK_unfold*:
PRINT_CHECK s x = *RETURN* (*print_check s x*)
unfolding *PRINT_CHECK_def comp_def* **..**

concrete_definition *check_invariant_buechi_impl*
uses *check_invariant_buechi'_alt_def* **is** _ ≡ *RETURN ?f*

schematic_goal *certify_no_buechi_run_alt_def*:
certify_no_buechi_run ≡ *RETURN ?f*
unfolding
certify_no_buechi_run_def check_all_pre_impl.refine
check_invariant_buechi_impl.refine

PRINT_CHECK'_def pure_unfolds
short_circuit_conv
.

concrete_definition *certify_no_buechi_run_pure1*
uses *certify_no_buechi_run_alt_def* **is** _ ≡ *RETURN ?f*

This is where we add parallel execution:

schematic_goal *certify_no_buechi_run_pure1_alt_def*:
certify_no_buechi_run_pure1 ≡ *?f*
unfolding *certify_no_buechi_run_pure1_def*
apply (*abstract_let check_invariant_buechi_impl check_invariant*)
apply (*abstract_let check_all_pre_impl check_all_pre_impl*)
apply (*time_it STR ''Time for state set preconditions check'' check_all_pre_impl*)
apply (*time_it STR ''Time for state space invariant check'' check_invariant_buechi_impl*)
unfolding
check_invariant_buechi_impl_def check_all_pre_impl_def
check_prop_impl_def
list_all_split
apply (*subst list_all_default_split*[**where** *xs* = *Li*, *folded Parallel.map_def*])
.

concrete_definition (**in** −) *certify_no_buechi_run_pure*
uses *Buechi_Impl_pure.certify_no_buechi_run_pure1_alt_def* **is** _ ≡ *?f*

end

locale *Buechi_Impl_pure_correct* =
Buechi_Impl_pure **where** *M* = *M* +
Buechi_Impl_correct **where** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S* **for** *M*
begin

theorem *certify_no_buechi_run_correct*:
certify_no_buechi_run ≤ *SPEC* (λ*r*. *r* ⟶ (∄ *xs* l_0 s_0.
(l_0, s_0) ∈ *inits* ∧ *Graph_Defs.run E* ((l_0, s_0) ## *xs*) ∧ *alw* (*ev* (*holds F*)) ((l_0, s_0) ## *xs*)))
if *L* = *dom M*
by (*rule Orderings.order.trans*[*OF check_all1_refine*[*OF that*]],
refine_vcg that check_buechi_correct')
(*auto intro*: *no_buechi_run*)

theorem *certify_no_buechi_run_impl_pure_correct*:
certify_no_buechi_run_pure get_succs Li lei Li_split Pi Fi Mi initsi ⟶ (∄ *xs* l_0 s_0.
(l_0, s_0) ∈ *inits* ∧ *Graph_Defs.run E* ((l_0, s_0) ## *xs*) ∧ *alw* (*ev* (*holds F*)) ((l_0, s_0) ## *xs*))
if *L* = *dom M*
using *certify_no_buechi_run_correct that*
unfolding
certify_no_buechi_run_pure1.refine
certify_no_buechi_run_pure.refine[*OF Buechi_Impl_pure_axioms*]
by *simp*

end

locale *Reachability_Impl_imp_to_pure_base* = *Certification_Impl_base*

```
  where K = K and A = A
  for K :: 'k ⇒ ('ki :: {hashable,heap}) ⇒ assn and A :: 's ⇒ ('si :: heap) ⇒ assn
  +
  fixes to_state :: 's1 ⇒ 'si Heap and from_state :: 'si ⇒ 's1 Heap
    and to_loc :: 'k1 ⇒ 'ki and from_loc :: 'ki ⇒ 'k1
  fixes lei
  fixes K_rel and A_rel
  fixes L_list :: 'ki list and Li :: 'k1 list and L :: 'k set and L' :: 'k list
  fixes Li_split :: 'k1 list list
  assumes Li: (L_list, L') ∈ ⟨the_pure K⟩list_rel (Li, L') ∈ ⟨K_rel⟩list_rel set L' = L
  assumes to_state_ht: (s1, s) ∈ A_rel ⟹ <emp> to_state s1 <λsi. A s si>
  assumes from_state_ht: <A s si> from_state si <λs'. ↑((s', s) ∈ A_rel)>t
  assumes from_loc: (li, l) ∈ the_pure K ⟹ (from_loc li, l) ∈ K_rel
  assumes to_loc: (l1, l) ∈ K_rel ⟹ (to_loc l1, l) ∈ the_pure K
  assumes K_rel: single_valued K_rel single_valued (K_rel⁻¹)
  assumes lei_less_eq: (lei, (≼)) ∈ A_rel → A_rel → bool_rel
  assumes full_split: set Li = (⋃ xs ∈ set Li_split. set xs)
begin

definition
  get_succs l xs ≡
    do {
      let li = to_loc l;
      xsi ← Heap_Monad.fold_map to_state xs;
      r ← succsi li xsi;
      Heap_Monad.fold_map
        (λ(li, xsi). do {xs ← Heap_Monad.fold_map from_state xsi; return (from_loc li, xs)}) r
    }

lemma get_succs:
  (run_heap oo get_succs, succs)
  ∈ K_rel → ⟨A_rel⟩list_set_rel → ⟨K_rel ×r ⟨A_rel⟩list_set_rel⟩list_rel
proof −
  {
    fix l :: ‹'k› and l1 :: ‹'k1› and xs :: ‹'s1 list› and S :: ‹'s set›
    assume ‹(l1, l) ∈ K_rel› ‹(xs, S) ∈ ⟨A_rel⟩list_set_rel›
    then obtain ys where ys: (xs, ys) ∈ ⟨A_rel⟩list_rel set ys = S
      by (elim list_set_relE)
    have 1: K = pure (the_pure K)
      using pure_K by auto
    let ?li = to_loc l1
   have <emp> get_succs l1 xs <λr. ↑((r, succs l S) ∈ ⟨K_rel ×r ⟨A_rel⟩list_set_rel⟩list_rel)>t
      unfolding get_succs_def
      apply sep_auto

       apply (rule Hoare_Triple.cons_pre_rule[rotated])
        apply (rule fold_map_ht3[where A = true and R = pure A_rel and Q = A and xs =
ys])
      apply (sep_auto heap: to_state_ht simp: pure_def; fail)
       apply (unfold list_assn_pure_conv, sep_auto simp: pure_def ys; fail)
      apply sep_auto

       apply (rule Hoare_Triple.cons_pre_rule[rotated], rule frame_rule[where R = true])
        apply (rule succsi[to_hnr, unfolded hn_refine_def hn_ctxt_def, simplified, of S _ l ?li])
```

subgoal
using *ys* ‹(*l1*, *l*) ∈ *K_rel*› **unfolding** *lso_assn_def hr_comp_def br_def* ‹*set ys* = _›[*symmetric*]
by (*subst 1*) (*sep_auto simp*: *pure_def to_loc*)

apply (*sep_auto simp*: *invalid_assn_def*)
apply (*rule cons_rule*[*rotated 2*])
apply (*rule frame_rule*)
apply (*rule fold_map_ht1* [**where**
A = *true* **and** *R* = (*K* $\times_a$ *lso_assn A*) **and** *xs* = *succs l S* **and**
Q = λ*x xi*. (*xi*, *x*) ∈ *K_rel* $\times_r$ ⟨*A_rel*⟩*list_set_rel*
])
subgoal
unfolding *lso_assn_def*
apply (*subst 1*, *subst pure_def*)
apply (*sep_auto simp*: *prod_assn_def hr_comp_def br_def split*: *prod.splits*)

apply (*rule Hoare_Triple.cons_pre_rule*[*rotated*])
apply (*rule fold_map_ht1* [**where** *A* = *true* **and** *R* = *A* **and** *Q* = λ*l l1*. (*l1*, *l*) ∈ *A_rel*])
apply (*rule cons_rule*[*rotated 2*], *rule frame_rule*, *rule from_state_ht*)
apply *frame_inference*
apply (*sep_auto*; *fail*)
apply *solve_entails*

using *list_all2_swap* **by** (*sep_auto simp*: *list.rel_eq list_set_rel_def from_loc list_rel_def*)
apply *solve_entails*
using *list_all2_swap* **by** (*sep_auto simp*: *list_rel_def*)
}
then show *?thesis*
by (*refine_rcg*, *clarsimp*, *rule hoare_triple_run_heapD*)
qed

definition
to_pair ≡ λ(*l*, *s*). *do* {*s* ← *to_state s*; *return* (*to_loc l*, *s*)}

lemma *to_pair_ht*:
<*emp*> *to_pair a1* <λ*ai*. (*K* $\times_a$ *A*) *a ai*> **if** (*a1*, *a*) ∈ *K_rel* $\times_r$ *A_rel*
using *that* **unfolding** *to_pair_def*
by (*cases a*, *cases a1*, *subst pure_the_pure*[*symmetric*, *OF pure_K*])
(*sep_auto heap*: *to_state_ht simp*: *pure_def to_loc prod_assn_def split*: *prod.splits*)

sublocale *pure*:
Reachability_Impl_pure_base2
where
get_succs = *run_heap oo get_succs* **and**
K = *K_rel* **and**
A = *A_rel* **and**
lei = *lei* **and**
Pi = λ*a*. *run_heap* (*do* {*a* ← *to_pair a*; *Pi a*}) **and**
Fi = λ*a*. *run_heap* (*do* {*a* ← *to_pair a*; *Fi a*}) ~~and l0i = from_loc (run_heap l0i) and s0i = run_heap (do {s ← s0i; from_state s})~~
apply *standard*
subgoal
by (*rule K_rel*)

```
  subgoal
    by (rule K_rel)
  subgoal
    using Li unfolding list_set_rel_def by auto
  subgoal
    by (rule lei_less_eq)
  subgoal
    using get_succs .
  subgoal
    by (rule full_split)
  subgoal
    apply standard
    apply (rule hoare_triple_run_heapD)
    apply (sep_auto heap: to_pair_ht simp del: prod_rel_simp prod_assn_pair_conv)
    apply (rule Hoare_Triple.cons_rule[rotated 2])
      apply (rule Pi_P'[to_hnr, unfolded hn_refine_def hn_ctxt_def, simplified], rule ent_refl)
    apply (sep_auto simp: pure_def)
    done
  subgoal
    apply standard
    apply (rule hoare_triple_run_heapD)
    apply (sep_auto heap: to_pair_ht simp del: prod_rel_simp prod_assn_pair_conv)
    apply (rule Hoare_Triple.cons_rule[rotated 2])
      apply (rule Fi_F[to_hnr, unfolded hn_refine_def hn_ctxt_def, simplified], rule ent_refl)
    apply (sep_auto simp: pure_def)
    done
  done

end

locale Reachability_Impl_imp_to_pure = Reachability_Impl where
  l_0 = l_0 and s_0 = s_0 and M = M and K = K and A = A
  + Reachability_Impl_imp_to_pure_base
  where K_rel = K_rel and A_rel = A_rel and K = K and A = A
  for l_0 :: 'k and s_0 :: 'a
  and K :: 'k ⇒ ('ki :: {hashable,heap}) ⇒ assn and A :: 'a ⇒ ('ai :: heap) ⇒ assn
  and M and K_rel :: ('k1 × 'k) set and A_rel :: ('a1 × 'a) set +
  fixes Mi :: 'k1 ⇒ 'a1 list option
  assumes Mi_M: (Mi, M) ∈ K_rel → ⟨⟨A_rel⟩list_set_rel⟩option_rel
begin

sublocale pure:
  Reachability_Impl_pure
  where
    M = M and
    Mi = Mi and
    get_succs = run_heap oo get_succs and
    K = K_rel and
    A = A_rel and
    lei = lei and
    Pi = λa. run_heap (do {a ← to_pair a; Pi a}) and
    Fi = λa. run_heap (do {a ← to_pair a; Fi a}) and
    l_0i = from_loc (run_heap l_0i) and
    s_0i = run_heap (do {s ← s_0i; from_state s})
```

apply *standard*
apply (*rule Mi_M*)
subgoal
apply (*rule from_loc*)
apply (*rule hoare_triple_run_heapD*)
using $l_0i\_l_0$[*to_hnr*, *unfolded hn_refine_def hn_ctxt_def*, *simplified*]
apply (*subst* (*asm*) *pure_the_pure*[*symmetric*, *OF pure_K*])
apply (*sep_auto simp*: *pure_def elim*!: *cons_post_rule*)
done
subgoal
using $s_0i\_s_0$[*to_hnr*, *unfolded hn_refine_def hn_ctxt_def*, *simplified*]
by − (*rule hoare_triple_run_heapD*, *sep_auto heap*: *from_state_ht*)
done

end

locale *Reachability_Impl_imp_to_pure_correct* =
Reachability_Impl_imp_to_pure **where** $M = M$
+ *Reachability_Impl_correct* **where** $M = \lambda x.$ *case M x of None* ⇒ {} | *Some S* ⇒ *S*
for M
begin

sublocale *pure*:
Reachability_Impl_pure_correct
where
$M = M$ **and**
$Mi = Mi$ **and**
get_succs = *run_heap oo get_succs* **and**
$K = K\_rel$ **and**
$A = A\_rel$ **and**
$lei = lei$ **and**
$Pi = \lambda a.$ *run_heap* (*do* {$a \leftarrow$ *to_pair a*; *Pi a*}) **and**
$Fi = \lambda a.$ *run_heap* (*do* {$a \leftarrow$ *to_pair a*; *Fi a*}) **and**
l_0i = *from_loc* (*run_heap* l_0i) **and**
s_0i = *run_heap* (*do* {$s \leftarrow s_0i$; *from_state s*})
by *standard*

end

locale *Buechi_Impl_imp_to_pure* = *Buechi_Impl_pre* **where**
$M = \lambda x.$ *case M x of None* ⇒ {} | *Some S* ⇒ *S*
+ *Reachability_Impl_imp_to_pure_base*
where $K\_rel = K\_rel$ **and** $A\_rel = A\_rel$
for $M :: 'k \Rightarrow ('a \times nat)$ *set option*
and $K\_rel :: ('ki \times 'k)$ *set* **and** $A\_rel :: ('ai \times 'a)$ *set* +
fixes *inits initsi*
assumes *initsi_inits*: (*uncurry0 initsi*, *uncurry0* (*RETURN* (*PR_CONST inits*)))
$\in$ $unit\_assn^k \rightarrow_a$ *list_assn* ($K \times_a A$)
fixes $Mi :: 'ki \Rightarrow ('ai \times nat)$ *list option*
assumes *Mi_M*: $(Mi, M) \in K\_rel \rightarrow \langle\langle A\_rel \times_r Id\rangle list\_set\_rel\rangle option\_rel$
begin

lemma (**in** −) *list_all2_flip*:
list_all2 P xs ys **if** *list_all2 Q ys xs* ($\bigwedge x\ y.\ Q\ y\ x \Longrightarrow P\ x\ y$)

```
  using that by (simp add: list_all2_conv_all_nth)

sublocale pure:
  Buechi_Impl_pure
  where
    M = M and
    Mi = Mi and
    get_succs = run_heap oo get_succs and
    K = K_rel and
    A = A_rel and
    lei = lei and
    Pi = λa. run_heap (do {a ← to_pair a; Pi a}) and
    Fi = λa. run_heap (do {a ← to_pair a; Fi a}) and
    inits = set inits and
    initsi = run_heap (do {
      xs ← initsi;
      Heap_Monad.fold_map (λ(l, s). do {s ← from_state s; return (from_loc l, s)}) xs})
  apply standard
    apply (rule Mi_M F_mono; assumption)+

  apply (rule hoare_triple_run_heapD)
  subgoal
  proof −
    have ***: <↑((li, l) ∈ the_pure K) * ↑ ((s1, s) ∈ A_rel) * true> return (from_loc li, s1)
          <λ(l1, s1). ↑ ((l1, l) ∈ K_rel) * ↑((s1, s) ∈ A_rel)>t for li l s1 s
      by (sep_auto intro: from_loc)

    have 1: (l, fst (a, b)) ∈ R if (l, a) ∈ R for l a b R
      using that by auto
    have 2: (l, snd (a, b)) ∈ R if (l, b) ∈ R for l a b R
      using that by auto
    have
      <emp> (do {
        xs ← initsi;
        Heap_Monad.fold_map (λ(l, s). do {s ← from_state s; return (from_loc l, s)}) xs})
      <λr. ↑ ((r, inits) ∈ ⟨K_rel ×r A_rel⟩list_rel)>t
      using initsi_inits[to_hnr, unfolded hn_refine_def hn_ctxt_def, simplified]
      apply (subst (asm) pure_the_pure[symmetric, OF pure_K])

      apply (sep_auto simp: pure_def list_rel_def heap: fold_map_ht1)

       prefer 3
       apply (sep_auto elim: list_all2_flip)
      prefer 2
      apply (rule cons_post_rule)
       apply (rule ***)
      prefer 2
      apply (sep_auto heap: from_state_ht elim: 1 2)+
     done

    then show ?thesis
      by (sep_auto simp: pure_def list_set_rel_def relcomp.simps elim!: cons_post_rule)
```

qed
done

end

locale *Buechi_Impl_imp_to_pure_correct* =
Buechi_Impl_imp_to_pure **where** $M = M$ +
Buechi_Impl_correct **where** $M = \lambda x.\ case\ M\ x\ of\ None \Rightarrow \{\} \mid Some\ S \Rightarrow S$
for M
begin

sublocale *pure*:
Buechi_Impl_pure_correct
where
$M = M$ **and**
$Mi = Mi$ **and**
$get\_succs = run\_heap\ oo\ get\_succs$ **and**
$K = K\_rel$ **and**
$A = A\_rel$ **and**
$lei = lei$ **and**
$Pi = \lambda a.\ run\_heap\ (do\ \{a \leftarrow to\_pair\ a;\ Pi\ a\})$ **and**
$Fi = \lambda a.\ run\_heap\ (do\ \{a \leftarrow to\_pair\ a;\ Fi\ a\})$ **and**
$inits = set\ inits$ **and**
~~$initsi = map\ (\lambda(l,\ s).\ do\ \{s \leftarrow from\_state\ s;\ return\ (from\_loc\ l,\ s)\})\ inits$~~ **and**
$initsi =$
$run\_heap\ (do\ \{$
$xs \leftarrow initsi;$
$Heap\_Monad.fold\_map\ (\lambda(l,\ s).\ do\ \{s \leftarrow from\_state\ s;\ return\ (from\_loc\ l,\ s)\})\ xs\})$
..

end

end

7 Simulations for Buechi Properties

theory *Simulation_Graphs2*
imports *Timed_Automata.Simulation_Graphs HOL−Eisbach.Eisbach*
begin

This theory essentially formalizes the concepts from Guangyuan Li's FORMATS 2009 paper "Checking Timed Büchi Automata Emptiness Using LU-Abstractions" [1]. However, instead of formalizing this directly for the notions of timed Büchi automata, time-abstract simulations, and zone graphs with abstractions, we use general notions of simulation graphs with certain properties.

7.1 Misc

lemma *map_eq_append_conv*:
$(map\ f\ xs = ys\ @\ zs) = (\exists\ as\ bs.\ xs = as\ @\ bs \wedge map\ f\ as = ys \wedge map\ f\ bs = zs)$
by (*induction ys arbitrary*: *xs*; *simp add*: *map_eq_Cons_conv*; *metis append_Cons*)

lemma *Cons_subseq_iff*:
$subseq\ (x\ \#\ xs)\ ys \longleftrightarrow (\exists\ as\ bs.\ ys = as\ @\ x\ \#\ bs \wedge subseq\ xs\ bs)$

using *list_emb_ConsD list_emb_append2* **by** *fastforce*

lemma *append_subseq_iff*:
subseq (*as* @ *bs*) *xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *zs* ∧ *subseq as ys* ∧ *subseq bs zs*)
by (*meson list_emb_appendD list_emb_append_mono*)

context *Graph_Defs*
begin

lemma *steps_append_singleE*:
assumes *steps* (*xs* @ [*x*])
obtains *xs* = [] | *ys y* **where** *xs* = *ys* @ [*y*] *steps xs y* → *x*
using *assms* **by** (*metis append_butlast_last_id list.distinct*(*1*) *list.sel*(*1*) *steps_decomp*)

lemma *steps_alt_induct2*[*consumes 1*, *case_names Single Snoc*]:
assumes
steps (*a* # *xs* @ [*b*]) (⋀*b*. *E a b* ⟹ *P a* [] *b*)
⋀*y x xs*. *E y x* ⟹ *steps* (*a* # *xs* @ [*y*]) ⟹ *P a xs y* ⟹ *P a* (*xs* @ [*y*]) *x*
shows *P a xs b*
using *assms*(*1*)
apply (*induction a* # *xs* @ [*b*] *arbitrary*: *xs b rule*: *steps_alt_induct*)
subgoal
apply *auto*
done
subgoal for *y x xs ys b*
apply (*cases ys* = [])
apply (*force intro*: *assms*(*2*); *fail*)
apply (*simp*, *metis append.simps*(*2*) *append1_eq_conv append_butlast_last_id assms*(*3*))
done
done

lemma *steps_singleI*:
steps [*a*,*b*] **if** *a* → *b*
using *that steps.intros* **by** *blast*

end

7.2 Backward Simulations

locale *Backward_Simulation* = *Simulation* **where** *A* = *E* **and** *B* = *E* **and** *sim* = *sim*
for *E* :: *'a* ⇒ *'a* ⇒ *bool* **and** *sim* (**infix** ≼ *60*) +
fixes *G* :: *'a set* ⇒ *'a set* ⇒ *bool*
assumes *simulation*: *b* ∈ *B* ⟹ *G A B* ⟹ ∃ *a* ∈ *A*. ∃ *b'*. *a* → *b'* ∧ *b* ≼ *b'*
and *refl*[*intro*, *simp*]: *a* ≼ *a* **and** *trans*: *transp* (≼)
begin

lemmas *A_simulation_steps* = *simulation_steps*

sublocale *Graph_Defs G* **.**

lemmas *sim_transD*[*intro*] = *transpD*[*OF trans*]

lemma *backward_simulation_reaches*:
∃ *a* ∈ *A*. ∃ *b'*. *E*** *a b'* ∧ *b* ≼ *b'* **if** *G*** *A B b* ∈ *B*

using *that*
proof (*induction arbitrary*: *b rule*: *rtranclp_induct*)
case *base*
then show *?case*
by *auto*
next
case (*step Y Z*)
from *simulation*[*OF* ‹$b \in Z$› ‹$G\ Y\ Z$›] **obtain** $a\ b'$ **where** $a \in Y\ a \to b'\ b \preceq b'$
by *safe*
from *step.IH*[*OF* ‹$a \in Y$›] **obtain** $a0\ a'$ **where** $a0 \in A\ a0 \to* a'\ a \preceq a'$
by *safe*
moreover from *A_B_step*[*OF* ‹$a \to b'$› ‹$a \preceq a'$›] **obtain** b'' **where** $a' \to b''\ b' \preceq b''$
by *safe*
with ‹$a0 \in A$› ‹$a0 \to* a'$› ‹$b \preceq b'$› **show** *?case*
by (*auto intro*: *rtranclp.intros*(*2*))
qed

lemma *backward_simulation_steps*:
$\exists a \in A.\ \exists as\ b'.\ A.steps\ (a \#\ as\ @\ [b'])\ \wedge\ b \preceq b'$ **if** *steps* $(A \#\ As\ @\ [B])\ b \in B$
using *that*
proof (*induction A As B arbitrary*: *b rule*: *steps_alt_induct2*)
case (*Single x*)
from *simulation*[*OF this*(*2*,*1*)] **show** *?case*
by (*safe*, *intro bexI exI*[**where** $x = []$]) *auto*
next
case (*Snoc B C As c*)
from *simulation*[*OF* ‹$c \in C$› ‹$G\ B\ C$›] **obtain** $b\ c'$ **where** $b \in B\ b \to c'\ c \preceq c'$
by *safe*
with *Snoc.IH* **obtain** $a\ as\ b'$ **where** $a \in A\ A.steps\ (a \#\ as\ @\ [b'])\ b \preceq b'$
by *blast*
moreover from ‹$b \preceq b'$› ‹$b \to c'$› **obtain** c'' **where** $b' \to c''\ c' \preceq c''$
by (*auto dest*: *A_B_step*)
ultimately have $a \in A\ c \preceq c''\ A.steps\ (a \#\ (as\ @\ [b'])\ @\ [c''])$
using ‹$c \preceq c'$› **by** *auto* (*metis A.steps_appendI append_Cons*)
then show *?case*
by *blast*
qed

lemma *backward_simulation_reaches1*:
$\exists a \in A.\ \exists b'.\ E^{++}\ a\ b' \wedge b \preceq b'$ **if** $G^{++}\ A\ B\ b \in B$
using *that* **unfolding** *A.reaches1_steps_iff reaches1_steps_iff*
by (*auto dest*!: *backward_simulation_steps*)

Corresponds to lemma 8 of [1].

lemma *steps_repeat*:
assumes *steps* $(A \#\ As\ @\ [A])\ a \in A\ \forall a \in A.\ P\ a\ \forall x\ y.\ x \preceq y \wedge x \in A \wedge P\ x \longrightarrow P\ y$
obtains $x\ y\ as\ xs$ **where**
subseq as xs list_all P as A.steps $(x \#\ xs\ @\ [y])$ *length as* $= n\ a \preceq y\ x \in A$
using ‹$a \in A$›
proof (*atomize_elim*, *induction n arbitrary*: *a*)
case *0*
from *backward_simulation_steps*[*OF* ‹*steps* $(A \#\ As\ @\ [A])$› ‹$a \in A$›] **obtain** $a'\ as\ b$ **where**
$a' \in A\ A.steps\ (a' \#\ as\ @\ [b])\ a \preceq b$
by *auto*

then show *?case*
by (*auto 4 4 intro*: *exI*[**where** $x = []$])
next
case (*Suc n b*)
from *backward_simulation_steps*[*OF assms*(*1*) ‹$b \in A$›] **obtain** a *as* b' **where**
$a \in A$ *A.steps* (a # *as* @ [b']) $b \preceq b'$
by *auto*
from *Suc.IH*[*OF* ‹$a \in A$› ‹$a \in A$›] **obtain** *as'* *xs* x y **where**
subseq as' xs list_all P as' A.steps (x # *xs* @ [y]) n = *length as'* $a \preceq y$ $x \in A$
by *auto*
moreover from *A_simulation_steps*[*OF* ‹*A.steps* (a # _)› ‹$a \preceq y$›] ‹$b \preceq b'$› **obtain** b'' *bs*
where
A.steps (y # *bs* @ [b'']) *list_all2* ($\preceq$) *as bs* $b \preceq b''$
unfolding *list_all2_append1 list_all2_Cons1* **by** *auto*
ultimately show *?case*
using *assms*(*3*,*4*) ‹$a \in A$›
by (*inst_existentials as'* @ [y] *xs* @ [y] @ *bs x b''*)
(*auto simp*: *list_emb_append_mono*, *metis A.steps_append2 append_Cons*)
qed

end

7.3 Self Simulation for a Finite Simulation Relation

This section makes the following abstractions:

- The timed automata semantics correspond to the transition system $\rightarrow$ (E).
- The finite time-abstract bisimulation $\equiv_M$ from the classic region construction corresponds to the simulation $\preceq$.

locale *Self_Simulation* =
Simulation_Invariant **where** $A = E$ **and** $B = E$ **and** *sim* = *sim* **and** $PA = P$ **and** $PB = P$
for $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** *sim* (**infix** $\preceq$ *60*) **and** P +
assumes *refl*: *reflp* ($\preceq$) **and** *trans*: *transp* ($\preceq$)
begin

lemma *sim_reflI*[*intro*, *simp*]:
$x \preceq x$
using *refl* **unfolding** *reflp_def* **by** *auto*

lemmas *sim_transD*[*intro*] = *transpD*[*OF trans*]

Corresponds to lemma 3 of [1].

lemma *pre_cycle_infinite_cycle*:
assumes *A.steps* (x # *xs* @ [y]) $x \preceq y$ P x P y
obtains w **where**
A.run (x ## w) *stream_all2* ($\preceq$) (*cycle* (x # *xs*)) (x ## w)
proof −
let *?R* = $\lambda a\ b.\ a \preceq b \wedge P\ a \wedge P\ b$
define *nxt* **where**
nxt ≡ $\lambda(xs, y)$. *SOME* (ys, z). *A.steps* (y # *ys* @ [z]) ∧ *list_all2 ?R xs ys* ∧ $y \preceq z$
have ∗: *A.steps* (y # *ys* @ [z]) ∧ *list_all2 ?R xs ys* ∧ $y \preceq z$
if *nxt* (xs, y) = (ys, z) *A.steps*(x # *xs* @ [y]) $x \preceq y$ P x P y **for** x y *xs ys z*

```
proof −
  from simulation_steps[OF that(2) ‹x ⪯ y›] ‹P x› ‹P y› obtain ws y′ where
    A.steps (y # ws @ [y′]) list_all2 ?R xs ws y ⪯ y′
    by (smt list_all2_Cons1 list_all2_Nil list_all2_append1)
  then show ?thesis
    using ‹nxt _ = _› unfolding nxt_def by (auto dest!: verit_sko_ex_indirect[OF sym])
qed
let ?w = flat (smap (λ(xs, y). xs @ [y]) (siterate nxt (xs, y)))
from assms have A.run ?w
proof (coinduction arbitrary: x xs y rule: A.run_flat_coinduct)
  case (run_shift as bs xss x xs y)
  obtain ys z where nxt (xs, y) = (ys, z)
    by (cases nxt (xs, y))
  with run_shift have as = xs @ [y] bs = ys @ [z]
    bs ## xss = smap (λ(xs, y). xs @ [y]) (siterate nxt (ys, z))
    by auto
  with ∗[OF ‹nxt _ = _› ‹A.steps (x # _)› ‹x ⪯ y›] run_shift(2−) show ?case
    by (inst_existentials ys z)
       (auto 4 3 dest: A.steps_ConsD PA_invariant.invariant_steps elim: A.steps.cases)
qed
with assms(1) have A.run (x ## ?w)
  apply −
  apply (cases smap (λ(xs, y). xs @ [y]) (siterate nxt (xs, y)))
  apply (simp only:)
  apply clarsimp
  by (smt A.run.cases A.run.intros A.steps.cases shift_simps(1) stream.sel(1)
       append_Nil append_is_Nil_conv hd_append2 list.distinct(1) list.sel)
obtain x′ xs′ where eqs: xs = xs′ x = x′
  by auto
with assms have A.steps (x′ # xs′ @ [y]) list_all2 (⪯) xs xs′ x ⪯ x′ x′ ⪯ y P x′ P y
  by (auto simp: zip_same list_all2_iff)
then have stream_all2 (⪯) (cycle (xs @ [x])) ?w
proof (rewrite in ‹(xs, y)› eqs, coinduction arbitrary: x′ xs′ y rule: stream_rel_coinduct_shift)
  case stream_rel
  obtain ys z where nxt (xs′, y) = (ys, z)
    by (cases nxt (xs′, y))
  with stream_rel show ?case
    apply −
    apply (frule (4) ∗)
    apply (inst_existentials xs @ [x] cycle (xs @ [x]) xs′ @ [y]
          flat (smap (λ(xs, y). xs @ [y]) (siterate nxt (ys, z))))
    subgoal
      by simp
        (metis cycle_Cons cycle_decomp cycle_rotated list.distinct(1) list.sel(1) list.sel(3))
    subgoal
      by (cases smap (λ(xs, y). xs @ [y]) (siterate nxt (xs′, y))) (simp only:, auto)
    apply (solves ‹auto simp: list_all2_iff›)+
    apply (auto 4 5 elim: list_all2_trans[rotated] dest: PA_invariant.invariant_steps)
    done
qed
then have stream_all2 (⪯) (cycle (x # xs)) (x ## ?w)
  by (simp add: cycle_Cons)
with ‹A.run (x ## ?w)› show ?thesis
  by (auto intro: that)
```

qed

end

locale *Self_Simulation_Finite* =
  Simulation_Invariant **where** $A = E$ **and** $B = E$ **and** *sim* = *sim* **and** $PA = P$ **and** $PB = P$
  for $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** *sim* (**infix** $\preceq$ *60*) **and** P +
  assumes *equiv_sim*: *equivp* ($\preceq$) **and** *finite_quotient*: *finite* (*UNIV* // {(*x*, *y*). $x \preceq y$})
begin

sublocale *Self_Simulation*
  apply *standard*
  using *equiv_sim* **apply** (*simp add*: *equivp_reflp_symp_transp*)+
  done

Roughly corresponds to lemmas 9, 10, and 11 of [1].

lemma *steps_cycle_run*:
  assumes *A.steps* (x # *xs*) *subseq as xs P x length as* > *card* (*UNIV* // {(*x*, *y*). $x \preceq y$})
    $\forall x \in set\ as.\ \varphi\ x\ \forall x \in set\ as.\ \forall y.\ x \preceq y \wedge \varphi\ x \longrightarrow \varphi\ y$
  obtains *w* **where** *A.run* (x ## *w*) *infs* φ *w*
proof −
  from *assms*(*3*) **obtain** *a b as' ys cs'* **where** ∗: $xs = as' @ a \# ys @ b \# cs'\ a \preceq b\ a \in set\ as$
  proof −
    let *?f* = $\lambda x.$ {*y*. $x \preceq y$}
    let *?as* = *map ?f as*
    have *card* (*set ?as*) ≤ *card* (*UNIV* // {(*x*, *y*). $x \preceq y$})
      using *quotientI*[*of* _ _ {(*x*, *y*). $x \preceq y$}]
      by (*auto intro*: *finite_quotient surj_card_le*[**where** $f = id$])
    also have ... < *length as*
      by (*rule assms*)
    finally have *card* (*set ?as*) < *length ?as*
      by *simp*
    then have ¬ *distinct ?as*
      using *distinct_card*[*of ?as*] **by** *auto*
    then obtain *a b as' ys cs'* **where** $as = as' @ a \# ys @ b \# cs'\ a \preceq b$
      using *that*
     by (*clarsimp simp*: *map_eq_append_conv map_eq_Cons_conv dest*!: *not_distinct_decomp*)
blast
    then show *?thesis*
      using *that*[**where** $a = a$ **and** $b = b$] ‹*subseq as xs*›
      by (*simp add*: *append_subseq_iff Cons_subseq_iff*) (*metis append.assoc*)
  qed
  have *A.steps* (x # as' @ [*a*]) *A.steps* (a # *ys* @ [*b*])
  proof −
    from *assms*(*1*) ∗ **show** *A.steps* (x # as' @ [*a*])
      using *A.steps_appendD1* **by** *auto*
    from ∗ **have** $as' @ (a \# ys) @ b \# cs' = xs$
       by *simp*
    have $b \# cs' \neq [] \wedge a \# ys \neq [] \vee (a \# ys) @ b \# cs' \neq [] \wedge \neg$ *A.steps* $((a \# ys) @ b \# cs')$
       ∨ *A.steps* $((a \# ys) @ [b])$
      by *blast*
    with ‹_ = *xs*› *assms*(*1*) **have** *A.steps* $((a \# ys) @ [b])$
      by (*metis* (*no_types*) *A.Single A.steps_ConsD A.steps_append_single A.steps_decomp*
          Nil_is_append_conv list.sel(*1*) *self_append_conv2*)

```
    then show A.steps (a # ys @ [b])
      by simp
  qed
  with ‹P x› have P a P b
    by (auto 4 3 dest: PA_invariant.invariant_steps)
  from pre_cycle_infinite_cycle[OF ‹A.steps (a # ys @ [b])› ‹a ⪯ b› this] obtain w where
    A.run (a ## w) stream_all2 (⪯) (cycle (a # ys)) (a ## w) .
  from this(2) have infs ((⪯) a) (a ## w)
    by (smt alw_ev_lockstep infs_cycle list.distinct(1) list.set_intros(1) sim_reflI sim_transD)
  then have infs φ (as' @− a ## w)
    using assms(5) ‹a ∈ set as› unfolding ‹xs = as' @ _› apply simp
    using assms(6) by (elim infs_mono[rotated]) blast
  moreover from ‹A.run _› ‹A.steps (x # as' @ [a])› have A.run (x ## as' @− a ## w)
    by (metis A.extend_run A.steps_decomp append_Cons list.distinct(1) list.sel(1,3)
        shift_simps stream.exhaust stream.sel)
  ultimately show ?thesis
    by (rule that[rotated])
qed

end
```

7.4 Combining Finite Simulation with Backward Simulation

Here, $\preceq$ is any time-abstract simulation $\preceq$, and $\preceq'$ corresponds to $\equiv_M$.

```
locale Backward_Double_Simulation = Backward_Simulation where E = E and sim = sim +
  finite: Self_Simulation_Finite where E = E and sim = sim'
  for E :: 'a ⇒ 'a ⇒ bool and sim (infix ⪯ 60) and sim' (infix ⪯'' 60)
begin
```

Corresponds to lemma 12 of [1].

```
lemma cycle_Buechi_run:
  assumes steps (A # As @ [A]) a ∈ A ∀ a ∈ A. P a ∀ a ∈ A. φ a
    ∀ x y. x ⪯ y ∧ x ∈ A ∧ φ x ⟶ φ y ∀ x y. x ⪯' y ∧ φ x ⟶ φ y
  obtains x xs where A.run (x ## xs) infs φ xs x ∈ A
proof −
  let ?n = card (UNIV // {(x, y). x ⪯' y}) + 1
  from steps_repeat[OF assms(1,2,4−5), where n = ?n] obtain x y as ys where ∗:
    subseq as ys list_all φ as A.steps (x # ys @ [y])
    length as = ?n a ⪯ y x ∈ A .
  with assms(4) have ∀ x ∈ set as. φ x
    by (auto simp: list_all_iff)
  with ∗ assms(3,6) obtain w where A.run (x ## w) infs φ w
    by − (erule finite.steps_cycle_run[where φ = φ], auto)
  then show ?thesis
    using ‹x ∈ A› by (elim that) simp
qed

end

lemma (in Simulation_Invariant) simulation_run':
  assumes A.run (x ## xs) x ∼ y PA x PB y
  shows ∃ ys. B.run (y ## ys) ∧ stream_all2 (λa b. a ∼ b ∧ PA a ∧ PB b) xs ys
  using simulation_run assms unfolding equiv'_def by blast
```

context *Graph_Invariant_Start*
begin

lemma *reachable_reaches_equiv*: $G'.reaches\ x\ y \longleftrightarrow x \rightarrow* y$ **if** *reachable* x **for** $x\ y$
using *invariant_reaches invariant_reaches_iff reachable_def that* **by** *auto*

lemma *reachable_reaches1_equiv*: $G'.reaches1\ x\ y \longleftrightarrow x \rightarrow^+ y$ **if** *reachable* x **for** $x\ y$
using *invariant_reaches invariant_reaches1_iff reachable_def that* **by** *auto*

lemma *reachable_steps_equiv*:
$G'.steps\ (x\ \#\ xs) \longleftrightarrow steps\ (x\ \#\ xs)$ **if** *reachable* x
using *invariant_reaches invariant_steps_iff reachable_def that* **by** *auto*

lemma *reachable_run_equiv*:
$G'.run\ (x\ \#\#\ xs) \longleftrightarrow run\ (x\ \#\#\ xs)$ **if** *reachable* x
— This proof is bit clumsy due to the name clash for *invariant_run*
by (*metis reaches_steps_iff Graph_Invariant.invariant_run Graph_Invariant_axioms*
invariant_reaches reachable_steps subgraph_run_iff that)

lemmas *invariant_subgraph_equivs* =
reachable_reaches_equiv reachable_reaches1_equiv reachable_steps_equiv reachable_run_equiv

end

Adding the assumption that the abstracted zone graph is finite and complete.

locale *Backward_Double_Simulation_Complete* =
A: *Graph_Defs* E + G: *Graph_Defs* G + G_inv: *Graph_Defs* $\lambda x\ y.\ G\ x\ y \wedge Q\ x \wedge Q\ y$ +
backward: *Backward_Double_Simulation* **where** $E = E$ **and** $G = \lambda x\ y.\ G\ x\ y \wedge Q\ x \wedge Q\ y$ +
complete: *Simulation_Invariant* **where** $A = E$ **and** $B = G$ **and** $PA = P$ **and** $PB = Q$ **and** $sim = (\in)$ +
Finite_Graph **where** $E = G$ **and** $x_0 = a_0$ +
Graph_Invariant **where** $E = G$ **and** $P = Q$
for $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $G :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ **and** a_0 **and** Q +
assumes Q_P: $Q\ a \Longrightarrow \forall x \in a.\ P\ x$ **and** $a_0\_invariant$: $Q\ a_0$
begin

sublocale G: *Graph_Invariant_Start* **where** $E = G$ **and** $P = Q$ **and** $s_0 = a_0$
by (*standard*) (*rule* $a_0\_invariant$)

lemmas *G_invariant_subgraph_equivs* =
G.invariant_subgraph_equivs[*unfolded complete.PB_invariant.E'_def*]

Corresponds to theorem 1 of [1].

theorem *Buechi_run_lasso_iff*:
assumes
$\forall x\ y.\ x \preceq' y \wedge \varphi\ x \longrightarrow \varphi\ y$
$\forall x\ y.\ x \preceq y\ \wedge \varphi\ x \longrightarrow \varphi\ y$
$\forall x\ y\ A.\ G.reaches\ a_0\ A \wedge x \in A \wedge y \in A \wedge \varphi\ x \longrightarrow \varphi\ y$
shows
$(\exists x_0\ xs.\ x_0 \in a_0 \wedge A.run\ (x_0\ \#\#\ xs) \wedge infs\ \varphi\ (x_0\ \#\#\ xs))$
$\longleftrightarrow (\exists as\ a\ bs.\ G.steps\ (a_0\ \#\ as\ @\ a\ \#\ bs\ @\ [a]) \wedge (\forall x \in a.\ \varphi\ x) \wedge a \neq \{\})$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

proof
assume *?lhs*
then obtain x_0 *xs* **where** $x_0 \in a_0$ *A.run* $(x_0$ *##* *xs*$)$ *infs* φ *xs*
by *auto*
have *backward_reaches_A*: *G.reaches* a_0 *A* **if** *G_inv.reaches* a_0 *A* **for** *A*
using *that* **by** (*simp add*: *G_invariant_subgraph_equivs*[*symmetric*])
from *complete.simulation_run'*[*OF* ‹*A.run* _› ‹$x_0 \in$ _›] *a*$_0$*_invariant* ‹$x_0 \in a_0$› **obtain** *as* **where**
G.run $(a_0$ *##* *as*$)$ *stream_all2* $(\lambda a\ b.\ a \in b \wedge P\ a \wedge Q\ b)$ *xs as*
by (*auto dest*: *Q_P*)
from ‹*G.run* $(a_0$ *##* *as*$)$› **have** *G_inv.run* $(a_0$ *##* *as*$)$
by (*simp add*: *G_invariant_subgraph_equivs*)
from ‹*infs* φ _› ‹*stream_all2* _ _ _› **have** *infs* $(\lambda a.\ \exists x \in a.\ \varphi\ x)$ *as*
by (*rule alw_ev_lockstep*) *fast*
then have *infs* $(\lambda a.\ (\forall x \in a.\ \varphi\ x) \wedge a \neq \{\})$ *as*
using *assms*(*3*) ‹*G_inv.run* $(a_0$ *##* *as*$)$›
by (*auto 4 5 simp*: *stream.pred_set dest*!: *G_inv.run_reachable backward_reaches_A*
elim!: *infs_mono*[*rotated*])
with ‹*G.run* _› **show** *?rhs*
apply −
apply (*erule buechi_run_lasso*)
apply (*simp*; *fail*)
by (*metis* (*lifting*)
G.reaches1_steps_append G.reaches1_steps_iff G.reaches_stepsE G.steps_reaches1
list.sel(*1*))
next
assume *?rhs*
then obtain *as a x bs* **where** *G.steps* $(a_0$ *#* *as* *@* *a* *#* *bs* *@* $[a])$ $\forall x \in a.\ \varphi\ x\ x \in a$
by *auto*
then have *G.reaches* a_0 *a G.reaches1* a_0 *a*
apply −
subgoal *A*
by (*smt G.steps_reaches append_Cons last_appendR last_snoc list.sel*(*1*))
subgoal
by (*metis A G.steps_ConsD G.steps_appendD2 G.steps_reaches1*
append_is_Nil_conv list.distinct(*1*) *rtranclpD*)
done
moreover from ‹*G.steps* $(a_0$ *#* *as* *@* *a* *#* *bs* *@* $[a])$› **have** *G.steps* $(a$ *#* *bs* *@* $[a])$
by (*metis append_is_Nil_conv list.distinct*(*1*) *G.steps_ConsD G.steps_appendD2*)
ultimately have *G_inv.reaches1* a_0 *a G_inv.steps* $(a$ *#* *bs* *@* $[a])$
by (*simp add*: *G_invariant_subgraph_equivs G.reachable_def*)+
from ‹*G.reaches* a_0 *a*› *a*$_0$*_invariant* **have** *Q a*
by (*rule complete.PB_invariant.invariant_reaches*)
with *assms*(*1*,*2*) ‹*G.reaches* _ _› ‹$\forall x \in a.\ \varphi\ x$› **obtain** *x xs* **where**
A.run $(x$ *##* *xs*$)$ *infs* φ *xs* $x \in a$
by − (*rule backward.cycle_Buechi_run*[*OF* ‹*G_inv.steps* $(a$ *#* *bs* *@* $[a])$› ‹$x \in a$›, **where** $\varphi = \varphi$],
(*blast dest*: *Q_P*)+)
from *backward.backward_simulation_reaches1*[*OF* ‹*G_inv.reaches1* a_0 *a*› ‹$x \in a$›] **obtain** x_0 x' **where**
$x_0 \in a_0$ $x \preceq x'$ $x_0 \rightarrow^+ x'$
by *auto*
then obtain *ys* **where** *A.steps* $(x_0$ *#* *ys* *@* $[x'])$
using *A.reaches1_steps_iff* **by** *auto*

from *backward.simulation_run*[*OF* ‹*A.run* _› ‹$x \preceq x'$›] **obtain** xs' **where**
$A.run\ (x' \#\#\ xs')$ *stream_all2* $(\preceq)$ *xs* xs'
by (*elim conjE exE*)
with ‹*A.steps* _› **have** $A.run\ (x_0 \#\#\ ys\ @-\ x' \#\#\ xs')$
by (*metis A.extend_run A.steps_decomp*
append_Cons list.distinct(1) *list.sel*(1,3) *shift_simps*(1,2) *stream.collapse*)
moreover from ‹*infs* _ _› ‹*stream_all2* _ _ _› *assms*(2) **have** *infs* φ xs'
by (*auto elim*!: *alw_ev_lockstep*)
ultimately show *?lhs*
using ‹$x_0 \in a_0$› **by** *auto*
qed

end

end

8 Simulations on Timed Automata

theory *TA_Simulation*
imports
Timed_Automata.Timed_Automata
Timed_Automata.Normalized_Zone_Semantics
Timed_Automata.Simulation_Graphs_TA
HOL−Eisbach.Eisbach
Simulation_Graphs2
Munta_Base.Normalized_Zone_Semantics_Impl_Semantic_Refinement
HOL−ex.Sketch_and_Explore
begin

This theory essentially formalizes the concepts from Guangyuan Li's FORMATS 2009 paper "Checking Timed Büchi Automata Emptiness Using LU-Abstractions" [1].

no_notation *dbm_le* (_ $\preceq$ _ [*51*, *51*] *50*)

8.1 Preliminaries

lemma
step_z_state_setI1: $l \in state\_set\ A$ **and**
step_z_state_setI2: $l' \in state\_set\ A$ **if** $A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle$
using *that* **unfolding** *step_z'_def* **by** (*force simp*: *state_set_def trans_of_def*)+

lemma *step_trans_z'_sound*:
$A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z'\rangle \Longrightarrow \forall\, u' \in Z'.\ \exists\, u \in Z.\ \exists\, d.\ \ A \vdash' \langle l, u\rangle \rightarrow^t \langle l', u'\rangle$
by (*fastforce dest*!: *step_trans_a_z_sound step_trans_t_z_sound elim*!: *step_trans_z'.cases*)

lemma *step_trans_z'_exact_strong*:
assumes $A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z'\rangle$
shows $Z' = \{u'.\ \exists\, u \in Z.\ A \vdash' \langle l, u\rangle \rightarrow^t \langle l', u'\rangle\}$
using *step_trans_z'_sound assms* **by** (*auto dest*: *step_trans_z'_exact step_trans_z'_sound*)

lemma *step_a_step_trans_iff*:
$A \vdash \langle l, u\rangle \rightarrow_a \langle l', u'\rangle \longleftrightarrow (\exists\, g\ r.\ A \vdash_t \langle l, u\rangle \rightarrow_{(g,a,r)} \langle l', u'\rangle)$
unfolding *step_a.simps step_trans.simps* **by** *fast*

lemma *step_trans'_step_trans_iff*:
(∃ *t*. *A* ⊢′ ⟨*l*, *u*⟩ →^{t} ⟨*l'*, *u'*⟩) ⟷ *A* ⊢′ ⟨*l*, *u*⟩ → ⟨*l'*, *u'*⟩
unfolding *step_trans'.simps step'.simps step_a_step_trans_iff* **by** *fast*

8.2 Time-Abstract Simulation

locale *Time_Abstract_Simulation* =
fixes *A* :: ('*a*, '*c*, '*t* :: *time*, '*l*) *ta*
fixes *sim* :: '*l* × ('*c* ⇒ '*t* :: *time*) ⇒ '*l* × ('*c* ⇒ '*t*) ⇒ *bool* (**infix** ≼ *60*)
assumes *sim*:
⋀*l l' l*$_1$ *u u' u*$_1$ *t*. (*l*, *u*) ≼ (*l'*, *u'*) ⟹ *A* ⊢′ ⟨*l*, *u*⟩ →^{t} ⟨*l*$_1$, *u*$_1$⟩
⟹ ∃ *u*$_1$′. *A* ⊢′ ⟨*l'*, *u'*⟩ →^{t} ⟨*l*$_1$, *u*$_1$′⟩ ∧ (*l*$_1$, *u*$_1$) ≼ (*l*$_1$, *u*$_1$′)
assumes *refl*: ⋀*u*. *u* ≼ *u* **and** *trans*: ⋀*u v w*. *u* ≼ *v* ⟹ *v* ≼ *w* ⟹ *u* ≼ *w*
begin

lemma *simE*:
assumes (*l*, *u*) ≼ (*l'*, *u'*) *A* ⊢′ ⟨*l*, *u*⟩ →^{t} ⟨*l*$_1$, *u*$_1$⟩
obtains *u*$_1$′ **where** *A* ⊢′ ⟨*l'*, *u'*⟩ →^{t} ⟨*l*$_1$, *u*$_1$′⟩ (*l*$_1$, *u*$_1$) ≼ (*l*$_1$, *u*$_1$′)
using *assms sim* **by** *blast*

definition *abs* :: '*l* ⇒ ('*c*, '*t*) *zone* ⇒ ('*c*, '*t*) *zone* (*α* _ _ [*71*,*71*] *71*) **where**
α l W = {*v*. ∃ *v'* ∈ *W*. (*l*, *v*) ≼ (*l*, *v'*)}

lemma *simulation_mono*:
assumes *α l Z* ⊆ *α l Z' A* ⊢′ ⟨*l*, *Z*⟩ ⇝^{t} ⟨*l*$_1$, *Z*$_1$⟩ *A* ⊢′ ⟨*l*, *Z'*⟩ ⇝^{t} ⟨*l*$_1$, *Z*$_1$′⟩
shows *α l*$_1$ *Z*$_1$ ⊆ *α l*$_1$ *Z*$_1$′
proof −
have
Z$_1$ = {*u'*. ∃ *u* ∈ *Z*. *A* ⊢′ ⟨*l*, *u*⟩ →^{t} ⟨*l*$_1$, *u'*⟩} *Z*$_1$′ = {*u'*. ∃ *u* ∈ *Z'*. *A* ⊢′ ⟨*l*, *u*⟩ →^{t} ⟨*l*$_1$, *u'*⟩}
by (*intro step_trans_z'_exact_strong assms*(*2*,*3*))+
show *?thesis*
unfolding *abs_def*
proof *safe*
fix *u v*
assume *v* ∈ *Z*$_1$ (*l*$_1$, *u*) ≼ (*l*$_1$, *v*)
with ‹*Z*$_1$ = _› **obtain** *u*$_0$ **where** *u*$_0$ ∈ *Z* **and** *step*: *A* ⊢′ ⟨*l*, *u*$_0$⟩ →^{t} ⟨*l*$_1$, *v*⟩
by *auto*
from ‹*u*$_0$ ∈ *Z*› ‹*α l Z* ⊆ _› **obtain** *v*$_0$ **where** *v*$_0$ ∈ *Z'* (*l*, *u*$_0$) ≼ (*l*, *v*$_0$)
unfolding *abs_def* **using** *refl*[*of* (*l*, *u*$_0$)] **by** *auto*
from *simE*[*OF* ‹(*l*, *u*$_0$) ≼ (*l*, *v*$_0$)› *step*] **obtain** *v'* **where**
A ⊢′ ⟨*l*, *v*$_0$⟩ →^{t} ⟨*l*$_1$, *v'*⟩ (*l*$_1$, *v*) ≼ (*l*$_1$, *v'*) .
with ‹*v*$_0$ ∈ *Z'*› ‹*Z*$_1$′ = _› **have** *v'* ∈ *Z*$_1$′
by *auto*
moreover from ‹_ ≼ (*l*$_1$, *v*)› ‹(*l*$_1$, *v*) ≼ _› **have** (*l*$_1$, *u*) ≼ (*l*$_1$, *v'*)
by (*rule trans*)
ultimately show ∃ *x*∈*Z*$_1$′. (*l*$_1$, *u*) ≼ (*l*$_1$, *x*)
by *fast*
qed
qed

lemma *simulation*:
assumes *α l Z* = *α l Z' A* ⊢′ ⟨*l*, *Z*⟩ ⇝^{t} ⟨*l'*, *Z*$_1$⟩ *A* ⊢′ ⟨*l*, *Z'*⟩ ⇝^{t} ⟨*l'*, *Z*$_1$′⟩
shows *α l' Z*$_1$ = *α l' Z*$_1$′
using *simulation_mono assms* **by** *blast*

lemma *simulation'*:
  assumes $\alpha\ l\ Z = \alpha\ l\ Z'$ $A \vdash' \langle l, Z\rangle \leadsto^t \langle l', Z_1\rangle$
  shows $\exists Z_1'.\ A \vdash' \langle l, Z'\rangle \leadsto^t \langle l', Z_1'\rangle \wedge \alpha\ l'\ Z_1 = \alpha\ l'\ Z_1'$
proof −
  from ‹$A \vdash' \langle l, Z\rangle \leadsto^t \langle l', Z_1\rangle$› **obtain** Z_1' **where** $A \vdash' \langle l, Z'\rangle \leadsto^t \langle l', Z_1'\rangle$
    by (*auto elim*!: *step_trans_z'.cases step_trans_z.cases*)
  with *simulation assms* **show** *?thesis*
    by *blast*
qed

lemma *abs_involutive*:
  $\alpha\ l\ (\alpha\ l\ Z) = \alpha\ l\ Z$
  unfolding *abs_def* **by** (*auto intro*: *refl trans*)

lemma *abs_widens*:
  $Z \subseteq \alpha\ l\ Z$
  unfolding *abs_def* **by** (*auto intro*: *refl*)

This is Lemma 4 from the paper "Better Abstractions for Timed Automata" (https://arxiv.org/abs/1110.3705)

corollary *transition_compatibility*:
  assumes $A \vdash' \langle l, \alpha\ l\ Z\rangle \leadsto^t \langle l', W\rangle$ $A \vdash' \langle l, Z\rangle \leadsto^t \langle l', Z'\rangle$
  shows $\alpha\ l'\ W = \alpha\ l'\ Z'$
  by (*rule simulation*[*OF* _ *assms*(*1*,*2*)], *rule abs_involutive*)

inductive *step_abs* ::
  $('a, 'c, 't, 'l)\ ta \Rightarrow 'l \Rightarrow ('c, 't)\ zone \Rightarrow 'a \Rightarrow 'l \Rightarrow ('c, 't)\ zone \Rightarrow bool$
(_ ⊢ ⟨_, _⟩ ↝$_{\alpha(\_)}$ ⟨_, _⟩ [*61*,*61*,*61*] *61*)
where
  step_alpha:
    $A \vdash \langle l, Z\rangle \leadsto_\tau \langle l', Z'\rangle \Longrightarrow A \vdash \langle l', Z'\rangle \leadsto_{\uparrow a} \langle l'', Z''\rangle$
  $\Longrightarrow A \vdash \langle l, \alpha\ l\ Z\rangle \leadsto_{\alpha(a)} \langle l'', \alpha\ l''\ Z''\rangle$

interpretation *sim1*: *Simulation* **where**
  $A = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \to \langle l', u'\rangle$ **and**
  $B = \lambda(l, Z)\ (l', Z').\ \exists a.\ A \vdash \langle l, Z\rangle \leadsto_{\alpha(a)} \langle l', Z'\rangle$ **and**
  $sim = \lambda(l, u)\ (l', Z).\ l' = l \wedge u \in Z \wedge \alpha\ l\ Z = Z$
  apply *standard*
  unfolding *step'.simps step_abs.simps*
  apply *clarsimp*
  subgoal premises *prems* **for** *l v l'' v'' Z d l' v' a*
  proof −
    from ‹$v \in Z$› ‹$A \vdash \langle l, v\rangle \to^d \langle l', v'\rangle$› **obtain** Z' **where**
      $A \vdash \langle l, Z\rangle \leadsto_\tau \langle l', Z'\rangle$ $v' \in Z'$
      by (*auto dest*: *step_t_z_complete*)
    moreover obtain Z'' **where**
      $A \vdash \langle l', Z'\rangle \leadsto_{\uparrow a} \langle l'', Z''\rangle$ $v'' \in Z''$
      using *prems* ‹$v' \in Z'$› **by** (*auto dest*: *step_a_z_complete*)
    ultimately show *?thesis*
      using ‹$\alpha\ l\ Z = Z$› *abs_involutive abs_widens* **by** *blast*
  qed
  done

interpretation *sim2*: *Simulation* **where**
  $A = \lambda(l,\ u)\ (l',\ u').\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle$ **and**
  $B = \lambda(l,\ Z)\ (l',\ Z').\ \exists\, a.\ A \vdash \langle l,\ \alpha\ l\ Z\rangle \rightsquigarrow_{\alpha(a)} \langle l',\ \alpha\ l'\ Z'\rangle$ **and**
  $sim = \lambda(l,\ u)\ (l',\ Z).\ l' = l \wedge u \in Z$
  apply *standard*
  unfolding *step'.simps step_abs.simps*
  apply *clarsimp*
  subgoal premises *prems* **for** *l v l'' v'' Z d l' v' a*
  proof −
    from ‹$v \in Z$› ‹$A \vdash \langle l,\ v\rangle \rightarrow^{d} \langle l',\ v'\rangle$› **obtain** Z' **where**
      $A \vdash \langle l,\ Z\rangle \rightsquigarrow_{\tau} \langle l',\ Z'\rangle\ v' \in Z'$
      by (*auto dest*: *step_t_z_complete*)
    moreover obtain Z'' **where**
      $A \vdash \langle l',\ Z'\rangle \rightsquigarrow_{\upharpoonright a} \langle l'',\ Z''\rangle\ v'' \in Z''$
      using *prems* ‹$v' \in Z'$› **by** (*auto dest*: *step_a_z_complete*)
    ultimately show *?thesis*
      by *fastforce*
  qed
  done

sublocale *self_simulation*: *Self_Simulation* **where**
  $E = \lambda(l,\ u)\ (l',\ u').\ A \vdash' \langle l,\ u\rangle \rightarrow \langle l',\ u'\rangle$ **and** $P = \lambda\_.\ True$
  apply *standard*
  apply (*force dest*: *sim simp*: *step_trans'_step_trans_iff*[*symmetric*])
  using *refl trans* **unfolding** *reflp_def transp_def* **by** *blast*+

end

context *Regions_TA*
begin

definition *sim_regions* (**infix** $\equiv_M$ *60*) **where**
  sim_regions $\equiv \lambda(l,\ u)\ (l',\ u').$
    $(l' = l \wedge l \in state\_set\ A \wedge (\exists\, R \in \mathcal{R}\ l.\ u \in R \wedge u' \in R))$
  $\vee\ (l \notin state\_set\ A \vee u \notin V) \wedge (l' \notin state\_set\ A \vee u' \notin V)$

abbreviation
  valid $\equiv \lambda(l,\ u).\ l \in state\_set\ A \wedge u \in V$

lemma $\mathcal{R}\_I$:
  assumes $l \in state\_set\ A\ u \in V$
  shows $\exists\, R \in \mathcal{R}\ l.\ u \in R$
  using *assms regions_partition*[**where** $\mathcal{R} = \langle\mathcal{R}\ l\rangle$ **and** $X = X$ **and** $k = k\ l$ **and** $u = u$] $\mathcal{R}\_def$[*of l*]
  unfolding *V_def* **by** *blast*

lemma *regions_finite*:
  finite $(\mathcal{R}\ l)$
  using *finite_*$\mathcal{R}$[*OF finite*] **unfolding** $\mathcal{R}\_def$ **.**

lemma *valid_iff*:
  valid $(l,\ u) \longleftrightarrow$ *valid* $(l',\ u')$ **if** $(l,\ u) \equiv_M (l',\ u')$

using *that* **unfolding** *sim_regions_def* **by** (*auto dest*: $\mathcal{R}\_V$)

lemma *refl*:
$(l, u) \equiv_M (l, u)$
unfolding *sim_regions_def* **by** (*cases valid* (l, u); *simp add*: $\mathcal{R}\_I$)

lemma *sym*:
$(l, u) \equiv_M (l', u') \longleftrightarrow (l', u') \equiv_M (l, u)$
unfolding *sim_regions_def* **by** *auto*

lemma *trans*:
$(l, u) \equiv_M (l'', u'')$ **if** $(l, u) \equiv_M (l', u')$ $(l', u') \equiv_M (l'', u'')$
proof (*cases valid* (l, u))
case *True*
with *that* **have** *valid* (l, u) *valid* (l', u') *valid* (l'', u'')
using *valid_iff* **by** *metis+*
then show *?thesis*
using *that* **unfolding** *sim_regions_def* **by** (*auto dest*: $\mathcal{R}$*_regions_distinct*[*rotated 2*])
next
case *False*
with *that* **have** $\neg$ *valid* (l, u) $\neg$ *valid* (l', u') $\neg$ *valid* (l'', u'')
using *valid_iff* **by** *metis+*
then show *?thesis*
unfolding *sim_regions_def* **by** *simp*
qed

lemma *equiv*:
equivp $(\equiv_M)$
using *refl sym trans* **by** − (*rule equivpI*; *unfold equivp_def reflp_def symp_def transp_def*; *fast*)

lemma *same_loc*:
$l' = l$ **if** $(l, u) \equiv_M (l', u')$ *valid* (l, u)
using *that* **unfolding** *sim_regions_def* **by** *auto*

lemma *regions_simI*:
$(l, u) \equiv_M (l, u')$ **if** $l \in$ *state_set* A $R \in \mathcal{R}$ l $u \in R$ $u' \in R$
using *that* **unfolding** *sim_regions_def* **by** *auto*

lemma *regions_simD*:
$u' \in R$ **if** $l \in$ *state_set* A $R \in \mathcal{R}$ l $u \in R$ $(l, u) \equiv_M (l', u')$
using *that* **unfolding** *sim_regions_def* **by** (*auto dest*: $\mathcal{R}\_V$ $\mathcal{R}$*_regions_distinct*)

lemma *finite_quotient*:
finite (*UNIV* // $\{(x, y).\ x \equiv_M y\}$)
proof −
let *?S* = *state_set* $A \times (\bigcup l \in$ *state_set* $A.\ \mathcal{R}\ l)$ **and** *?f* = $\lambda(l, R).$ *from_R* l R
let *?invalid* = $\{(l, u).\ \neg$*valid* $(l, u)\}$
have *Collect* $((\equiv_M)\ (l, u)) \in$ *?f* ‘ *?S*
if *valid* (l, u) **for** l u
proof −
from *that refl*[*of* l u] **obtain** R **where** $*$: $l \in$ *state_set* A $R \in \mathcal{R}$ l $u \in R$
unfolding *sim_regions_def* **by** *auto*
with ‹*valid* _› **have** *Collect* $((\equiv_M)\ (l, u)) =$ *from_R* l R

unfolding *from_R_def* **by** (*auto simp*: *same_loc intro*: *regions_simI regions_simD*)
with ∗ **show** *?thesis*
by *auto*
qed
moreover have *Collect* ((≡$_M$) (*l*, *u*)) = *?invalid* **if** ¬ *valid* (*l*, *u*) **for** *l u*
using *that* **unfolding** *sim_regions_def* **by** (*auto simp*: $\mathcal{R}$_*V*)
ultimately have *UNIV* // {(*x*, *y*). *x* ≡$_M$ *y*} ⊆ (*?f* ‘ *?S*) ∪ {*?invalid*}
apply −
apply (*rule subsetI*)
apply (*erule quotientE*)
apply *clarsimp*
by *blast*
also have *finite* . . .
by (*blast intro*: *finite_state_set regions_finite*)+
finally show *?thesis* **.**
qed

sublocale *region_self_simulation*: *Self_Simulation* **where**
E = λ(*l*, *u*) (*l′*, *u′*). *A* ⊢′ ⟨*l*, *u*⟩ → ⟨*l′*, *u′*⟩ **and** *sim* = (≡$_M$) **and** *P* = *valid*
apply (*standard*; *clarsimp?*)
subgoal *simulation* **premises** *prems* **for** *l u l1 u1 l′ u′*
proof −
from ‹*u* ∈ *V*› ‹*A* ⊢′ ⟨*l*, *u*⟩ → ⟨*l1*, *u1*⟩›[*THEN step_r′_complete_spec*] **obtain** *a R1* **where**
u1 ∈ *R1 A*,$\mathcal{R}$ ⊢ ⟨*l*, [*u*]$_l$⟩ ⇝$_a$ ⟨*l1*, *R1*⟩
by *blast*
moreover from *prems* **have** *u′* ∈ [*u*]$_l$
unfolding *V_def* **by** (*auto elim*: *regions_simD dest*: *region_cover′*)
ultimately obtain *u1′* **where** *u1′* ∈ *R1 A* ⊢′ ⟨*l*, *u′*⟩ → ⟨*l1*, *u1′*⟩
by (*auto 4 3 dest*: *step_r′_sound*)
moreover from ‹*u1* ∈ *R1*› ‹*u1′* ∈ *R1*› ‹*A*,$\mathcal{R}$ ⊢ ⟨*l*, [*u*]$_l$⟩ ⇝$_a$ ⟨*l1*, *R1*⟩› **have** (*l1*, *u1*) ≡$_M$ (*l1*, *u1′*)
by (*meson regions_simI step_r′_*$\mathcal{R}$ *step_r′_state_set*)
moreover from *prems* **have** *valid* (*l′*, *u′*)
using *valid_iff* **by** *auto*
moreover from *prems* **have** *l′* = *l*
by − (*erule same_loc*, *simp*)
ultimately show *?thesis*
using ‹(*l*, *u*) ≡$_M$ (*l′*, *u′*)› **by** *blast*
qed
subgoal *invariant*
by (*meson* $\mathcal{R}$_*V step_r′_*$\mathcal{R}$ *step_r′_complete_spec step_r′_state_set*)
using *refl trans* **unfolding** *reflp_def transp_def* **by** *fast*+

end

8.3 LU-Simulation

definition
constraints_of A l = ⋃ (*set* ‘ *insert* (*inv_of A l*) {*g*. ∃ *a r l′*. (*l*, *g*, *a*, *r*, *l′*) ∈ *trans_of A*})

definition
is_lower A L ≡
∀ *l*. ∀ *ac* ∈ *constraints_of A l*. *case ac of*
GT c x ⇒ *L l c* ≥ *x* |

$GE\ c\ x \Rightarrow L\ l\ c \geq x \mid$
$EQ\ c\ x \Rightarrow L\ l\ c \geq x \mid$
$\_ \Rightarrow True$

definition
$is\_upper\ A\ U \equiv$
$\forall\, l.\ \forall\, ac \in constraints\_of\ A\ l.\ case\ ac\ of$
$LT\ c\ x \Rightarrow U\ l\ c \geq x \mid$
$LE\ c\ x \Rightarrow U\ l\ c \geq x \mid$
$EQ\ c\ x \Rightarrow U\ l\ c \geq x \mid$
$\_ \Rightarrow True$

definition
$is\_locally\_consistent\ A\ k \equiv$
$\forall\, (l,\ g,\ a,\ r,\ l') \in trans\_of\ A.\ \forall\, x \in clk\_set\ A - set\ r.\ k\ l\ x \geq k\ l'\ x$

lemma *is_locally_consistentD*:
assumes $is\_locally\_consistent\ A\ k\ A \vdash l \longrightarrow^{g,a,r} l_1$
shows $\forall\, x \in clk\_set\ A - set\ r.\ k\ l\ x \geq k\ l_1\ x$
using *assms* **unfolding** *is_locally_consistent_def* **by** *fast*

locale $TA\_LU =$
fixes $A :: ('a,\ 'c,\ 't :: time,\ 'l)\ ta$
fixes $L :: 'l \Rightarrow 'c \Rightarrow 't$ **and** $U :: 'l \Rightarrow 'c \Rightarrow 't$
assumes *is_lower*: $is\_lower\ A\ L$ **and** *is_upper*: $is\_upper\ A\ U$
and *locally_consistent*: $is\_locally\_consistent\ A\ L\ is\_locally\_consistent\ A\ U$
begin

definition $sim :: 'l \times ('c \Rightarrow 't :: time) \Rightarrow 'l \times ('c \Rightarrow 't) \Rightarrow bool$ (**infix** $\preceq$ *60*) **where**
$sim \equiv \lambda(l,\ v)\ (l',\ v').$
$l' = l \wedge (\forall\, x \in clk\_set\ A.\ (v'\ x < v\ x \longrightarrow v'\ x > L\ l\ x) \wedge (v'\ x > v\ x \longrightarrow v\ x > U\ l\ x))$

lemma *simE*:
assumes $(l,\ v) \preceq (l',\ v')\ x \in clk\_set\ A$
obtains $l' = l\ v\ x = v'\ x$
$\mid l' = l\ v\ x > v'\ x\ v'\ x > L\ l\ x$
$\mid l' = l\ v\ x < v'\ x\ v\ x > U\ l\ x$
using *assms* **unfolding** *sim_def* **by** *force*

lemma *sim_locD*:
$l' = l$ **if** $(l,\ v) \preceq (l',\ v')$
using *that* **unfolding** *sim_def* **by** *auto*

lemma *sim_nonneg*:
$u\ x \geq 0$ **if** $(l,\ u) \preceq (l',\ u')\ u'\ x \geq 0\ x \in clk\_set\ A\ U\ l\ x \geq 0$
using *that* **by** (*auto elim*: *simE*)

lemma *sim_time_shift*:
$(l,\ v \oplus d) \preceq (l',\ v' \oplus d)$ **if** $(l,\ v) \preceq (l',\ v')\ d \geq 0$
using *that* **unfolding** *cval_add_def sim_def* **by** *simp* (*metis add.commute add_strict_increasing2*)

lemma *constraints_of_clk_set*:
assumes $g \in constraints\_of\ A\ l$
shows

$g = LT\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = LE\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = EQ\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = GE\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = GT\ c\ x \Longrightarrow c \in clk\_set\ A$
using *assms*
unfolding *constraints_of_def*
unfolding *collect_clkvt_def clkp_set_def*
unfolding
Timed_Automata.clkp_set_def Timed_Automata.collect_clki_def Timed_Automata.collect_clkt_def
unfolding *collect_clock_pairs_def*
by *auto* (*smt UnCI Union_iff constraint_pair.simps fst_conv image_eqI mem_Collect_eq*)+

lemma *constraint_simulation*:
assumes $g \in constraints\_of\ A\ l\ (l,\ v) \preceq (l',\ v')\ v \vdash_a g$
shows $v' \vdash_a g$
using *assms*(*3*,*1*,*2*) *is_lower is_upper* **unfolding** *is_lower_def is_upper_def*
by *cases*(*all* ‹*frule* (*1*) *constraints_of_clk_set*; *erule* (*1*) *simE*; *fastforce simp*: *clock_val_a.simps*›)

lemma *inv_simulation*:
assumes $v \vdash inv\_of\ A\ l\ (l,\ v) \preceq (l',\ v')$
shows $v' \vdash inv\_of\ A\ l$
proof −
from *assms*(*1*) **have** $\forall\, ac \in set\ (inv\_of\ A\ l).\ v \vdash_a ac$
unfolding *clock_val_def list_all_iff* **by** *auto*
moreover have $\forall\, ac \in set\ (inv\_of\ A\ l).\ ac \in constraints\_of\ A\ l$
unfolding *constraints_of_def* **by** *auto*
ultimately show *?thesis*
using ‹_ ≼ _› **unfolding** *clock_val_def list_all_iff* **by** (*auto intro*: *constraint_simulation*)
qed

lemma *guard_simulation*:
assumes $A \vdash l \longrightarrow^{g,a,r} l_1\ v \vdash g\ (l,\ v) \preceq (l',\ v')$
shows $v' \vdash g$
proof −
from *assms*(*2*) **have** $\forall\, ac \in set\ g.\ v \vdash_a ac$
unfolding *clock_val_def list_all_iff* **by** *auto*
moreover from *assms*(*1*) **have** $\forall\, ac \in set\ g.\ ac \in constraints\_of\ A\ l$
unfolding *constraints_of_def* **by** *auto*
ultimately show *?thesis*
using ‹_ ≼ _› **unfolding** *clock_val_def list_all_iff* **by** (*auto intro*: *constraint_simulation*)
qed

lemma *sim_delay*:
assumes $(l,\ v) \preceq (l',\ v')\ d \geq 0$
shows $(l,\ v \oplus d) \preceq (l',\ v' \oplus d)$
using *assms* **unfolding** *cval_add_def sim_def* **by** (*auto simp*: *add_strict_increasing2 gt_swap*)

lemma *clock_set_iff*:
$([r{\rightarrow}0]v)\ c = (if\ c \in set\ r\ then\ 0\ else\ v\ c)$
by *auto*

lemma *sim_reset*:
assumes $A \vdash l \longrightarrow^{g,a,r} l_1\ v \vdash g\ (l,\ v) \preceq (l',\ v')$

```
  shows (l₁, [r→0]v) ≼ (l₁, [r→0]v')
proof −
  from assms(1) have
    ∀ x ∈ clk_set A − set r. L l x ≥ L l₁ x ∀ x ∈ clk_set A − set r. U l x ≥ U l₁ x
    using locally_consistent by − (intro is_locally_consistentD; assumption)+
  then show ?thesis
    using assms(2,3) unfolding sim_def by (auto simp: clock_set_iff) force+
qed

lemma step_t_simulation:
  (l, u) ≼ (l', u') ⟹ A ⊢ ⟨l, u⟩ →^d ⟨l₁, u₁⟩
  ⟹ ∃ u₁'. A ⊢ ⟨l₁, u'⟩ →^d ⟨l₁, u₁'⟩ ∧ (l₁, u₁) ≼ (l, u₁')
  unfolding step_t.simps by (auto dest: sim_delay inv_simulation sim_locD)

lemma step_a_simulation:
  (l, u) ≼ (l', u') ⟹ A ⊢ ⟨l, u⟩ →_a ⟨l₁, u₁⟩
  ⟹ ∃ u₁'. A ⊢ ⟨l, u'⟩ →_a ⟨l₁, u₁'⟩ ∧ (l₁, u₁) ≼ (l₁, u₁')
  unfolding step_a.simps
  apply clarsimp
  apply (frule (2) guard_simulation)
  apply (drule (2) sim_reset[rotated −1])
  apply (frule (1) inv_simulation)
  apply auto
  done

lemma step_trans_simulation:
  (l, u) ≼ (l', u') ⟹ A ⊢_t ⟨l, u⟩ →_t ⟨l₁, u₁⟩
  ⟹ ∃ u₁'. A ⊢_t ⟨l, u'⟩ →_t ⟨l₁, u₁'⟩ ∧ (l₁, u₁) ≼ (l₁, u₁')
  unfolding step_trans.simps
  apply clarsimp
  apply (frule (2) guard_simulation)
  apply (drule (2) sim_reset[rotated −1])
  apply (frule (1) inv_simulation)
  apply auto
  done

sublocale Time_Abstract_Simulation A sim
  apply standard
  subgoal
    unfolding step_trans'.simps using step_t_simulation step_trans_simulation
    by simp (metis sim_locD)
  subgoal
    unfolding sim_def by auto
  subgoal premises prems for u v w
  proof −
    define clks where clks = clk_set A
    from prems show ?thesis
      unfolding sim_def clks_def[symmetric] by fastforce+
  qed
  done

end
```

8.4 Simulation on Reachability Invariants

locale *Invariant_Simulation* =
  fixes $L :: 'l\ set$ **and** $M :: 'l \Rightarrow 's\ set$
    and $SE\ E\ SE'\ E'\ sim :: ('l \times 's) \Rightarrow ('l \times 's) \Rightarrow bool$
  assumes *SE_SE'*:
    $\bigwedge l\ l'\ x\ y\ x'.\ sim\ (l,\ x)\ (l,\ x') \Longrightarrow SE\ (l,\ x)\ (l',\ y)$
    $\Longrightarrow \exists\, y'.\ SE'\ (l,\ x')\ (l',\ y') \wedge sim\ (l',\ y)\ (l',\ y')$
  assumes *SE'_SE*:
    $\bigwedge l\ l'\ x\ y\ x'\ y'.\ sim\ (l,\ x)\ (l,\ x') \Longrightarrow sim\ (l',\ y)\ (l',\ y') \Longrightarrow SE'\ (l,\ x')\ (l',\ y')$
    $\Longrightarrow SE\ (l,\ x)\ (l',\ y)$
  and *E'_E*:
    $\bigwedge l\ l'\ a\ a'\ b'.\ sim\ (l,\ a)\ (l,\ a') \Longrightarrow E'\ (l,\ a')\ (l',\ b')$
    $\Longrightarrow (\exists\, b.\ E\ (l,\ a)\ (l',\ b) \wedge sim\ (l',\ b)\ (l',\ b'))$
begin

definition
  $M' \equiv \lambda l.\ \{x'.\ \exists\, x \in M\ l.\ sim\ (l,\ x)\ (l,\ x')\}$

lemma *invariant_simulation*:
  assumes
    $\forall\, l \in L.\ \forall\, s \in M\ l.\ \forall\, l'\ s'.\ E\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s'' \in M\ l'.\ SE\ (l',\ s')\ (l',\ s''))$
  shows
    $\forall\, l \in L.\ \forall\, s \in M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s'' \in M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
  apply *safe*
  subgoal
    using *assms* **unfolding** *M'_def* **by** (*auto dest*: *E'_E*)
  subgoal premises *prems*
  proof −
    have $\exists\, s''.\ (\exists\, x{\in}M\ l'.\ sim\ (l',\ x)\ (l',\ s'')) \wedge SE'\ (l',\ s')\ (l',\ s'')$
      if $l \in L\ E'\ (l,\ s)\ (l',\ s')\ x \in M\ l\ sim\ (l,\ x)\ (l,\ s)$
      for $l :: 'l$ **and** $s :: 's$ **and** $l' :: 'l$ **and** $s' :: 's$ **and** $x :: 's$
    proof −
      from *that E'_E* **obtain** x' **where** $sim\ (l',\ x')\ (l',\ s')\ E\ (l,\ x)\ (l',\ x')$
        by *force*
      with ‹$l \in L$› ‹$x \in M\ l$› *assms* **obtain** x'' **where** $x'' \in M\ l'\ SE\ (l',\ x')\ (l',\ x'')$
        by *force*
      from *this*(*2*) ‹$sim\ (l',\ x')$ _› **obtain** s'' **where**
        $SE'\ (l',\ s')\ (l',\ s'')\ sim\ (l',\ x'')\ (l',\ s'')$
        by *atomize_elim* (*rule SE_SE'*)
      with ‹$x'' \in$ _› **show** $\exists\, s''.\ (\exists\, x{\in}M\ l'.\ sim\ (l',\ x)\ (l',\ s'')) \wedge SE'\ (l',\ s')\ (l',\ s'')$
        by *auto*
    qed
    with *prems* **show** *?thesis*
      unfolding *M'_def* **by** *auto*
  qed
  done

interpretation *Simulation* **where**
  $A = E'$ **and**
  $B = E$ **and**
  $sim = \lambda(l,\ s)\ (l',\ s').\ l' = l \wedge sim\ (l,\ s')\ (l',\ s)$
  by *standard* (*auto dest*: *E'_E*)

context
fixes $f :: 'l \times 's \Rightarrow nat$
begin

definition
$f' \equiv \lambda(l, s).\ Max\ (\{f\ (l, s') \mid s'.\ sim\ (l, s')\ (l, s) \wedge s' \in M\ l\})$

context
assumes *finite*: $finite\ L \forall\ l \in L.\ finite\ (M\ l)$
assumes *f_topo*: $\bigwedge l\ s\ l1\ s1\ l2\ s2.$
$l \in L \Longrightarrow s \in M\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M\ l2 \Longrightarrow E\ (l, s)\ (l1, s1) \Longrightarrow SE\ (l1, s1)\ (l2, s2)$
$\Longrightarrow$
$f\ (l, s) \leq f\ (l2, s2)$
begin

lemma *topo_simulation*: $\bigwedge l\ s\ l1\ s1\ l2\ s2.$
$l \in L \Longrightarrow s \in M'\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M'\ l2 \Longrightarrow E'\ (l, s)\ (l1, s1) \Longrightarrow SE'\ (l1, s1)\ (l2, s2)$
$\Longrightarrow$
$f'\ (l, s) \leq f'\ (l2, s2)$
subgoal premises *prems* **for** $l\ s\ l1\ s1\ l2\ s2$
proof −
have $f'\ (l, s) \leq f'\ (l'', s'')$
if $l \in L$
and $l'' \in L$
and $E'\ (l, s)\ (l', s')$
and $SE'\ (l', s')\ (l'', s'')$
and $x \in M\ l$
and $sim\ (l, x)\ (l, s)$
and $x'' \in M\ l''$
and $sim\ (l'', x'')\ (l'', s'')$
for $l\ s\ l'\ s'\ l''\ s''\ x\ x''$
proof −
let $?S = \lambda\ l''\ s''.\ \{f\ (l'', s')\ |s'.\ sim\ (l'', s')\ (l'', s'') \wedge s' \in M\ l''\}$
have *finiteI*: $finite\ (?S\ l\ s)$ **if** $l \in L$ **for** $l\ s$
using *finite that* **using** [[*simproc add*: *finite_Collect*]] **by** *simp*
have $Max\ (?S\ l\ s) \in ?S\ l\ s$
using ‹$l \in L$› ‹$sim\ (l, x)$ _› ‹$x \in$ _› **by** (*intro Max_in*) (*auto intro*: *finiteI*)
then obtain y **where** $f'\ (l, s) = f\ (l, y)\ sim\ (l, y)\ (l, s)\ y \in M\ l$
unfolding *f'_def* **by** *auto*
with *E'_E* ‹$E'\ (l,s)$ _› ‹$sim\ (l, x)$ _› **obtain** x' **where**
$E\ (l, y)\ (l', x')\ sim\ (l', x')\ (l', s')$
by *metis*
moreover from ‹$SE'\ (l',s')$ _› ‹$sim\ (l', x')$ _› ‹$sim\ (l'', x'')$ _› **have** $SE\ (l', x')\ (l'', x'')$
using *SE'_SE* **by** *metis*
ultimately have $f\ (l, y) \leq f\ (l'', x'')$
using *that f_topo*[*of l y l'' x'' l' x'*] ‹$y \in M\ l$› **by** *auto*
with ‹$\_ = f\ (l, y)$›
have $f'\ (l, s) \leq f\ (l'', x'')$
by *simp*
also from ‹$x'' \in$ _› ‹$l'' \in$ _› ‹$sim\ (l'', x'')$ _› **have** $\ldots \leq f'\ (l'', s'')$
unfolding *f'_def* **by** (*auto intro*: *finiteI Max_ge*)
finally show *?thesis* **.**
qed
then show *?thesis*
using *prems* **unfolding** *M'_def* **by** *auto*

qed
done

end

end

end

8.5 Abstraction-Simulation on Reachability Invariants

locale *Abstraction_Simulation* =
fixes $L :: 'l\ set$ **and** $M :: 'l \Rightarrow 's\ set$
and $SE\ E\ SE' :: ('l \times 's) \Rightarrow ('l \times 's) \Rightarrow bool$
and $\alpha :: 'l \Rightarrow 's \Rightarrow 's$
assumes SE_SE': $\bigwedge l\ l'\ x\ y.\ SE\ (l,\ x)\ (l',\ \alpha\ l'\ y) \Longrightarrow SE'\ (l,\ \alpha\ l\ x)\ (l',\ \alpha\ l'\ y)$
assumes SE'_SE: $\bigwedge l\ l'\ x\ y.\ SE'\ (l,\ \alpha\ l\ x)\ (l',\ \alpha\ l'\ y) \Longrightarrow SE\ (l,\ x)\ (l',\ \alpha\ l'\ y)$
and *simulation*:
$\bigwedge l\ l'\ a\ a'\ b.$
$\alpha\ l\ a = \alpha\ l\ a' \Longrightarrow E\ (l,\ a)\ (l',\ b) \Longrightarrow (\exists\, b'.\ E\ (l,\ a')\ (l',\ b') \wedge \alpha\ l'\ b = \alpha\ l'\ b')$
begin

definition
$M' \equiv \lambda l.\ \alpha\ l\ `\ M\ l$

inductive E' **where**
$E\ (l,\ s)\ (l',\ s') \Longrightarrow E'\ (l,\ \alpha\ l\ s)\ (l',\ \alpha\ l'\ s')$

sublocale *sim*: *Invariant_Simulation* **where**
$sim = \lambda(l,\ x)\ (l',\ y).\ l' = l \wedge y = \alpha\ l\ x$ **and**
$SE = \lambda(l,\ x)\ (l',\ y).\ SE\ (l,\ x)\ (l',\ \alpha\ l'\ y)$ **and**
$SE' = \lambda(l,\ x)\ (l',\ y).\ SE'\ (l,\ x)\ (l',\ y)$ **and**
$E = E$ **and**
$E' = E'$
apply *standard*
subgoal for $l\ l'\ x\ y\ x'$
apply *clarsimp*
by (*rule SE_SE'*)
subgoal for $l\ l'\ x\ y\ x'\ y'$
apply *clarsimp*
by (*rule SE'_SE*)
subgoal for $l\ l'\ a\ a'\ b'$
using *simulation* **by** (*auto elim*!: *E'.cases*)
done

interpretation *sim2*: *Simulation* **where**
$sim = \lambda(l,\ x)\ (l',\ y).\ l' = l \wedge y = \alpha\ l\ x$ **and**
$A = E$ **and**
$B = E'$
by *standard* (*force simp*: *E'.simps*)

interpretation *bisim*: *Bisimulation* **where**
$sim = \lambda(l,\ x)\ (l',\ y).\ l' = l \wedge y = \alpha\ l\ x$ **and**
$A = E$ **and**

$B = E'$
apply *standard*
subgoal
using *sim2.A_B_step* **.**
subgoal for $a\ a'\ b'$
by (*cases* b') (*auto dest*: *sim.E'_E*[*rotated*])
done

lemma *simulationE*:
assumes $\alpha\ l\ a = \alpha\ l\ a'\ E\ (l,\ a)\ (l',\ b)$
obtains b' **where** $E\ (l,\ a')\ (l',\ b')\ \alpha\ l'\ b = \alpha\ l'\ b'$
using *assms simulation* **by** *blast*

lemma *M'_eq*:
$sim.M' = M'$
unfolding *sim.M'_def M'_def* **by** *auto*

lemma *invariant_simulation*:
assumes
$\forall\, l{\in}L.\ \forall\, s{\in}M\ l.\ \forall\, l'\ s'.\ E\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M\ l'.\ SE\ (l',\ s')\ (l',\ \alpha\ l'\ s''))$
shows
$\forall\, l{\in}L.\ \forall\, s{\in}M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
using *sim.invariant_simulation assms* **unfolding** *M'_eq* **by** *fast*

lemma — Alternative proof of: $\forall\, l{\in}L.\ \forall\, s{\in}M\ l.\ \forall\, l'\ s'.\ (l,\ s) \to (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M\ l'.\ SE\ (l',\ s')\ (l',\ \alpha\ l'\ s'')) \Longrightarrow \forall\, l{\in}L.\ \forall\, s{\in}M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
assumes
$\forall\, l{\in}L.\ \forall\, s{\in}M\ l.\ \forall\, l'\ s'.\ E\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M\ l'.\ SE\ (l',\ s')\ (l',\ \alpha\ l'\ s''))$
shows
$\forall\, l{\in}L.\ \forall\, s{\in}M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
proof −
{ fix $x\ \ l\ s\ l'\ s'$
assume $l \in L\ x \in M\ l\ (l,\ s) \to (l',\ s')\ \alpha\ l\ x = \alpha\ l\ s$
then have $l' \in L$
using *assms simulation* **by** *metis*
}
moreover
{ fix $l\ l' :: 'l$ **and** $s\ s'\ x :: 's$
assume $l \in L\ E\ (l,\ s)\ (l',\ s')\ \alpha\ l\ x = \alpha\ l\ s\ x \in M\ l$
with *simulation* **obtain** x' **where** $\alpha\ l'\ s' = \alpha\ l'\ x'\ E\ (l,\ x)\ (l',\ x')$
by *metis*
with ‹$l \in L$› ‹$x \in M\ l$› *assms* **obtain** x'' **where** $x'' \in M\ l'\ SE\ (l',\ x')\ (l',\ \alpha\ l'\ x'')$
by *force*
from *this*(*2*) **have** $SE'\ (l',\ \alpha\ l'\ x')\ (l',\ \alpha\ l'\ x'')$
by (*rule SE_SE'*)
with ‹$x'' \in \_$› ‹$\alpha\ l'\ s' = \_$› **have** $\exists\, s''{\in}M\ l'.\ SE'\ (l',\ \alpha\ l'\ s')\ (l',\ \alpha\ l'\ s'')$
by *auto*
}
ultimately show *?thesis*
unfolding *M'_def* **by** − (*safe*; (*erule E'.cases*; *clarsimp*))
qed

context
fixes $f :: 'l \times 's \Rightarrow nat$
assumes *finite*: *finite* $L \forall\ l \in L$. *finite* $(M\ l)$
assumes *f_topo*: $\bigwedge l\ s\ l1\ s1\ l2\ s2$.
$l \in L \Longrightarrow s \in M\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M\ l2 \Longrightarrow E\ (l, s)\ (l1, s1) \Longrightarrow SE\ (l1, s1)\ (l2, \alpha\ l2\ s2)$
$\Longrightarrow f\ (l, s) \leq f\ (l2, s2)$
begin

definition
$f' \equiv \lambda(l, s).\ Max\ (\{f\ (l, s') \mid s'.\ \alpha\ l\ s' = s \wedge s' \in M\ l\})$

lemma *f'_eq*:
sim.f' f = *f'*
unfolding *sim.f'_def f'_def* **by** (*rule ext*; *clarsimp*; *metis*)

lemma *topo_simulation*: $\bigwedge l\ s\ l1\ s1\ l2\ s2$.
$l \in L \Longrightarrow s \in M'\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M'\ l2 \Longrightarrow E'\ (l, s)\ (l1, s1) \Longrightarrow SE'\ (l1, s1)\ (l2, s2) \Longrightarrow$
$f'\ (l, s) \leq f'\ (l2, s2)$
by (*rule sim.topo_simulation*[**where** *f* = *f*, *OF finite*, *unfolded M'_eq f'_eq*])
((*rule f_topo*; *simp*; *fail*), *simp*+)

end

end

8.6 Instantiation for Abstractions based on Time-Abstraction Simulations

context *Time_Abstract_Simulation*
begin

context
fixes *SE* :: $('l \times ('c, 't)\ zone) \Rightarrow ('l \times ('c, 't)\ zone) \Rightarrow bool$
assumes *SE_subsumption*: $\bigwedge l\ l'\ Z\ Z'$. *SE* $(l, Z)\ (l', Z') \Longrightarrow l' = l \wedge Z \subseteq \alpha\ l'\ Z'$
and *SE_determ*:
$\bigwedge l\ l'\ Z\ Z'\ W$. *SE* $(l, Z)\ (l', Z') \Longrightarrow \alpha\ l\ Z = \alpha\ l\ W \Longrightarrow SE\ (l, W)\ (l', Z')$
begin

lemma *step_z'_step_trans_z'_iff*:
$A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z''\rangle \longleftrightarrow (\exists\, t.\ A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z''\rangle)$
using *step_trans_z'_step_z' step_z'_step_trans_z'* **by** *metis*

interpretation *Abstraction_Simulation* **where**
$SE = \lambda(l, Z)\ (l', Z').\ \exists\ W.\ Z' = \alpha\ l\ W \wedge SE\ (l, Z)\ (l', W)$ **and**
$E = \lambda(l, Z)\ (l', Z').\ A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle$ **and**
$SE' = \lambda(l, Z)\ (l', Z').\ \exists\ W\ W'.\ Z = \alpha\ l\ W \wedge Z' = \alpha\ l'\ W' \wedge SE\ (l, W)\ (l', W')$ **and**
$\alpha = abs$
apply (*standard*; *clarsimp*)
subgoal
using *SE_subsumption* **by** *metis*
subgoal for *l l' x y W W'*

using *SE_subsumption SE_determ* **by** *metis*
subgoal
unfolding *step_z'_step_trans_z'_iff* **using** *simulation'* **by** *metis*
done

interpretation *inv*: *Invariant_Simulation* **where**
$SE = \lambda(l, Z)\ (l', Z').\ \exists W.\ l' = l \wedge \alpha\ l\ Z' = \alpha\ l\ W \wedge SE\ (l, Z)\ (l', W)$ **and**
$E = \lambda(l, Z)\ (l', Z').\ A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle$ **and** $E' = E'$ **and**
$SE' = \lambda(l, Z)\ (l', Z').\ \exists W\ W'.\ l' = l \wedge Z = \alpha\ l\ W \wedge Z' = \alpha\ l'\ W' \wedge SE\ (l, W)\ (l', W')$ **and**
$sim = \lambda(l, Z)\ (l', Z').\ l' = l \wedge Z' = \alpha\ l\ Z$
apply (*standard*; *clarsimp simp*: *abs_involutive*)
subgoal for $l\ Z'\ W$
by *blast*
subgoal for $l\ Z\ W\ Z'\ W'$
using *SE_determ* **by** *auto*
subgoal for $l\ l'\ Z\ Z'$
using *sim.E'_E* **by** *auto*
done

end

end

8.7 "Sandwiches" of Abstraction-Simulations

locale *Time_Abstract_Simulation_Sandwich* =
Regions_TA **where** $A = A$ +
Time_Abstract_Simulation **where** $A = A$ **for** $A :: ('a, 'c, real, 'l)\ ta$ +
assumes *sim_V*: $(l, u) \preceq (l', u') \Longrightarrow u' \in V \Longrightarrow u \in V$

fixes $I\ \beta$
assumes *I_invariant*: $I\ Z \Longrightarrow A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle \Longrightarrow I\ Z'$
assumes β_α: $I\ Z \Longrightarrow Z \subseteq V \Longrightarrow l \in state\_set\ A \Longrightarrow \beta\ l\ Z \subseteq \alpha\ l\ Z$
and *β_widens*: $I\ Z \Longrightarrow Z \subseteq V \Longrightarrow l \in state\_set\ A \Longrightarrow Z \subseteq \beta\ l\ Z$
and *β_I*: $I\ Z \Longrightarrow Z \subseteq V \Longrightarrow l \in state\_set\ A \Longrightarrow I\ (\beta\ l\ Z)$
and *finite_abstraction*: *finite* $\{\beta\ l\ Z \mid l\ Z.\ I\ Z \wedge Z \subseteq V \wedge l \in state\_set\ A\}$

fixes $l_0 :: 'l$ **and** $Z_0 :: ('c, real)\ zone$
assumes l_0*_state_set*: $l_0 \in state\_set\ A$ **and** Z_0*_V*: $Z_0 \subseteq V$ **and** Z_0*_I*: $I\ Z_0$
begin

inductive *step_beta* ::
$('a, 'c, real, 'l)\ ta \Rightarrow 'l \Rightarrow ('c, real)\ zone \Rightarrow 'a \Rightarrow 'l \Rightarrow ('c, real)\ zone \Rightarrow bool$
$(\_ \vdash \langle\_, \_\rangle \rightsquigarrow_{\beta(\_)} \langle\_, \_\rangle\ [61,61,61]\ 61)$
where
step_beta:
$A \vdash \langle l, Z\rangle \rightsquigarrow_{\tau} \langle l', Z'\rangle \Longrightarrow A \vdash \langle l', Z'\rangle \rightsquigarrow_{\uparrow a} \langle l'', Z''\rangle$
$\Longrightarrow A \vdash \langle l, Z\rangle \rightsquigarrow_{\beta(a)} \langle l'', \beta\ l''\ Z''\rangle$

no_notation *step_z_beta* $(\_ \vdash \langle\_, \_\rangle \rightsquigarrow_{\beta(\_)} \langle\_, \_\rangle\ [61,61,61,61]\ 61)$

no_notation *step_z_alpha* $(\_ \vdash \langle\_, \_\rangle \rightsquigarrow_{\alpha(\_)} \langle\_, \_\rangle\ [61,61,61]\ 61)$

lemma *step_beta_alt_def*:

$(\exists\, a.\ A \vdash \langle l,\ Z\rangle \leadsto_{\beta(a)} \langle l',\ W\rangle) \longleftrightarrow (\exists\, Z'.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge\ W = \beta\ l'\ Z')$
unfolding *step_beta.simps step_z'_def* **by** *auto*

lemma *step_betaE*:
assumes $A \vdash \langle l,\ Z\rangle \leadsto_{\beta(a)} \langle l',\ W\rangle$
obtains Z' **where** $A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle\ W = \beta\ l'\ Z'$
using *step_beta_alt_def assms* **by** *metis*

definition
loc_is l s $\equiv \forall\, (l', \_) \in s.\ l' = l$

lemma α_V:
$\alpha\ l\ Z \subseteq V$ **if** $Z \subseteq V$
using *that sim_V* **unfolding** *V_def abs_def* **by** *auto*

lemma β_V:
$\beta\ l\ Z \subseteq V$ **if** $Z \subseteq V\ I\ Z\ l \in$ *state_set A*
using $\beta\_\alpha\ \alpha\_V$ *that* **by** *blast*

lemma *step_z'_V*:
$Z' \subseteq V$ **if** $A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle\ Z \subseteq V$
by (*meson step_z_V step_z'_def that*)

Corresponds to lemma 6 of [1].

lemma *backward_simulation*:
assumes
$b \in S'$ *loc_is l S loc_is l' S' A* $\vdash \langle l,$ *R_of S*$\rangle \leadsto_{\beta(a)} \langle l',$ *R_of S'*$\rangle$
I (*R_of S*) *R_of S* $\subseteq V$
shows $\exists\, a{\in}S.\ \exists\, b'.$ (*case a of* $(l,\ u) \Rightarrow \lambda(l',\ u').\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle)\ b' \wedge b \preceq b'$
proof −
let *?Z* = *R_of S* **and** *?Z'* = *R_of S'*
obtain *u1* **where** $b = (l',\ u1)$
using *assms*(*1*,*3*) **unfolding** *loc_is_def* **by** (*cases b*) *auto*
then have *u1* ∈ *?Z'*
using *assms*(*1*) **by** *blast*
from *assms*(*4*) **obtain** Z' **where** $A \vdash \langle l,$ *?Z*$\rangle \rightsquigarrow \langle l',\ Z'\rangle$ *?Z'* = $\beta\ l'\ Z'$
by (*erule step_betaE*)
then have $\beta\ l'\ Z' \subseteq \alpha\ l'\ Z'$
using *assms*(*5*,*6*) **by** (*intro* β_α) (*auto dest*: *I_invariant step_z'_V step_z_state_setI2*)
with ‹*u1* ∈ *?Z'*› ‹*?Z'* = _› **obtain** *u1'* **where** *u1'* ∈ *Z'* $(l',\ u1) \preceq (l',$ *u1'*$)$
unfolding *abs_def* **by** *auto*
with ‹$A \vdash \langle l,$ *?Z*$\rangle \rightsquigarrow \langle l',$ *Z'*$\rangle$› **obtain** u **where** $A \vdash' \langle l,\ u\rangle \to \langle l',$ *u1'*$\rangle\ u \in$ *?Z*
by (*meson step_z_sound'*)
with ‹_ ≼ _› **show** *?thesis*
using *assms*(*2*) **unfolding** *R_of_def loc_is_def* ‹*b* = _› **by** *fastforce*
qed

lemma *step'_step_beta*:
assumes
$(l,\ u) \in a'\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$ *loc_is l1 a' R_of a'* $\subseteq V\ I$ (*R_of a'*)
shows
$\exists\, b'.$ ($\exists\, a\ l\ l'.$ *loc_is l a'* $\wedge$ *loc_is l' b'* $\wedge\ a' \neq \{\} \wedge\ b' \neq \{\} \wedge$

```
        A ⊢ ⟨l, R_of a'⟩ ↝_βa ⟨l', R_of b'⟩) ∧ (l', u') ∈ b'
proof −
  let ?Z = R_of a'
  from ‹(l, u) ∈ _› ‹loc_is _ _› have [simp]: l1 = l
    unfolding loc_is_def by auto
  from assms(1) have u ∈ ?Z
    unfolding R_of_def by force
  with assms(2) obtain Z' where step: A ⊢ ⟨l, ?Z⟩ ⇝ ⟨l', Z'⟩ u' ∈ Z'
    by (metis step_z_complete')
  then obtain a where A ⊢ ⟨l, R_of a'⟩ ↝_βa ⟨l', β l' Z'⟩
    by atomize_elim (unfold step_beta_alt_def, fast)
  moreover from β_widens have u' ∈ β l' Z'
    using step ‹_ ⊆ V› ‹I ?Z› by (blast dest: step_z'_V I_invariant step_z_state_setI2)
  ultimately show ?thesis
    using ‹loc_is _ _› ‹(l, u) ∈ _›
    by (inst_existentials from_R l' (β l' Z') a l l')
       (auto simp: from_R_def loc_is_def R_of_def image_def)
qed

definition beta_step where
  beta_step ≡ λs s'. ∃ a l l'. loc_is l s ∧ loc_is l' s' ∧ s ≠ {} ∧ s' ≠ {} ∧
     A ⊢ ⟨l, R_of s⟩ ↝_β(a) ⟨l', R_of s'⟩

lemma beta_step_inv:
  assumes beta_step a b ∃ l∈state_set A. loc_is l a ∧ R_of a ⊆ V ∧ I (R_of a)
  shows ∃ l∈state_set A. loc_is l b ∧ R_of b ⊆ V ∧ I (R_of b)
  using assms unfolding beta_step_def
  using β_V step_z'_V step_z_state_setI2 β_I I_invariant step_betaE by metis

lemma from_R_R_of:
  assumes loc_is l S
  shows from_R l (R_of S) = S
  using assms from_R_R_of unfolding loc_is_def by force

interpretation backward_simulation: Backward_Double_Simulation_Complete where
  E = λ(l, u) (l', u'). A ⊢' ⟨l, u⟩ → ⟨l', u'⟩ and
  G = beta_step and
  sim' = (≡_M) and
  P = valid and
  Q = λs. ∃ l ∈ state_set A. loc_is l s ∧ R_of s ⊆ V ∧ I (R_of s) and
  a_0 = from_R l_0 Z_0
proof (standard, goal_cases)
  case (1 a b a')
  then show ?case
    by (intro self_simulation.A_B_step TrueI)
next
  case (2 b B A)
  then show ?case
    unfolding beta_step_def by clarify (rule backward_simulation)
next
  case (3 a)
  then show ?case
    by (rule refl)
next
```

```
  case 4
  then show ?case
    by (rule self_simulation.trans)
next
  case 5
  then show ?case
    by (rule equiv)
next
  case 6
  then show ?case
    by (rule finite_quotient)
next
  case (7 a b a')
  then show ?case
    unfolding beta_step_def by clarify (rule step'_step_beta)
next
  case (8 a b)
  then show ?case
    by (rule region_self_simulation.PA_invariant.invariant)
next
  case (9 a b)
  then show ?case
    by (rule beta_step_inv[rotated])
next
  case 10
  let ?S = {from_R l (β l Z) | l Z. l ∈ state_set A ∧ Z ⊆ V ∧ I Z}
  have {x. beta_step** (from_R l₀ Z₀) x} ⊆ ?S ∪ {from_R l₀ Z₀}
    apply (rule subsetI)
    apply simp
    subgoal
    proof (induction from_R l₀ Z₀ _ rule: rtranclp.induct)
      case rtrancl_refl
      then show ?case
        by simp
    next
      case (rtrancl_into_rtrancl b c)
      let ?Z = R_of b and ?Z' = R_of c
      from ‹beta_step b c› obtain a l l' where step:
        loc_is l b loc_is l' c b ≠ {} c ≠ {}
        A ⊢ ⟨l, R_of b⟩ ⇝βa ⟨l', R_of c⟩
        unfolding beta_step_def by blast
      with rtrancl_into_rtrancl(2) ‹loc_is l b› have l ∈ state_set A ?Z ⊆ V I ?Z
        apply −
        subgoal
          by (metis step_betaE step_z_state_setI1)
        subgoal
          using Z₀_V β_V by (metis R_of_from_R)
        subgoal
          using Z₀_I β_I by (metis R_of_from_R)
        done
      with step(1,2,5) show ?case
        using from_R_R_of step_z_state_setI2 step_z'_V step_betaE I_invariant by metis
    qed
    done
```

moreover have *finite* (*?S* ∪ {*from_R* l_0 Z_0})
proof −
let *?T* = (λ(*l*, *Z*). *from_R l Z*) ‘ (*state_set A* × {β *l Z* |*l Z*. *I Z* ∧ *Z* ⊆ *V* ∧ *l* ∈ *state_set A*})
have *?S* ⊆ *?T*
by *auto*
also from *finite_state_set finite_abstraction* **have** *finite ?T*
by *auto*
finally show *?thesis*
by *fast*
qed
ultimately show *?case*
by (*rule finite_subset*)
next
case (*11 a*)
then show *?case*
unfolding *loc_is_def* **by** *auto*
next
case *12*
then show *?case*
using l_0*_state_set* Z_0*_V* Z_0*_I* **by** (*auto simp*: *V_def loc_is_def from_R_def R_of_def image_def*)
qed

end

8.8 Invariants on DBM-based Model Checking

context *Regions*
begin

inductive *step_z_dbm′* ::
('*a*, '*c*, '*t*, '*s*) *ta* ⇒ '*s* ⇒ '*t* :: {*linordered_cancel_ab_monoid_add*,*uminus*} *DBM*
⇒ '*a* ⇒ '*s* ⇒ '*t DBM* ⇒ *bool*
(_ ⊢′′ ⟨_, _⟩ ⇝_ ⟨_, _⟩ [*61*,*61*,*61*] *61*) **for** *A l D a l′′ D′′*
where
A ⊢′ ⟨*l*,*D*⟩ ⇝$_a$ ⟨*l′′*,*D′′*⟩ **if** *A* ⊢ ⟨*l*,*D*⟩ ⇝$_{v,n,\tau}$ ⟨*l′*,*D′*⟩ *A* ⊢ ⟨*l′*,*D′*⟩ ⇝$_{v,n,\upharpoonright a}$ ⟨*l′′*,*D′′*⟩

lemmas *step_z_dbm′_def* = *step_z_dbm′.simps*

inductive *step_impl′* ::
('*a*, *nat*, '*t* :: *linordered_ab_group_add*, '*s*) *ta* ⇒ '*s* ⇒ '*t DBM′*
⇒ '*a* ⇒ '*s* ⇒ '*t DBM′* ⇒ *bool*
(_ ⊢$_I$′′ ⟨_, _⟩ ⇝_ ⟨_, _⟩ [*61*,*61*,*61*] *61*) **for** *A l D a l′′ D′′*
where
A ⊢$_I$′ ⟨*l*,*D*⟩ ⇝$_a$ ⟨*l′′*,*D′′*⟩ **if** *A* ⊢$_I$ ⟨*l*, *D*⟩ ⇝$_{n,\tau}$ ⟨*l′*,*D′*⟩ *A* ⊢$_I$ ⟨*l′*,*D′*⟩ ⇝$_{n,\upharpoonright a}$ ⟨*l′′*,*D′′*⟩

lemmas *step_impl′_def* = *step_impl′.simps*

end

definition
dbm_nonneg n M ≡ ∀ *i* ≤ *n*. *i* > *0* ⟶ *M 0 i* ≤ *0*

named_theorems *dbm_nonneg*

lemma *dbm_nonneg_And*[*dbm_nonneg*]:
assumes *dbm_nonneg n M dbm_nonneg n M'*
shows *dbm_nonneg n* (*And M M'*)
using *assms* **by** (*auto simp*: *dbm_nonneg_def min_def*)

lemma *dbm_nonneg_abstra*[*dbm_nonneg*]:
assumes *dbm_nonneg n M*
shows *dbm_nonneg n* (*abstra ac M v*)
using *assms* **by** (*cases ac*) (*auto simp*: *dbm_nonneg_def min_def*)

lemma *dbm_nonneg_abstr*[*dbm_nonneg*]:
assumes *dbm_nonneg n M*
shows *dbm_nonneg n* (*abstr g M v*)
using *assms*(*1*) **unfolding** *abstr.simps*
by (*rule fold_acc_preserv'*[**where** *P* = *dbm_nonneg n*, *rotated*]) (*rule dbm_nonneg_abstra*)

lemma *dbm_nonneg_up*[*dbm_nonneg*]:
dbm_nonneg n (*up M*) **if** *dbm_nonneg n M*
using *that* **unfolding** *dbm_nonneg_def up_def* **by** *auto*

lemma *dbm_nonneg_reset*[*dbm_nonneg*]:
fixes *M* :: *'t* :: *time DBM*
assumes *dbm_nonneg n M x* > *0*
shows *dbm_nonneg n* (*reset M n x 0*)
using *assms* **unfolding** *reset_def dbm_nonneg_def* **by** (*auto simp*: *neutral min_def*)

lemma *dbm_nonneg_reset'*[*dbm_nonneg*]:
fixes *M* :: *'t* :: *time DBM*
assumes *dbm_nonneg n M* ∀ *c* ∈ *set r*. *v c* > *0*
shows *dbm_nonneg n* (*reset' M n r v 0*)
using *assms* **by** (*induction r*) (*auto intro*: *dbm_nonneg_reset*)

lemma *dbm_nonneg_fw_upd*[*dbm_nonneg*]:
dbm_nonneg n (*fw_upd M k' i j*) **if** *dbm_nonneg n M*
using *that* **unfolding** *dbm_nonneg_def fw_upd_def upd_def min_def* **by** *auto*

lemma *dbm_nonneg_fwi*[*dbm_nonneg*]:
dbm_nonneg n (*fwi M n k' i j*) **if** *dbm_nonneg n M*
using *that* **by** (*induction M _ _ _ _ rule*: *fwi.induct*) (*auto intro*!: *dbm_nonneg_fw_upd*)

lemma *dbm_nonneg_fw*[*dbm_nonneg*]:
dbm_nonneg n (*fw M n k*) **if** *dbm_nonneg n M*
using *that* **by** (*induction k*) (*auto intro*!: *dbm_nonneg_fwi*)

lemma *dbm_nonneg_FW*[*dbm_nonneg*]:
dbm_nonneg n (*FW M n*) **if** *dbm_nonneg n M*
using *that* **by** (*rule dbm_nonneg_fw*)

definition
empty_dbm ≡ λ_ _. *Lt 0*

lemma *neg_diag_zero_empty_dbmI*:
 assumes *M 0 0 < 0*
 shows $[M]_{v,n} = \{\}$
 using *assms*
 unfolding *DBM_zone_repr_def DBM_val_bounded_def DBM.neutral DBM.less_eq*[*symmetric*]
by *auto*

lemma *empty_dbm_empty_zone*:
 $[empty\_dbm]_{v,n} = \{\}$
 unfolding *empty_dbm_def* **by** (*rule neg_diag_zero_empty_dbmI*) (*simp add*: *DBM.neutral*)

lemma *canonical_empty_dbm*:
 canonical empty_dbm n
 unfolding *empty_dbm_def* **by** (*auto simp*: *DBM.add*)

lemma *dbm_int_empty_dbm*:
 dbm_int empty_dbm n
 unfolding *empty_dbm_def* **by** *auto*

lemma *dbm_nonneg_empty_dbm*:
 dbm_nonneg n empty_dbm
 unfolding *dbm_nonneg_def empty_dbm_def DBM.neutral* **by** *simp*

lemmas [*simp*] = *any_le_inf*

lemma *DBM_val_boundedD1*:
 assumes $u \vdash_{v,n} M\ v\ c \leq n$
 shows *Le* $(-\ u\ c) \leq M\ 0\ (v\ c)$
 using *assms* **unfolding** *dbm_entry_val.simps DBM_val_bounded_def* **by** *auto*

lemma *DBM_val_boundedD2*:
 assumes $u \vdash_{v,n} M\ v\ c \leq n$
 shows *Le* $(u\ c) \leq M\ (v\ c)\ 0$
 using *assms* **unfolding** *dbm_entry_val.simps DBM_val_bounded_def* **by** *auto*

lemma *DBM_val_boundedD3*:
 assumes $u \vdash_{v,n} M\ v\ c1 \leq n\ v\ c2 \leq n$
 shows *Le* $(u\ c1 - u\ c2) \leq M\ (v\ c1)\ (v\ c2)$
 using *assms* **unfolding** *dbm_entry_val.simps DBM_val_bounded_def* **by** *force*

lemma *dbm_default_And*:
 assumes *dbm_default M n dbm_default M' n*
 shows *dbm_default* (*And M M'*) *n*
 using *assms* **by** *auto*

lemma *dbm_default_abstra*:
 assumes *dbm_default M n constraint_pair ac* = (x, m) $v\ x \leq n$
 shows *dbm_default* (*abstra ac M v*) *n*
 using *assms* **by** (*cases ac*) *auto*

lemma *dbm_default_abstr*:
 assumes *dbm_default M n* $\forall (x, m) \in$ *collect_clock_pairs g*. $v\ x \leq n$
 shows *dbm_default* (*abstr g M v*) *n*
 using *assms*(*1*) **unfolding** *abstr.simps*

```
proof (rule fold_acc_preserv'[where P = λM. dbm_default M n, rotated], goal_cases)
  case (1 ac acc)
  then obtain x m where constraint_pair ac = (x, m)
    by force
  with assms(2) 1 show ?case
    by (intro dbm_default_abstra) (auto simp: collect_clock_pairs_def)
qed

lemma dbm_entry_dense:
  fixes a b :: 't :: time DBMEntry
  assumes a + b ≥ 0 b > 0
  obtains x where x > 0 b ≥ Le x Le (−x) ≤ a
proof −
  consider a = ∞ | a ≠ ∞ a + b = 0 | a ≠ ∞ a + b > 0
    using assms(1) by force
  then show ?thesis
  proof cases
    case 1
    with assms show ?thesis
    proof (cases b)
      case (Le x)
      with assms(2) 1 show ?thesis
        by (intro that[of x]) (auto simp: DBM.neutral DBM.add)
    next
      case (Lt x2)
      with assms(2) 1 show ?thesis
        using that time_class.dense by (auto simp: DBM.neutral DBM.add)
    next
      case INF
      with 1 assms(2) that show ?thesis
      by (metis add_inf(2) any_le_inf dbm_less_eq_simps(2) neg_less_iff_less sum_gt_neutral_dest)
    qed
  next
    case 2
    then obtain x where a = Le (−x) b = Le x
      unfolding neutral by (cases a; cases b; simp add: DBM.add eq_neg_iff_add_eq_0)
    with ‹b > 0› show ?thesis
      by (intro that[of x]; simp add: neutral)
  next
    case 3
    note intro = that
    have 1: 0 < x + y ⟹ − y ≤ x for x y :: 't
       by (metis ab_semigroup_add_class.add.commute add.right_inverse add_le_cancel_left
less_imp_le)
    have 2: thesis if 0 < x + y 0 < y a = Le x b = Lt y for x y :: 't
    proof (cases x > 0)
      case True
      then have − x ≤ 0
        by auto
      from ‹y > 0› dense obtain z where 0 < z z < y
        by auto
      with that ‹− x ≤ 0› show ?thesis
        using dual_order.trans by (intro intro[of z]; fastforce)
    next
```

```
    case False
    from that have −x < y
      using 1 minus_less_iff by fastforce
    with dense obtain z where − x < z z < y
      by auto
    with False that show ?thesis
      by (intro intro[of z]; simp add: less_imp_le minus_less_iff)
        (meson leI less_le_trans neg_less_0_iff_less)
  qed
  include no_library_syntax
  have 3: thesis if a = Le x b = ∞ for x :: 't
  by (smt 3(2) Le_le_LtI Lt_le_LeI add.inverse_inverse any_le_inf intro neg_0_less_iff_less
      non_trivial_neg not_less order_trans sum_gt_neutral_dest that(2))
  have 4: thesis if a = Lt x b = ∞ for x :: 't
   by (metis that ‹0 < a + b› add.inverse_inverse dbm_less_eq_simps(2) dbm_less_simps(2)
intro leI
      less_imp_le less_le_trans neg_0_less_iff_less sum_gt_neutral_dest)
  have 5: thesis if 0 < x + y 0 < y a = Lt x b = Le y for x y
    by (metis that Le_le_LtI antisym_conv1 diff_0_right diff_less_eq intro less_irrefl mi-
nus_diff_eq)
  have 6: thesis if 0 < y a = Lt x b = Lt y for x y
    using that ‹a + b > 0›
    apply −
    apply (drule sum_gt_neutral_dest)
    apply safe
    subgoal for d
      by (cases d ≥ 0, cases d = 0)
        (smt intro Le_le_LtI Lt_le_LeI
          add.inverse_inverse not_less neg_less_0_iff_less dense order_trans)+
    done
  have 7: thesis if 0 < x + y 0 < y a = Le x b = Le y for x y
    using that by (intro intro[of y]) (auto simp: DBM.add intro: 1)
  from ‹a + b > 0› ‹b > 0› ‹a ≠ ∞› show thesis
    by (cases a; cases b) (auto simp: DBM.add neutral intro: 2 3 4 5 6 7)
 qed
qed

lemma canonical_diag:
 fixes M :: 't :: time DBM
 assumes canonical M n i ≤ n
 shows M i i ≥ Lt 0
proof (rule ccontr)
 assume ¬ M i i ≥ Lt 0
 then have M i i < Lt 0
   by auto
 then have M i i + M i i < M i i
   by (cases M i i) (auto simp: DBM.add)
 with assms show False
   by force
qed

lemma canonical_diag_nonnegI:
 fixes M :: 't :: time DBM
 assumes canonical M n ∀ i ≤ n. M i i ≠ Lt 0
```

shows $\forall\, i \leq n.\ M\ i\ i \geq 0$
proof *clarify*
fix i **assume** $i \leq n$
then show $M\ i\ i \geq 0$
using *canonical_diag*[*OF assms*(*1*) ‹$i \leq n$›] *assms*(*2*) **by** (*cases M i i*; *auto simp*: *DBM.neutral*)
qed

context *Regions_common*
begin

lemma *canonical_non_emptyI*:
assumes $[M]_{v,n} \neq \{\}$
shows *canonical* (*FW M n*) *n*
by (*simp add*: *assms fw_shortest non_empty_cyc_free*)

definition
canonical_dbm M ≡ *canonical M n* ∧ *dbm_nonneg n M* ∧ *dbm_int M n*

abbreviation
vabstr' (*Z* :: ('*c*, *t*) *zone*) *M* ≡ *Z* = $[M]_{v,n}$ ∧ *canonical_dbm M*

lemma *V_structuralI*:
assumes *dbm_nonneg n M*
shows $[M]_{v,n} \subseteq V$
using *clock_numbering*(*3*) *assms* **unfolding** *V_def DBM_zone_repr_def*
by (*clarsimp simp*: *neutral*) (*meson assms clock_numbering*(*1*) *dbm_positive dbm_nonneg_def*)

lemma *canonical_dbm_valid*:
valid_dbm M **if** *canonical_dbm M*
using *that* **unfolding** *canonical_dbm_def* **by** (*auto dest*: *V_structuralI*)

lemma *dbm_nonnegI*:
assumes *canonical M n* $[M]_{v,n} \subseteq V$ $\forall\, i \leq n.\ M\ i\ i \neq Lt\ 0$
shows *dbm_nonneg n M*
proof (*rule ccontr*)
assume *A*: ¬ *dbm_nonneg n M*
from *assms*(*1*,*3*) **have** *diag_nonneg*: $\forall\, i \leq n.\ M\ i\ i \geq 0$
by (*rule canonical_diag_nonnegI*)
from *A* **obtain** *i* **where** $i > 0$ $i \leq n$ $M\ 0\ i > 0$
unfolding *dbm_nonneg_def* **by** *auto*
moreover have $M\ i\ 0 + M\ 0\ i \geq 0$
proof −
from *assms*(*1*) ‹$i \leq n$› **have** $M\ i\ i \geq Lt\ 0$
by (*rule canonical_diag*)
with ‹$i \leq n$› *assms*(*3*) **have** $M\ i\ i \geq 0$
by (*cases M i i*; *auto simp*: *neutral*)
also from ‹*canonical M n*› ‹$i \leq n$› **have** $M\ i\ 0 + M\ 0\ i \geq M\ i\ i$
by *auto*
finally show *?thesis* **.**
qed
ultimately obtain *x* **where** $x > 0$ $M\ 0\ i \geq Le\ x$ $Le\ (-x) \leq M\ i\ 0$
by − (*rule dbm_entry_dense*)
moreover from ‹$i > 0$› ‹$i \leq n$› **obtain** *c* **where** $c \in X\ v\ c \leq n\ i = v\ c$
using *clock_numbering* **by** *auto*

moreover from *clock_numbering(1) cn_weak assms(1)* **have** *cycle_free M n*
by (*intro non_empty_cycle_free canonical_nonneg_diag_non_empty diag_nonneg*) *auto*
ultimately obtain *u* **where** $u \in [M]_{v,n}$ $u\ c < 0$
using *assms(1)*
by (*auto simp*: *clock_numbering(1) intro*: *canonical_saturated_1*[**where** $M = M$ **and** $v = v$])
with *assms(2)* ‹$c \in X$› **show** *False*
unfolding *V_def DBM_zone_repr_def* **by** *force*
qed

lemma *vabstr'_V*:
obtains *M* **where** *vabstr' V M*
proof −
interpret *beta_regions*: *Beta_Regions'*
where $k = k\ l$
apply −
apply *unfold_locales*
apply (*rule finite*)
apply (*simp add*: *non_empty*)
apply (*rule clock_numbering cn_weak not_in_X*)+
done
have *V_eq*: *beta_regions.V* = *V*
unfolding *beta_regions.V_def V_def* **..**
let *?M* = *beta_regions.V_dbm*
from *beta_regions.V_dbm_eq_V beta_regions.V_dbm_int beta_regions.normalized_V_dbm*
have ∗:
$[?M]_{v,n} = V$ *dbm_int ?M n beta_regions.normalized ?M*
unfolding *V_eq* **by** *auto*
moreover have *dbm_nonneg n ?M*
unfolding *beta_regions.V_dbm_def dbm_nonneg_def DBM.neutral* **by** *simp*
ultimately show *?thesis*
apply −
apply (*rule that*[*of FW ?M n*])
apply (*rule conjI*)
subgoal
using *FW_zone_equiv_spec* **by** *blast*
unfolding *canonical_dbm_def*
apply (*intro conjI dbm_nonneg_FW FW_int_preservation canonical_non_emptyI*)
apply (*auto simp*: *V_def*)
done
qed

lemma *canonical_dbm_empty_dbm*:
canonical_dbm empty_dbm
unfolding *canonical_dbm_def*
by (*intro conjI canonical_empty_dbm dbm_int_empty_dbm dbm_nonneg_empty_dbm*)

lemma *vabstr'_empty_dbm*:
vabstr' {} *empty_dbm*
by (*intro conjI empty_dbm_empty_zone*[*symmetric*] *canonical_dbm_empty_dbm*)

lemma *vabstr'I*:
assumes *dbm_int M' n Z'* ⊆ *V Z'* = $[M']_{v,n}$
obtains *M'* **where** *vabstr' Z' M'*

proof (*cases* $Z' = \{\}$)
case *True*
then show *?thesis*
by (*intro that*[*of empty_dbm*], *simp only*: *vabstr'_empty_dbm*)
next
case *False*
with *assms* **obtain** M' **where** $*$: $Z' = [M']_{v,n}$ *dbm_int* M' *n canonical* M' *n*
by (*metis FW_canonical' FW_valid_preservation FW_zone_equiv_spec*
dbm_non_empty_diag valid_dbm.simps)
with *assms*(*2*) **have** *dbm_nonneg n* M'
by − (*rule dbm_nonnegI*; *smt* ‹$Z' \neq \{\}$› *dbm_less_eq_simps*(*2*) *dbm_non_empty_diag neutral*)
let *?M* = *FW* M' *n*
from $*$ ‹*dbm_nonneg n* M'› **show** *?thesis*
apply (*intro that*[*of ?M*] *conjI*)
apply (*simp add*: *FW_zone_equiv_spec*[*symmetric*])
unfolding *canonical_dbm_def*
unfolding *dbm_nonneg_def*[*symmetric*]
apply (*intro conjI FW_canonical' dbm_nonneg_FW FW_int_preservation*; *assumption?*)
subgoal *diag_nonneg*
using *FW_zone_equiv_spec False dbm_non_empty_diag* **by** *blast*
done
qed

end

context *Regions_TA*
begin

The following does not hold:

lemma
fixes M :: *real DBM*
assumes *canonical* M *n*
and $\forall\, i \leq n.\ M\ 0\ i \leq 0$
shows $\forall\, i \leq n.\ 0 \leq M\ i\ i$
oops

lemma *global_clock_numbering*:
global_clock_numbering A v n
using *clock_numbering*(*1*) *clock_numbering_le cn_weak valid_abstraction* **by** *blast*

sublocale *step_z'_bisim_step_z_dbm'*: *Bisimulation*
$\lambda(l,\ Z)\ (l',\ Z').\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle$
$\lambda(l,\ M)\ (l',\ M').\ \exists\, a.\ A \vdash' \langle l,M\rangle \rightsquigarrow_a \langle l',M'\rangle$
$\lambda(l,\ Z)\ (l',\ M).\ l' = l \wedge Z = [M]_{v,n}$
apply (*standard*; *clarsimp simp*: *step_z_dbm'_def step_z'_def*)
subgoal
using *global_clock_numbering* **by** (*auto elim*!: *step_z_dbm_DBM*)
subgoal
by (*blast dest*: *step_z_dbm_sound*[*OF _ global_clock_numbering*])
done

lemma *step_z_dbm_preserves_int*:
dbm_int M' *n* **if** $A \vdash \langle l,M\rangle \rightsquigarrow_{v,n,a} \langle l',M'\rangle$ *dbm_int M n*

apply (*rule step_z_dbm_preserves_int*; (*rule that global_clock_numbering*)*?*)
using *valid_abstraction valid_abstraction_pairsD* **by** *fastforce*

lemma *step_z_dbm'_preserves_int*:
dbm_int M' n **if** $A \vdash' \langle l,M\rangle \rightsquigarrow_a \langle l',M'\rangle$ *dbm_int M n*
using *that* **by** *cases* (*erule step_z_dbm_preserves_int*)+

lemma *step_z'_vabstr'*:
$\exists M.$ *vabstr' Z' M* **if** $\exists M.$ *vabstr' Z M A* $\vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle$
proof −
from *that* **obtain** *M* **where** *vabstr' Z M*
by *auto*
with *step_z'_bisim_step_z_dbm'.A_B_step*[*of* (*l*, *Z*) _ (*l*, *M*)] *that*(*2*) **obtain** *a M'* **where** ∗:
$A \vdash' \langle l, M\rangle \rightsquigarrow_a \langle l', M'\rangle$ *Z'* = $[M']_{v,n}$
by *force*
with ‹*vabstr' Z M*› **have** *dbm_int M' n*
by − (*rule step_z_dbm'_preserves_int*, *auto simp*: *canonical_dbm_def*)
moreover from ∗(*1*) ‹*vabstr' Z M*› **have** $Z' \subseteq V$
by (*metis canonical_dbm_valid step_z'_def step_z_V that*(*2*) *valid_dbm.simps*)
ultimately obtain *M'* **where** *vabstr' Z' M'*
using ‹*Z'* = _› **by** − (*rule vabstr'I*)
then show *?thesis*
by (*intro exI*)
qed

end

8.9 Instantiating the "Sandwich" for Extrapolations on DBMs

locale *TA_Extrapolation* =
Regions_TA **where** *A* = *A* +
Time_Abstract_Simulation **where** *A* = *A* **for** *A* :: ('*a*, '*c*, *real*, '*l*) *ta* +
assumes *simulation_nonneg*: *u'* ∈ *V* ⟹ (*l*, *u*) ≼ (*l'*, *u'*) ⟹ *u* ∈ *V*
fixes *extra* :: '*l* ⇒ *real DBM* ⇒ *real DBM* **and** l_0 **and** M_0
assumes *extra_widens*: *vabstr' Z M* ⟹ *Z* ⊆ $[$*extra l M*$]_{v,n}$
and *extra_α*: *vabstr' Z M* ⟹ $[$*extra l M*$]_{v,n} \subseteq \alpha\ l\ Z$
and *extra_finite*: *finite* {*extra l M* | *M*. *canonical_dbm M*}
and *extra_int*: *dbm_int M n* ⟹ *dbm_int* (*extra l M*) *n*
assumes l_0*_state_set*: $l_0 \in$ *state_set A* **and** M_0*_V*: $[M_0]_{v,n} \subseteq V$ **and** M_0*_int*: *dbm_int* M_0 *n*
begin

definition *apx* **where**
apx l Z ≡ *let M* = (*SOME M*. *canonical_dbm M* ∧ *Z* = $[M]_{v,n}$) *in* $[$*extra l M*$]_{v,n}$

lemma *apx_widens*:
$[M]_{v,n} \subseteq$ *apx l* ($[M]_{v,n}$) **if** *canonical_dbm M*
by (*smt apx_def extra_widens someI_ex that*)

lemma *apx_abs*:
apx l ($[M]_{v,n}$) ⊆ α *l* ($[M]_{v,n}$) **if** *canonical_dbm M*
by (*smt apx_def extra_α someI_ex that*)

```
lemma α_V:
  α l Z ⊆ V if Z ⊆ V
  using that simulation_nonneg unfolding V_def abs_def by auto

lemma apx_V:
  apx l ([M]_{v,n}) ⊆ V if canonical_dbm M
proof −
  from that have apx l ([M]_{v,n}) ⊆ α l ([M]_{v,n})
    by (rule apx_abs)
  moreover from that have ... ⊆ V
    unfolding canonical_dbm_def by (intro α_V V_structuralI, elim conjE)
  finally show ?thesis .
qed

lemma apx_empty:
  apx l {} = {}
proof −
  have vabstr' {} empty_dbm
    by (rule vabstr'_empty_dbm)
  from canonical_dbm_empty_dbm have apx l {} ⊆ α l {}
    by (subst empty_dbm_empty_zone[symmetric])+ (rule apx_abs)
  also have ... = {}
    unfolding abs_def by auto
  finally show ?thesis
    by simp
qed

lemma apx_ex:
  assumes canonical_dbm M
  shows ∃ M'. apx l ([M]_{v,n}) = [extra l M']_{v,n} ∧ canonical_dbm M'
  using assms unfolding apx_def by (smt someI_ex)

lemma vabstr'_apx:
  assumes vabstr' Z M Z ⊆ V
  obtains M where vabstr' (apx l Z) M
proof (cases Z = {})
  case False
  from apx_ex assms obtain M' where ∗:
    apx l Z = [extra l M']_{v,n} canonical_dbm M'
    using apx_ex by blast
  with FW_zone_equiv_spec have apx l Z = [FW (extra l M') n]_{v,n}
    by auto
  moreover from ‹canonical_dbm M'› have canonical_dbm (FW (extra l M') n)
  proof −
    from assms have Z ⊆ apx l Z
      by (auto intro!: apx_widens)
    with ‹Z ≠ {}› have apx l Z ≠ {}
      by auto
    then have canonical (FW (extra l M') n) n
      unfolding ∗ by (rule canonical_non_emptyI)
    moreover have dbm_nonneg n (FW (extra l M') n)
      using ‹apx l Z = [FW (extra l M') n]_{v,n}› ‹apx l Z ≠ {}› ‹canonical (FW _ _) _›
      apply (intro dbm_nonnegI)
      apply assumption
```

```
    subgoal
      using apx_V assms(1) by blast
    subgoal
      by (metis DBMEntry.distinct(1) Le_less_Lt  antisym_conv dbm_non_empty_diag leI
neutral)
    done
  moreover have dbm_int (FW (extra l M') n) n
    using ∗(2) unfolding canonical_dbm_def by (intro FW_int_preservation extra_int, elim
conjE)
  ultimately show ?thesis
    unfolding canonical_dbm_def by (intro conjI)
 qed
 ultimately show ?thesis
  by (auto intro: that)
next
 case True
 then show ?thesis
  by (intro that[of empty_dbm])(auto simp: empty_dbm_empty_zone apx_empty canonical_dbm_empty_dbm)
qed

lemma apx_finite:
 finite {apx l Z |l Z. ∃ M. vabstr' Z M ∧ l ∈ state_set A} (is finite ?S)
proof −
 { fix l assume l ∈ state_set A
  from extra_finite have finite {[extra l M]v,n | M. canonical_dbm M}
  proof −
   have {[extra l M]v,n | M. canonical_dbm M}
     = (λM. [M]v,n) ‘ {extra l M | M. canonical_dbm M}
    by auto
   with extra_finite show ?thesis
    by simp
  qed
  also from apx_ex have {apx l Z |Z. ∃ M. vabstr' Z M} ⊆ ...
   by auto
  finally (finite_subset[rotated]) have finite {apx l Z |Z. ∃ M. vabstr' Z M} .
 } note ∗ = this
 have ?S = (⋃ l ∈ state_set A. {apx l Z |Z. ∃ M. vabstr' Z M})
  by auto
 also have finite ...
  using ∗ finite_state_set by auto
 finally show ?thesis .
qed

sublocale Time_Abstract_Simulation_Sandwich
 where β = apx and I = λZ. ∃ M. vabstr' Z M and Z0 = [M0]v,n
proof standard
 show u ∈ V if (l, u) ≼ (l', u') u' ∈ V for l l' :: 'l and u u' :: 'c ⇒ real
  using that by (rule simulation_nonneg[rotated])
 show ∃ M. vabstr' Z' M
  if ∃ M. vabstr' Z M and A ⊢ ⟨l, Z⟩ ⇝ ⟨l', Z'⟩ for Z Z' :: ('c ⇒ real) set and l l' :: 'l
  using that by (rule step_z'_vabstr')
 show apx l Z ⊆ α l Z
  if ∃ M. vabstr' Z M Z ⊆ V for Z :: ('c ⇒ real) set and l :: 'l
  using that apx_abs by auto
```

show $Z \subseteq apx\ l\ Z$ **if** $\exists M.\ vabstr'\ Z\ M\ Z \subseteq V$ **for** $Z :: ('c \Rightarrow real)\ set$ **and** $l :: 'l$
using *that apx_widens* **by** *auto*
show $\exists M.\ vabstr'\ (apx\ l\ Z)\ M$
if $\exists M.\ vabstr'\ Z\ M\ Z \subseteq V$ **for** $Z :: ('c \Rightarrow real)\ set$ **and** $l :: 'l$
using *that* **by** (*elim exE*) (*rule vabstr'_apx*, *auto*)
show *finite* $\{apx\ l\ Z\ |l\ Z.\ (\exists M.\ vabstr'\ Z\ M) \wedge Z \subseteq V \wedge l \in state\_set\ A\}$
using *apx_finite* **by** (*rule finite_subset*[*rotated*]) *auto*
show $l_0 \in state\_set\ A$
by (*rule* l_0*_state_set*)
show $[M_0]_{v,n} \subseteq V$
by (*rule* M_0*_V*)
show $\exists M.\ vabstr'\ ([M_0]_{v,n})\ M$
by (*rule vabstr'I*; (*rule* M_0*_int* M_0*_V*)*?*) *auto*
qed

end

end
theory *Normalized_Zone_Semantics_Certification*
imports *Munta_Base.Normalized_Zone_Semantics_Impl_Semantic_Refinement*
begin

no_notation *TA_Start_Defs.step_impl'* ($\langle \_, \_ \rangle \rightsquigarrow \_\ \langle \_, \_ \rangle$ [*61*,*61*,*61*] *61*)

context *TA_Start_Defs*
begin

definition
E_precise_op l r g l' M $\equiv$
let
$M' = FW'\ (abstr\_upd\ (inv\_of\ A\ l)\ (up\_canonical\_upd\ M\ n))\ n$;
$M'' = FW'\ (abstr\_upd\ (inv\_of\ A\ l')\ (reset'\_upd\ (FW'\ (abstr\_upd\ g\ M')\ n)\ n\ r\ 0))\ n$
in M''

definition
E_precise_op' l r g l' M $\equiv$
let
M1 $= abstr\_repair\ (inv\_of\ A\ l)\ (up\_canonical\_upd\ M\ n)$;
M2 $= filter\_diag\ (\lambda\ M.\ abstr\_repair\ g\ M)\ M1$;
M3 $= filter\_diag\ (\lambda\ M.\ abstr\_repair\ (inv\_of\ A\ l')\ (reset'\_upd\ M\ n\ r\ 0))\ M2$
in M3

lemma *E_precise_op'_alt_def*:
E_precise_op' l r g l' M $\equiv$
let
$M' = abstr\_repair\ (inv\_of\ A\ l)\ (up\_canonical\_upd\ M\ n)$;
f1 $= \lambda\ M.\ abstr\_repair\ g\ M$;
f2 $= \lambda\ M.\ abstr\_repair\ (inv\_of\ A\ l')\ (reset'\_upd\ M\ n\ r\ 0)$
in filter_diag (*filter_diag f2 o f1*) M'
unfolding *E_precise_op'_def filter_diag_def*
by (*rule HOL.eq_reflection*) (*auto simp*: *Let_def check_diag_marker*)

no_notation *step_impl'* ($\langle \_, \_ \rangle \rightsquigarrow \_\ \langle \_, \_ \rangle$ [*61*,*61*,*61*] *61*)

abbreviation *step_impl_precise* ($\langle \_, \_\rangle \rightsquigarrow_{\_} \langle \_, \_\rangle$ [61,61,61] 61)
where
$\langle l, Z\rangle \rightsquigarrow_a \langle l'', Z''\rangle \equiv \exists\ l'\ Z'.$
$A \vdash_I \langle l, Z\rangle \rightsquigarrow_{n,\tau} \langle l', Z'\rangle \wedge A \vdash_I \langle l', Z'\rangle \rightsquigarrow_{n,\downarrow a} \langle l'', Z''\rangle$

definition *E_precise* $\equiv (\lambda(l, Z)\ (l'', Z'').\ \exists\, a.\ \langle l, Z\rangle \rightsquigarrow_a \langle l'', Z''\rangle)$

end

locale *E_Precise_Bisim* = *E_From_Op* +
assumes *op_bisim*:
$A \vdash l \longrightarrow^{g,a,r} l' \Longrightarrow$ *wf_dbm* $M \Longrightarrow f\ l\ r\ g\ l'\ M \simeq$ *E_precise_op* $l\ r\ g\ l'\ M$
and *op_wf*:
$A \vdash l \longrightarrow^{g,a,r} l' \Longrightarrow$ *wf_dbm* $M \Longrightarrow$ *wf_dbm* $(f\ l\ r\ g\ l'\ M)$
begin

lemma *E_precise_E_op*:
E_precise $= (\lambda(l, M)\ (l', M'').\ \exists\, g\ a\ r.\ A \vdash l \longrightarrow^{g,a,r} l' \wedge M'' = E\_precise\_op\ l\ r\ g\ l'\ M)$
unfolding *E_precise_op_def E_precise_def* **by** (*intro ext*) (*auto elim*!: *step_impl.cases*)

lemma *E_E_from_op_step*:
$\exists\, c.$ *E_from_op* $a\ c \wedge b \sim c$ **if** *E_precise* $a\ b$ *wf_state* a
using *that* **unfolding** *E_precise_E_op E_from_op_def wf_state_def state_equiv_def*
by (*blast intro*: *op_bisim dbm_equiv_sym*)

lemma *E_from_op_E_step*:
$\exists\, c.$ *E_precise* $a\ c \wedge b \sim c$ **if** *E_from_op* $a\ b$ *wf_state* a
using *that* **unfolding** *E_precise_E_op E_from_op_def wf_state_def state_equiv_def*
by (*blast intro*: *op_bisim*)

lemma *E_from_op_wf_state*:
wf_state b **if** *wf_state* a *E_from_op* $a\ b$
using *that* **unfolding** *E_E_op E_from_op_def wf_state_def state_equiv_def* **by** (*blast intro*: *op_wf*)

lemma *E_precise_wf_dbm*[*intro*]:
wf_dbm D' **if** *E_precise* $(l, D)\ (l', D')$ *wf_dbm* D
using *that* **unfolding** *wf_state_def E_def E_precise_def* **by** (*auto intro*: *step_impl_wf_dbm*)

lemma *E_precise_wf_state*:
wf_state $a \Longrightarrow$ *E_precise* $a\ b \Longrightarrow$ *wf_state* b
unfolding *wf_state_def* **by** *auto*

theorem *E_from_op_reachability_check*:
$(\exists\ l'\ D'.$ *E_precise*** $a_0\ (l', D') \wedge$ *F_rel* $(l', D'))$
$\longleftrightarrow (\exists\ l'\ D'.$ *E_from_op*** $a_0\ (l', D') \wedge$ *F_rel* $(l', D'))$
oops

lemma *E_from_op_mono*:
assumes *E_from_op* (*l*,*D*) (*l'*,*D'*)
and *wf_dbm D wf_dbm M*
and $[curry\ (conv\_M\ D)]_{v,n} \subseteq [curry\ (conv\_M\ M)]_{v,n}$
shows $\exists\ M'.$ *E_from_op* (*l*,*M*) (*l'*,*M'*) $\wedge$ $[curry\ (conv\_M\ D')]_{v,n} \subseteq [curry\ (conv\_M\ M')]_{v,n}$

oops

lemma *E_from_op_mono'*:
assumes *E_from_op* (*l*,*D*) (*l'*,*D'*)
and *wf_dbm D wf_dbm M*
and *dbm_subset n D M*
shows $\exists\ M'.$ *E_from_op* (*l*,*M*) (*l'*,*M'*) $\wedge$ *dbm_subset n D' M'*

oops

lemma *E_equiv*:
$\exists\ b'.$ *E_precise b b'* $\wedge$ $a' \sim b'$ **if** *E_precise a a' a* $\sim$ *b wf_state a wf_state b*
using *that*
unfolding *wf_state_def E_precise_def*
apply *safe*
apply (*frule* (*2*) *step_impl_equiv*, *erule state_equiv_D*)
by (*safe*, *drule step_impl_equiv*, *auto intro*: *step_impl_wf_dbm simp*: *state_equiv_def*)

lemma *E_from_op_bisim*:
Bisimulation_Invariant E_precise E_from_op ($\sim$) *wf_state wf_state*
apply *standard*
subgoal
by (*drule E_equiv*, *assumption*+) (*auto dest*!: *E_E_from_op_step*)
subgoal
by (*drule* (*1*) *E_from_op_E_step*, *safe*, *drule E_equiv*) (*auto 4 4 intro*: *state_equiv_sym*)
apply (*rule E_precise_wf_state*; *assumption*)
apply (*rule E_from_op_wf_state*; *assumption*)
done

lemma *E_from_op_bisim_empty*:
Bisimulation_Invariant
($\lambda(l,\ M)\ (l',\ M').$ *E_precise* (*l*, *M*) (*l'*, *M'*) $\wedge$ $\neg$ *check_diag n M'*)
($\lambda(l,\ M)\ (l',\ M').$ *E_from_op* (*l*, *M*) (*l'*, *M'*) $\wedge$ $\neg$ *check_diag n M'*)
($\sim$) *wf_state wf_state*
using *E_from_op_bisim*
apply (*rule Bisimulation_Invariant_filter*[
where *FA* = $\lambda(l,\ M).$ $\neg$ *check_diag n M* **and** *FB* = $\lambda(l,\ M).$ $\neg$ *check_diag n M*
])
subgoal
unfolding *wf_state_def state_equiv_def*
apply *clarsimp*
apply (*subst canonical_check_diag_empty_iff*[*symmetric*], *erule wf_dbm_altD*(*1*))
apply (*subst canonical_check_diag_empty_iff*[*symmetric*], *erule wf_dbm_altD*(*1*))
apply (*simp add*: *dbm_equiv_def*)
done
apply (*solves auto*)+
done

end

definition *step_z_dbm'* ::
(*'a*, *'c*, *'t*, *'s*) *ta* ⇒ *'s* ⇒ *'t* :: {*linordered_cancel_ab_monoid_add*,*uminus*} *DBM*
⇒ (*'c* ⇒ *nat*) ⇒ *nat* ⇒ *'a* ⇒ *'s* ⇒ *'t DBM* ⇒ *bool*
(_ ⊢'' ⟨_, _⟩ ⇝_,_,_ ⟨_, _⟩ [*61*,*61*,*61*,*61*] *61*) **where**
A ⊢' ⟨*l*,*D*⟩ ⇝*v*,*n*,*a* ⟨*l''*,*D''*⟩ ≡
∃ *l' D'*. *A* ⊢ ⟨*l*, *D*⟩ ⇝*v*,*n*,*τ* ⟨*l'*, *D'*⟩ ∧ *A* ⊢ ⟨*l'*, *D'*⟩ ⇝*v*,*n*,↑*a* ⟨*l''*, *D''*⟩

context *TA_Start*
begin

lemma *E_precise_op'_bisim*:
E_precise_op' l r g l' M ≃ *E_precise_op l r g l' M* **if** *A* ⊢ *l* ⟶^*g*,*a*,*r* *l' wf_dbm M*
proof −
note *intros* =
dbm_equiv_refl dbm_equiv_trans[*OF filter_diag_equiv*, *rotated*]
wf_dbm_abstr_repair_equiv_FW[*rotated*] *reset'_upd_equiv*
have ∀ *c*∈*constraint_clk* ‘ *set* (*inv_of A l*). *0* < *c* ∧ *c* ≤ *n*
using *clock_range collect_clks_inv_clk_set*[*of A l*] **unfolding** *collect_clks_def* **by** *blast*
moreover have
∀ *c*∈*constraint_clk* ‘ *set* (*inv_of A l'*). *0* < *c* ∧ *c* ≤ *n*
∀ *c*∈*constraint_clk* ‘ *set* (*inv_of A l'*). *0* < *c*
using *clock_range collect_clks_inv_clk_set*[*of A l'*] **unfolding** *collect_clks_def* **by** *blast+*
moreover have ∀ *c*∈*constraint_clk* ‘ *set g*. *0* < *c* ∧ *c* ≤ *n* ∀ *c*∈*constraint_clk* ‘ *set g*. *0* < *c*
using *clock_range collect_clocks_clk_set*[*OF that(1)*] **unfolding** *collect_clks_def* **by** *blast+*
moreover have ∀ *i*∈*set r*. *0* < *i* ∧ *i* ≤ *n* ∀ *i*∈*set r*. *0* < *i*
using *clock_range reset_clk_set*[*OF that(1)*] **unfolding** *collect_clks_def* **by** *blast+*
moreover note *side_conds* = *calculation that(2)*
note *wf_intros* =
wf_dbm_abstr_repair wf_dbm_reset'_upd wf_dbm_up_canonical_upd filter_diag_wf_dbm
wf_dbm_FW'_abstr_upd
note *check_diag_intros* =
reset'_upd_check_diag_preservation abstr_repair_check_diag_preservation
show *?thesis* **unfolding** *E_precise_op'_def E_precise_op_def*
by *simp* (*intro intros check_diag_intros side_conds wf_intros order.refl*)
qed

lemma *step_z_'_step_z_dbm'_equiv*:
Bisimulation_Invariant
(λ (*l*, *D*) (*l'*, *D'*). ∃ *a*. *step_z'* (*conv_A A*) *l D l' D'*)
(λ (*l*, *D*) (*l'*, *D'*). ∃ *a*. *step_z_dbm'* (*conv_A A*) *l D v n a l' D'*)
(λ(*l*, *Z*) (*l'*, *M*). *l* = *l'* ∧ [*M*]*v*,*n* = *Z*)
(λ(*l*, *Z*). *True*)
(λ(*l*, *y*). *True*)
proof (*standard*, *goal_cases*)
case *prems*: (*1 a b a'*)
obtain *l Z l' Z' l1 M* **where** *unfolds*[*simp*]: *a* = (*l*, *Z*) *b* = (*l'*, *Z'*) *a'* = (*l1*, *M*)
by *force+*
from *prems* **have** [*simp*]: *l1* = *l*
by *auto*
from *prems* **have** *conv_A A* ⊢ ⟨*l*, *Z*⟩ ⇝ ⟨*l'*, *Z'*⟩ [*M*]*v*,*n* = *Z*
by *auto*

```
  from this(1) obtain Z1 a where
    conv_A A ⊢ ⟨l, Z⟩ ↝_τ ⟨l, Z1⟩
    conv_A A ⊢ ⟨l, Z1⟩ ↝_↑a ⟨l', Z'⟩
    unfolding step_z'_def by safe
  note Z1 = this
  from step_z_dbm_DBM[OF Z1(1)[folded ‹_ = Z›]] obtain M1 where M1:
    conv_A A ⊢ ⟨l, M⟩ ↝_v,n,τ ⟨l, M1⟩ Z1 = [M1]_v,n .
  from step_z_dbm_DBM[OF Z1(2)[unfolded ‹Z1 = _›]] obtain Z2 where Z2:
    conv_A A ⊢ ⟨l, M1⟩ ↝_v,n,↑a ⟨l', Z2⟩ Z' = [Z2]_v,n .
  with M1 Z1 show ?case
    unfolding step_z_dbm'_def by auto
next
  case prems: (2 a a' b')
  obtain l M l' M' l1 Z where unfolds[simp]: a' = (l, M) b' = (l', M') a = (l1, Z)
    by force+
  from prems have [simp]: l1 = l
    by auto
  from prems obtain a1 where conv_A A ⊢' ⟨l, M⟩ ↝_v,n,a1 ⟨l', M'⟩ [M]_v,n = Z
    by auto
  from this(1) obtain l1' Z1 where Z1:
    conv_A A ⊢ ⟨l, M⟩ ↝_v,n,τ ⟨l1', Z1⟩
    conv_A A ⊢ ⟨l1', Z1⟩ ↝_v,n,↑a1 ⟨l', M'⟩
    unfolding step_z_dbm'_def by atomize_elim
  then have [simp]: l1' = l
    by (intro step_z_dbm_delay_loc)
  from Z1 ‹_ = Z› show ?case
    unfolding step_z'_def by (auto dest: step_z_dbm_sound)
next
  case (3 a b)
  then show ?case
    by auto
next
  case (4 a b)
  then show ?case
    by auto
qed

lemma step_z_dbm'_step_impl_precise_equiv:
  Bisimulation_Invariant
    (λ (l, D) (l', D'). ∃ a. step_z_dbm' (conv_A A) l D v n a l' D')
    (λ(l, D) (l', D'). ∃ a. ⟨l, D⟩ ↝_a ⟨l', D'⟩)
    (λ(l, M) (l', D). l = l' ∧ [curry (conv_M D)]_v,n = [M]_v,n)
    (λ(l, y). valid_dbm y)
    wf_state
proof (standard, goal_cases)
  case prems: (1 a b a')
  obtain l M l' M' l1 D where unfolds[simp]: a = (l, M) b = (l', M') a' = (l1, D)
    by force+
  from prems have [simp]: l1 = l
    by auto
  from prems obtain a1 where
    step_z_dbm' (conv_A A) l M v n a1 l' M'
    by auto
  then obtain l2 M1 where steps:
```

conv_A A ⊢ ⟨*l*, *M*⟩ ⇝$_{v,n,\tau}$ ⟨*l2*, *M1*⟩
conv_A A ⊢ ⟨*l2*, *M1*⟩ ⇝$_{v,n,\uparrow a1}$ ⟨*l'*, *M'*⟩
unfolding *step_z_dbm'_def* **by** *auto*
from *step_z_dbm_equiv'*[*OF steps*(*1*), *of curry* (*conv_M D*)] *prems*(*2*−) **obtain** *M2* **where**
conv_A A ⊢ ⟨*l*, *curry* (*conv_M D*)⟩ ⇝$_{v,n,\tau}$ ⟨*l2*, *M2*⟩ *wf_dbm D* [*M1*]$_{v,n}$ = [*M2*]$_{v,n}$
by (*auto simp*: *wf_state_def*)
with *step_impl_complete''_improved*[*OF this*(*1*)] **obtain** *D2* **where** *D2*:
A ⊢$_I$ ⟨*l*, *D*⟩ ⇝$_{n,\tau}$ ⟨*l2*, *D2*⟩ [*curry* (*conv_M D2*)]$_{v,n}$ = [*M1*]$_{v,n}$
by *auto*
from *step_impl_wf_dbm*[*OF D2*(*1*) ‹*wf_dbm D*›] **have** *wf_dbm D2* **.**
from *step_z_dbm_equiv'*[*OF steps*(*2*) *sym*[*OF D2*(*2*)]] **obtain** *D3* **where** *D3*:
conv_A A ⊢ ⟨*l2*, *curry* (*conv_M D2*)⟩ ⇝$_{v,n,\uparrow a1}$ ⟨*l'*, *D3*⟩ [*M'*]$_{v,n}$ = [*D3*]$_{v,n}$
by (*elim conjE exE*)
from *step_impl_complete''_improved*[*OF D3*(*1*) ‹*wf_dbm D2*›] **obtain** *D4* **where** *D4*:
A ⊢$_I$ ⟨*l2*, *D2*⟩ ⇝$_{n,\uparrow a1}$ ⟨*l'*, *D4*⟩ [*curry* (*conv_M D4*)]$_{v,n}$ = [*D3*]$_{v,n}$
by *auto*
with *D2*(*1*) **have** ⟨*l*, *D*⟩ ⇝$_{a1}$ ⟨*l'*, *D4*⟩
by *auto*
with *D4*(*2*) *D3* **show** *?case*
by *force*
next
case *prems*: (*2 a a' b'*)
obtain *l M l' D' l1 D* **where** *unfolds*[*simp*]: *a* = (*l*, *M*) *b'* = (*l'*, *D'*) *a'* = (*l1*, *D*)
by *force*+
from *prems* **have** [*simp*]: *l1* = *l*
by *auto*
from *prems* **obtain** *a1* **where** ⟨*l*, *D*⟩ ⇝$_{a1}$ ⟨*l'*, *D'*⟩
by *auto*
then obtain *D1* **where** *steps*:
A ⊢$_I$ ⟨*l*, *D*⟩ ⇝$_{n,\tau}$ ⟨*l*, *D1*⟩ *A* ⊢$_I$ ⟨*l*, *D1*⟩ ⇝$_{n,\uparrow a1}$ ⟨*l'*, *D'*⟩
by (*auto dest*: *step_impl_delay_loc_eq*)
from *prems* **have** *wf_dbm D*
by (*auto simp*: *wf_state_def*)
with *steps* **have** *wf_dbm D1*
by (*blast intro*: *step_impl_wf_dbm*)
from *step_impl_sound'*[*OF steps*(*1*)] ‹*wf_dbm D*› **obtain** *M2* **where** *M2*:
conv_A A ⊢ ⟨*l*, *curry* (*conv_M D*)⟩ ⇝$_{v,n,\tau}$ ⟨*l*, *M2*⟩
[*curry* (*conv_M D1*)]$_{v,n}$ = [*M2*]$_{v,n}$
using *wf_dbm_D* **by** *auto*
from *step_impl_sound'*[*OF steps*(*2*)] ‹*wf_dbm D1*› **obtain** *M3* **where** *M3*:
step_z_dbm (*conv_A A*) *l* (*curry* (*conv_M D1*)) *v n* (↑*a1*) *l' M3*
[*curry* (*conv_M D'*)]$_{v,n}$ = [*M3*]$_{v,n}$
using *wf_dbm_D* **by** *auto*
from *step_z_dbm_equiv'*[*OF M2*(*1*), *of M*] *prems*(*2*) **obtain** *M2'* **where** *M2'*:
conv_A A ⊢ ⟨*l*, *M*⟩ ⇝$_{v,n,\tau}$ ⟨*l*, *M2'*⟩ [*M2*]$_{v,n}$ = [*M2'*]$_{v,n}$
by *auto*
from *step_z_dbm_equiv'*[*OF M3*(*1*), *of M2'*] *M2*(*2*) *M2'*(*2*) **obtain** *M3'* **where**
conv_A A ⊢ ⟨*l*, *M2'*⟩ ⇝$_{v,n,\uparrow a1}$ ⟨*l'*, *M3'*⟩ [*M3*]$_{v,n}$ = [*M3'*]$_{v,n}$
by *auto*
with *M2'*(*1*) *M3*(*2*) **show** *?case*
unfolding *step_z_dbm'_def* **by** *auto*
next
case (*3 a b*)
then show *?case*

unfolding *step_z_dbm'_def* **using** *step_z_norm_valid_dbm'_spec step_z_valid_dbm'* **by** *auto*
next
case (*4 a b*)
then show *?case*
by (*clarsimp simp*: *norm_step_wf_dbm step_impl_wf_dbm wf_state_def*)
qed

lemma *E_precise_op'_wf*:
assumes $A \vdash l \longrightarrow^{g,a,r} l'$ *wf_dbm M*
shows *wf_dbm* (*E_precise_op' l r g l' M*)
proof −
have $\forall\, c{\in}$*constraint_clk ' set* (*inv_of A l*). $0 < c \wedge c \leq n$
using *clock_range collect_clks_inv_clk_set*[*of A l*] **unfolding** *collect_clks_def* **by** *blast*
moreover have $\forall\, c{\in}$*constraint_clk ' set* (*inv_of A l'*). $0 < c \wedge c \leq n$
using *clock_range collect_clks_inv_clk_set*[*of A l'*] **unfolding** *collect_clks_def* **by** *blast*
moreover have $\forall\, c{\in}$*constraint_clk ' set g*. $0 < c \wedge c \leq n$
using *clock_range collect_clocks_clk_set*[*OF assms*(*1*)] **unfolding** *collect_clks_def* **by** *blast*
moreover have $\forall\, i{\in}$*set r*. $0 < i \wedge i \leq n$
using *clock_range reset_clk_set*[*OF assms*(*1*)] **unfolding** *collect_clks_def* **by** *blast*
moreover note *side_conds* = *calculation assms*(*2*)
note *wf_intros* =
wf_dbm_abstr_repair wf_dbm_reset'_upd wf_dbm_up_canonical_upd filter_diag_wf_dbm
show *?thesis*
unfolding *E_precise_op'_def* **by** *simp* (*intro wf_intros side_conds order.refl*)
qed

sublocale *E_precise_op'*: *E_Precise_Bisim* _ _ _ _ *E_precise_op'*
by *standard* (*rule E_precise_op'_bisim E_precise_op'_wf*; *assumption*)+

lemma *step_z_dbm'_final_bisim*:
Bisimulation_Invariant
(λ (*l*, *D*) (*l'*, *D'*). ∃ *a*. *step_z_dbm'* (*conv_A A*) *l D v n a l' D'*)
E_precise_op'.E_from_op
(λ (*l*, *M*) (*l'*, *D'*). $l' = l \wedge [\mathit{curry}\ (\mathit{conv\_M}\ D')]_{v,n} = [M]_{v,n}$)
(λ(*l*, *y*). *valid_dbm y*) *wf_state*
by (*rule Bisimulation_Invariant_sim_replace*,
rule Bisimulation_Invariant_composition[
OF step_z_dbm'_step_impl_precise_equiv[*folded E_precise_def*] *E_precise_op'.E_from_op_bisim*
]) (*auto simp add*: *state_equiv_def dbm_equiv_def*)

end

context *E_Precise_Bisim*
begin

theorem *step_z_dbm'_mono*:
assumes *conv_A A* $\vdash' \langle l, M\rangle \rightsquigarrow_{v,n,a} \langle l', M'\rangle$ **and** $[M]_{v,n} \subseteq [D]_{v,n}$
shows $\exists\, D'.$ *conv_A A* $\vdash' \langle l, D\rangle \rightsquigarrow_{v,n,a} \langle l', D'\rangle \wedge [M']_{v,n} \subseteq [D']_{v,n}$
using *assms* **unfolding** *step_z_dbm'_def*
apply *clarsimp*
apply (*drule step_z_dbm_mono*, *assumption*)
apply *safe*
apply (*drule step_z_dbm_mono*, *assumption*)

apply *auto*
done

interpretation
Bisimulation_Invariant
$(\lambda\ (l,\ D)\ (l',\ D').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ D\ v\ n\ a\ l'\ D')$
E_from_op
$\lambda\ (l,\ M)\ (l',\ D').\ l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = [M]_{v,n}$
$\lambda(l,\ y).\ valid\_dbm\ y$
wf_state
by (*rule Bisimulation_Invariant_sim_replace*,
rule Bisimulation_Invariant_composition[
OF step_z_dbm'_step_impl_precise_equiv[*folded E_precise_def*] *E_from_op_bisim*
]) (*auto simp add*: *state_equiv_def dbm_equiv_def*)

lemmas *step_z_dbm'_E_from_op_bisim* = *Bisimulation_Invariant_axioms*

definition
$E\_from\_op\_empty \equiv \lambda(l,\ D)\ (l',\ D').\ E\_from\_op\ (l,\ D)\ (l',\ D') \wedge \neg\ check\_diag\ n\ D'$

interpretation *bisim_empty*:
Bisimulation_Invariant
$\lambda(l,\ M)\ (l',\ M').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ M\ v\ n\ a\ l'\ M' \wedge [M']_{v,n} \neq \{\}$
$\lambda(l,\ D)\ (l',\ D').\ E\_from\_op\ (l,\ D)\ (l',\ D') \wedge \neg\ check\_diag\ n\ D'$
$\lambda(l,\ M)\ (l',\ D').\ l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = [M]_{v,n}$
$\lambda(l,\ y).\ valid\_dbm\ y$
wf_state
using *step_z_dbm'_E_from_op_bisim* **apply** (*rule Bisimulation_Invariant_filter*[
where $FA = \lambda(l',\ M').\ [M']_{v,n} \neq \{\}$ **and** $FB = \lambda(l',\ D').\ \neg\ check\_diag\ n\ D'$
])
using *canonical_check_diag_empty_iff canonical_empty_check_diag'* **by** (*auto simp*: *wf_state_def*)

lemmas *step_z_dbm'_E_from_op_bisim_empty* =
bisim_empty.Bisimulation_Invariant_axioms[*folded E_from_op_empty_def*]

lemma *E_from_op_mono*:
assumes *E_from_op* (*l*,*D*) (*l'*,*D'*)
and *wf_dbm D wf_dbm M*
and $[curry\ (conv\_M\ D)]_{v,n} \subseteq [curry\ (conv\_M\ M)]_{v,n}$
shows $\exists\ M'.\ E\_from\_op\ (l,M)\ (l',M') \wedge [curry\ (conv\_M\ D')]_{v,n} \subseteq [curry\ (conv\_M\ M')]_{v,n}$
proof −
from *B_A_step*[*OF assms*(*1*), *of* (*l*, *curry* (*conv_M D*))] *assms*(*2*) **obtain** *a D1* **where** *D1*:
$conv\_A\ A \vdash' \langle l,\ curry\ (conv\_M\ D)\rangle \rightsquigarrow_{v,n,a} \langle l',\ D1\rangle\ [D1]_{v,n} = [curry\ (conv\_M\ D')]_{v,n}$
unfolding *wf_state_def* **by** (*force dest*: *wf_dbm_D*)
from *step_z_dbm'_mono*[*OF this*(*1*) *assms*(*4*)] **obtain** *M1* **where** *M1*:
$conv\_A\ A \vdash' \langle l,\ curry\ (conv\_M\ M)\rangle \rightsquigarrow_{v,n,a} \langle l',\ M1\rangle$
$[D1]_{v,n} \subseteq [M1]_{v,n}$
by *atomize_elim*
with *A_B_step*[*of* (*l*, *curry* (*conv_M M*)) (*l'*, *M1*) (*l*, *M*)] *assms*(*3*) **obtain** *M2* **where**
$E\_from\_op\ (l,\ M)\ (l',\ M2)\ [curry\ (conv\_M\ M2)]_{v,n} = [M1]_{v,n}$
unfolding *wf_state_def* **by** (*force dest*: *wf_dbm_D*)
with *assms*(*3*) *M1*(*2*) *D1*(*2*) **show** *?thesis*
by *auto*
qed

lemma *E_from_op_mono′*:
assumes *E_from_op* (*l*,*D*) (*l′*,*D′*)
and *wf_dbm D wf_dbm M*
and *dbm_subset n D M*
shows ∃ *M′*. *E_from_op* (*l*,*M*) (*l′*,*M′*) ∧ *dbm_subset n D′ M′*
using *assms*
apply −
apply (*frule E_from_op_mono*[**where** *M* = *M*], *assumption*+)
apply (*subst dbm_subset_correct′′*, *assumption*+)
apply (*rule dbm_subset_conv*, *assumption*)
apply *safe*
apply (*subst* (*asm*) *dbm_subset_correct′′*)
subgoal
using *B_invariant*[*unfolded wf_state_def*] **by** *auto*
subgoal
using *B_invariant*[*unfolded wf_state_def*] **by** *auto*
apply (*blast intro*: *dbm_subset_conv_rev*)
done

lemma *E_from_op_empty_mono′*:
assumes *E_from_op_empty* (*l*,*D*) (*l′*,*D′*)
and *wf_dbm D wf_dbm M*
and *dbm_subset n D M*
shows ∃ *M′*. *E_from_op_empty* (*l*,*M*) (*l′*,*M′*) ∧ *dbm_subset n D′ M′*
using *assms* **unfolding** *E_from_op_empty_def* **using** *check_diag_subset E_from_op_mono′*
by *blast*

end

end

9 Checker Implementation for Single Automata

theory *Normalized_Zone_Semantics_Certification_Impl*
imports
Munta_Base.Normalized_Zone_Semantics_Impl_Refine
Normalized_Zone_Semantics_Certification
Collections.Refine_Dflt_ICF
Unreachability_Certification2
Unreachability_Certification
HOL−Library.IArray
Munta_Model_Checker.Deadlock_Impl
Munta_Base.More_Methods
HOL−Library.Rewrite
begin

Misc **lemma** (**in** *Graph_Defs*) *run_first_reaches*:
pred_stream (*reaches x*) *xs* **if** *run* (*x* ## *xs*)
proof −
from *that* **obtain** *a* **where** *run* (*a* ## *xs*) *reaches x a*
by *auto*

then show *?thesis*
by (*coinduction arbitrary*: *a xs rule*: *stream_pred_coinduct*) (*auto 4 3 elim*: *run.cases*)
qed

lemma (**in** *Graph_Start_Defs*) *run_reachable*:
pred_stream reachable xs **if** *run* (s_0 ## *xs*)
using *run_first_reaches*[*OF that*] **unfolding** *reachable_def* **.**

lemma *pred_stream_stream_all2_combine*:
assumes *pred_stream P xs stream_all2 Q xs ys* $\bigwedge x\ y.\ P\ x \Longrightarrow Q\ x\ y \Longrightarrow R\ x\ y$
shows *stream_all2 R xs ys*
using *assms* **by** (*auto intro*: *stream_all2_combine simp*: *stream.pred_rel eq_onp_def*)

lemma *stream_all2_pred_stream_combine*:
assumes *stream_all2 Q xs ys pred_stream P ys* $\bigwedge x\ y.\ Q\ x\ y \Longrightarrow P\ y \Longrightarrow R\ x\ y$
shows *stream_all2 R xs ys*
using *assms* **by** (*auto intro*: *stream_all2_combine simp*: *stream.pred_rel eq_onp_def*)

lemma *map_set_rel*:
assumes *list_all P xs* (*f*, *g*) ∈ {(*xs*, *ys*). *xs* = *ys* ∧ *P xs*} → *B*
shows (*map f xs*, *g* ‘ *set xs*) ∈ ⟨*B*⟩*list_set_rel*
unfolding *list_set_rel_def*
apply (*rule relcompI*[**where** *b* = *map g xs*])
apply *parametricity*
using *assms* **unfolding** *list_rel_def list_all_iff* **by** (*auto intro*: *list.rel_refl_strong*)

context *TA_Start*
begin

lemma *V_I*:
assumes ∀ *i* ∈ {*1*..<*Suc n*}. *M 0 i* ≤ *0*
shows $[M]_{v,n} \subseteq V$
unfolding *V_def DBM_zone_repr_def*
proof (*safe*, *goal_cases*)
case *prems*: (*1 u i*)
then have *v i* = *i*
using *X_alt_def X_def triv_numbering* **by** *blast*
with *prems* **have** *v i* > *0 v i* ≤ *n* **by** *auto*
with *prems* **have** *dbm_entry_val u None* (*Some i*) (*M 0* (*v i*))
unfolding *DBM_val_bounded_def* **by** *auto*
moreover from *assms* ‹*v i* > *0*› ‹*v i* ≤ *n*› **have** *M 0* (*v i*) ≤ *0* **by** *auto*
ultimately
show *?case*
apply (*cases M 0* (*v i*))
unfolding *neutral less_eq dbm_le_def*
by (*auto elim*!: *dbm_lt.cases simp*: ‹*v i* = *i*›)
qed

end

lemma *Simulation_Composition*:
fixes *A B C*
assumes

Simulation A B sim1 Simulation B C sim2 $\bigwedge a\ c.\ (\exists\ b.\ sim1\ a\ b \wedge sim2\ b\ c) \longleftrightarrow sim\ a\ c$
shows *Simulation A C sim*
proof −
interpret *A*: *Simulation A B sim1*
by (*rule assms*)
interpret *B*: *Simulation B C sim2*
by (*rule assms*)
show *?thesis*
by *standard* (*auto dest*!: *B.A_B_step A.A_B_step simp*: *assms*(*3*)[*symmetric*])
qed

lemma *fold_generalize_start*:
assumes $\bigwedge a.\ P\ a \Longrightarrow Q\ (fold\ g\ xs\ a)\ P\ a$
shows *Q* (*fold g xs a*)
using *assms* **by** *auto*

definition
set_of_list xs = *SPEC* (λ*S*. *set xs* = *S*)

lemma *set_of_list_hnr*:
(*return o id*, *set_of_list*) ∈ (*list_assn A*)^d →$_a$ *lso_assn A*
unfolding *set_of_list_def lso_assn_def hr_comp_def br_def* **by** *sepref_to_hoare sep_auto*

lemma *set_of_list_alt_def*:
set_of_list = *RETURN o set*
unfolding *set_of_list_def* **by** *auto*

lemmas *set_of_list_hnr′* = *set_of_list_hnr*[*unfolded set_of_list_alt_def*]

lemmas *amtx_copy_hnr* = *amtx_copy_hnr*[*unfolded op_mtx_copy_def*, *folded COPY_def*[*abs_def*]]

lemma *lso_op_set_is_empty_hnr*[*sepref_fr_rules*]:
(*return o* (λ*xs*. *xs* = []), *RETURN o op_set_is_empty*) ∈ (*lso_assn AA*)^k →$_a$ *bool_assn*
unfolding *lso_assn_def hr_comp_def br_def* **by** *sepref_to_hoare sep_auto*

context
fixes *n* :: *nat*
begin

qualified definition
dbm_tab M ≡ λ (*i*, *j*). *if i* ≤ *n* ∧ *j* ≤ *n then M* ! ((*n* + *1*) ∗ *i* + *j*) *else 0*

private lemma
shows *mtx_nonzero_dbm_tab_1*: (*a*, *b*) ∈ *mtx_nonzero* (*dbm_tab M*) ⟹ *a* < *Suc n*
and *mtx_nonzero_dbm_tab_2*: (*a*, *b*) ∈ *mtx_nonzero* (*dbm_tab M*) ⟹ *b* < *Suc n*
unfolding *mtx_nonzero_def dbm_tab_def* **by** (*auto split*: *if_split_asm*)

definition
list_to_dbm M = *op_amtx_new* (*Suc n*) (*Suc n*) (*dbm_tab M*)

lemma [*sepref_fr_rules*]:
(*return o dbm_tab*, *RETURN o PR_CONST dbm_tab*) ∈ *id_assn*^{k} →$_a$ *pure* (*nat_rel* ×$_r$ *nat_rel* → *Id*)

by *sepref_to_hoare sep_auto*

lemmas [*sepref_opt_simps*] = *dbm_tab_def*

sepref_register *dbm_tab*

sepref_definition *list_to_dbm_impl*
is *RETURN o PR_CONST list_to_dbm* :: *id_assn*^{k} →$_a$ *mtx_assn n*
supply *mtx_nonzero_dbm_tab_1*[*simp*] *mtx_nonzero_dbm_tab_2*[*simp*]
unfolding *PR_CONST_def list_to_dbm_def* **by** *sepref*

lemma *the_pure_Id*:
the_pure (λ*a c*. ↑ (*c* = *a*)) = *Id*
by (*subst is_pure_the_pure_id_eq*) (*auto simp*: *pure_def intro*: *is_pureI*)

lemma *of_list_list_to_dbm*:
(*Array.of_list*, (*RETURN* ∘∘ *PR_CONST*) *list_to_dbm*)
∈ [λ*a*. *length a* = *Suc n* ∗ *Suc n*]$_a$ *id_assn*^{k} → *mtx_assn n*
apply *sepref_to_hoare*
apply *sep_auto*
unfolding *amtx_assn_def hr_comp_def IICF_Array_Matrix.is_amtx_def*
apply (*sep_auto eintros del*: *exI*)
subgoal for *xs p*
apply (*rule exI*[**where** *x* = *list_to_dbm xs*])
apply (*rule exI*[**where** *x* = *xs*])
apply (*sep_auto simp*: *list_to_dbm_def dbm_tab_def the_pure_Id*)
apply (*simp add*: *algebra_simps*; *fail*)
apply *sep_auto*
done
done

end

definition
array_freeze a = *do* {*xs* ← *Array.freeze a*; *Heap_Monad.return* (*IArray xs*)}

definition
array_unfreeze a = *Array.of_list* (*IArray.list_of a*)

definition
iarray_mtx_rel n m c a ≡
IArray.length a = *n* ∗ *m*
∧ (∀ *i*<*n*.∀ *j*<*m*. *c* (*i*, *j*) = *IArray.sub a* (*i* ∗ *m* + *j*))
∧ (∀ *i j*. *i* ≥ *n* ∨ *j* ≥ *m* ⟶ *c* (*i*, *j*) = *0*)

lemma *iarray_mtx_rel_is_amtx*:
x ↦$_a$ *IArray.list_of a* ⟹$_A$ *IICF_Array_Matrix.is_amtx n m c x* **if** *iarray_mtx_rel n m c a*
using *that* **unfolding** *is_amtx_def iarray_mtx_rel_def* **by** *simp solve_entails*

lemma *array_unfreeze_ht*:
<*emp*> *array_unfreeze a* <*amtx_assn n m id_assn c*> **if** *iarray_mtx_rel n m c a*
using *that* **unfolding** *array_unfreeze_def amtx_assn_def* **by** (*sep_auto intro*: *iarray_mtx_rel_is_amtx*)

lemma *array_freeze_ht*:

```
  <amtx_assn n m id_assn c a> array_freeze a <λa. ↑(iarray_mtx_rel n m c a)>t
  unfolding array_freeze_def amtx_assn_def iarray_mtx_rel_def is_amtx_def
  by (sep_auto intro: iarray_mtx_rel_is_amtx)


datatype ('a, 'b) frozen_hm =
  Frozen_Hash_Map (array_of_hm: ('a, 'b) list_map iarray) (size_of_hm: nat)

definition diff_array_freeze :: 'a array ⇒ 'a iarray where
  diff_array_freeze a = IArray (list_of_array a)

definition hm_freeze where
  hm_freeze ≡ λhm. case hm of Impl_Array_Hash_Map.HashMap a _
    ⇒ Frozen_Hash_Map (diff_array_freeze a) (array_length a)

definition frozen_hm_lookup where
  frozen_hm_lookup ≡ λkey hm.
    case hm of Frozen_Hash_Map a n ⇒
      let
        code = bounded_hashcode_nat n key;
        bucket = IArray.sub a code
      in
        list_map_lookup (=) key bucket


lemma list_of_array_nth:
  list_of_array a ! n = array_get a n
  by (cases a) simp

lemma frozen_hm_lookup_hm_freeze:
 frozen_hm_lookup k (hm_freeze m) = Impl_Array_Hash_Map.ahm_lookup (=) bounded_hashcode_nat
k m
proof −
  obtain a n where m = Impl_Array_Hash_Map.HashMap a n
    by (cases m) auto
  have IArray.sub (diff_array_freeze a) i = array_get a i for i
    unfolding diff_array_freeze_def by (simp add: list_of_array_nth)
  then show ?thesis
    unfolding frozen_hm_lookup_def ‹m = _› hm_freeze_def by simp
qed

context
  fixes M :: ('a :: hashable ∗ 'b) list
begin

definition map_of_list = fold (λ(k, v) a. a(k ↦ v)) M Map.empty

lemma M_id_refine[autoref_rules]:
  (M, M) ∈ ⟨Id ×r Id⟩list_rel
  by simp

schematic_goal M_impl:
  (?M::?'c, map_of_list:::r⟨Id,Id⟩dflt_ahm_rel) ∈ ?R
  unfolding map_of_list_def
```

apply (*autoref* (*trace*))
done

concrete_definition *hashmap_of_list* **uses** *M_impl*

definition
frozen_hm_of_list ≡ *hm_freeze hashmap_of_list*

theorem *hashmap_of_list_lookup*:
(λ*k*. *Impl_Array_Hash_Map.ahm_lookup* (=) *bounded_hashcode_nat k hashmap_of_list*, *map_of_list*)
∈ *Id* → ⟨*Id*⟩*option_rel* (**is** (λ*k*. *?f k hashmap_of_list*, _) ∈ *?R*)
proof −
have ∗: (*?f*, *Intf_Map.op_map_lookup*) ∈ *Id* → *ahm_map_rel′ bounded_hashcode_nat* → *Id*
using *Impl_Array_Hash_Map.ahm_lookup_impl*[*OF hashable_bhc_is_bhc*] **by** *simp*
{ fix *k* :: *′a*
have *abstract_bounded_hashcode Id bounded_hashcode_nat* = *bounded_hashcode_nat*
unfolding *abstract_bounded_hashcode_def* **by** (*intro ext*) *auto*
from *hashmap_of_list.refine* **obtain** *hm* **where**
(*hashmap_of_list*, *hm*) ∈ ⟨*Id*, *Id*⟩*ahm_map_rel*
(*hm*, *map_of_list*) ∈ *ahm_map_rel′ bounded_hashcode_nat*
unfolding *ahm_rel_def* **by** (*clarsimp simp*: ‹_ = *bounded_hashcode_nat*›)
with ∗ **have** *?f k hm* = *Intf_Map.op_map_lookup k map_of_list*
by (*auto dest*!: *fun_relD*)
with ‹(*hashmap_of_list*, *hm*) ∈ _› **have** *?f k hashmap_of_list* = *map_of_list k*
unfolding *ahm_map_rel_def array_rel_def* **by** *clarsimp*
}
then show *?thesis*
by *simp*
qed

theorem *frozen_hm_of_list_lookup*:
(λ*k*. *frozen_hm_lookup k frozen_hm_of_list*, *map_of_list*) ∈ *Id* → ⟨*Id*⟩*option_rel*
using *hashmap_of_list_lookup* **unfolding** *frozen_hm_of_list_def*
by (*simp add*: *frozen_hm_lookup_hm_freeze*)

end

definition
array_all2 n P as bs ≡ ∀ *i* < *n*. *P* (*IArray.sub as i*) (*IArray.sub bs i*)

lemma *iarray_mtx_relD*:
assumes *iarray_mtx_rel n m M a i* < *n j* < *m*
shows *M* (*i*, *j*) = *IArray.sub a* (*i* ∗ *m* + *j*)
using *assms* **unfolding** *iarray_mtx_rel_def* **by** *auto*

lemma *array_all2_iff_pointwise_cmp*:
assumes *iarray_mtx_rel* (*Suc n*) (*Suc n*) *M a iarray_mtx_rel* (*Suc n*) (*Suc n*) *M′ b*
shows *array_all2* (*Suc n* ∗ *Suc n*) *P a b* ⟷ *pointwise_cmp P n* (*curry M*) (*curry M′*)
proof −
have ∗: ‹*i* + *i* ∗ *n* + *j* < *Suc* (*n* + (*n* + *n* ∗ *n*))› **if** ‹*i* ≤ *n*› **and** ‹*j* ≤ *n*› **for** *i j* :: ‹*nat*›
using *that* **by** (*simp add*: *algebra_simps*) (*intro le_imp_less_Suc add_mono*; *simp*)
have ∗∗: ∃ *i* ≤ *n*. ∃ *j* ≤ *n*. *k* = *i* + *i* ∗ *n* + *j* **if** *k* < *Suc n* ∗ *Suc n* **for** *k*
apply (*inst_existentials k div Suc n k mod Suc n*)

```
    subgoal
      by (meson less_Suc_eq_le less_mult_imp_div_less that)
    subgoal
      by simp
    subgoal
      by (metis div_mod_decomp mult_Suc_right)
    done
  from assms show ?thesis
    unfolding pointwise_cmp_def array_all2_def
    by (auto dest: *[simplified] **[simplified] simp: iarray_mtx_relD)
qed


context TA_Impl
begin

interpretation DBM_Impl n .

sepref_definition E_precise_op'_impl is
  uncurry4 (λ l r. RETURN ooo E_precise_op' l r) :: op_impl_assn
  unfolding
   E_precise_op'_def FW''_def[symmetric] reset'_upd_def inv_of_A_def[symmetric] PR_CONST_def
    filter_diag_def
  by sepref

end



locale TA_Impl_Precise =
  TA_Impl _ _ _ l₀ _ _ _ _ _ l₀i
  + op_precise: E_Precise_Bisim _ l₀ for l₀ :: 's and l₀i :: 'si:: {hashable,heap} +
  fixes op_impl and states_mem_impl
  assumes op_impl: (uncurry4 op_impl, uncurry4 (λ l r. RETURN ooo PR_CONST f l r)) ∈
op_impl_assn
      and states_mem_impl: (states_mem_impl, (λl. l ∈ states')) ∈ loc_rel → bool_rel

locale Reachability_Problem_Impl_Precise =
  TA_Impl_Precise _ show_state A
  for show_state :: 'si:: {hashable,heap} ⇒ string and A :: ('a, nat, int, 's) ta+
  fixes F :: 's × (nat × nat ⇒ int DBMEntry) ⇒ bool and F' and F1 and F_impl
  assumes F_mono: ⋀ a b.
    (λ(l, M). l ∈ states' ∧ wf_dbm M) a ⟹ F a ⟹
    (λ(l, s) (l', s'). l' = l ∧ dbm_subset n s s') a b ⟹ (λ(l, M). l ∈ states' ∧ wf_dbm M) b
    ⟹ F b
      and F_F1: ⋀l D Z. op_precise.E_from_op_empty** (l₀, init_dbm) (l, D)
          ⟹ dbm.zone_of (curry (conv_M D)) = Z ⟹ F (l, D) = F1 (l, Z)
      and F'_F1: ⋀l u Z. u ∈ Z ⟹ F' (l, u) ⟹ F1 (l, Z)
      and F_impl: (F_impl, RETURN o PR_CONST F) ∈ state_assn'^d →ₐ bool_assn

context TA_Impl_Precise
begin

lemma E_precise_E_op:
```

E_precise = (λ(*l*, *M*) (*l'*, *M'''*). ∃ *g a r*. *A* ⊢ *l* ⟶^{g,a,r} *l'* ∧ *M'''* = *E_precise_op l r g l' M*)
unfolding *E_precise_op_def E_precise_def* **by** (*intro ext*) (*auto elim*!: *step_impl.cases*)

definition *succs_precise* **where**
succs_precise ≡ λ*l S*.
if S = {} *then* []
else rev [
(*l'*, {*D'* | *D' D*. *D* ∈ *S* ∧ *D'* = *f l r g l' D* ∧ ¬ *check_diag n D'*}). (*g*,*a*,*r*,*l'*) ← *trans_fun l*]

definition *succs_precise_inner* **where**
succs_precise_inner l r g l' S ≡ *do* {
xs ← *SPEC* (λ*xs*. *set xs* = *S*);
p ← *nfoldli xs* (λ_. *True*) (λ*D xs*.
do {*let D'* = *PR_CONST f l r g l' D*; *if check_diag n D' then RETURN xs else RETURN* (*D'* # *xs*)}
) [];
S' ← *SPEC* (λ*S*. *set p* = *S*);
RETURN S'
}

definition *succs_precise'* **where**
succs_precise' ≡ λ*l S*. *if S* = {} *then RETURN* [] *else do* {
nfoldli (*trans_fun l*) (λ _. *True*) (λ (*g*,*a*,*r*,*l'*) *xxs*.
do {
S' ← *PR_CONST succs_precise_inner l r g l'* (*COPY S*);
RETURN ((*l'*, *S'*) # *xxs*)
}
) []
}

lemma *succs_precise_inner_rule*:
succs_precise_inner l r g l' S
≤ *RETURN* {*D'* | *D' D*. *D* ∈ *S* ∧ *D'* = *f l r g l' D* ∧ ¬ *check_diag n D'*}
unfolding *succs_precise_inner_def*
by (*refine_vcg nfoldli_rule*[**where**
I = λ*l1 l2* σ. σ = *rev* (*filter* (λ*D'*. ¬ *check_diag n D'*) (*map* (*f l r g l'*) *l1*))
]) *auto*

lemma *succs_precise'_refine*:
succs_precise' l S ≤ *RETURN* (*succs_precise l S*)
unfolding *succs_precise_def succs_precise'_def*
unfolding *rev_map_fold fold_eq_nfoldli PR_CONST_def*
apply (*cases S* = {})
apply (*simp*; *fail*)
apply (*simp only*: *if_False fold_eq_nfoldli*)
apply (*refine_vcg nfoldli_mono*)
apply (*rule order.trans*)
apply (*rule succs_precise_inner_rule*)
apply *auto*
done

lemma *succs_precise'_correct*:
(*uncurry succs_precise'*, *uncurry* (*RETURN oo PR_CONST succs_precise*)) ∈ *Id* ×$_r$ *Id* → ⟨*Id*⟩*nres_rel*

using *succs_precise'_refine* **by** (*clarsimp simp*: *pw_le_iff pw_nres_rel_iff*)

sepref_register *PR_CONST f* ::
's ⇒ *nat list* ⇒ (*nat*, *int*) *acconstraint list* ⇒ *'s* ⇒ *int DBMEntry i_mtx* ⇒ *int DBMEntry i_mtx*

lemma *aux3*:
ID D x'a TYPE(*nat* × *nat* ⇒ *int DBMEntry*) = *ID D x'a TYPE*(*int DBMEntry i_mtx*)
by *simp*

lemmas [*sepref_fr_rules*] =
set_of_list_hnr Leadsto_Impl.lso_id_hnr
op_impl

interpretation *DBM_Impl n* .

sepref_definition *succs_precise_inner_impl* **is**
uncurry4 (*PR_CONST succs_precise_inner*)
:: *location_assn*^{k} $*_a$ (*list_assn clock_assn*)^k $*_a$
(*list_assn* (*acconstraint_assn clock_assn int_assn*))^k $*_a$
location_assn^k $*_a$ (*lso_assn mtx_assn*)^d
$\rightarrow_a$ *lso_assn mtx_assn*
unfolding *PR_CONST_def*
unfolding *succs_precise_inner_def*
list_of_set_def[*symmetric*] *set_of_list_def*[*symmetric*]
apply (*rewrite HOL_list.fold_custom_empty*)
apply *sepref_dbg_keep*
apply *sepref_dbg_id_keep*
unfolding *aux3*
apply *sepref_dbg_id_step+*
apply *sepref_dbg_monadify*
apply *sepref_dbg_opt_init*
apply *sepref_dbg_trans_keep*
apply *sepref_dbg_opt*
apply *sepref_dbg_cons_solve*
apply *sepref_dbg_cons_solve*
apply *sepref_dbg_constraints*
done

sepref_register *succs_precise_inner*

lemmas [*sepref_fr_rules*] = *succs_precise_inner_impl.refine*

lemmas [*sepref_fr_rules*] = *copy_list_lso_assn_refine*[*OF amtx_copy_hnr*]

sepref_definition *succs_precise'_impl* **is**
uncurry succs_precise'
:: *location_assn*^{k} $*_a$ (*lso_assn mtx_assn*)^d
$\rightarrow_a$ *list_assn* (*prod_assn location_assn* (*lso_assn mtx_assn*))
unfolding *PR_CONST_def*
unfolding
comp_def succs_precise'_def
FW''_def[*symmetric*] *rev_map_fold inv_of_A_def*[*symmetric*]

```
    list_of_set_def[symmetric] set_of_list_def[symmetric]
  unfolding HOL_list.fold_custom_empty by sepref

lemmas succs_precise_impl_refine = succs_precise'_impl.refine[FCOMP succs_precise'_correct]

lemma succs_precise_finite:
  ∀ l S. ∀ (l', S')∈set (succs_precise l S). finite S ⟶ finite S'
  unfolding succs_precise_def by auto

definition
  wf_dbm' D ≡ (canonical' D ∨ check_diag n D) ∧
     (list_all (λi. D (i, i) ≤ 0) [0..<n+1]) ∧ list_all (λi. D (0, i) ≤ 0) [0..<n+1]

theorem wf_dbm'_wf_dbm:
  fixes D :: nat × nat ⇒ int DBMEntry
  assumes wf_dbm' D
  shows wf_dbm D
  using assms
  unfolding wf_dbm'_def wf_dbm_def valid_dbm_def list_all_iff canonical'_conv_M_iff
  unfolding valid_dbm.simps
  apply (elim conjE)
  apply (rule conjI)
   apply blast
  apply (rule conjI)
  subgoal
    by (intro impI diag_conv_M) auto
  apply (inst_existentials curry (conv_M D))
    apply (rule HOL.refl)
   apply (rule V_I)
  subgoal
    apply (clarsimp del: disjE)
    subgoal for i
      apply (subgoal_tac D (0, i) ≤ Le 0)
      subgoal
        apply (auto dest: conv_dbm_entry_mono simp: neutral del: disjE)
        done
        apply (drule conv_dbm_entry_mono)
        apply (clarsimp simp: neutral)
        apply (cases i = n)
       apply auto
      done
    done
  apply (rule dbm_int_conv)
  done

lemma canonical'_compute:
  canonical' M =
  list_all (λi.
    list_all (λj.
      list_all (λk.
        M (i, k) ≤ M (i, j) + M (j, k)
  )[0..<n+1]) [0..<n+1]) [0..<n+1]

  unfolding list_all_iff by auto force
```

sepref_definition *canonical'_impl* **is**
RETURN o PR_CONST canonical' :: $mtx\_assn^k \to_a bool\_assn$
unfolding *canonical'_compute list_all_foldli PR_CONST_def* **by** *sepref*

sepref_thm *wf_dbm'_impl* **is**
RETURN o PR_CONST wf_dbm' :: $mtx\_assn^k \to_a bool\_assn$
unfolding *wf_dbm'_def canonical'_compute list_all_foldli PR_CONST_def* **by** *sepref*

definition
states_mem l ≡ *l* ∈ *states'*

definition
P ≡ λ (*l*, *M*). *PR_CONST states_mem l* ∧ *wf_dbm' M*

lemma *P_correct*:
l ∈ *states'* ∧ *wf_dbm M* **if** *P* (*l*, *M*)
using *that* **unfolding** *P_def states_mem_def* **by** (*auto intro*: *wf_dbm'_wf_dbm*)

lemmas [*sepref_import_param*] = *states_mem_impl*[*folded states_mem_def*]

sepref_register *states_mem*

sepref_register *wf_dbm'* :: *'c DBMEntry i_mtx* ⇒ *bool*

lemmas [*sepref_fr_rules*] =

wf_dbm'_impl.refine_raw

sepref_definition *P_impl* **is**
RETURN o PR_CONST P :: $(prod\_assn\ (pure\ loc\_rel)\ mtx\_assn)^k \to_a bool\_assn$
unfolding *PR_CONST_def P_def* **by** *sepref*

lemma *P_impl_refine*:
(*P_impl*, (*RETURN* ∘∘ *PR_CONST*) *P*) ∈ $(location\_assn \times_a mtx\_assn)^k \to_a bool\_assn$
apply *sepref_to_hoare*
apply *sep_auto*
subgoal for *l M l' M'*
using *P_impl.refine*[*to_hnr*, *unfolded hn_refine_def hn_ctxt_def*, *rule_format*, *of* (*l*, *M*)]
by (*sep_auto simp*: *pure_def*)
done

lemma *E_from_op_states*:
l' ∈ *states'* **if** *op_precise.E_from_op* (*l*, *M*) (*l'*, *M'*) *l* ∈ *states'*
using *that* **unfolding** *op_precise.E_from_op_def* **by** *auto*

lemmas [*safe_constraint_rules*] = *location_assn_constraints*

end

context *Reachability_Problem_Impl_Precise*
begin

context
  fixes *L_list* :: *'si list* **and** *P_loc*
  assumes *state_impl_abstract*: $\bigwedge li.\ P\_loc\ li \Longrightarrow \exists\, l.\ (li,\ l) \in loc\_rel$
  assumes *P_loc*: *list_all* $(\lambda x.\ P\_loc\ x \wedge states\_mem\_impl\ x)$ *L_list*
begin

definition
  $L \equiv map\ (\lambda li.\ SOME\ l.\ (li,\ l) \in loc\_rel)\ L\_list$

lemma *mem_states'I*:
  $l \in states'$ **if** *states_mem_impl li* $(li,\ l) \in loc\_rel$ **for** *l li*
  using *states_mem_impl that* **by** (*auto dest*: *fun_relD*)

lemma *L_list_rel*:
  $(L\_list,\ L) \in \langle location\_rel \rangle list\_rel$
  unfolding *list_rel_def L_def*
  using *P_loc*
  apply (*clarsimp simp*: *list.pred_rel list.rel_map*)
  apply (*elim list_all2_mono*)
  apply (*clarsimp simp*: *eq_onp_def*)
  apply (*meson someI_ex state_impl_abstract*)
  apply (*erule mem_states'I*, *meson someI_ex state_impl_abstract*)
  done

lemma *L_list_hnr*:
  $(uncurry0\ (return\ L\_list),\ uncurry0\ (RETURN\ (PR\_CONST\ (set\ L))))$
  $\in unit\_assn^k \rightarrow_a lso\_assn\ location\_assn$
proof −
  have $(\lambda a\ c.\ \uparrow((c,\ a) \in loc\_rel \wedge a \in states')) = pure\ location\_rel$
    unfolding *pure_def* **by** *auto*
  then have *list_assn* $(\lambda a\ c.\ \uparrow((c,\ a) \in loc\_rel \wedge a \in states')) = pure\ (\langle location\_rel \rangle list\_rel)$
    by (*simp add*: *fcomp_norm_unfold*)
  then have $emp \Longrightarrow_A list\_assn\ (\lambda a\ c.\ \uparrow((c,\ a) \in loc\_rel \wedge a \in states'))\ L\ L\_list * true$
    by (*sep_auto simp*: *pure_def intro*: *L_list_rel*)
  then show *?thesis*
    by *sepref_to_hoare* (*sep_auto simp*: *lso_assn_def hr_comp_def br_def*)
qed

sepref_register *list_to_dbm n*

lemmas [*sepref_fr_rules*] = *of_list_list_to_dbm*[*of n*]

sepref_register *set*

lemmas [*sepref_fr_rules*] = *set_of_list_hnr'*

lemmas *step_z_dbm_complete* = *step_z_dbm_complete*[*OF global_clock_numbering'*]

interpretation *A*:
  Simulation

$\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$
$\lambda\ (l,\ Z)\ (l',\ Z').\ \exists\ a.\ conv\_A\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge Z' \neq \{\}$
$\lambda\ (l,\ u)\ (l',\ Z).\ l' = l \wedge u \in Z$
by *standard* (*auto dest*!: *step_z_complete'*)

interpretation *B*:
Simulation
$\lambda\ (l,\ Z)\ (l',\ Z').\ \exists\ a.\ conv\_A\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge Z' \neq \{\}$
$\lambda\ (l,\ M)\ (l',\ M').\ \exists\ a.\ conv\_A\ A \vdash' \langle l,\ M\rangle \rightsquigarrow_{v,n,a} \langle l',\ M'\rangle \wedge \lceil M'\rceil_{v,n} \neq \{\}$
$\lambda\ (l,\ Z)\ (l',\ M).\ l' = l \wedge Z = \lceil M\rceil_{v,n}$
by *standard* (*force simp*: *step_z'_def step_z_dbm'_def elim*!: *step_z_dbm_DBM*)

interpretation
Simulation
$\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$
$\lambda\ (l,\ M)\ (l',\ M').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ M\ v\ n\ a\ l'\ M' \wedge \lceil M'\rceil_{v,n} \neq \{\}$
$\lambda\ (l,\ u)\ (l',\ M).\ l' = l \wedge u \in \lceil M\rceil_{v,n}$
by (*rule Simulation_Composition*, *rule A.Simulation_axioms*, *rule B.Simulation_axioms*) *auto*

lemma *op_precise_buechi_run_correct*:
assumes
$(\nexists\ xs.$
Graph_Defs.run op_precise.E_from_op_empty $((l_0',\ init\_dbm)\ \#\#\ xs)$
$\wedge$ *alw* (*ev* (*holds F*)) $((l_0',\ init\_dbm)\ \#\#\ xs))$
and *F_F1*: $\bigwedge l\ D\ Z.$ *op_precise.E_from_op_empty*** $(l_0',\ init\_dbm)\ (l,\ D)$
$\Longrightarrow$ *dbm.zone_of* (*curry* (*conv_M D*)) $= Z \Longrightarrow F\ (l,\ D) = F1\ (l,\ Z)$
shows
$\nexists\ u\ xs.\ (\forall\ c \leq n.\ u\ c = 0)$
$\wedge$ *Graph_Defs.run* $(\lambda(l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle)\ ((l_0',\ u)\ \#\#\ xs)$
$\wedge$ *alw* (*ev* (*holds F'*)) $((l_0',\ u)\ \#\#\ xs)$
proof −
let *?E* $= \lambda(l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$
define *E* **where** $E \equiv \lambda(l,\ M)\ (l',\ M').\ \exists a.\ conv\_A\ A \vdash' \langle l,\ M\rangle \rightsquigarrow_{v,n,a} \langle l',\ M'\rangle \wedge \lceil M'\rceil_{v,n} \neq \{\}$
interpret *Bisimulation_Invariant*
E
op_precise.E_from_op_empty
$\lambda(l,\ M)\ (l',\ D).\ l' = l \wedge [curry\ (conv\_M\ D)]_{v,n} = [M]_{v,n}$
$\lambda(l,\ y).\ valid\_dbm\ y$
wf_state
unfolding *E_def* **by** (*rule op_precise.step_z_dbm'_E_from_op_bisim_empty*)
have *init*: $u \in dbm.zone\_of\ (curry\ init\_dbm)$ **if** $\forall\ c \leq n.\ u\ c = 0$ **for** $u :: \_ \Rightarrow real$
by (*simp add*: *init_dbm_zone that*)
let *?F1* $= \lambda(l,\ M).\ F1\ (l,\ \lceil M\rceil_{v,n})$
have *alw* (*ev* (*holds F*)) *ys*
if *stream_all2 equiv' xs ys*
alw (*ev* (*holds* $(\lambda(l,\ M).\ F1\ (l,\ dbm.zone\_of\ M))$)) *xs*
B.run $((l_0',\ init\_dbm)\ \#\#\ ys)$
for *xs ys*
proof −
from *that*(*3*) **have** *pred_stream* $(B.reaches\ (l_0',\ init\_dbm))$ *ys*
by (*rule B.run_first_reaches*)
with *that*(*1*) **have** *stream_all2* $(\lambda x\ y.\ equiv'\ x\ y \wedge B.reaches\ (l_0',\ init\_dbm)\ y)$ *xs ys*
by (*rule stream_all2_pred_stream_combine*) (*rule conjI*)
with *that*(*2*) **show** *?thesis*

```
    apply (rule alw_ev_lockstep)
    unfolding equiv'_def using F_F1 by blast
  qed
  with assms have ∄xs.
    Graph_Defs.run E ((l₀', curry (conv_M init_dbm)) ## xs)
  ∧ alw (ev (holds ?F1)) ((l₀', curry (conv_M init_dbm)) ## xs)
    apply safe
    apply (drule bisim.A_B.simulation_run[where y = (l₀', init_dbm)])
    using valid_init_dbm unfolding equiv'_def
    by (auto simp: wf_state_def dest: bisim.A_B.simulation_run[where y = (l₀', init_dbm)])
  then show ?thesis
    unfolding E_def
    by (auto
        intro: F'_F1
        dest: alw_ev_lockstep[where R = ?F1]
        dest!: simulation_run[where y = (l₀', curry init_dbm)] init
        )
qed

lemma op_precise_unreachable_correct:
  assumes ∄s'. op_precise.E_from_op_empty** (l₀, init_dbm) s' ∧ F s'
  shows ∄u l' u'. (∀c ≤ n. u c = 0) ∧ conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩ ∧ F' (l', u')
proof −
  define E where E ≡ λ(l, M) (l', M'). ∃ a. conv_A A ⊢' ⟨l, M⟩ ⇝_{v,n,a} ⟨l', M'⟩ ∧ [M']_{v,n} ≠
{}
  interpret Bisimulation_Invariant
    E
    op_precise.E_from_op_empty
    λ(l, M) (l', D). l' = l ∧ [curry (conv_M D)]_{v,n} = [M]_{v,n}
    λ(l, y). valid_dbm y
    wf_state
    unfolding E_def by (rule op_precise.step_z_dbm'_E_from_op_bisim_empty)
  have 1: reaches (l₀, u) (l', u') if conv_A A ⊢' ⟨l₀, u⟩ →* ⟨l', u'⟩ for u u' l'
    by (simp add: steps'_iff that)
  have 2: u ∈ dbm.zone_of (curry init_dbm) if ∀c ≤ n. u c = 0 for u :: _ ⇒ real
    by (simp add: init_dbm_zone that)
  from assms have
    ∄l' M'. E** (l₀, curry (conv_M init_dbm)) (l', M') ∧ F1 (l', [M']_{v,n}) ∧ [M']_{v,n} ≠ {}
    apply (clarsimp simp: )
    apply (drule bisim.A_B_reaches[where b = (l₀, init_dbm)])
    subgoal
      using valid_init_dbm unfolding equiv'_def
      by (auto simp: wf_state_def)
    unfolding equiv'_def using canonical_check_diag_empty_iff
    using F_F1 by blast
  then show ?thesis
   unfolding E_def by (fastforce dest: dbm.check_diag_empty F'_F1 dest!: simulation_reaches
1 2)
qed

lemma op_precise_unreachable_correct':
  (uncurry0 (SPEC (λr. r ⟶
     (∄s'. op_precise.E_from_op_empty** (l₀, init_dbm) s' ∧ F s'))),
    uncurry0 (SPEC (λr. r ⟶
```

(∄ u l′ u′. (∀ c ≤ n. u c = 0) ∧ conv_A A ⊢′ ⟨l₀, u⟩ →* ⟨l′, u′⟩ ∧ F′ (l′, u′)))))
∈ Id → ⟨Id⟩nres_rel
using *op_precise_unreachable_correct* **by** (*clarsimp simp*: *pw_le_iff pw_nres_rel_iff*)

lemma *IArray_list_to_dbm_rel*[*param*]:
(*IArray*, *list_to_dbm n*)
∈ {(*xs*, *ys*). *xs* = *ys* ∧ *length xs* = *Suc n* ∗ *Suc n*} → {(*a*, *b*). *iarray_mtx_rel* (*Suc n*) (*Suc n*) *b a*}
unfolding *list_to_dbm_def op_amtx_new_def iarray_mtx_rel_def*
Normalized_Zone_Semantics_Certification_Impl.dbm_tab_def
by (*auto simp*: *algebra_simps*)

lemma *IArray_list_to_dbm_rel′*:
(*map IArray xs*, *list_to_dbm n* ‘ *set xs*)
∈ ⟨{(*a*, *b*). *iarray_mtx_rel* (*Suc n*) (*Suc n*) *b a*}⟩*list_set_rel*
if *list_all* (λ*xs*. *length xs* = *Suc n* ∗ *Suc n*) *xs*
using *that* **by** (*rule map_set_rel*) (*rule IArray_list_to_dbm_rel*)

context
fixes *M_list* :: (′*si* × ′*v list*) *list* **and** *g* :: ′*v* ⇒ ′*v1* **and** *h* :: ′*v* ⇒ ′*v2*
and *P* :: ′*v* ⇒ *bool* **and** *R* :: (′*v2* × ′*v1*) *set*
begin

definition
M_list1 ≡ *map* (λ(*li*, *xs*). (*SOME l*. (*li*, *l*) ∈ *loc_rel*, *xs*)) *M_list*

definition
M1 = *fold* (λ*p M*.
let
s = *fst p*; *xs* = *snd p*;
xs = *rev* (*map g xs*);
S = *set xs in fun_upd M s* (*Some S*)
) (*PR_CONST M_list1*) *IICF_Map.op_map_empty*

definition
M1i = *hashmap_of_list* (*map* (λ(*k*, *dbms*). (*k*, *map h dbms*)) *M_list*)

context
assumes *M_list_covered*: *fst* ‘ *set M_list* ⊆ *set L_list*
and *M_P*: *list_all* (λ(*l*, *xs*). *list_all P xs*) *M_list*
assumes *g_h_param*: (*h*, *g*) ∈ {(*x*, *y*). *x* = *y* ∧ *P x*} → *R*
begin

lemma *M1_finite*:
∀ *S*∈*ran M1*. *finite S*
unfolding *M1_def*
apply (*rule fold_generalize_start*[**where** *P* = λ*M*. ∀ *S*∈*ran M*. *finite S*])
subgoal for *a*
unfolding *M_list1_def*

```
    apply (induction M_list arbitrary: a)
     apply (simp; fail)
    apply (simp, rprem, auto dest: ran_upd_cases)
    done
  apply (simp; fail)
  done

lemma P_loc1:
  list_all
    (λ(l, xs). P_loc l ∧ states_mem_impl l ∧ list_all P xs) M_list
  using P_loc ‹_ ⊆ set L_list› M_P unfolding list_all_iff by auto

lemma M_list_rel1:
  (M_list, M_list1) ∈ ⟨location_rel ×r ⟨br id P⟩list_rel⟩list_rel
  unfolding list_rel_def M_list1_def
  using P_loc1
  apply (clarsimp simp: list.pred_rel list.rel_map br_def)
  apply (elim list_all2_mono)
  apply (clarsimp simp: eq_onp_def)
  apply (meson someI_ex state_impl_abstract)
   apply (erule mem_states'I, meson someI_ex state_impl_abstract)
  apply (elim list_all2_mono, clarsimp)
  done

lemma dom_M_eq1_aux:
  dom (fold (λp M.
    let s = fst p; xs = snd p;
    xs = rev (map g xs); S = set xs in fun_upd M s (Some S)
  ) xs m) = dom m ∪ fst ‘ set xs for xs m
    by (induction xs arbitrary: m) auto

lemma dom_M_eq1:
  dom M1 = fst ‘ set M_list1
  unfolding dom_M_eq1_aux M1_def by simp

lemma L_dom_M_eqI1:
  assumes fst ‘ set M_list = set L_list
  shows set L = dom M1
proof −
  show ?thesis
    unfolding dom_M_eq1
  proof (safe)
    fix l assume l ∈ set L
    with L_list_rel assms obtain l' where l' ∈ fst ‘ set M_list (l', l) ∈ location_rel
    by (fastforce simp: list_all2_append2 list_all2_Cons2 list_rel_def elim!: in_set_list_format)
    with M_list_rel1 obtain l1 where l1 ∈ fst ‘ set M_list1 (l', l1) ∈ location_rel
    by (fastforce simp: list_all2_append1 list_all2_Cons1 list_rel_def elim!: in_set_list_format)
    with ‹(l', l) ∈ location_rel› show l ∈ fst ‘ set M_list1
      using loc_rel_right_unique by auto
  next
    fix l M assume (l, M) ∈ set M_list1
    with M_list_rel1 assms obtain l' where l' ∈ set L_list (l', l) ∈ location_rel
    by (fastforce simp: list_all2_append2 list_all2_Cons2 list_rel_def elim!: in_set_list_format)
    with L_list_rel obtain l1 where l1 ∈ set L (l', l1) ∈ location_rel
```

```
    by (fastforce simp: list_all2_append1 list_all2_Cons1 list_rel_def elim!: in_set_list_format)
    with ‹(l', l) ∈ location_rel› show fst (l, M) ∈ set L
      using loc_rel_right_unique by auto
  qed
qed

lemma map_of_M_list_M_rel1:
  (map_of_list (map (λ(k, dbms). (k, map h dbms)) M_list), M1)
∈ location_rel → ⟨⟨R⟩list_set_rel⟩option_rel
  unfolding M1_def M_list1_def
  unfolding map_of_list_def
  unfolding PR_CONST_def
proof goal_cases
  case 1
  let (fold ?f ?xs Map.empty, fold ?g ?ys _) ∈ ?R = ?case
  have ∗: l' = (SOME l'. (l, l') ∈ loc_rel)
    if (l, l') ∈ loc_rel states_mem_impl l l' ∈ states' for l l'
  proof −
    from that have (l, SOME l'. (l, l') ∈ loc_rel) ∈ loc_rel
      by (intro someI)
    moreover then have (SOME l'. (l, l') ∈ loc_rel) ∈ states'
      using that(2) by (elim mem_states'I)
    ultimately show ?thesis
      using that right_unique_location_rel unfolding single_valued_def by auto
  qed
  have (fold ?f ?xs m, fold ?g ?ys m') ∈ ?R
    if (m, m') ∈ ?R for m m'
    using that P_loc1
  proof (induction M_list arbitrary: m m')
    case Nil
    then show ?case
      by simp
  next
    case (Cons x M_list)
    obtain l M where x = (l, M)
      by force
    from Cons.IH ‹list_all _ (x # M_list)› show ?case
      apply (simp split:)
      apply rprems
      unfolding ‹x = _›
      apply simp
      apply (rule fun_relI)
      apply (clarsimp; safe)
      subgoal
        using g_h_param by (auto dest!: fun_relD intro!: map_set_rel)
      subgoal
        by (frule ∗) auto
      subgoal
        using left_unique_location_rel unfolding IS_LEFT_UNIQUE_def single_valued_def
        by (auto dest: someI_ex[OF state_impl_abstract])
      subgoal
        using Cons.prems(1)
        apply −
        apply (drule fun_relD)
```

```
      by simp
    done
  qed
  then show ?case
    by rprems auto
qed

lemma Mi_M1:
  (λk. Impl_Array_Hash_Map.ahm_lookup (=) bounded_hashcode_nat k M1i, M1)
  ∈ location_rel → ⟨⟨R⟩list_set_rel⟩option_rel
(is (?f, M1) ∈ ?R)
proof −
  let ?g = map_of_list (map (λ(k, dbms). (k, map h dbms)) M_list)
  have (?f, ?g) ∈ Id → ⟨Id⟩option_rel
    unfolding M1i_def by (rule hashmap_of_list_lookup)
  moreover have (?g, M1) ∈ ?R
    by (rule map_of_M_list_M_rel1)
  ultimately show ?thesis
    by auto
qed

end

end



context
  fixes M_list :: ('si × int DBMEntry list list) list
  assumes M_list_covered: fst ‘ set M_list ⊆ set L_list
      and M_dbm_len: list_all (λ(l, xs). list_all (λM. length M = Suc n ∗ Suc n) xs) M_list
begin

lemmas M_assms = M_list_covered M_dbm_len IArray_list_to_dbm_rel

definition
  M_list' ≡ M_list1 M_list

definition
  M = fold (λp M.
    let s = fst p; xs = snd p; xs = rev (map (list_to_dbm n) xs); S = set xs in fun_upd M s
(Some S)
  ) (PR_CONST M_list') IICF_Map.op_map_empty

lemma M_alt_def:
  M = M1 TYPE(int DBMEntry list) M_list (list_to_dbm n)
  unfolding M_def M1_def M_list'_def ..

lemma M_finite:
  ∀ S∈ran M. finite S
  unfolding M_alt_def by (rule M1_finite[OF M_assms])

lemmas M_list_rel = M_list_rel1[OF M_assms, folded M_list'_def]
```

lemma *M_list_hnr*[*sepref_fr_rules*]:
  (*uncurry0* (*return M_list*), *uncurry0* (*RETURN* (*PR_CONST M_list'*)))
    ∈ *id_assn*^{k} →$_a$ *list_assn* (
        location_assn ×$_a$ *list_assn* (*pure* (*b_rel Id* (λ*xs*. *length xs* = *Suc n* ∗ *Suc n*))))
proof −
  let *?R1* = ⟨*br id* (λ*xs*. *length xs* = *Suc n* ∗ *Suc n*)⟩*list_rel*
  let *?R2* = λ*a c*. ↑ (*a* = *c* ∧ *length c* = *Suc* (*n* + (*n* + *n* ∗ *n*)))
  let *?R* = (λ*a c*. ↑ ((*c*, *a*) ∈ *loc_rel* ∧ *a* ∈ *states'*)) ×$_a$ *list_assn ?R2*
  have *b_rel Id* = *br id*
    unfolding *br_def b_rel_def* **by** *auto*
  have ∗: *list_assn* (λ*a c*. ↑ (*a* = *c* ∧ *length c* = *Suc* (*n* + (*n* + *n* ∗ *n*)))) = *pure ?R1*
    unfolding *fcomp_norm_unfold* **by** (*simp add*: *pure_def br_def*)
  have *?R* = *pure* (*location_rel* ×$_r$ *?R1*)
    unfolding ∗ *pure_def prod_assn_def* **by** (*intro ext*) *auto*
  then have ∗∗: *list_assn ?R* = *pure* (⟨*location_rel* ×$_r$ *?R1*⟩*list_rel*)
    unfolding *fcomp_norm_unfold* **by** *simp* (*fo_rule HOL.arg_cong*)
  have *emp* ⟹$_A$ *list_assn ?R M_list' M_list* ∗ *true*
    using *M_list_rel* **unfolding** ∗∗ **by** (*sep_auto simp*: *pure_def*)
  then show *?thesis*
    by *sepref_to_hoare* (*sep_auto simp*: *lso_assn_def hr_comp_def br_def* ‹*b_rel Id* = *br id*›)
qed

sepref_register *PR_CONST M_list'*

interpretation *DBM_Impl n* **.**

sepref_definition *M_table* **is**
  uncurry0 (*RETURN M*) :: *unit_assn*^{k} →$_a$ *hm.hms_assn' location_assn* (*lso_assn mtx_assn*)
  unfolding *M_def set_of_list_def*[*symmetric*] *rev_map_fold*
    HOL_list.fold_custom_empty hm.op_hms_empty_def[*symmetric*]
  by *sepref*

lemmas *dom_M_eq* = *dom_M_eq1*[*OF M_assms*, *folded M_alt_def M_list'_def*]

interpretation
  Reachability_Impl
  where *A* = *mtx_assn*
    and *F* = *F*
    and *l*$_0$*i* = *return l*$_0$*i*
    and *s*$_0$ = *init_dbm*
    and *s*$_0$*i* = *init_dbm_impl*
    and *succs* = *succs_precise*
    and *succsi* = *succs_precise'_impl*
    and *less* = λ *x y*. *dbm_subset n x y* ∧ ¬ *dbm_subset n y x*
    and *less_eq* = *dbm_subset n*
    and *Lei* = *dbm_subset_impl n*
    and *E* = *op_precise.E_from_op_empty*
    and *Fi* = *F_impl*
    and *K* = *location_assn*
    and *keyi* = *return o fst*
    and *copyi* = *amtx_copy*
    and *P* = λ(*l*, *M*). *l* ∈ *states'* ∧ *wf_dbm M*
    and *P'* = *P*
    and *Pi* = *P_impl*

```
  and L = set L
  and M = M
 apply standard
                  apply (rule HOL.refl; fail)
                  apply (rule dbm_subset_refl; fail)
                  apply (rule dbm_subset_trans; assumption)
 subgoal
  unfolding succs_precise_def op_precise.E_from_op_empty_def op_precise.E_from_op_def
  apply simp
  apply safe
  subgoal
    by (drule trans_impl_trans_of) auto
  apply simp
  apply (drule trans_of_trans_impl, solves simp)
  apply (intro exI conjI, erule image_eqI[rotated])
  apply auto
  done
 subgoal
  by (auto dest: P_correct)
 subgoal
  by (rule F_mono)
 subgoal
  ..
 subgoal
  by (rule M_finite)
 subgoal
  by (rule succs_precise_finite)
 subgoal
  unfolding succs_precise_def by auto
 subgoal
  by sepref_to_hoare sep_auto
       apply (rule amtx_copy_hnr; fail)
 subgoal
  by (rule P_impl_refine)
 subgoal
  by (rule F_impl)
 subgoal
  using succs_precise_impl_refine unfolding b_assn_pure_conv .
    apply (rule dbm_subset_impl.refine; fail)
     apply (rule location_assn_constraints; fail)+
 subgoal
  by (auto dest: op_precise.E_from_op_empty_mono')
 subgoal
  by (clarsimp simp: op_precise.E_from_op_empty_def, frule op_precise.E_from_op_wf_state[rotated])
    (auto dest: E_from_op_states simp: wf_state_def)
 subgoal
  using init_impl states'_states by sepref_to_hoare sep_auto
   apply (unfold PR_CONST_def, rule init_dbm_impl.refine; fail)
 done

lemmas reachability_impl = Reachability_Impl_axioms

definition
  Mi = hashmap_of_list (map (λ(k, dbms). (k, map IArray dbms)) M_list)
```

lemma *Mi_alt_def*:
Mi = *M1i TYPE*(*int DBMEntry list*) *M_list IArray*
unfolding *Mi_def M1i_def* **..**

lemmas *map_of_M_list_M_rel* = *map_of_M_list_M_rel1*[*OF M_assms*, *folded M_alt_def*]

lemmas *Mi_M* = *Mi_M1*[*OF M_assms*, *folded M_alt_def Mi_alt_def*]

lemmas *L_dom_M_eqI* = *L_dom_M_eqI1*[*OF M_assms*, *folded M_alt_def*]

context
fixes *Li_split* :: *'si list list*
assumes *full_split*: *set L_list* = (⋃ *xs* ∈ *set Li_split*. *set xs*)
begin

interpretation *Reachability_Impl_imp_to_pure_correct*
where *A* = *mtx_assn*
and *F* = *F*
and $l_0 i$ = *return* l_0*i*
and s_0 = *init_dbm*
and $s_0 i$ = *init_dbm_impl*
and *succs* = *succs_precise*
and *succsi* = *succs_precise'_impl*
and *less* = λ *x y*. *dbm_subset n x y* ∧ ¬ *dbm_subset n y x*
and *less_eq* = *dbm_subset n*
and *Lei* = *dbm_subset_impl n*
and *lei* = λ*as bs*.
(∃ *i*≤*n*. *IArray.sub as* (*i* + *i* ∗ *n* + *i*) < *Le 0*) ∨ *array_all2* (*Suc n* ∗ *Suc n*) (≤) *as bs*
and *E* = *op_precise.E_from_op_empty*
and *Fi* = *F_impl*
and *K* = *location_assn*
and *keyi* = *return o fst*
and *copyi* = *amtx_copy*
and *P* = λ(*l*, *M*). *l* ∈ *states'* ∧ *wf_dbm M*
and *P'* = *P*
and *Pi* = *P_impl*
and *L* = *set L*
and *M* = *M*
and *to_loc* = *id*
and *from_loc* = *id*
and *L_list* = *L_list*
and *K_rel* = *location_rel*
and *L'* = *L*
and *Li* = *L_list*
and *to_state* = *array_unfreeze*
and *from_state* = *array_freeze*
and *A_rel* = {(*a*, *b*). *iarray_mtx_rel* (*Suc n*) (*Suc n*) *b a*}
and *Mi* = λ*k*. *Impl_Array_Hash_Map.ahm_lookup* (=) *bounded_hashcode_nat k Mi*
apply *standard*
subgoal
using *L_list_rel* **by** *simp*
subgoal
by (*rule L_list_rel*)

subgoal
..
subgoal for *s1 s*
by (*rule array_unfreeze_ht*) *simp*
subgoal for *si s*
by (*sep_auto heap*: *array_freeze_ht*)
subgoal
by *simp*
subgoal
by *simp*
subgoal
by (*rule right_unique_location_rel*)
subgoal
using *left_unique_location_rel* **unfolding** *IS_LEFT_UNIQUE_def* .
subgoal
unfolding *dbm_subset_def check_diag_def*
by (*auto simp*: *array_all2_iff_pointwise_cmp*[*symmetric*] *iarray_mtx_relD*)
subgoal
using *full_split* .
using *Mi_M* .

concrete_definition *certify_unreachable_pure*
uses *pure.certify_unreachable_impl_pure_correct*[*unfolded to_pair_def get_succs_def*] **is** *?f* $\longrightarrow$ _

lemma *certify_unreachable_pure_refine*:
assumes *fst* ‘ *set M_list* = *set L_list certify_unreachable_pure*
shows $\nexists u\ l'\ u'.\ (\forall c{\le}n.\ u\ c = 0) \wedge conv\_A\ A \vdash' \langle l_0, u\rangle \rightarrow* \langle l', u'\rangle \wedge F'\ (l', u')$
using *certify_unreachable_pure.refine*[*OF L_dom_M_eqI*] *assms op_precise_unreachable_correct*
by *simp*

end

context
fixes *splitter* :: *'s list* $\Rightarrow$ *'s list list* **and** *splitteri* :: *'si list* $\Rightarrow$ *'si list list*
assumes *full_split*: *set xs* = $(\bigcup xs \in set\ (splitter\ xs).\ set\ xs)$
and *same_split*:
$\bigwedge L\ Li.$
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
begin

lemmas *certify_unreachable_impl_hnr* =
certify_unreachable_impl.refine[
OF Reachability_Impl_axioms L_list_hnr, *unfolded PR_CONST_def*, *OF M_table.refine*,
OF full_split same_split,
FCOMP op_precise_unreachable_correct'
]

definition
unreachability_checker ≡
let
Fi = *F_impl*;
Pi = *P_impl*;
copyi = *amtx_copy*;

```
    Lei = dbm_subset_impl n;
    l_0i = Heap_Monad.return l_0i;
    s_0i = init_dbm_impl;
    succsi = succs_precise'_impl;
    M_table = M_table
  in
    certify_unreachable_impl Fi Pi copyi Lei succsi l_0i s_0i L_list M_table splitteri

lemma unreachability_checker_alt_def:
  unreachability_checker ≡
  let
    Fi = F_impl;
    Pi = P_impl;
    copyi = amtx_copy;
    Lei = dbm_subset_impl n;
    l_0i = Heap_Monad.return l_0i;
    s_0i = init_dbm_impl;
    succsi = succs_precise'_impl
  in do {
    M_table ← M_table;
    certify_unreachable_impl_inner Fi Pi copyi Lei succsi l_0i s_0i splitteri L_list M_table
  }
  unfolding unreachability_checker_def certify_unreachable_impl_def Let_def .

lemmas unreachability_checker_hnr =
  certify_unreachable_impl_hnr[folded unreachability_checker_def[unfolded Let_def]]

lemmas unreachability_checker_alt_def' = unreachability_checker_alt_def[unfolded M_table_def]

definition
  unreachability_checker2 ≡
  let
    Fi = F_impl;
    Pi = P_impl;
    copyi = amtx_copy;
    Lei = dbm_subset_impl n;
    l_0i = Heap_Monad.return l_0i;
    s_0i = init_dbm_impl;
    succsi = succs_precise'_impl;
    M_table = M_table
  in
    certify_unreachable_impl2 Fi Pi copyi Lei succsi l_0i s_0i splitteri L_list M_table

lemmas unreachability_checker2_refine = certify_unreachable_impl2_refine[
    of L_list M_table splitter splitteri,
    OF L_list_hnr _ full_split same_split L_dom_M_eqI[symmetric],
    unfolded PR_CONST_def, OF M_table.refine,
    folded unreachability_checker2_def[unfolded Let_def],
    THEN mp, THEN op_precise_unreachable_correct
]

end

end
```

context
  fixes *M_list* :: (*'si* × (*int DBMEntry list* × *nat*) *list*) *list*
  assumes *M_list_covered*: *fst* ‘ *set M_list* ⊆ *set L_list*
    and *M_dbm_len*: *list_all* (λ(*l*, *xs*). *list_all* (λ(*M*, _). *length M* = *Suc n* ∗ *Suc n*) *xs*) *M_list*
begin

lemma *conversions_param*:
  (λ(*M*, *i*). (*IArray M*, *i*), λ(*M*, *i*). (*list_to_dbm n M*, *i*))
∈ {(*x*, *y*). *x* = *y* ∧ (λ(*M*, _). *length M* = *Suc n* ∗ *Suc n*) *x*} →
  {(*a*, *b*). *iarray_mtx_rel* (*Suc n*) (*Suc n*) *b a*} ×$_r$ *Id*
  using *IArray_list_to_dbm_rel* **by** (*auto dest*!: *fun_relD*)

lemmas *M2_assms* =
  M_list_covered
  M_dbm_len
  conversions_param

definition
  M_list2 ≡ *map* (λ(*li*, *xs*). (*SOME l*. (*li*, *l*) ∈ *loc_rel*, *xs*)) *M_list*

definition
  M2 = *fold* (λ*p M*.
    let
      s = *fst p*; *xs* = *snd p*;
      xs = *rev* (*map* (λ(*M*, *i*). (*list_to_dbm n M*, *i*)) *xs*);
      S = *set xs in fun_upd M s* (*Some S*)
) (*PR_CONST M_list2*) *IICF_Map.op_map_empty*

lemma *M_list2_alt_def*:
  M_list2 = *M_list1 M_list*
  unfolding *M_list2_def M_list1_def* **..**

lemma *M2_alt_def*:
  M2 = *M1 TYPE*(*int DBMEntry list* × *nat*) *M_list* (λ(*M*, *i*). (*list_to_dbm n M*, *i*))
  unfolding *M1_def M2_def M_list1_def M_list2_def* **..**

lemmas *M2_finite* = *M1_finite*[*OF M2_assms*, *folded M2_alt_def*]

lemmas *L_dom_M_eqI2* = *L_dom_M_eqI1*[*OF M2_assms*, *folded M2_alt_def M_list2_alt_def*]

definition
  M2i = *hashmap_of_list* (*map* (λ(*k*, *dbms*). (*k*, *map* (λ(*M*, *i*). (*IArray M*, *i*)) *dbms*)) *M_list*)

lemma *M2i_alt_def*:
  M2i = *M1i TYPE*(*int DBMEntry list* × *nat*) *M_list* (λ(*M*, *i*). (*IArray M*, *i*))
  unfolding *M2i_def M1i_def* **..**

lemmas *map_of_M_list_M_rel2* = *map_of_M_list_M_rel1*[*OF M2_assms*, *folded M2_alt_def*]

lemmas *Mi_M2* = *Mi_M1*[*OF M2_assms*, *folded M2i_alt_def M2_alt_def*]

interpretation

```
  Buechi_Impl_pre
  where F = F
    and succs = succs_precise
    and less = λ x y. dbm_subset n x y ∧ ¬ dbm_subset n y x
    and less_eq = dbm_subset n
    and E = op_precise.E_from_op_empty
    and P = λ(l, M). l ∈ states' ∧ wf_dbm M
    and P' = P
    and L = set L
    and M = λx. case M2 x of None ⇒ {} | Some S ⇒ S
  apply standard
                  apply (rule HOL.refl; fail)
                  apply (rule dbm_subset_refl; fail)
                  apply (rule dbm_subset_trans; assumption)
  subgoal
   unfolding succs_precise_def op_precise.E_from_op_empty_def op_precise.E_from_op_def
    apply simp
    apply safe
    subgoal
      by (drule trans_impl_trans_of) auto
    apply simp
    apply (drule trans_of_trans_impl, solves simp)
    apply (intro exI conjI, erule image_eqI[rotated])
    apply auto
    done
  subgoal
    by (auto dest: P_correct)
  subgoal
    by (rule F_mono)
  subgoal
    ..
  subgoal
    using M2_finite by (auto split: option.split intro: ranI)
  done


context
  fixes Li_split :: 'si list list
  assumes full_split: set L_list = (⋃ xs ∈ set Li_split. set xs)
  fixes init_locsi :: 'si list and init_locs :: 's set
  assumes init_locs_in_states: init_locs ⊆ states'
  assumes initsi_inits:
    (init_locsi, init_locs) ∈ ⟨loc_rel⟩list_set_rel
begin

definition
  init_locs1 = (SOME xs. set xs = init_locs ∧ (init_locsi, xs) ∈ ⟨loc_rel⟩list_rel)

lemma init_locs1:
  set init_locs1 = init_locs ∧ (init_locsi, init_locs1) ∈ ⟨loc_rel⟩list_rel
  using initsi_inits unfolding list_set_rel_def
  apply (elim relcompE)
  unfolding init_locs1_def
  apply (rule someI)
```

apply *auto*
done

lemma *init_locsi_init_locs1*:
(*init_locsi*, *init_locs1*) ∈ ⟨*location_rel*⟩*list_rel*
using *init_locs1 init_locs_in_states* **unfolding** *b_rel_def*
unfolding *list_rel_def* **by** (*auto elim*!: *list.rel_mono_strong*)

lemma [*sepref_fr_rules*]:
(*uncurry0* (*return li*), *uncurry0* (*RETURN* (*PR_CONST l*)))
∈ *unit_assn*^{k} →$_a$ *location_assn* **if** (*li*, *l*) ∈ *loc_rel l* ∈ *states*′
using *that* **by** *sepref_to_hoare sep_auto*

lemma *init_locsi_refine*[*sepref_fr_rules*]:
(*uncurry0* (*return init_locsi*), *uncurry0* (*RETURN* (*PR_CONST init_locs1*)))
∈ *unit_assn*^{k} →$_a$ *list_assn location_assn*
proof −
let *?x* = *list_assn* (λ*a c*. ↑ ((*c*, *a*) ∈ *loc_rel* ∧ *a* ∈ *states*′))
have *?x* = *pure* (⟨*location_rel*⟩*list_rel*)
unfolding *fcomp_norm_unfold* **unfolding** *b_assn_def pure_def* **by** *simp*
then have *emp* ⟹$_A$ *?x init_locs1 init_locsi* ∗ *true*
using *init_locsi_init_locs1* **by** (*simp add*: *pure_app_eq*)
then show *?thesis*
by *sepref_to_hoare sep_auto*
qed

definition
inits = *map* (λ*l*. (*l*, *init_dbm*)) *init_locs1*

interpretation *DBM_Impl n* **.**

sepref_register *init_locs1*

sepref_definition *initsi*
is *uncurry0* (*RETURN* (*PR_CONST inits*))
:: *unit_assn*^{k} →$_a$ *list_assn* (*prod_assn location_assn mtx_assn*)
unfolding *inits_def PR_CONST_def*
unfolding *map_by_foldl*[*symmetric*] *foldl_conv_fold HOL_list.fold_custom_empty*
by *sepref*

interpretation *Buechi_Impl_imp_to_pure_correct*
where *A* = *mtx_assn*
and *F* = *F*
and *succs* = *succs_precise*
and *succsi* = *succs_precise*′*_impl*
and *less* = λ *x y*. *dbm_subset n x y* ∧ ¬ *dbm_subset n y x*
and *less_eq* = *dbm_subset n*
and *Lei* = *dbm_subset_impl n*
and *lei* = λ*as bs*.
(∃ *i*≤*n*. *IArray.sub as* (*i* + *i* ∗ *n* + *i*) < *Le 0*) ∨ *array_all2* (*Suc n* ∗ *Suc n*) (≤) *as bs*
and *E* = *op_precise.E_from_op_empty*
and *Fi* = *F_impl*
and *K* = *location_assn*
and *keyi* = *return o fst*

and *copyi* = *amtx_copy*
and $P = \lambda(l,\ M).\ l \in states' \wedge wf\_dbm\ M$
and $P' = P$
and *Pi* = *P_impl*
and *L* = *set L*
and *M* = *M2*
and *to_loc* = *id*
and *from_loc* = *id*
and *L_list* = *L_list*
and *K_rel* = *location_rel*
and $L' = L$
and *Li* = *L_list*
and *to_state* = *array_unfreeze*
and *from_state* = *array_freeze*
and $A\_rel = \{(a,\ b).\ iarray\_mtx\_rel\ (Suc\ n)\ (Suc\ n)\ b\ a\}$
and *Mi* = $\lambda k.$ *Impl_Array_Hash_Map.ahm_lookup* (=) *bounded_hashcode_nat k M2i*
and *inits* = *inits*
and *initsi* = *initsi*
apply *standard*
subgoal
by *sepref_to_hoare sep_auto*
apply (*rule amtx_copy_hnr*; *fail*)
subgoal
by (*rule P_impl_refine*)
subgoal
by (*rule F_impl*)
subgoal
using *succs_precise_impl_refine* **unfolding** *b_assn_pure_conv* **.**
apply (*rule dbm_subset_impl.refine*; *fail*)
apply (*rule location_assn_constraints*; *fail*)+
subgoal
using *L_list_rel* **by** *simp*
subgoal
by (*rule L_list_rel*)
subgoal
..
subgoal for *s1 s*
by (*rule array_unfreeze_ht*) *simp*
subgoal for *si s*
by (*sep_auto heap*: *array_freeze_ht*)
subgoal
by *simp*
subgoal
by *simp*
subgoal
by (*rule right_unique_location_rel*)
subgoal
using *left_unique_location_rel* **unfolding** *IS_LEFT_UNIQUE_def* **.**
subgoal
unfolding *dbm_subset_def check_diag_def*
by (*auto simp*: *array_all2_iff_pointwise_cmp*[*symmetric*] *iarray_mtx_relD*)
subgoal
using *full_split* **.**
subgoal

using *initsi.refine* **.**
subgoal
using *Mi_M2* **.**
subgoal
by (*auto dest*: *op_precise.E_from_op_empty_mono′*)
subgoal
by (*clarsimp simp*: *op_precise.E_from_op_empty_def*, *frule op_precise.E_from_op_wf_state*[*rotated*])
(*auto dest*: *E_from_op_states simp*: *wf_state_def*)
done

concrete_definition *certify_no_buechi_run_pure*
uses *pure.certify_no_buechi_run_impl_pure_correct*[*unfolded to_pair_def get_succs_def*]
is *?f* ⟶ _

lemma *certify_no_buechi_run_pure_refine*:
assumes *fst* ‘ *set M_list* = *set L_list certify_no_buechi_run_pure*
and *F_F1*: ⋀l_0 *l D Z*. l_0 ∈ *init_locs* ⟹ *op_precise.E_from_op_empty*** (l_0, *init_dbm*) (*l*, *D*)
⟹ *dbm.zone_of* (*curry* (*conv_M D*)) = *Z* ⟹ *F* (*l*, *D*) = *F1* (*l*, *Z*)
shows ∄l_0 *u xs*. l_0 ∈ *init_locs* ∧ (∀ *c*≤*n*. *u c* = *0*) ∧ *run* ((l_0, *u*) ## *xs*) ∧ *alw* (*ev* (*holds F′*)) ((l_0, *u*) ## *xs*)
using *certify_no_buechi_run_pure.refine*[*OF L_dom_M_eqI2*] *assms*(*1*,*2*)
op_precise_buechi_run_correct[*OF* _ *F_F1*]
unfolding *inits_def* **using** *init_locs1* **by** *simp*

end

end

end

end

context *TA_Impl_Precise*
begin

lemma (**in** *TA_Impl*) *dbm_subset_correct*:
assumes *wf_dbm D* **and** *wf_dbm M*
shows [*curry* (*conv_M D*)]$_{v,n}$ ⊆ [*curry* (*conv_M M*)]$_{v,n}$ ⟷ *dbm_subset n D M*
unfolding *dbm_subset_correct′′*[*OF assms*] **using** *dbm_subset_conv_rev dbm_subset_conv* **..**

lemma *empty_steps_states′*:
l′ ∈ *states′* **if** *op_precise.E_from_op_empty*** (*l*, *D*) (*l′*, *D′*) *l* ∈ *states′*
using *that*
proof (*induction* (*l*, *D*) (*l′*, *D′*) *arbitrary*: *l′ D′*)
case *rtrancl_refl*
then show *?case*
by *simp*
next
case (*rtrancl_into_rtrancl b*)
then show *?case*
by (*cases b*) (*auto simp add*: *op_precise.E_from_op_empty_def intro*: *E_from_op_states*)

qed

interpretation *deadlock*: *Reachability_Problem_Impl_Precise* **where**
F = λ(*l*, *D*). ¬ (*check_deadlock_dbm l D*) **and**
F1 = λ(*l*, *Z*). ¬ (*TA.check_deadlock l Z*) **and**
F' = *deadlocked* **and**
F_impl = λ(*l*, *M*). *do* {*r* ← *check_deadlock_impl l M*; *return* (¬ *r*)}
apply *standard*

subgoal for *a b*
apply *clarsimp*
apply (*auto dest*: *TA.check_deadlock_anti_mono simp*:
dbm_subset_correct[*symmetric*] *check_deadlock_dbm_correct'*[*symmetric*, *unfolded wf_state_def*])
done

subgoal
using
Bisimulation_Invariant.B_steps_invariant[*OF op_precise.step_z_dbm'_E_from_op_bisim_empty*]
wf_state_init states'_states
unfolding a_0*_def*
by *simp* (*subst check_deadlock_dbm_correct'*[*symmetric*], *auto elim*: *empty_steps_states'*)

subgoal for *l u Z*
unfolding *TA.check_deadlock_correct_step' deadlocked_def* **by** *auto*

subgoal
proof −
define *location_assn'* **where** *location_assn'* = *location_assn*
define *mtx_assn'* :: _ ⇒ *int DBMEntry Heap.array* ⇒ _ **where** *mtx_assn'* = *mtx_assn n*
note [*sep_heap_rules*] = *check_deadlock_impl.refine*[
to_hnr, *unfolded hn_refine_def hn_ctxt_def*,
folded location_assn'_def mtx_assn'_def, *simplified*]
show *?thesis*
unfolding *location_assn'_def*[*symmetric*] *mtx_assn'_def*[*symmetric*]
by *sepref_to_hoare* (*sep_auto simp*: *pure_def*)
qed
done

lemma *deadlock_unreachability_checker2_hnr*:
fixes *P_loc* :: *'si* ⇒ *bool*
and *L_list* :: *'si list*
and *M_list* :: (*'si* × *int DBMEntry list list*) *list*
fixes *splitter* :: *'s list* ⇒ *'s list list* **and** *splitteri* :: *'si list* ⇒ *'si list list*
assumes ⋀*li*. *P_loc li* ⟹ ∃ *l*. (*li*, *l*) ∈ *loc_rel*
and *list_all* (λ*x*. *P_loc x* ∧ *states_mem_impl x*) *L_list*
and *list_all* (λ(*l*, *xs*). *list_all* (λ*M*. *length M* = *Suc n* ∗ *Suc n*) *xs*) *M_list*
and *fst* ‘ *set M_list* = *set L_list*
assumes *full_split*: ⋀*xs*. *set xs* = (⋃ *xs* ∈ *set* (*splitter xs*). *set xs*)
and *same_split*:
⋀*L Li*.
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
shows
deadlock.unreachability_checker2 L_list M_list splitteri
⟶ (∀ *u*. (∀ *c*≤*n*. *u c* = *0*) ⟶ ¬ *deadlock* (l_0, *u*))

using *deadlock.unreachability_checker2_refine*[
OF assms(1−2) *assms*(4)[*THEN equalityD1*] *assms*(3) *full_split same_split assms*(4)
]
unfolding *deadlock_def steps'_iff*[*symmetric*] **by** *auto*

lemma *deadlock_unreachability_checker3_hnr*:
fixes *P_loc* :: *'si* ⇒ *bool*
and *L_list* :: *'si list*
and *M_list* :: (*'si* × *int DBMEntry list list*) *list*
fixes *Li_split* :: *'si list list*
assumes ⋀*li*. *P_loc li* ⟹ ∃ *l*. (*li*, *l*) ∈ *loc_rel*
and *list_all* (λ*x*. *P_loc x* ∧ *states_mem_impl x*) *L_list*
and *list_all* (λ(*l*, *xs*). *list_all* (λ*M*. *length M* = *Suc n* ∗ *Suc n*) *xs*) *M_list*
and *fst* ‘ *set M_list* = *set L_list*
assumes *full_split*: *set L_list* = (⋃ *xs* ∈ *set Li_split*. *set xs*)
shows
deadlock.certify_unreachable_pure L_list M_list Li_split
⟶ (∀ *u*. (∀ *c*≤*n*. *u c* = *0*) ⟶ ¬ *deadlock* (*l*$_0$, *u*))
using *deadlock.certify_unreachable_pure_refine*[
OF assms(1−2) *assms*(4)[*THEN equalityD1*] *assms*(3) *full_split assms*(4)
]
unfolding *deadlock_def steps'_iff*[*symmetric*] **by** *auto*

lemmas *deadlock_unreachability_checker2_def* = *deadlock.unreachability_checker2_def*

lemmas *deadlock_unreachability_checker_alt_def* = *deadlock.unreachability_checker_alt_def*

lemmas *deadlock_unreachability_checker3_def* = *deadlock.certify_unreachable_pure_def*

lemma *deadlock_unreachability_checker_hnr*:
fixes *P_loc* :: *'si* ⇒ *bool*
and *L_list* :: *'si list*
and *M_list* :: (*'si* × *int DBMEntry list list*) *list*
fixes *splitter* :: *'s list* ⇒ *'s list list* **and** *splitteri* :: *'si list* ⇒ *'si list list*
assumes ⋀*li*. *P_loc li* ⟹ ∃ *l*. (*li*, *l*) ∈ *loc_rel*
and *list_all* (λ*x*. *P_loc x* ∧ *states_mem_impl x*) *L_list*
and *fst* ‘ *set M_list* ⊆ *set L_list*
and *list_all* (λ(*l*, *xs*). *list_all* (λ*M*. *length M* = *Suc n* ∗ *Suc n*) *xs*) *M_list*
and *set* (*deadlock.L L_list*) = *dom* (*deadlock.M M_list*)
assumes *full_split*: ⋀*xs*. *set xs* = (⋃ *xs* ∈ *set* (*splitter xs*). *set xs*)
and *same_split*:
⋀*L Li*.
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
shows
(*uncurry0*
(*Reachability_Problem_Impl_Precise.unreachability_checker n trans_impl l*$_0$*i op_impl*
states_mem_impl (λ(*l*, *M*). *check_deadlock_impl l M* ⤜ (λ*r*. *return* (¬ *r*))) *L_list*
M_list splitteri),
uncurry0
(*SPEC*
(λ*r*. *r* ⟶ (∀ *u*. (∀ *c*≤*n*. *u c* = *0*) ⟶ ¬ *deadlock* (*l*$_0$, *u*)))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
using *deadlock.unreachability_checker_hnr*[*OF assms*(1−4), *OF _ full_split same_split assms*(5)]
unfolding *deadlock_def steps'_iff*[*symmetric*] **by** *simp linarith*

end

concrete_definition (**in** −) *unreachability_checker*
uses *Reachability_Problem_Impl_Precise.unreachability_checker_alt_def*

end
theory *Normalized_Zone_Semantics_Certification_Impl2*
imports
Normalized_Zone_Semantics_Certification_Impl
Unreachability_Certification
begin

context *Reachability_Problem_Impl_Precise*
begin

context
fixes *L_list* :: *'si list* **and** *P_loc* **and** *M_list* :: (*'si* × *int DBMEntry list list*) *list*
assumes *state_impl_abstract*: ⋀*li*. *P_loc li* ⟹ ∃ *l*. (*li*, *l*) ∈ *loc_rel*
assumes *P_loc*: *list_all* (λ*x*. *P_loc x* ∧ *states_mem_impl x*) *L_list*
assumes *M_list_covered*: *fst* ‘ *set M_list* ⊆ *set L_list*
and *M_dbm_len*: *list_all* (λ(*l*, *xs*). *list_all* (λ*M*. *length M* = *Suc n* ∗ *Suc n*) *xs*) *M_list*
begin

lemmas *table_axioms* = *state_impl_abstract P_loc M_list_covered M_dbm_len*

context
fixes *splitter* :: *'s list* ⇒ *'s list list* **and** *splitteri* :: *'si list* ⇒ *'si list list*
assumes *full_split*: *set xs* = (⋃ *xs* ∈ *set* (*splitter xs*). *set xs*)
and *same_split*:
⋀*L Li*.
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
begin

lemma *certify_unreachable_impl_hnr*:
assumes *set* (*L L_list*) = *dom* (*M M_list*)
shows
(*uncurry0*
(*certify_unreachable_impl F_impl P_impl amtx_copy* (*dbm_subset_impl n*) *succs_precise'_impl*
(*return* $l_0 i$) *init_dbm_impl L_list* (*M_table M_list*) *splitteri*),
uncurry0 (*SPEC* (λ*r*. *r* ⟶
(∄ *u l' u'*. (∀ *c*≤*n*. *u c* = *0*) ∧ *conv_A A* ⊢' ⟨l_0, *u*⟩ →∗ ⟨*l'*, *u'*⟩ ∧ *F'* (*l'*, *u'*)))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
by (*rule certify_unreachable_impl_hnr*)
(*rule table_axioms full_split same_split L_dom_M_eqI*[*symmetric*] *assms* | *assumption*)+

definition
unreachability_checker' ≡
let
Fi = *F_impl*;
Pi = *P_impl*;
copyi = *amtx_copy*;
Lei = *dbm_subset_impl n*;

$l_0 i$ = *Heap_Monad.return* $l_0 i$;
$s_0 i$ = *init_dbm_impl*;
succsi = *succs_precise'_impl*;
M_table = *M_table M_list*
in
certify_unreachable_impl Fi Pi copyi Lei succsi $l_0 i$ $s_0 i$ *L_list M_table splitteri*

lemma *unreachability_checker_alt_def*:
unreachability_checker' ≡
let
Fi = *F_impl*;
Pi = *P_impl*;
copyi = *amtx_copy*;
Lei = *dbm_subset_impl n*;
$l_0 i$ = *Heap_Monad.return* $l_0 i$;
$s_0 i$ = *init_dbm_impl*;
succsi = *succs_precise'_impl*
in do {
M_table ← *M_table M_list*;
certify_unreachable_impl_inner Fi Pi copyi Lei succsi $l_0 i$ $s_0 i$ *splitteri L_list M_table*
}
unfolding *unreachability_checker'_def certify_unreachable_impl_def Let_def* .

lemmas *unreachability_checker_hnr* =
certify_unreachable_impl_hnr[*folded unreachability_checker'_def*[*unfolded Let_def*]]

lemmas *unreachability_checker_alt_def'* = *unreachability_checker_alt_def*[*unfolded M_table_def*]

definition
unreachability_checker2' ≡
let
Fi = *F_impl*;
Pi = *P_impl*;
copyi = *amtx_copy*;
Lei = *dbm_subset_impl n*;
$l_0 i$ = *Heap_Monad.return* $l_0 i$;
$s_0 i$ = *init_dbm_impl*;
succsi = *succs_precise'_impl*;
M_table = *M_table M_list*
in
certify_unreachable_impl2 Fi Pi copyi Lei succsi $l_0 i$ $s_0 i$ *splitteri L_list M_table*

lemma *unreachability_checker2_refine*:
assumes *fst* ‘ *set M_list* = *set L_list unreachability_checker2 L_list M_list splitteri*
shows ∄*u l' u'*. (∀ *c*≤*n*. *u c* = *0*) ∧ *conv_A A* ⊢' ⟨l_0, *u*⟩ →* ⟨*l'*, *u'*⟩ ∧ *F'* (*l'*, *u'*)
by (*rule unreachability_checker2_refine table_axioms assms full_split same_split* | *assumption*)+

lemma *unreachability_checker2'_is_unreachability_checker2*:
unreachability_checker2' = *unreachability_checker2 L_list M_list splitteri*
unfolding *unreachability_checker2'_def*
by (*subst unreachability_checker2_def*)
(*rule table_axioms full_split same_split HOL.refl* | *assumption*)+

end

end

end

context *TA_Impl_Precise*
begin

lemma (**in** *TA_Impl*) *dbm_subset_correct*:
assumes *wf_dbm D* **and** *wf_dbm M*
shows $[curry\ (conv\_M\ D)]_{v,n} \subseteq [curry\ (conv\_M\ M)]_{v,n} \longleftrightarrow dbm\_subset\ n\ D\ M$
unfolding *dbm_subset_correct''*[*OF assms*] **using** *dbm_subset_conv_rev dbm_subset_conv* **..**

lemma *empty_steps_states'*:
$l' \in states'$ **if** *op_precise.E_from_op_empty*** $(l, D)\ (l', D')\ l \in states'$
using *that*
proof (*induction* $(l, D)\ (l', D')$ *arbitrary*: $l'\ D'$)
case *rtrancl_refl*
then show *?case*
by *simp*
next
case (*rtrancl_into_rtrancl b*)
then show *?case*
by (*cases b*) (*auto simp add*: *op_precise.E_from_op_empty_def intro*: *E_from_op_states*)
qed

interpretation *deadlock*: *Reachability_Problem_Impl_Precise* **where**
$F = \lambda(l, D).\ \neg\ (check\_deadlock\_dbm\ l\ D)$ **and**
$F1 = \lambda(l, Z).\ \neg\ (TA.check\_deadlock\ l\ Z)$ **and**
$F' = deadlocked$ **and**
$F\_impl = \lambda(l, M).\ do\ \{r \leftarrow check\_deadlock\_impl\ l\ M;\ return\ (\neg\ r)\}$
apply *standard*

subgoal for *a b*
apply *clarsimp*
apply (*auto dest*: *TA.check_deadlock_anti_mono simp*:
dbm_subset_correct[*symmetric*] *check_deadlock_dbm_correct'*[*symmetric*, *unfolded wf_state_def*])
done

subgoal
using
Bisimulation_Invariant.B_steps_invariant[*OF op_precise.step_z_dbm'_E_from_op_bisim_empty*]
wf_state_init states'_states
unfolding a_0*_def*
by *simp* (*subst check_deadlock_dbm_correct'*[*symmetric*], *auto elim*: *empty_steps_states'*)

subgoal for *l u Z*
unfolding *TA.check_deadlock_correct_step' deadlocked_def* **by** *auto*

subgoal
proof −

```
    define location_assn' where location_assn' = location_assn
    define mtx_assn' :: _ ⇒ int DBMEntry Heap.array ⇒ _ where mtx_assn' = mtx_assn n
    note [sep_heap_rules] = check_deadlock_impl.refine[
        to_hnr, unfolded hn_refine_def hn_ctxt_def,
        folded location_assn'_def mtx_assn'_def, simplified]
    show ?thesis
      unfolding location_assn'_def[symmetric] mtx_assn'_def[symmetric]
      by sepref_to_hoare (sep_auto simp: pure_def)
  qed
  done

lemma deadlock_unreachability_checker2_hnr:
  fixes P_loc :: 'si ⇒ bool
    and L_list :: 'si list
    and M_list :: ('si × int DBMEntry list list) list
  fixes splitter :: 's list ⇒ 's list list and splitteri :: 'si list ⇒ 'si list list
  assumes ⋀li. P_loc li ⟹ ∃ l. (li, l) ∈ loc_rel
    and list_all (λx. P_loc x ∧ states_mem_impl x) L_list
    and list_all (λ(l, xs). list_all (λM. length M = Suc n ∗ Suc n) xs) M_list
    and fst ‘ set M_list = set L_list
  assumes full_split: ⋀xs. set xs = (⋃ xs ∈ set (splitter xs). set xs)
    and same_split:
    ⋀L Li.
      list_assn (list_assn location_assn) (splitter L) (splitteri Li) = list_assn location_assn L Li
  shows
    deadlock.unreachability_checker2 L_list M_list splitteri
    ⟶ (∀ u. (∀ c≤n. u c = 0) ⟶ ¬ deadlock (l0, u))
  using deadlock.unreachability_checker2_refine[
      OF assms(1−2) assms(4)[THEN equalityD1] assms(3) full_split same_split assms(4)
      ]
  unfolding deadlock_def steps'_iff[symmetric] by auto

lemma deadlock_unreachability_checker3_hnr:
  fixes P_loc :: 'si ⇒ bool
    and L_list :: 'si list
    and M_list :: ('si × int DBMEntry list list) list
  fixes Li_split :: 'si list list
  assumes ⋀li. P_loc li ⟹ ∃ l. (li, l) ∈ loc_rel
    and list_all (λx. P_loc x ∧ states_mem_impl x) L_list
    and list_all (λ(l, xs). list_all (λM. length M = Suc n ∗ Suc n) xs) M_list
    and fst ‘ set M_list = set L_list
  assumes full_split: set L_list = (⋃ xs ∈ set Li_split. set xs)
  shows
    deadlock.certify_unreachable_pure L_list M_list Li_split
    ⟶ (∀ u. (∀ c≤n. u c = 0) ⟶ ¬ deadlock (l0, u))
  using deadlock.certify_unreachable_pure_refine[
      OF assms(1−2) assms(4)[THEN equalityD1] assms(3) full_split assms(4)
      ]
  unfolding deadlock_def steps'_iff[symmetric] by auto

lemmas deadlock_unreachability_checker2_def = deadlock.unreachability_checker2_def

lemmas deadlock_unreachability_checker_alt_def = deadlock.unreachability_checker_alt_def
```

lemmas *deadlock_unreachability_checker3_def = deadlock.certify_unreachable_pure_def*

lemma *deadlock_unreachability_checker_hnr*:
fixes *P_loc* :: *'si ⇒ bool*
and *L_list* :: *'si list*
and *M_list* :: *('si × int DBMEntry list list) list*
fixes *splitter* :: *'s list ⇒ 's list list* **and** *splitteri* :: *'si list ⇒ 'si list list*
assumes ⋀*li*. *P_loc li ⟹* ∃ *l*. (*li*, *l*) ∈ *loc_rel*
and *list_all* (λ*x*. *P_loc x* ∧ *states_mem_impl x*) *L_list*
and *fst ' set M_list* ⊆ *set L_list*
and *list_all* (λ(*l*, *xs*). *list_all* (λ*M*. *length M = Suc n* ∗ *Suc n*) *xs*) *M_list*
and *set* (*deadlock.L L_list*) = *dom* (*deadlock.M M_list*)
assumes *full_split*: ⋀*xs*. *set xs* = (⋃ *xs* ∈ *set* (*splitter xs*). *set xs*)
and *same_split*:
⋀*L Li*.
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
shows
(*uncurry0*
(*Reachability_Problem_Impl_Precise.unreachability_checker' n trans_impl* l_0*i op_impl*
states_mem_impl (λ(*l*, *M*). *check_deadlock_impl l M* ⤜ (λ*r*. *return* (¬ *r*))) *L_list*
M_list splitteri),
uncurry0
(*SPEC*
(λ*r*. *r* ⟶ (∀ *u*. (∀ *c*≤*n*. *u c = 0*) ⟶ ¬ *deadlock* (l_0, *u*)))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
using *deadlock.unreachability_checker_hnr*[*OF assms*(1−4), *OF _ full_split same_split assms*(5)]
unfolding *deadlock_def steps'_iff*[*symmetric*] **by** *simp linarith*

end

concrete_definition (**in** −) *unreachability_checker*
uses *Reachability_Problem_Impl_Precise.unreachability_checker_alt_def*

end

10 Extracting Certificates From Munta

theory *Extract_Certificate*
imports
Worklist_Algorithms.Unified_PW_Impl
Worklist_Algorithms.Next_Key
Difference_Bound_Matrices.DBM_Operations_Impl_Refine
Worklist_Algorithms.Leadsto_Impl
begin

10.1 Turning a map into a list

definition *list_of_map* **where**
list_of_map m ≡ *do*
{
(*xs*, *m*) ← *WHILEIT*
(λ (*xs*, *m'*). *finite* (*dom m'*) ∧ *m = m'* ++ *map_of xs* ∧ *dom m'* ∩ *dom* (*map_of xs*) =

```
{})
          (λ (_, m). Map.empty ≠ m)
          (λ (xs, m). do
            {
              k ← next_key m;
              let (x, m) = op_map_extract k m;
              ASSERT (x ≠ None);
              RETURN ((k, the x) # xs, m)
            }
          )
          ([], m);
      RETURN xs
    }


context
begin

private definition
  ran_of_map_var = (inv_image (measure (card o dom)) (λ (a, b). b))

private lemma wf_ran_of_map_var:
  wf ran_of_map_var
  unfolding ran_of_map_var_def by auto

private lemma insert_restrict_ran:
  insert v (ran (m |` (- {k}))) = ran m  if m k = Some v
  using that unfolding ran_def restrict_map_def by force

private lemma 1:
  (m |` (- {x}))(x ↦ y) = m if m x = Some y
  using that unfolding restrict_map_def by auto

private lemma 2:
  (m1 ++ m2)(x ↦ y) = m1(x ↦ y) ++ m2 if x ∉ dom m2
  using that by auto

lemma list_of_map_correct[refine]:
  list_of_map m ≤ SPEC (λ r. map_of r = m) if finite (dom m)
  using that unfolding list_of_map_def next_key_def
  apply (refine_vcg wf_ran_of_map_var)
        apply (clarsimp; fail)+
  subgoal for s xs m' x v xs' xs1 xs'1
    unfolding dom_def apply (clarsimp simp: map_upd_eq_restrict)
    apply (subst 2, solves auto)
    apply (subst 1)
     apply auto
    done
  unfolding ran_of_map_var_def by (fastforce intro: card_Diff1_less split: if_split_asm)+

end — End of private context for auxiliary facts and definitions

context
  fixes K :: _ ⇒ _ :: {hashable, heap} ⇒ assn
```

assumes *is_pure_K*[*safe_constraint_rules*]: *is_pure K*
and *left_unique_K*[*safe_constraint_rules*]: *IS_LEFT_UNIQUE* (*the_pure K*)
and *right_unique_K*[*safe_constraint_rules*]: *IS_RIGHT_UNIQUE* (*the_pure K*)
notes [*sepref_fr_rules*] = *hm_it_next_key_next_key''*[*OF is_pure_K*]
begin

sepref_definition *list_of_map_impl* **is**
list_of_map :: (*hm.hms_assn' K A*)^d →$_a$ (*list_assn* (*K* ×$_a$ *A*))
unfolding *list_of_map_def hm.hms_assn'_id_hms_assn*[*symmetric*]
unfolding *op_map_is_empty_def*[*symmetric*]
unfolding *hm.hms_fold_custom_empty HOL_list.fold_custom_empty*
by *sepref*

end

lemmas *list_of_map_impl_code*[*code*] =
list_of_map_impl_def[*of pure Id*, *simplified*, *OF Sepref_Constraints.safe_constraint_rules*(*41*)]

context
notes [*sepref_fr_rules*] = *hm_it_next_key_next_key'*[*folded hm.hms_assn'_id_hms_assn*]
begin

sepref_definition *list_of_map_impl'* **is**
list_of_map :: (*hm.hms_assn A*)^d →$_a$ (*list_assn* (*id_assn* ×$_a$ *A*))
unfolding *list_of_map_def hm.hms_assn'_id_hms_assn*[*symmetric*]
unfolding *op_map_is_empty_def*[*symmetric*]
unfolding *hm.hms_fold_custom_empty HOL_list.fold_custom_empty*
by *sepref*

end

context *Worklist_Map2_Impl*
begin

definition *extract_certificate* :: _ *nres* **where**
extract_certificate = *do* {
(_, *passed*) ← *pw_algo_map2*;
list_of_map passed
}

context
begin

private definition
pw_algo_map2_copy = *pw_algo_map2*

sepref_register *pw_algo_map2_copy*

lemma *pw_algo_map2_copy_fold*:
PR_CONST pw_algo_map2_copy = *pw_algo_map2*
unfolding *pw_algo_map2_copy_def* **by** *simp*

lemmas [*sepref_fr_rules*] =
pw_algo_map2_impl.refine_raw[*folded pw_algo_map2_copy_fold*]

list_of_map_impl.refine[*OF pure_K left_unique_K right_unique_K*]

sepref_thm *extract_certificate_impl* **is**
uncurry0 extract_certificate :: *unit_assn*^{k} $\rightarrow_a$ (*list_assn* (*K* $\times_a$ *lso_assn A*))
unfolding *extract_certificate_def pw_algo_map2_copy_def*[*symmetric*] **by** *sepref*

end

end

concrete_definition *extract_certificate_impl*
uses *Worklist_Map2_Impl.extract_certificate_impl.refine_raw*

end
theory *Simple_Network_Language_Certificate*
imports *Extract_Certificate Munta_Model_Checker.Simple_Network_Language_Model_Checking*
begin

context *Simple_Network_Impl_nat_ceiling_start_state*
begin

definition
state_space ≡
let
succsi = *impl.succs_impl*;
$a_0 i$ = *impl.*a_0*_impl*;
Fi = *impl.F_impl*;
Lei = *impl.subsumes_impl*;
emptyi = *impl.emptiness_check_impl*;
keyi = *return o fst*;
copyi = *impl.state_copy_impl*;
trace = *impl.tracei*
in extract_certificate_impl succsi $a_0 i$ *Fi Lei emptyi keyi copyi trace*

schematic_goal *state_space_alt_def*:
state_space ≡ *?impl*
unfolding *state_space_def*
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.*a_0*_impl_def*
unfolding *impl.F_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

end

concrete_definition *state_space*
uses *Simple_Network_Impl_nat_ceiling_start_state.state_space_alt_def*

end

11 A Verified Certificate Checker for the Simple Networks Lanuage

theory *Simple_Network_Language_Certificate_Checking*
imports
Extract_Certificate
Normalized_Zone_Semantics_Certification_Impl
Munta_Model_Checker.Simple_Network_Language_Export_Code
Munta_Base.Trace_Timing
begin

unbundle *no_library_syntax*
notation *fun_rel_syn* (**infixr** $\rightarrow$ *60*)
no_notation *Omega_Words_Fun.build* (**infixr** ‹##› *65*)
no_notation *Assertions.models* (**infix** $\models$ *50*)

Misc **lemma** *list_set_rel_singleton_iff*:
$([a], \{b\}) \in \langle R \rangle list\_set\_rel \longleftrightarrow (a, b) \in R$
unfolding *list_set_rel_def relcomp.simps* **by** (*auto simp*: *list_all2_Cons1 list_rel_def*)

definition (**in** *Graph_Defs*)
alw_ev φ *x* $\equiv \forall$ *xs*. *run* (*x* ## *xs*) $\longrightarrow$ *alw* (*ev* (*holds* φ)) (*x* ## *xs*)

lemma (**in** *Graph_Defs*) *alw_ev_to_ctl_star*:
alw_ev φ *x* $\longleftrightarrow$ *models_state* (*All* (*G* (*F* (*State* (*PropS* φ))))) *x*
unfolding *alw_ev_def* **by** (*simp add*: *models_state.simps*[*abs_def*])

context *Simple_Network_Rename_Formula*
begin

lemma *buechi_models_state_compatible*:
assumes Φ = *formula.EX* φ
shows
Graph_Defs.models_state
(λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *rename.renum.sem* $\vdash$ $\langle$*L*, *s*, *u*$\rangle$ $\rightarrow$ $\langle$*L'*, *s'*, *u'*$\rangle$)
(*map_state_formula* ($\lambda\varphi$ (*L*, *s*, _).
check_sexp (*map_sexp* (λ*p*. *renum_states p*) *renum_vars id* φ) *L* (*the o s*))
(*All* (*G* (*F* (*State* (*PropS* φ)))))))
$a_0{}'$
$\longleftrightarrow$ *Graph_Defs.models_state*
(λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *sem* $\vdash$ $\langle$*L*, *s*, *u*$\rangle$ $\rightarrow$ $\langle$*L'*, *s'*, *u'*$\rangle$)
(*map_state_formula* ($\lambda\varphi$ (*L*, *s*, _). *check_sexp* φ *L* (*the o s*)) (*All* (*G* (*F* (*State* (*PropS* φ)))))))
a_0 (**is** *Graph_Defs.models_state* _ *?*φ _ $\longleftrightarrow$ _)
proof −
have *?*φ = *map_state_formula*
(λ*P* (*L*, *s*, _).
check_sexp
(*map_sexp* (λ*p*. *extend_bij* (*renum_states p*) *loc_set*) (*extend_bij renum_vars var_set*) *id P*)

```
      L (the ∘ s))
   (All (G (F (State (PropS φ)))))
    using assms formula_dom by (auto simp: sexp_eq)
  show ?thesis
  unfolding a₀_def a₀'_def ‹?φ = _› using assms formula_dom
  by − (rule sym, rule models_state_compatible, auto)
qed

lemma buechi_compatible:
  assumes Φ = formula.EX φ
  shows
  Graph_Defs.alw_ev
    (λ (L, s, u) (L', s', u'). rename.renum.sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
    (λ (L, s, _). check_sexp (map_sexp (λp. renum_states p) renum_vars id φ) L (the o s)) a₀'
⟷ Graph_Defs.alw_ev
    (λ (L, s, u) (L', s', u'). sem ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
    (λ (L, s, _). check_sexp φ L (the o s)) a₀
  using buechi_models_state_compatible[OF assms] unfolding Graph_Defs.alw_ev_to_ctl_star
by simp

lemmas buechi_compatible' =
   buechi_compatible[unfolded rename_N_eq_sem, folded N_eq_sem, unfolded a₀_def a₀'_def
Φ'_def]

end

lemma stream_all2_flip:
  assumes stream_all2 R xs ys
  shows stream_all2 (λy x. R x y) ys xs
  using assms by (coinduction arbitrary: xs ys) auto

lemma (in Bisimulation_Invariant) alw_ev_iff:
  fixes φ :: 'a ⇒ bool and ψ :: 'b ⇒ bool
  assumes compatible: ⋀a b. A_B.equiv' a b ⟹ φ a ⟷ ψ b
    and that: A_B.equiv' a b
  shows A.alw_ev φ a ⟷ B.alw_ev ψ b
  unfolding Graph_Defs.alw_ev_def
  apply safe
  subgoal
    using that
    by − (
      drule B_A.simulation_run,
      auto simp: compatible intro: equiv'_rotate_1 dest!: equiv'_rotate_2
      elim!: alw_ev_lockstep stream_all2_flip)
  subgoal for xs
    using that
    by − (
        drule A_B.simulation_run,
        auto simp: compatible intro: equiv'_rotate_2 elim!: alw_ev_lockstep stream_all2_flip)
  done

Splitters  context
  fixes f :: 'a ⇒ nat and width :: nat
```

begin

fun *split_size* :: *nat* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list list* **where**
  split_size _ *acc* [] = [*acc*]
| *split_size n acc* (*x* # *xs*) =
   (*let k* = *f x in if n* < *width then split_size* (*n* + *k*) (*x* # *acc*) *xs else acc* # *split_size k* [*x*] *xs*)

lemma *split_size_full_split*:
  (⋃ *x* ∈ *set* (*split_size n acc xs*). *set x*) = *set xs* ∪ *set acc*
  by (*induction n acc xs rule*: *split_size.induct*) *auto*

end

definition *split_eq_width* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
  split_eq_width n ≡ *split_size* (λ_. *1* :: *nat*) *n 0* []

lemma *list_all2_split_size_1*:
  assumes *list_all2 R xs ys list_all2 R acc acc'*
  shows *list_all2* (*list_all2 R*)
    (*split_size* (λ_. *1* :: *nat*) *k n acc xs*) (*split_size* (λ_. *1* :: *nat*) *k n acc' ys*)
  using *assms* **by** (*induction arbitrary*: *acc acc' n rule*: *list_all2_induct*) *auto*

fun *zip2* **where**
  zip2 [] [] = []
| *zip2* [] *xs* = []
| *zip2 ys* [] = []
| *zip2* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip2 xs ys*

lemma *length_hd_split_size_mono*:
  length (*hd* (*split_size f k n acc xs*)) ≥ *length acc*
  apply (*induction xs arbitrary*: *acc n*)
   apply (*solves auto*)
  apply *clarsimp*
  apply (*rule order.trans*[*rotated*])
   apply *rprems*
  apply *simp*
  done

lemma *split_size_non_empty*[*simp*]:
  split_size f k n acc xs = [] ⟷ *False*
  by (*induction xs arbitrary*: *acc n*) *auto*

lemma *list_all2_split_size_2*:
  assumes *length acc* = *length acc'*
  shows
  list_all2 (*list_all2 R*)
    (*split_size* (λ_. *1* :: *nat*) *k n acc xs*) (*split_size* (λ_. *1* :: *nat*) *k n acc' ys*)
⟷ *list_all2 R xs ys* ∧ *list_all2 R acc acc'*
  using *assms*
proof (*induction xs ys arbitrary*: *acc acc' n rule*: *zip2.induct*)
  case (*2 y ys*)
  { assume *A*: *list_all2* (*list_all2 R*) [*acc*] (*split_size* (λ_. *Suc 0*) *k* (*Suc n*) (*y* # *acc'*) *ys*)
    then obtain *x* **where** *split_size* (λ_. *Suc 0*) *k* (*Suc n*) (*y* # *acc'*) *ys* = [*x*]
      by (*cases split_size* (λ_. *Suc 0*) *k* (*Suc n*) (*y* # *acc'*) *ys*) *auto*

```
    with length_hd_split_size_mono[of y # acc' (λ_. Suc 0) k Suc n ys] have
      length x ≥ length (y # acc')
      by auto
    with A ‹_ = [x]› ‹length acc = length acc'› have False
      by (auto dest: list_all2_lengthD)
  }
  with 2 show ?case
    by clarsimp
next
  case (3 y ys)
  { assume A: list_all2 (list_all2 R) (split_size (λ_. Suc 0) k (Suc n) (y # acc) ys) [acc']
    then obtain x where split_size (λ_. Suc 0) k (Suc n) (y # acc) ys = [x]
      by (cases split_size (λ_. Suc 0) k (Suc n) (y # acc) ys) auto
    with length_hd_split_size_mono[of y # acc (λ_. Suc 0) k Suc n ys] have
      length x ≥ length (y # acc)
      by auto
    with A ‹_ = [x]› ‹length acc = length acc'› have False
      by (auto dest: list_all2_lengthD)
  }
  with 3 show ?case
    by clarsimp
qed auto

lemma list_all2_split_eq_width:
  shows list_all2 R xs ys ⟷ list_all2 (list_all2 R) (split_eq_width k xs) (split_eq_width k ys)
  unfolding split_eq_width_def by (subst list_all2_split_size_2; simp)

lemma length_split_size_1:
  sum_list (map length (split_size (λ_. 1 :: nat) k n acc xs)) = length xs + length acc
  by (induction xs arbitrary: acc n) auto

lemma length_sum_list:
  sum_list (map length (split_eq_width k xs)) = length xs
  unfolding split_eq_width_def by (subst length_split_size_1; simp)

definition split_k :: nat ⇒ 'a list ⇒ 'a list list where
  split_k k xs ≡ let
    width = length xs div k;
    width = (if length xs mod k = 0 then width else width + 1)
  in split_eq_width width xs

lemma length_hd_split_size:
  length (hd (split_size (λ_. 1 :: nat) k n acc xs))
  = (if n + length xs < k then n + length xs else max k n) if length acc = n
  using that by (induction xs arbitrary: n acc) auto

lemma length_hd_split_eq_width:
  length (hd (split_eq_width width xs)) = (if length xs < width then length xs else width)
  unfolding split_eq_width_def by (subst length_hd_split_size; simp)

lemma list_all2_split_k:
  list_all2 R xs ys ⟷ list_all2 (list_all2 R) (split_k k xs) (split_k k ys)
proof (cases length xs = length ys)
  case True
```

```
  show ?thesis
    unfolding split_k_def Let_def True by (rule list_all2_split_eq_width)
next
  case False
  { assume list_all2 (list_all2 R) (split_k k xs) (split_k k ys)
    then have list_all2 R (concat (split_k k xs)) (concat (split_k k ys))
      by (meson concat_transfer rel_funD)
    then have length (concat (split_k k xs)) = length (concat (split_k k ys))
      by (rule list_all2_lengthD)
    then have length xs = length ys
      unfolding split_k_def by (simp add: length_concat length_sum_list)
  }
  with False show ?thesis
    by (auto dest: list_all2_lengthD)
qed

definition split_k' :: nat ⇒ ('a × 'b list) list ⇒ 'a list list where
  split_k' k xs ≡ let
    width = sum_list (map (length o snd) xs) div k;
    width = (if length xs mod k = 0 then width else width + 1)
  in map (map fst) (split_size (length o snd) width 0 [] xs)

lemma split_eq_width_full_split:
  set xs = (⋃x ∈ set (split_eq_width n xs). set x)
  unfolding split_eq_width_def by (auto simp add: split_size_full_split)

lemma split_k_full_split:
  set xs = (⋃x ∈ set (split_k n xs). set x)
  unfolding split_k_def by (simp add: split_eq_width_full_split)

lemma split_k'_full_split:
  fst ` set xs = (⋃x ∈ set (split_k' n xs). set x)
  unfolding split_k'_def by (simp add: split_size_full_split image_UN[symmetric])


lemma (in Graph_Defs) Ex_ev_reaches:
  ∃ y. x →* y ∧ φ y if Ex_ev φ x
  using that unfolding Ex_ev_def
proof safe
  fix xs assume prems: run (x ## xs) ev (holds φ) (x ## xs)
  show ∃y. x →* y ∧ φ y
  proof (cases φ x)
    case True
    then show ?thesis
      by auto
  next
    case False
    with prems obtain y ys zs where
      φ y xs = ys @− y ## zs y ∉ set ys
      unfolding ev_holds_sset by (auto elim!:split_stream_first')
    with prems have steps (x # ys @ [y])
      by (auto intro: run_decomp[THEN conjunct1])
    with ‹φ y› show ?thesis
      including graph_automation by (auto 4 3)
```

qed
qed

More debugging auxiliaries

concrete_definition (**in** −) *M_table*
uses *Reachability_Problem_Impl_Precise.M_table_def*

definition
check_nonneg n M ≡ *imp_for 0* (*n* + *1*) *Heap_Monad.return*
(λ*xc σ*. *mtx_get* (*Suc n*) *M* (*0*, *xc*) ⤜ (λ*x'e*. *Heap_Monad.return* (*x'e* ≤ *Le 0*))) *True*

definition
check_diag_nonpos n M ≡ *imp_for 0* (*n* + *1*) *Heap_Monad.return*
(λ*xc σ*. *mtx_get* (*Suc n*) *M* (*xc*, *xc*) ⤜ (λ*x'd*. *Heap_Monad.return* (*x'd* ≤ *Le 0*))) *True*

Complete DBM printing

context
fixes *n* :: *nat*
fixes *show_clock* :: *nat* ⇒ *string*
and *show_num* :: *'a* :: {*linordered_ab_group_add*,*heap*} ⇒ *string*
notes [*id_rules*] = *itypeI*[*of n TYPE* (*nat*)]
and [*sepref_import_param*] = *IdI*[*of n*]
begin

definition
make_string' e i j ≡
let
i = (*if i* > *0 then show_clock i else ''0''*);
j = (*if j* > *0 then show_clock j else ''0''*)
in
case e of
DBMEntry.Le a ⇒ *i* @ *'' − ''* @ *j* @ *'' <= ''* @ *show_num a*
| *DBMEntry.Lt a* ⇒ *i* @ *'' − ''* @ *j* @ *'' < ''* @ *show_num a*
| *_* ⇒ *i* @ *'' − ''* @ *j* @ *'' < inf''*

definition
dbm_list_to_string' xs ≡
(*concat o intersperse ''*, *'' o rev o snd o snd*) $ *fold* (λ*e* (*i*, *j*, *acc*).
let
s = *make_string' e i j*;
j = (*j* + *1*) *mod* (*n* + *1*);
i = (*if j* = *0 then i* + *1 else i*)
in (*i*, *j*, *s* # *acc*)
) *xs* (*0*, *0*, [])

lemma [*sepref_import_param*]:
(*dbm_list_to_string'*, *PR_CONST dbm_list_to_string'*) ∈ ⟨*Id*⟩*list_rel* → ⟨*Id*⟩*list_rel*
by *simp*

definition *show_dbm'* **where**
show_dbm' M ≡ *PR_CONST dbm_list_to_string'* (*dbm_to_list n M*)

sepref_register *PR_CONST dbm_list_to_string'*

lemmas [*sepref_fr_rules*] = *dbm_to_list_impl.refine*

sepref_definition *show_dbm_impl_all* **is**
Refine_Basic.RETURN o show_dbm' :: $(mtx\_assn\ n)^k \rightarrow_a list\_assn\ id\_assn$
unfolding *show_dbm'_def* **by** *sepref*

end

definition
abstr_repair_impl m ≡
λ*ai*. *imp_nfoldli ai* (λσ. *Heap_Monad.return True*)
(λ*ai bi*. *abstra_upd_impl m ai bi* ⤜ (λ*x'*. *repair_pair_impl m x' 0* (*constraint_clk ai*)))

definition *E_op_impl* ::
nat ⇒ (*nat*, *int*) *acconstraint list* ⇒ *nat list* ⇒ _
where
E_op_impl m l_inv r g l'_inv M ≡
do {
M1 ← *up_canonical_upd_impl m M m*;
M2 ← *abstr_repair_impl m l_inv M1*;
is_empty1 ← *check_diag_impl m M2*;
M3 ← (*if is_empty1 then mtx_set* (*Suc m*) *M2* (*0*, *0*) (*Lt 0*) *else abstr_repair_impl m g M2*);
is_empty3 ← *check_diag_impl m M3*;
if is_empty3 then
mtx_set (*Suc m*) *M3* (*0*, *0*) (*Lt 0*)
else do {
M4 ←
imp_nfoldli r (λσ. *Heap_Monad.return True*) (λ*xc* σ. *reset_canonical_upd_impl m* σ *m xc 0*) *M3*;
abstr_repair_impl m l'_inv M4
}
}

Full Monomorphization of *E_op_impl* **definition** *min_int* :: *int* ⇒ *int* ⇒ *int* **where**
min_int x y ≡ *if x* ≤ *y then x else y*

named_theorems *int_folds*

lemma *min_int_fold*[*int_folds*]:
min = *min_int*
unfolding *min_int_def min_def* **..**

fun *min_int_entry* :: *int DBMEntry* ⇒ *int DBMEntry* ⇒ *int DBMEntry* **where**
min_int_entry (*Lt x*) (*Lt y*) = (*if x* ≤ *y then Lt x else Lt y*)
| *min_int_entry* (*Lt x*) (*Le y*) = (*if x* ≤ *y then Lt x else Le y*)
| *min_int_entry* (*Le x*) (*Lt y*) = (*if x* < *y then Le x else Lt y*)
| *min_int_entry* (*Le x*) (*Le y*) = (*if x* ≤ *y then Le x else Le y*)
| *min_int_entry* ∞ *x* = *x*
| *min_int_entry x* ∞ = *x*

export_code *min_int_entry* **in** *SML*

lemma *min_int_entry*[*int_folds*]:
min = *min_int_entry*
apply (*intro ext*)
subgoal for *a b*
by (*cases a*; *cases b*; *simp add*: *dbm_entry_simps*)
done

fun *dbm_le_int* :: *int DBMEntry* ⇒ *int DBMEntry* ⇒ *bool* **where**
dbm_le_int (*Lt a*) (*Lt b*) ⟷ *a* ≤ *b*
| *dbm_le_int* (*Lt a*) (*Le b*) ⟷ *a* ≤ *b*
| *dbm_le_int* (*Le a*) (*Lt b*) ⟷ *a* < *b*
| *dbm_le_int* (*Le a*) (*Le b*) ⟷ *a* ≤ *b*
| *dbm_le_int* _ ∞ ⟷ *True*
| *dbm_le_int* _ _ ⟷ *False*

lemma *dbm_le_dbm_le_int*[*int_folds*]:
dbm_le = *dbm_le_int*
apply (*intro ext*)
subgoal for *a b*
by (*cases a*; *cases b*; *auto simp*: *dbm_le_def*)
done

fun *dbm_lt_int* :: *int DBMEntry* ⇒ *int DBMEntry* ⇒ *bool*
where
dbm_lt_int (*Le a*) (*Le b*) ⟷ *a* < *b* |
dbm_lt_int (*Le a*) (*Lt b*) ⟷ *a* < *b* |
dbm_lt_int (*Lt a*) (*Le b*) ⟷ *a* ≤ *b* |
dbm_lt_int (*Lt a*) (*Lt b*) ⟷ *a* < *b* |
dbm_lt_int ∞ _ = *False* |
dbm_lt_int _ ∞ = *True*

lemma *dbm_lt_dbm_lt_int*[*int_folds*]:
dbm_lt = *dbm_lt_int*
apply (*intro ext*)
subgoal for *a b*
by (*cases a*; *cases b*; *auto*)
done

definition [*symmetric*, *int_folds*]:
dbm_lt_0 x ≡ *x* < (*Le* (*0* :: *int*))

lemmas [*int_folds*] = *dbm_lt_0_def*[*unfolded DBM.less*]

lemma *dbm_lt_0_code_simps* [*code*]:
dbm_lt_0 (*Le x*) ⟷ *x* < *0*
dbm_lt_0 (*Lt x*) ⟷ *x* ≤ *0*
dbm_lt_0 ∞ = *False*
unfolding *dbm_lt_0_def*[*symmetric*] *DBM.less dbm_lt_dbm_lt_int* **by** *simp+*

definition *abstra_upd_impl_int*
:: *nat* ⇒ (*nat*, *int*) *acconstraint* ⇒ *int DBMEntry Heap.array* ⇒ *int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:

abstra_upd_impl_int = abstra_upd_impl

schematic_goal *abstra_upd_impl_int_code*[*code*]:
abstra_upd_impl_int ≡ ?i
unfolding *abstra_upd_impl_int_def*[*symmetric*] *abstra_upd_impl_def*
unfolding *int_folds*
.

definition *fw_upd_impl_int*
:: *nat ⇒ int DBMEntry Heap.array ⇒ nat ⇒ nat ⇒ nat ⇒ int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:
fw_upd_impl_int = fw_upd_impl

lemmas [*int_folds*] = *DBM.add dbm_add_int*

schematic_goal *fw_upd_impl_int_code* [*code*]:
fw_upd_impl_int ≡ ?i
unfolding *fw_upd_impl_int_def*[*symmetric*] *fw_upd_impl_def*
unfolding *int_folds*
.

definition *fwi_impl_int*
:: *nat ⇒ int DBMEntry Heap.array ⇒ nat ⇒ int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:
fwi_impl_int = fwi_impl

schematic_goal *fwi_impl_int_code* [*code*]:
fwi_impl_int ≡ ?i
unfolding *fwi_impl_int_def*[*symmetric*] *fwi_impl_def* **unfolding** *int_folds* **.**

definition *fw_impl_int*
:: *nat ⇒ int DBMEntry Heap.array ⇒ int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:
fw_impl_int = fw_impl

schematic_goal *fw_impl_int_code* [*code*]:
fw_impl_int ≡ ?i
unfolding *fw_impl_int_def*[*symmetric*] *fw_impl_def* **unfolding** *int_folds* **.**

definition *repair_pair_impl_int*
:: *nat ⇒ int DBMEntry Heap.array ⇒ nat ⇒ nat ⇒ int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:
repair_pair_impl_int ≡ repair_pair_impl

schematic_goal *repair_pair_impl_int_code*[*code*]:
repair_pair_impl_int ≡ ?i
unfolding *repair_pair_impl_int_def*[*symmetric*] *repair_pair_impl_def*
unfolding *int_folds*
.

definition *abstr_repair_impl_int*
:: *nat ⇒* (*nat*, *int*) *acconstraint list ⇒ int DBMEntry Heap.array ⇒ int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:

abstr_repair_impl_int = *abstr_repair_impl*

schematic_goal *abstr_repair_impl_int_code*[*code*]:
abstr_repair_impl_int ≡ *?i*
unfolding *abstr_repair_impl_int_def*[*symmetric*] *abstr_repair_impl_def*
unfolding *int_folds*
.

definition *check_diag_impl_int*
:: *nat* ⇒ *int DBMEntry Heap.array* ⇒ *bool Heap*
where [*symmetric*, *int_folds*]:
check_diag_impl_int = *check_diag_impl*

schematic_goal *check_diag_impl_int_code*[*code*]:
check_diag_impl_int ≡ *?i*
unfolding *check_diag_impl_int_def*[*symmetric*] *check_diag_impl_def*
unfolding *int_folds*
.

definition *check_diag_impl′_int*
:: *nat* ⇒ *nat* ⇒ *int DBMEntry Heap.array* ⇒ *bool Heap*
where [*symmetric*, *int_folds*]:
check_diag_impl′_int = *check_diag_impl′*

schematic_goal *check_diag_impl′_int_code*[*code*]:
check_diag_impl′_int ≡ *?i*
unfolding *check_diag_impl′_int_def*[*symmetric*] *check_diag_impl′_def*
unfolding *int_folds*
.

definition *reset_canonical_upd_impl_int*
:: *nat* ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
reset_canonical_upd_impl_int = *reset_canonical_upd_impl*

schematic_goal *reset_canonical_upd_impl_int_code*[*code*]:
reset_canonical_upd_impl_int ≡ *?i*
unfolding *reset_canonical_upd_impl_int_def*[*symmetric*] *reset_canonical_upd_impl_def*
unfolding *int_folds*
.

definition *up_canonical_upd_impl_int*
:: *nat* ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
up_canonical_upd_impl_int = *up_canonical_upd_impl*

schematic_goal *up_canonical_upd_impl_int_code*[*code*]:
up_canonical_upd_impl_int ≡ *?i*
unfolding *up_canonical_upd_impl_int_def*[*symmetric*] *up_canonical_upd_impl_def*
.

schematic_goal *E_op_impl_code*[*code*]:
E_op_impl ≡ *?i*
unfolding *E_op_impl_def*

unfolding *int_folds*
.

definition *free_impl_int* :: *nat* ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
free_impl_int = *free_impl*

schematic_goal *free_impl_int_code*[*code*]:
free_impl_int ≡ *?i*
unfolding *free_impl_int_def*[*symmetric*] *free_impl_def*
unfolding *int_folds*
.

definition *down_impl_int* :: *nat* ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
down_impl_int = *down_impl*

schematic_goal *down_impl_int_code*[*code*]:
down_impl_int ≡ *?i*
unfolding *down_impl_int_def*[*symmetric*] *down_impl_def*
unfolding *int_folds*
.

fun *neg_dbm_entry_int* **where**
neg_dbm_entry_int (*Le* (*a* :: *int*)) = *Lt* (−*a*) |
neg_dbm_entry_int (*Lt a*) = *Le* (−*a*) |
neg_dbm_entry_int DBMEntry.INF = *DBMEntry.INF*

lemma *neg_dbm_entry_int_fold* [*int_folds*]:
neg_dbm_entry = *neg_dbm_entry_int*
apply (*intro ext*)
subgoal for *x*
by (*cases x*; *auto*)
done

schematic_goal *and_entry_impl_code* [*code*]:
and_entry_impl ≡ *?impl*
unfolding *and_entry_impl_def* **unfolding** *int_folds* .

schematic_goal *and_entry_repair_impl_code* [*code*]:
and_entry_repair_impl ≡ *?impl*
unfolding *and_entry_repair_impl_def* **unfolding** *int_folds* .

definition *upd_entry_impl_int* :: _ ⇒ _ ⇒ _ ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
upd_entry_impl_int = *upd_entry_impl*

schematic_goal *upd_entry_impl_int_code* [*code*]:
upd_entry_impl_int ≡ *?i*
unfolding *upd_entry_impl_int_def*[*symmetric*] *upd_entry_impl_def* **unfolding** *int_folds* .

schematic_goal *upd_entries_impl_code* [*code*]:
upd_entries_impl ≡ *?impl*
unfolding *upd_entries_impl_def int_folds* .

definition *get_entries_impl_int* :: *nat* ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
get_entries_impl_int = *get_entries_impl*

schematic_goal *get_entries_impl_int_code*[*code*]:
get_entries_impl_int ≡ *?i*
unfolding *get_entries_impl_int_def*[*symmetric*] *get_entries_impl_def* **unfolding** *int_folds*
.

schematic_goal *dbm_minus_canonical_impl_code* [*code*]:
dbm_minus_canonical_impl ≡ *?i*
unfolding *dbm_minus_canonical_impl_def* **unfolding** *int_folds* .

definition *abstr_upd_impl_int*
:: *nat* ⇒ (*nat*, *int*) *acconstraint list* ⇒ *int DBMEntry Heap.array* ⇒ *int DBMEntry Heap.array Heap*
where [*symmetric*, *int_folds*]:
abstr_upd_impl_int = *abstr_upd_impl*

schematic_goal *abstr_upd_impl_int_code*[*code*]:
abstr_upd_impl_int ≡ *?i*
unfolding *abstr_upd_impl_int_def*[*symmetric*] *abstr_upd_impl_def* **unfolding** *int_folds* .

definition *abstr_FW_impl_int* :: _ ⇒ _ ⇒ *int DBMEntry Heap.array* ⇒ _
where [*symmetric*, *int_folds*]:
abstr_FW_impl_int = *abstr_FW_impl*

schematic_goal *abstr_FW_impl_int_code* [*code*]:
abstr_FW_impl_int ≡ *?i*
unfolding *abstr_FW_impl_int_def*[*symmetric*] *abstr_FW_impl_def* **unfolding** *int_folds* .

Extracting executable implementations **lemma** *hfkeep_hfdropI*:
assumes (*fi*, *f*) ∈ A^k →$_a$ *B*
shows (*fi*, *f*) ∈ A^d →$_a$ *B*
supply [*sep_heap_rules*] =
assms[*to_hnr*, *unfolded hn_refine_def*, *simplified hn_ctxt_def*, *simplified*, *rule_format*]
by *sepref_to_hoare sep_auto*

context *Simple_Network_Impl_nat_ceiling_start_state* — slow: 70s
begin

sublocale *impl*: *Reachability_Problem_Impl_Precise*
where *trans_fun* = *trans_from*
and *inv_fun* = *inv_fun*

and *ceiling* = *k_impl*
and *A* = *prod_ta*
and l_0 = l_0
and $l_0 i$ = $l_0 i$

```
  and n = m
  and k = k_fun
  and trans_impl = trans_impl
  and states' = states'
  and loc_rel = loc_rel
  and f = reach.E_precise_op'
  and op_impl = PR_CONST impl.E_precise_op'_impl
  and states_mem_impl = states'_memi
  and F = λ(l, _). F l
  and F1 = F o fst
  and F' = F o fst
  and F_impl = impl.F_impl
  apply standard
    apply (unfold PR_CONST_def, rule impl.E_precise_op'_impl.refine, rule states'_memi_correct)
  subgoal
    by auto
  subgoal
    apply simp
    done
subgoal
    apply simp
  done
  subgoal
    using impl.F_impl.refine by (intro hfkeep_hfdropI) simp
  done

lemma state_impl_abstract':
  assumes states'_memi li
  shows ∃ l. (li, l) ∈ loc_rel
proof −
  obtain Li si where li = (Li, si)
    by force
  with state_impl_abstract[of Li si] show ?thesis
    using assms unfolding states'_memi_def states_def by auto
qed

interpretation Bisimulation_Invariant
  (λ(l, u) (l', u'). conv_A prod_ta ⊢' ⟨l, u⟩ → ⟨l', u'⟩)
  (λ(L, s, u) (L', s', u'). Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
  (λ((L, s), u) (L', s', u'). L = L' ∧ u = u' ∧ s = s')
  (λ((L, s), u). conv.all_prop L s)
  (λ(L, s, u). conv.all_prop L s)
  by (rule prod_bisim)

lemma unreachability_prod:
  assumes
    formula = formula.EX φ
    (∄ u l' u'. (∀ c≤m. u c = 0) ∧ conv_A prod_ta ⊢' ⟨l0, u⟩ →* ⟨l', u'⟩ ∧ PR_CONST F l')
  shows ¬ Simple_Network_Language.conv A,(L0, map_of s0, λ_. 0) ⊨ formula
proof −
  let ?check = ¬ B.Ex_ev (λ(L, s, _). check_sexp φ L (the ∘ s)) (L0, map_of s0, λ_. 0)
  have ∗: PR_CONST F l ⟷ (λ((L, s), _). check_sexp φ L (the o s)) (l, u)
    for l and u :: nat ⇒ real
    unfolding assms(1) by auto
```

have *conv.all_prop* L_0 (*map_of* s_0)
using *all_prop_start* **unfolding** *conv_all_prop* **.**
then have
?check $\longleftrightarrow \neg$ *reach.Ex_ev* (λ ((L, s), _). *check_sexp* φ L (*the* o s)) (l_0, u_0)
by (*auto intro*!: *Ex_ev_iff*[*symmetric*, *unfolded A_B.equiv'_def*])
also from *assms*(*2*) **have** ...
apply −
apply *standard*
apply (*drule reach.Ex_ev_reaches*)
unfolding *impl.reaches_steps'*[*symmetric*]
apply (*subst* (*asm*) ∗)
apply *force*
done
finally show *?thesis*
using *assms* **unfolding** *models_def* **by** *simp*
qed

lemma *no_buechi_run_prod*:
assumes
formula = *formula.EX* φ
$\neg$ *has_deadlock* (*Simple_Network_Language.conv* A) (L_0, *map_of* s_0, λ_. 0)
$\nexists$ u xs. ($\forall$ $c \leq m$. u c = 0) $\wedge$ *reach.run* ((l_0, u) ## xs) $\wedge$ *alw* (*ev* (*holds* ($F \circ$ *fst*))) ((l_0, u) ## xs)
shows $\neg$ *Graph_Defs.alw_ev*
(λ (L, s, u) (L', s', u'). *Simple_Network_Language.conv* $A \vdash \langle L, s, u\rangle \rightarrow \langle L', s', u'\rangle$)
(λ (L, s, _). *check_sexp* φ L (*the* o s)) (L_0, *map_of* s_0, λ_. 0)
proof −
let *?F* = λ((L, s), _). *check_sexp* φ L (*the* o s)
have ∗: *?F* = ($F \circ$ *fst*)
using *assms*(*1*) **by** *auto*
have *conv.all_prop* L_0 (*map_of* s_0)
using *all_prop_start* **unfolding** *conv_all_prop* **.**
then have
$\neg$ *B.alw_ev* (λ(L, s, _). *check_sexp* φ L (*the* $\circ$ s)) (L_0, *map_of* s_0, λ_. 0)
$\longleftrightarrow \neg$ *reach.alw_ev* *?F* (l_0, u_0)
by (*auto intro*!: *alw_ev_iff*[*symmetric*, *unfolded A_B.equiv'_def*])
also have ...
proof −
from *assms*(*2*) **have** $\neg$ *reach.deadlock* (l_0, λ_. 0)
unfolding *has_deadlock_def deadlock_start_iff* **.**
then obtain xs **where** *reach.run* ((l_0, λ_. 0) ## xs)
using *reach.no_deadlock_run_extend*[*of* (l_0, λ_. 0) []] **by** *auto*
with *assms*(*3*) **show** *?thesis*
unfolding *reach.alw_ev_def* ‹*?F* = _› **by** *force*
qed
finally show *?thesis*
using *assms* **by** *simp*
qed

lemma *deadlock_prod*:
$\neg$ *reach.deadlock* (l_0, λ_. 0)
$\longleftrightarrow \neg$ *has_deadlock* (*Simple_Network_Language.conv* A) (L_0, *map_of* s_0, λ_. 0)
unfolding *has_deadlock_def deadlock_start_iff* **..**

lemma *list_assn_split*:
list_assn (*list_assn impl.location_assn*) (*split_k num_split L*) (*split_k num_split Li*) =
list_assn impl.location_assn L Li
proof −
have *1*: *impl.location_assn* = *pure* (*the_pure impl.location_assn*)
using *impl.pure_K* **by** *auto*
have (*split_k num_split Li*, *split_k num_split L*) ∈ ⟨⟨*the_pure impl.location_assn*⟩*list_rel*⟩*list_rel*
⟷ (*Li*, *L*) ∈ ⟨*the_pure impl.location_assn*⟩*list_rel*
unfolding *list_rel_def* **by** (*auto simp*: *list_all2_split_k*[*symmetric*])
then show *?thesis*
apply (*subst* (*2*) *1*, *subst 1*)
unfolding *list_assn_pure_conv* **unfolding** *pure_def* **by** *auto*
qed

theorem *unreachability_checker_hnr*:
assumes ⋀*li*. *P_loc li* ⟹ *states'_memi li*
and *list_all* (λ*x*. *P_loc x* ∧ *states'_memi x*) *L_list*
and *list_all* (λ(*l*, *y*). *list_all* (λ*M*. *length M* = *Suc m* ∗ *Suc m*) *y*) *M_list*
and *fst* ‘ *set M_list* = *set L_list*
and *formula* = *formula.EX* φ
shows (
uncurry0 (*impl.unreachability_checker L_list M_list* (*split_k num_split*)),
uncurry0 (*SPEC* (λ*r*. *r* ⟶
¬ *Simple_Network_Language.conv A*,(*L*$_0$, *map_of s*$_0$, λ_. *0*) ⊨ *formula*)))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
proof −
define *checker* **where** *checker* ≡ *impl.unreachability_checker L_list M_list*
from *assms*(*4*) **have** *fst* ‘ *set M_list* ⊆ *set L_list*
by *blast*
note [*sep_heap_rules*] =
impl.unreachability_checker_hnr[
OF state_impl_abstract',
OF assms(*1*,*2*) *this assms*(*3*) *split_k_full_split list_assn_split*
impl.L_dom_M_eqI[*OF state_impl_abstract'*, *OF assms*(*1*,*2*) *this assms*(*3*,*4*)],
to_hnr, *unfolded hn_refine_def*, *rule_format*, *folded checker_def*
]
show *?thesis*
unfolding *checker_def*[*symmetric*] **using** *unreachability_prod*[*OF assms*(*5*)]
by *sepref_to_hoare* (*sep_auto simp*: *pure_def*)
qed

theorem *unreachability_checker2_refine*:
assumes ⋀*li*. *P_loc li* ⟹ *states'_memi li*
and *list_all* (λ*x*. *P_loc x* ∧ *states'_memi x*) *L_list*
and *list_all* (λ(*l*, *y*). *list_all* (λ*M*. *length M* = *Suc m* ∗ *Suc m*) *y*) *M_list*
and *fst* ‘ *set M_list* = *set L_list*
and *formula* = *formula.EX* φ
shows
impl.unreachability_checker2 L_list M_list (*split_k num_split*) ⟶
¬ *Simple_Network_Language.conv A*,(*L*$_0$, *map_of s*$_0$, λ_. *0*) ⊨ *formula*
using *impl.unreachability_checker2_refine*[*OF state_impl_abstract'*,
OF assms(*1*,*2*) *assms*(*4*)[*THEN equalityD1*] *assms*(*3*) *split_k_full_split list_assn_split*
assms(*4*)
]

```
    unreachability_prod[OF assms(5)]
  by auto

theorem unreachability_checker3_refine:
  assumes ⋀li. P_loc li ⟹ states'_memi li
    and list_all (λx. P_loc x ∧ states'_memi x) L_list
    and list_all (λ(l, y). list_all (λM. length M = Suc m * Suc m) y) M_list
    and fst ‘ set M_list = set L_list
    and formula = formula.EX φ
  shows
  impl.certify_unreachable_pure L_list M_list (split_k' num_split M_list) ⟶
    ¬ Simple_Network_Language.conv A,(L₀, map_of s₀, λ_. 0) ⊨ formula
  using impl.certify_unreachable_pure_refine[
      OF state_impl_abstract', OF assms(1,2) assms(4)[THEN equalityD1] assms(3)
        split_k'_full_split[of M_list, unfolded assms(4)] assms(4)
      ]
    unreachability_prod[OF assms(5)]
  by auto

lemma init_conds: {l₀} ⊆ states' ([l₀i], {l₀}) ∈ ⟨loc_rel⟩list_set_rel
  unfolding list_set_rel_singleton_iff using impl.init_impl by blast+

theorem no_buechi_run_checker_refine:
  assumes ⋀li. P_loc li ⟹ states'_memi li
    and list_all (λx. P_loc x ∧ states'_memi x) L_list
    and list_all (λ(l, y). list_all (λ(M, _). length M = Suc m * Suc m) y) M_list
    and fst ‘ set M_list = set L_list
    and formula = formula.EX φ
    and ¬ has_deadlock (Simple_Network_Language.conv A) (L₀, map_of s₀, λ_. 0)
  shows
  impl.certify_no_buechi_run_pure L_list M_list (split_k' num_split M_list) [l₀i] ⟶
  ¬ Graph_Defs.alw_ev
    (λ (L, s, u) (L', s', u'). Simple_Network_Language.conv A ⊢ ⟨L, s, u⟩ → ⟨L', s', u'⟩)
    (λ (L, s, _). check_sexp φ L (the o s)) (L₀, map_of s₀, λ_. 0)
proof −
  have
    (case (l, D) of (l, _) ⇒ F l) = (F ∘ fst) (l, Z)
    if l₀' ∈ {l₀} reach1.E_precise_op'.E_from_op_empty** (l₀', init_dbm) (l, D)
      dbm.zone_of (curry (conv_M D)) = Z for l₀' l D Z
    unfolding comp_def by simp
  with init_conds show ?thesis
    using impl.certify_no_buechi_run_pure_refine[
        OF state_impl_abstract', OF assms(1,2) assms(4)[THEN equalityD1] assms(3)
          split_k'_full_split[of M_list, unfolded assms(4)] _ _ assms(4),
          of {(L₀, map_of s₀)} [l₀i]
        ]
    using no_buechi_run_prod[OF assms(5, 6)] by auto
qed

lemma abstr_repair_impl_refine:
  impl.abstr_repair_impl = abstr_repair_impl m
  unfolding abstr_repair_impl_def impl.abstr_repair_impl_def  impl.abstra_repair_impl_def
..
```

```
lemma E_op_impl_refine:
  impl.E_precise_op'_impl l r g l' M = E_op_impl m (inv_fun l) r g (inv_fun l') M
  unfolding impl.E_precise_op'_impl_def E_op_impl_def abstr_repair_impl_refine ..

definition
  succs1 n_ps invs ≡
  let
    inv_fun = λ(L, a). concat (map (λi. invs !! i !! (L ! i)) [0..<n_ps]);
    E_op_impl = (λl r g l' M. E_op_impl m (inv_fun l) r g (inv_fun l') M)
  in (λL_s M.
    if M = [] then Heap_Monad.return []
    else imp_nfoldli (trans_impl L_s) (λσ. Heap_Monad.return True)
      (λc σ. case c of (g, a1a, r, L_s') ⇒ do {
        M ← heap_map amtx_copy M;
        Ms ← imp_nfoldli M (λσ. Heap_Monad.return True)
          (λxb σ. do {
            x'c ← E_op_impl L_s r g L_s' xb;
            x'e ← check_diag_impl m x'c;
            Heap_Monad.return (if x'e then σ else op_list_prepend x'c σ)
          }) [];
        Heap_Monad.return (op_list_prepend (L_s', Ms) σ)
      })
    []) for n_ps :: nat and invs :: (nat, int) acconstraint list iarray iarray

lemma succs1_refine:
  impl.succs_precise'_impl = succs1 n_ps invs2
  unfolding impl.succs_precise'_impl_def impl.succs_precise_inner_impl_def
  unfolding succs1_def Let_def PR_CONST_def E_op_impl_refine
  unfolding inv_fun_alt_def ..

schematic_goal trans_impl_alt_def:
  trans_impl ≡ ?impl
  unfolding trans_impl_def
  apply (abstract_let int_trans_impl int_trans_impl)
  apply (abstract_let bin_trans_from_impl bin_trans_impl)
  apply (abstract_let broad_trans_from_impl broad_trans_impl)
  unfolding int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def
  apply (abstract_let trans_in_broad_grouped trans_in_broad_grouped)
  apply (abstract_let trans_out_broad_grouped trans_out_broad_grouped)
  apply (abstract_let trans_in_map trans_in_map)
  apply (abstract_let trans_out_map trans_out_map)
  apply (abstract_let int_trans_from_all_impl int_trans_from_all_impl)
  unfolding int_trans_from_all_impl_def
  apply (abstract_let int_trans_from_vec_impl int_trans_from_vec_impl)
  unfolding int_trans_from_vec_impl_def
  apply (abstract_let int_trans_from_loc_impl int_trans_from_loc_impl)
  unfolding int_trans_from_loc_impl_def
  apply (abstract_let trans_i_map trans_i_map)
  unfolding trans_out_broad_grouped_def trans_out_broad_map_def
  unfolding trans_in_broad_grouped_def trans_in_broad_map_def
  unfolding trans_in_map_def trans_out_map_def
  unfolding trans_i_map_def
  apply (abstract_let trans_map trans_map)
  .
```

schematic_goal *succs1_alt_def*:
succs1 ≡ *?impl*
unfolding *succs1_def*
apply (*abstract_let trans_impl trans_impl*)
unfolding *trans_impl_alt_def*
.

schematic_goal *succs_impl_alt_def*:
impl.succs_precise'_impl ≡ *?impl*
unfolding *impl.succs_precise'_impl_def impl.succs_precise_inner_impl_def*
apply (*abstract_let impl.E_precise_op'_impl E_op_impl*)
unfolding *impl.E_precise_op'_impl_def fw_impl'_int*
apply (*abstract_let trans_impl trans_impl*)
unfolding *trans_impl_def*
apply (*abstract_let int_trans_impl int_trans_impl*)
apply (*abstract_let bin_trans_from_impl bin_trans_impl*)
apply (*abstract_let broad_trans_from_impl broad_trans_impl*)
unfolding *int_trans_impl_def bin_trans_from_impl_def broad_trans_from_impl_def*
apply (*abstract_let trans_in_broad_grouped trans_in_broad_grouped*)
apply (*abstract_let trans_out_broad_grouped trans_out_broad_grouped*)
apply (*abstract_let trans_in_map trans_in_map*)
apply (*abstract_let trans_out_map trans_out_map*)
apply (*abstract_let int_trans_from_all_impl int_trans_from_all_impl*)
unfolding *int_trans_from_all_impl_def*
apply (*abstract_let int_trans_from_vec_impl int_trans_from_vec_impl*)
unfolding *int_trans_from_vec_impl_def*
apply (*abstract_let int_trans_from_loc_impl int_trans_from_loc_impl*)
unfolding *int_trans_from_loc_impl_def*
apply (*abstract_let trans_i_map trans_i_map*)
unfolding *trans_out_broad_grouped_def trans_out_broad_map_def*
unfolding *trans_in_broad_grouped_def trans_in_broad_map_def*
unfolding *trans_in_map_def trans_out_map_def*
unfolding *trans_i_map_def*
apply (*abstract_let trans_map trans_map*)
apply (*abstract_let inv_fun :: nat list × int list ⇒ _ inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
apply (*abstract_let n_ps n_ps*)
by (*rule Pure.reflexive*)

end

concrete_definition (**in** −) *succs_impl*
uses *Simple_Network_Impl_nat_ceiling_start_state.succs1_alt_def*

context *Simple_Network_Impl_nat_ceiling_start_state* — slow: 100s
begin

schematic_goal *check_deadlock_impl_alt_def*:
impl.check_deadlock_impl ≡ *?impl*
unfolding *impl.check_deadlock_impl_def*
apply (*abstract_let trans_impl trans_impl*)

unfolding *trans_impl_alt_def*
apply (*abstract_let inv_fun :: nat list × int list ⇒ _ inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
apply (*abstract_let n_ps n_ps*)
.

context
fixes *L_list*
assumes *L_list*: *list_all states'_memi L_list*
begin

private lemma *A*:
list_all (*λx. states'_memi x ∧ states'_memi x*) *L_list*
using *L_list* **by** *simp*

context
fixes *M_list* :: ((*nat list × int list*) *× int DBMEntry list list*) *list*
assumes *assms*: *fst ' set M_list ⊆ set L_list*
list_all (*λ*(*l*, *y*). *list_all* (*λM. length M = Suc m ∗ Suc m*) *y*) *M_list*
begin

lemmas *assms = L_list assms*

lemma *unreachability_checker_def*:
impl.unreachability_checker L_list M_list (*split_k num_split*) ≡
let Fi = impl.F_impl; *Pi = impl.P_impl*; *copyi = amtx_copy*; *Lei = dbm_subset_impl m*;
l$_0$*i = Heap_Monad.return l*$_0$*i*; *s*$_0$*i = impl.init_dbm_impl*; *succsi = impl.succs_precise'_impl*
in do {
let _ = start_timer ();
M_table ← impl.M_table M_list;
let _ = save_time STR ''Time for loading certificate'';
r ← certify_unreachable_impl_inner
Fi Pi copyi Lei succsi l$_0$*i s*$_0$*i* (*split_k num_split*) *L_list M_table*;
Heap_Monad.return r
}
by (*subst impl.unreachability_checker_alt_def*[*OF*
state_impl_abstract', *OF _ A assms*(*2*,*3*) *split_k_full_split list_assn_split*];
simp)

schematic_goal *unreachability_checker_alt_def*:
impl.unreachability_checker L_list M_list (*split_k num_split*) ≡ *?x*
apply (*subst unreachability_checker_def*)
apply (*subst impl.M_table_def*[*OF state_impl_abstract'*, *of states'_memi*, *OF _ A assms*(*2*,*3*)])
apply *assumption*
unfolding *impl.F_impl_def impl.P_impl_def*
apply (*abstract_let states'_memi check_states*)
unfolding *states'_memi_def states_mem_compute'*
apply (*abstract_let map states_i* [*0..<n_ps*] *states_i*)
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*

```
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
  unfolding impl.init_dbm_impl_def impl.a₀_impl_def
  unfolding impl.subsumes_impl_def
  unfolding impl.emptiness_check_impl_def
  unfolding impl.state_copy_impl_def
  by (rule Pure.reflexive)

definition no_deadlock_certifier where
  no_deadlock_certifier ≡
  Reachability_Problem_Impl_Precise.unreachability_checker
    m trans_impl l₀i (PR_CONST impl.E_precise_op'_impl)
    states'_memi (λ(l, M). impl.check_deadlock_impl l M ⤜ (λr. Heap_Monad.return (¬ r)))

lemma no_deadlock_certifier_alt_def1:
  no_deadlock_certifier L_list M_list (split_k num_split) ≡
   let
      Fi = (λ(l, M). impl.check_deadlock_impl l M ⤜ (λr. Heap_Monad.return (¬ r)));
      Pi = impl.P_impl; copyi = amtx_copy; Lei = dbm_subset_impl m;
      l₀i = Heap_Monad.return l₀i; s₀i = impl.init_dbm_impl;
      succsi = impl.succs_precise'_impl
    in do {
     let _ = start_timer ();
     M_table ← impl.M_table M_list;
     let _ = save_time STR ''Time for loading certificate'';
     r ← certify_unreachable_impl_inner
       Fi Pi copyi Lei succsi l₀i s₀i (split_k num_split) L_list M_table;
     Heap_Monad.return r
   }
   unfolding no_deadlock_certifier_def
   by (subst impl.deadlock_unreachability_checker_alt_def[
        OF state_impl_abstract', OF _ A assms(2,3) split_k_full_split list_assn_split
        ];
       simp)

schematic_goal no_deadlock_certifier_alt_def:
  no_deadlock_certifier L_list M_list (split_k num_split) ≡ ?x
  apply (subst no_deadlock_certifier_alt_def1)
 apply (subst impl.M_table_def[OF state_impl_abstract', of states'_memi, OF _ A assms(2,3)])
  apply assumption
  unfolding check_deadlock_impl_alt_def impl.P_impl_def
  apply (abstract_let states'_memi check_states)
  unfolding states'_memi_def states_mem_compute'
  apply (abstract_let map states_i [0..<n_ps] states_i)
  unfolding succs_impl_alt_def
  unfolding k_impl_alt_def k_i_def

  unfolding impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def
  unfolding impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def
  unfolding impl.init_dbm_impl_def impl.a₀_impl_def
  unfolding impl.subsumes_impl_def
  unfolding impl.emptiness_check_impl_def
  unfolding impl.state_copy_impl_def
  .
```

lemmas *no_deadlock_certifier_hnr'* =
impl.deadlock_unreachability_checker_hnr[*folded no_deadlock_certifier_def*,
OF state_impl_abstract',
OF _ A assms(2,3) *impl.L_dom_M_eqI*[*OF state_impl_abstract' A assms*(2,3)]
split_k_full_split list_assn_split,
unfolded deadlock_prod
]

schematic_goal *unreachability_checker2_alt_def*:
impl.unreachability_checker2 L_list M_list (*split_k num_split*) ≡ *?x*
apply (*subst impl.unreachability_checker2_def*[
OF state_impl_abstract', *OF _ A assms*(2,3) *split_k_full_split list_assn_split*
], (*simp*; *fail*))
apply (*subst impl.M_table_def*[*OF state_impl_abstract'*, *of states'_memi*, *OF _ A assms*(2,3)])
apply *assumption*
unfolding *check_deadlock_impl_alt_def impl.P_impl_def impl.F_impl_def*
apply (*abstract_let states'_memi check_states*)
unfolding *states'_memi_def states_mem_compute'*
apply (*abstract_let map states_i* [*0..<n_ps*] *states_i*)
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.a*$_0$*_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

definition *no_deadlock_certifier2* **where**
no_deadlock_certifier2 ≡
Reachability_Problem_Impl_Precise.unreachability_checker2
m trans_impl l$_0$*i* (*PR_CONST impl.E_precise_op'_impl*)
states'_memi (λ(*l*, *M*). *impl.check_deadlock_impl l M* ⤜ (λ*r*. *Heap_Monad.return* (¬ *r*)))

schematic_goal *no_deadlock_certifier2_alt_def*:
no_deadlock_certifier2 L_list M_list (*split_k num_split*) ≡ *?x*
unfolding *no_deadlock_certifier2_def*
apply (*subst impl.deadlock_unreachability_checker2_def*[
OF state_impl_abstract', *OF _ A assms*(2,3) *split_k_full_split list_assn_split*
], (*simp*; *fail*))
apply (*subst impl.M_table_def*[*OF state_impl_abstract'*, *of states'_memi*, *OF _ A assms*(2,3)])
apply *assumption*
unfolding *check_deadlock_impl_alt_def impl.P_impl_def*
apply (*abstract_let states'_memi check_states*)
unfolding *states'_memi_def states_mem_compute'*
apply (*abstract_let map states_i* [*0..<n_ps*] *states_i*)
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*

unfolding *impl.start_inv_check_impl_def unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.init_dbm_impl_def impl.a$_0$_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
by (*rule Pure.reflexive*)

lemmas *no_deadlock_certifier2_refine' =*
impl.deadlock_unreachability_checker2_hnr[
folded no_deadlock_certifier2_def,
OF state_impl_abstract' A assms(*3*) _ *split_k_full_split list_assn_split*
]

schematic_goal *unreachability_checker3_alt_def*:
impl.certify_unreachable_pure L_list M_list (*split_k' num_split M_list*) ≡ *?x*
if *fst ' set M_list = set L_list*
apply (*subst impl.certify_unreachable_pure_def*[
OF state_impl_abstract', OF _ A assms(*2*,*3*) *split_k'_full_split*[*of M_list*, *unfolded that*]
], (*simp*; *fail*))
apply (*abstract_let impl.init_dbm_impl* :: *int DBMEntry Heap.array Heap init_dbm*)
unfolding *impl.init_dbm_impl_def*
apply (*abstract_let impl.Mi M_list Mi*)
apply (*subst impl.Mi_def*[*OF state_impl_abstract', of states'_memi, OF _ A assms*(*2*,*3*)])
apply *assumption*
unfolding *check_deadlock_impl_alt_def impl.P_impl_def impl.F_impl_def*
apply (*abstract_let states'_memi check_state*)
unfolding *states'_memi_def states_mem_compute'*
apply (*abstract_let map states_i* [*0*..<*n_ps*] *states_i*)
apply (*abstract_let impl.succs_precise'_impl succsi*)
unfolding *succs_impl.refine*[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*] *succs1_refine*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
apply (*abstract_let n_ps n_ps*)
apply (*abstract_let n_vs n_vs*)
.

schematic_goal *check_deadlock_impl_alt_def2*:
impl.check_deadlock_impl ≡ *?impl*
unfolding *impl.check_deadlock_impl_def*
apply (*abstract_let trans_impl trans_impl*)
unfolding *trans_impl_alt_def*
apply (*abstract_let inv_fun* :: *nat list* × *int list* ⇒ _ *inv_fun*)
unfolding *inv_fun_alt_def*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
apply (*abstract_let n_ps n_ps*)
.

definition *no_deadlock_certifier3* **where**
no_deadlock_certifier3 ≡
Reachability_Problem_Impl_Precise.certify_unreachable_pure
m trans_impl l$_0$i (*PR_CONST impl.E_precise_op'_impl*)
states'_memi (λ(*l*, *M*). *impl.check_deadlock_impl l M* ⤜ (λ*r*. *Heap_Monad.return* (¬ *r*)))

definition
check_deadlock_fold = (λ*a*. *run_heap* ((*case a of* (*l*, *s*) ⇒ *array_unfreeze s*
⤜ (λ*s*. *Heap_Monad.return* (*id l*, *s*)))
⤜ (λ*a*. *case a of* (*l*, *M*) ⇒ *impl.check_deadlock_impl l M*
⤜ (λ*r*. *Heap_Monad.return* (¬ *r*)))))

schematic_goal *no_deadlock_certifier3_alt_def*:
no_deadlock_certifier3 L_list M_list (*split_k′ num_split M_list*) ≡ *?x*
if *fst ‘ set M_list* = *set L_list*
unfolding *no_deadlock_certifier3_def*
apply (*subst impl.deadlock_unreachability_checker3_def*[
OF state_impl_abstract′, *OF _ A assms*(*2*,*3*) *split_k′_full_split*[*of M_list*, *unfolded that*]
], (*simp*; *fail*))
unfolding *check_deadlock_fold_def*[*symmetric*]
apply (*abstract_let check_deadlock_fold check_deadlock1*)
unfolding *check_deadlock_fold_def*
apply (*abstract_let impl.check_deadlock_impl check_deadlock*)
apply (*abstract_let impl.init_dbm_impl* :: *int DBMEntry Heap.array Heap init_dbm*)
unfolding *impl.init_dbm_impl_def*
apply (*abstract_let impl.Mi M_list Mi*)
apply (*subst impl.Mi_def*[*OF state_impl_abstract′*, *of states′_memi*, *OF _ A assms*(*2*,*3*)])
apply *assumption*
apply (*abstract_let* (λ*a* :: (*nat list* × *int list*) × *int DBMEntry iarray*. *run_heap*
((*case a of* (*l*, *s*) ⇒ *array_unfreeze s* ⤜ (λ*s*. *Heap_Monad.return* (*id l*, *s*))) ⤜ *impl.P_impl*))
P)
apply (*abstract_let impl.P_impl* :: *_* × *int DBMEntry Heap.array* ⇒ *_ P_impl*)
apply (*abstract_let* (λ(*as* :: *int DBMEntry iarray*) *bs*.
(∃ *i*≤*m*. *as* !! (*i* + *i* ∗ *m* + *i*) < *Le 0*) ∨ *array_all2* (*Suc m* ∗ *Suc m*) (≤) *as bs*)
subsumption)
unfolding *check_deadlock_impl_alt_def2*
unfolding *impl.P_impl_def impl.F_impl_def*
apply (*abstract_let states′_memi check_states*)
unfolding *states′_memi_def states_mem_compute′*
apply (*abstract_let map states_i* [*0*..<*n_ps*] *states_i*)
apply (*abstract_let impl.succs_precise′_impl succsi*)
unfolding *succs_impl.refine*[*OF Simple_Network_Impl_nat_ceiling_start_state_axioms*] *succs1_refine*
apply (*abstract_let invs2 invs*)
unfolding *invs2_def*
apply (*abstract_let n_ps n_ps*)
apply (*abstract_let n_vs n_vs*)
.

lemma *no_deadlock_certifier3_refine′*:
no_deadlock_certifier3 L_list M_list (*split_k′ num_split M_list*)
⟶ (∀ *u*. (∀ *c*≤*m*. *u c* = *0*) ⟶ ¬ *reach.deadlock* (*l*$_0$, *u*)) **if** *fst ‘ set M_list* = *set L_list*
by (*rule impl.deadlock_unreachability_checker3_hnr*[
folded no_deadlock_certifier3_def, *OF state_impl_abstract′ A assms*(*3*)
]) (*simp add*: *that split_k′_full_split*[*symmetric*])+

end

context
fixes *M_list* :: ((*nat list* × *int list*) × (*int DBMEntry list* × *nat*) *list*) *list*

```
  assumes assms:
    fst ‘ set M_list ⊆ set L_list
    list_all (λ(l, y). list_all (λ(M, _). length M = Suc m ∗ Suc m) y) M_list
begin

schematic_goal no_buechi_run_checker_alt_def:
  impl.certify_no_buechi_run_pure L_list M_list (split_k′ num_split M_list) [l0 i] ≡ ?x
  if fst ‘ set M_list = set L_list
  apply (subst impl.certify_no_buechi_run_pure_def[
        OF state_impl_abstract′, OF _ A assms split_k′_full_split[of M_list, unfolded that]
init_conds
      ], assumption)
  apply (subst impl.initsi_def[
        OF state_impl_abstract′, OF _ A assms split_k′_full_split[of M_list, unfolded that]
init_conds
      ], assumption)
  apply (abstract_let impl.init_dbm_impl :: int DBMEntry Heap.array Heap init_dbm)
  unfolding impl.init_dbm_impl_def
  apply (abstract_let impl.M2i M_list Mi)
  apply (subst impl.M2i_def[OF state_impl_abstract′, of states′_memi, OF _ A assms(1,2)])
   apply assumption
  unfolding check_deadlock_impl_alt_def impl.P_impl_def impl.F_impl_def
  apply (abstract_let states′_memi check_state)
  unfolding states′_memi_def states_mem_compute′
  apply (abstract_let map states_i [0..<n_ps] states_i)
  apply (abstract_let impl.succs_precise′_impl succsi)
 unfolding succs_impl.refine[OF Simple_Network_Impl_nat_ceiling_start_state_axioms] succs1_refine
  apply (abstract_let invs2 invs)
  unfolding invs2_def
  apply (abstract_let n_ps n_ps)
  apply (abstract_let n_vs n_vs)
  .

end

end

theorem no_deadlock_certifier_hnr:
  assumes list_all states′_memi L_list
      and list_all (λ(l, y). list_all (λM. length M = Suc m ∗ Suc m) y) M_list
      and fst ‘ set M_list = set L_list
  shows (
  uncurry0 (no_deadlock_certifier L_list M_list (split_k num_split)),
  uncurry0 (SPEC (λr. r ⟶
    ¬ has_deadlock (Simple_Network_Language.conv A) (L0, map_of s0, λ_. 0))))
  ∈ unit_assn^k →a bool_assn
proof −
  define checker where checker ≡ no_deadlock_certifier L_list M_list
  define P where P ≡ reach.deadlock
  note ∗[sep_heap_rules] =
    no_deadlock_certifier_hnr′[
      OF assms(1) _ assms(2),
      to_hnr, unfolded hn_refine_def, rule_format, folded checker_def P_def
      ]
```

```
  from assms(3) show ?thesis
    unfolding checker_def[symmetric] deadlock_prod[symmetric] P_def[symmetric]
    by sepref_to_hoare (sep_auto simp: pure_def)
qed

theorem no_deadlock_certifier2_refine:
  assumes list_all states'_memi L_list
      and list_all (λ(l, y). list_all (λM. length M = Suc m * Suc m) y) M_list
      and fst ‘ set M_list = set L_list
  shows no_deadlock_certifier2 L_list M_list (split_k num_split) ⟶
    ¬ has_deadlock (Simple_Network_Language.conv A) (L₀, map_of s₀, λ_. 0)
  using no_deadlock_certifier2_refine'[OF assms(1) _ assms(2), of num_split] assms(3)
  unfolding deadlock_prod[symmetric] by auto

theorem no_deadlock_certifier3_refine:
  assumes list_all states'_memi L_list
      and list_all (λ(l, y). list_all (λM. length M = Suc m * Suc m) y) M_list
      and fst ‘ set M_list = set L_list
  shows no_deadlock_certifier3 L_list M_list (split_k' num_split M_list) ⟶
    ¬ has_deadlock (Simple_Network_Language.conv A) (L₀, map_of s₀, λ_. 0)
  using no_deadlock_certifier3_refine' assms unfolding deadlock_prod[symmetric] by auto

definition
  show_dbm_impl' M ≡ do {
  s ← show_dbm_impl m show_clock show M;
  Heap_Monad.return (String.implode s)
}

definition
  show_state_impl l ≡ do {
    let s = show_state l;
    let s = String.implode s;
    Heap_Monad.return s
  }

definition
  trace_table M_table ≡ do {
    M_list' ← list_of_map_impl M_table;
    let _ = println STR ''Inverted table'';
    Heap_Monad.fold_map (λ (l, xs). do {
      s1 ← show_state_impl l;
      let _ = println (s1 + STR '':'');
      Heap_Monad.fold_map (λM.  do {
        s2 ← show_dbm_impl_all m show_clock show M;
        let _ = println (STR ''  '' + String.implode s2);
        Heap_Monad.return ()
      }) xs;
      Heap_Monad.return ()
    }) M_list';
    Heap_Monad.return ()
  } for M_table


definition
```

```
  check_prop_fail L_list M_list ≡ let
    P_impl = impl.P_impl;
    copy = amtx_copy;
    show_dbm = show_dbm_impl';
    show_state = show_state_impl
   in do {
    M_table ← M_table M_list;

    trace (table M_list);

    r ← check_prop_fail_impl P_impl copy show_dbm show_state L_list M_table;
    case r of None ⇒ Heap_Monad.return () | Some (l, M) ⇒ do {
      let b = states'_memi l;
      let _ = println (if b then STR ''State passed'' else STR ''State failed'');
      b ← check_diag_impl m M;
      let _ = println (if b then STR ''DBM passed diag'' else STR ''DBM failed diag'');
      b ← check_diag_nonpos m M;
       let _ = println (if b then STR ''DBM passed diag nonpos'' else STR ''DBM failed diag
nonpos'');
       b ← check_nonneg m M;
      let _ = println (if b then STR ''DBM passed nonneg'' else STR ''DBM failed nonneg'');
      s ← show_dbm_impl_all m show_clock show M;
      let _ = println (STR ''DBM: '' + String.implode s);
      Heap_Monad.return ()
    }
   }
```

definition

```
  check_deadlock_fail L_list M_list ≡ let
    P_impl = (λ(l, M). impl.check_deadlock_impl l M);
    copy = amtx_copy;
    show_dbm = show_dbm_impl';
    show_state = show_state_impl
   in do {
    M_table ← M_table M_list;
    r ← check_prop_fail_impl P_impl copy show_dbm show_state L_list M_table;
    case r of None ⇒ Heap_Monad.return () | Some (l, M) ⇒ do {
      let _ = println (STR ''↩The following state is deadlocked'');
      s ← show_state l;
      let _ = println s;
      s ← show_dbm_impl_all m show_clock show M;
      let _ = println (String.implode s);
      Heap_Monad.return ()
    }
   }
```

definition

```
  check_invariant_fail ≡ λL_list M_list. let
    copy = amtx_copy;
    succs = impl.succs_precise'_impl;
    Lei = dbm_subset_impl m;
    show_state = show_state_impl;
```

```
  show_dbm = show_dbm_impl_all m show_clock show
in do {
  M_table ← M_table M_list;
  r ← check_invariant_fail_impl copy Lei succs L_list M_table;
  case r of None ⇒ Heap_Monad.return ()
  | Some (Inl (Inl (l, l', xs))) ⇒ do {
      let _ = println (STR ''The successor is not contained in L:'');
      s ← show_state l;
      let _ = println (STR ''  '' + s);
      s ← show_state l';
      let _ = println (STR ''  '' + s);
      Heap_Monad.fold_map (λM. do {
        s ← show_dbm M;
        let _ = println (STR '' '' + String.implode s);
        Heap_Monad.return ()
      }) xs;
      Heap_Monad.return ()
    }
  | Some (Inl (Inr (l, l', xs))) ⇒ do {
      let _ = println (STR ''The successor is not empty:'');
      s ← show_state l;
      let _ = println (STR ''  '' + s);
      s ← show_state l';
      let _ = println (STR ''  '' + s);
      Heap_Monad.fold_map (λM. do {
        s ← show_dbm M;
        let _ = println (STR '' '' + String.implode s);
        Heap_Monad.return ()
      }) xs;
      Heap_Monad.return ()
    }
  | Some (Inr (l, as, l', M, xs)) ⇒ do {
      s1 ← show_state l;
      s2 ← show_state l';
      s3 ← show_dbm M;
      let _ = println (STR ''↩A successor of the zones for:↩  '' + s1);
      Heap_Monad.fold_map (λM. do {
        s ← show_dbm M;
        let _ = println (STR ''↩'' + String.implode s);
        Heap_Monad.return ()
      }) as;
      let _ = println (STR ''↩is not subsumed:↩  '' + s2 + STR ''↩'');
      let _ = println (String.implode s3 + STR ''↩'');
      let _ = println (STR ''These are the candidate dbms:'');
      Heap_Monad.fold_map (λM. do {
        s ← show_dbm M;
        let _ = println (STR ''↩'' + String.implode s);
        Heap_Monad.return ()
      }) xs;
      let _ = println (STR '''');
      Heap_Monad.return ()
    }
}
```

schematic_goal *check_prop_fail_alt_def*:
check_prop_fail ≡ *?t*
unfolding *check_prop_fail_def*
unfolding *M_table_def trace_table_def*
unfolding *impl.P_impl_def*
unfolding *show_dbm_impl'_def*
unfolding *show_state_impl_def*
apply (*abstract_let states'_memi check_states*)
unfolding *states'_memi_def states_mem_compute'*
apply (*abstract_let map states_i* [*0..<n_ps*] *states_i*)
by (*rule Pure.reflexive*)

schematic_goal *check_deadlock_fail_alt_def*:
check_deadlock_fail ≡ *?t*
unfolding *check_deadlock_fail_def*
unfolding *M_table_def trace_table_def*
unfolding *check_deadlock_impl_alt_def*
unfolding *succs_impl_alt_def*
unfolding *k_impl_alt_def k_i_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *unbounded_dbm_impl_def unbounded_dbm'_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
unfolding *show_dbm_impl'_def*
unfolding *show_state_impl_def*
.

schematic_goal *check_invariant_fail_alt_def*:
check_invariant_fail ≡ *?t*
unfolding *check_invariant_fail_def*
unfolding *M_table_def*
unfolding *succs_impl_alt_def*

unfolding *impl.E_op''_impl_def impl.abstr_repair_impl_def impl.abstra_repair_impl_def*
unfolding *impl.subsumes_impl_def*
unfolding *impl.emptiness_check_impl_def*
unfolding *impl.state_copy_impl_def*
unfolding *show_dbm_impl'_def show_state_impl_def*
by (*rule Pure.reflexive*)

end

concrete_definition *unreachability_checker* **uses**
Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker_alt_def

concrete_definition *no_deadlock_certifier* **uses**
Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier_alt_def

concrete_definition *unreachability_checker2* **uses**
Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker2_alt_def

concrete_definition *no_deadlock_certifier2* **uses**

Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier2_alt_def

concrete_definition *unreachability_checker3* **uses**
Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker3_alt_def

concrete_definition *no_deadlock_certifier3* **uses**
Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier3_alt_def

concrete_definition *no_buechi_run_checker* **uses**
Simple_Network_Impl_nat_ceiling_start_state.no_buechi_run_checker_alt_def

concrete_definition *check_prop_fail* **uses**
Simple_Network_Impl_nat_ceiling_start_state.check_prop_fail_alt_def

concrete_definition *check_deadlock_fail* **uses**
Simple_Network_Impl_nat_ceiling_start_state.check_deadlock_fail_alt_def

concrete_definition *check_invariant_fail* **uses**
Simple_Network_Impl_nat_ceiling_start_state.check_invariant_fail_alt_def

lemma *states'_memi_alt_def*:
Simple_Network_Impl_nat_defs.states'_memi broadcast bounds' automata = (
λ(*L*, *s*).
let
n_ps = *length automata*;
n_vs = *Simple_Network_Impl_Defs.n_vs bounds'*;
states_i = *map* (*Simple_Network_Impl_nat_defs.states_i automata*) [*0*..<*n_ps*]
in
length L = *n_ps* $\wedge$ ($\forall$ *i*<*n_ps*. *L* ! *i* $\in$ *states_i* ! *i*) $\wedge$
length s = *n_vs* $\wedge$ *Simple_Network_Impl_nat_defs.check_boundedi bounds' s*
)

unfolding *Simple_Network_Impl_nat_defs.states'_memi_def*
unfolding *Simple_Network_Impl_nat_defs.states_mem_compute'*
unfolding *Prod_TA_Defs.n_ps_def list_all_iff*
by (*intro ext*) *simp*

definition
certificate_checker_pre
L_list M_list broadcast bounds' automata m num_states num_actions k L_0 s_0 *formula*
$\equiv$
let
_ = *start_timer* ();
check1 = *Simple_Network_Impl_nat_ceiling_start_state*
broadcast bounds' automata m num_states num_actions k L_0 s_0 *formula*;
_ = *save_time STR ''Time to check ceiling''*;
n_ps = *length automata*;
n_vs = *Simple_Network_Impl_Defs.n_vs bounds'*;
states_i = *map* (*Simple_Network_Impl_nat_defs.states_i automata*) [*0*..<*n_ps*];
_ = *start_timer* ();
check2 = *list_all* (λ(*L*, *s*). *length L* = *n_ps* $\wedge$ ($\forall$ *i*<*n_ps*. *L* ! *i* $\in$ *states_i* ! *i*) $\wedge$
length s = *n_vs* $\wedge$ *Simple_Network_Impl_nat_defs.check_boundedi bounds' s*
) *L_list*;
_ = *save_time STR ''Time to check states''*;

```
    _ = start_timer ();
    n_sq = Suc m * Suc m;
    check3 = list_all (λ(l, xs). list_all (λM. length M = n_sq) xs) M_list;
    _ = save_time STR ''Time to check DBMs'';
    check4 = (case formula of formula.EX _ ⇒ True | _ ⇒ False)
  in check1 ∧ check2 ∧ check3 ∧ check4

definition
  certificate_checker num_split dc
    M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
    ////return states//return runs//return clocks/
  ≡
  let
    L_list = map fst M_list;
    check = certificate_checker_pre
      L_list M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
    ;////show c#//show clock//return clocks;////show st#//show state//return states/
//return runs/
  in if check then
  do {
    r ←
      if dc then
        no_deadlock_certifier
          broadcast bounds' automata m num_states num_actions L₀ s₀ L_list M_list num_split
      else
        unreachability_checker
            broadcast bounds' automata m num_states num_actions L₀ s₀ formula L_list M_list
num_split;
    Heap_Monad.return (Some r)
  } else Heap_Monad.return None

definition
  certificate_checker2 num_split dc
    M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  ≡
  let
    L_list = map fst M_list;
    check = certificate_checker_pre
      L_list M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  in if check then
    if dc then
        no_deadlock_certifier2
          broadcast bounds' automata m num_states num_actions L₀ s₀ L_list M_list num_split
      else
        unreachability_checker2
            broadcast bounds' automata m num_states num_actions L₀ s₀ formula L_list M_list
num_split
  else False

definition
  certificate_checker3 num_split dc
    M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  ≡
  let
```

```
    L_list = map fst M_list;
    check = certificate_checker_pre
      L_list M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  in if check then
    if dc then
       no_deadlock_certifier3
         broadcast bounds' automata m num_states num_actions L₀ s₀ L_list M_list num_split
      else
       unreachability_checker3
          broadcast bounds' automata m num_states num_actions L₀ s₀ formula L_list M_list
num_split
  else False

definition
  certificate_checker_dbg num_split dc
    (show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ char list))
    M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  ≡
  let
    _ = start_timer ();
    check1 = Simple_Network_Impl_nat_ceiling_start_state
      broadcast bounds' automata m num_states num_actions k L₀ s₀ formula;
    _ = save_time STR ''Time to check ceiling'';
    L_list = map fst M_list;
    n_ps = length automata;
    n_vs = Simple_Network_Impl_Defs.n_vs bounds';
    states_i = map (Simple_Network_Impl_nat_defs.states_i automata) [0..<n_ps];
    _ = start_timer ();
    check2 = list_all (λ(L, s). length L = n_ps ∧ (∀ i<n_ps. L ! i ∈ states_i ! i) ∧
      length s = n_vs ∧ Simple_Network_Impl_nat_defs.check_boundedi bounds' s
    ) L_list;
    _ = save_time STR ''Time to check states'';
    check3 = (case formula of formula.EX _ ⇒ True | _ ⇒ False)
  in if check1 ∧ check2 ∧ check3 then
  do {
    check_prop_fail broadcast bounds' automata m show_clock show_state L_list M_list;
    check_invariant_fail broadcast bounds' automata m
      num_states num_actions show_clock show_state L_list M_list;
    (if dc then
      check_deadlock_fail broadcast bounds' automata m
        num_states num_actions show_clock show_state L_list M_list
    else Heap_Monad.return ());
    r ← unreachability_checker
         broadcast bounds' automata m num_states num_actions L₀ s₀ formula L_list M_list
num_split;
    Heap_Monad.return (Some r)
  } else Heap_Monad.return None for show_clock show_state

definition
  print_fail s b ≡ if b then () else println (s + STR '' precondition check failed!'')

definition
  certificate_checker_pre1
    L_list M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
```

```
  ≡
  let
    _ = start_timer ();
    check1 = Simple_Network_Impl_nat_ceiling_start_state
      broadcast bounds' automata m num_states num_actions k L₀ s₀ formula;
    _ = save_time STR ''Time to check ceiling'';
    n_ps = length automata;
    n_vs = Simple_Network_Impl_Defs.n_vs bounds';
    states_i = map (Simple_Network_Impl_nat_defs.states_i automata) [0..<n_ps];
    _ = start_timer ();
    check2 = list_all (λ(L, s). length L = n_ps ∧ (∀ i<n_ps. L ! i ∈ states_i ! i) ∧
      length s = n_vs ∧ Simple_Network_Impl_nat_defs.check_boundedi bounds' s
    ) L_list;
    _ = save_time STR ''Time to check states'';
    _ = start_timer ();
    n_sq = Suc m * Suc m;
    check3 = list_all (λ(l, xs). list_all (λ(M, _). length M = n_sq) xs) M_list;
    _ = save_time STR ''Time to check DBMs'';
    check4 = (case formula of formula.EX _ ⇒ True | _ ⇒ False);
    _ = map (λ(a, b). print_fail a b) [
      (STR ''Ceiling'', check1),
      (STR ''States'',  check2),
      (STR ''DBM'',     check3),
      (STR ''Formula'', check4)
    ]
  in check1 ∧ check2 ∧ check3 ∧ check4

definition
  buechi_certificate_checker num_split
    M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  ≡
  let
    L_list = map fst M_list;
    check = certificate_checker_pre1
      L_list M_list broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
  in if check then
    no_buechi_run_checker
          broadcast bounds' automata m num_states num_actions L₀ s₀ formula L_list M_list
num_split
  else let _ = println STR ''Checking Buechi preconditions failed'' in False

theorem certificate_check:
  <emp> certificate_checker num_split False
    M_list broadcast bounds automata m num_states num_actions k L₀ s₀ formula
    ////////////////////////////////////////////////////////////////////////////
   <λ Some r ⇒ ↑(r ⟶ ¬ N broadcast automata bounds,(L₀, map_of s₀, λ_ . 0) ⊨ formula)
    | None ⇒ true>ₜ
proof −
  define A where A ≡ N broadcast automata bounds
  define check where check ≡ A,(L₀, map_of s₀, λ_ . 0) ⊨ formula
  { fix φ assume A: formula = formula.EX φ
    note
      Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker_hnr[
        of broadcast bounds automata m num_states num_actions k L₀ s₀ formula
```

```
      Simple_Network_Impl_nat_defs.states'_memi broadcast bounds automata
      map fst M_list M_list,
    folded A_def check_def,
    to_hnr, unfolded hn_refine_def, rule_format,
    OF _ _ _ _ _ A, unfolded A]
 } note *[sep_heap_rules] = this[simplified, unfolded hd_of_formulai.simps[abs_def]]
 show ?thesis
  unfolding A_def[symmetric] check_def[symmetric]
  unfolding certificate_checker_def certificate_checker_pre_def
  unfolding Let_def
 unfolding states'_memi_alt_def[unfolded Let_def, symmetric, of automata bounds broadcast]
 by (sep_auto simp: unreachability_checker.refine[symmetric] pure_def split: formula.split_asm)
qed

theorem certificate_deadlock_check:
 <emp> certificate_checker num_split True
  M_list broadcast bounds automata m num_states num_actions k L₀ s₀ formula
  ////////////////////////////////////////////////////////////////////////////
  <λ Some r ⇒ ↑(r ⟶ ¬ has_deadlock (N broadcast automata bounds) (L₀, map_of s₀, λ_ .
0))
  | None ⇒ true>t
proof −
 define A where A ≡ N broadcast automata bounds
 define check where check ≡ ¬ has_deadlock A (L₀, map_of s₀, λ_ . 0)
 note *[sep_heap_rules] = Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier_hnr[
    of broadcast bounds automata m num_states num_actions k L₀ s₀ formula
      map fst M_list M_list,
    folded A_def check_def,
    to_hnr, unfolded hn_refine_def, rule_format, unfolded hd_of_formulai.simps[abs_def]
 ]
 show ?thesis
  unfolding A_def[symmetric] check_def[symmetric]
  unfolding certificate_checker_def certificate_checker_pre_def
  unfolding Let_def
 unfolding states'_memi_alt_def[unfolded Let_def, symmetric, of automata bounds broadcast]
 by (sep_auto simp: no_deadlock_certifier.refine[symmetric] pure_def split: formula.split_asm)
qed

theorem certificate_check2:
 certificate_checker2 num_split False
  M_list broadcast bounds automata m num_states num_actions k L₀ s₀ formula
 ⟶ ¬ N broadcast automata bounds,(L₀, map_of s₀, λ_ . 0) ⊨ formula
 using Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker2_refine[
    of broadcast bounds automata m num_states num_actions k L₀ s₀ formula
      Simple_Network_Impl_nat_defs.states'_memi broadcast bounds automata
      map fst M_list M_list
      ]
 unfolding certificate_checker2_def certificate_checker_pre_def
 unfolding Let_def
 unfolding states'_memi_alt_def[unfolded Let_def, symmetric, of automata bounds broadcast]
 by (clarsimp split: formula.split_asm simp: unreachability_checker2.refine[symmetric])

theorem certificate_deadlock_check2:
 certificate_checker2 num_split True
```

M_list broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
$\longrightarrow \neg$ *has_deadlock* (*N broadcast automata bounds*) (L_0, *map_of* s_0, $\lambda\_$. *0*)
using *Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier2_refine*[
of broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
map fst M_list M_list num_split
]
unfolding *certificate_checker2_def certificate_checker_pre_def Let_def*
unfolding *states'_memi_alt_def*[*unfolded Let_def*, *symmetric*, *of automata bounds broadcast*]
by (*clarsimp simp*: *no_deadlock_certifier2.refine*[*symmetric*])

theorem *certificate_check3*:
certificate_checker3 num_split False
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
$\longrightarrow \neg$ *N broadcast automata bounds*,(L_0, *map_of* s_0, $\lambda\_$. *0*) $\models$ *formula*
using *Simple_Network_Impl_nat_ceiling_start_state.unreachability_checker3_refine*[
of broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
Simple_Network_Impl_nat_defs.states'_memi broadcast bounds automata
map fst M_list M_list
]
unfolding *certificate_checker3_def certificate_checker_pre_def*
unfolding *Let_def*
unfolding *states'_memi_alt_def*[*unfolded Let_def*, *symmetric*, *of automata bounds broadcast*]
by (*clarsimp split*: *formula.split_asm simp*: *unreachability_checker3.refine*[*symmetric*])

theorem *certificate_deadlock_check3*:
certificate_checker3 num_split True
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
$\longrightarrow \neg$ *has_deadlock* (*N broadcast automata bounds*) (L_0, *map_of* s_0, $\lambda\_$. *0*)
using *Simple_Network_Impl_nat_ceiling_start_state.no_deadlock_certifier3_refine*[
of broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
map fst M_list M_list
]
unfolding *certificate_checker3_def certificate_checker_pre_def Let_def*
unfolding *states'_memi_alt_def*[*unfolded Let_def*, *symmetric*, *of automata bounds broadcast*]
by (*clarsimp simp*: *no_deadlock_certifier3.refine*[*symmetric*])

fun *sexp_of_Ex* **where**
sexp_of_Ex (*formula.EX s*) = *s*

theorem *buechi_certificate_check*:
buechi_certificate_checker num_split
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
$\longrightarrow \neg$ *has_deadlock* (*N broadcast automata bounds*) (L_0, *map_of* s_0, $\lambda\_$. *0*)
$\longrightarrow (\exists \varphi.$ *formula* = *formula.EX* $\varphi) \wedge \neg$ *Graph_Defs.alw_ev*
($\lambda(L, s, u)$ (L', x, y).
(*N broadcast automata bounds*) $\vdash \langle L, s, u\rangle \rightarrow \langle L', x, y\rangle$)
($\lambda(L, s, \_)$. *check_sexp* (*sexp_of_Ex formula*) L (*the* $\circ$ s))
(L_0, *map_of* s_0, $\lambda\_$. *0*)
using *Simple_Network_Impl_nat_ceiling_start_state.no_buechi_run_checker_refine*[
of broadcast bounds automata m num_states num_actions k L_0 s_0 *formula*
Simple_Network_Impl_nat_defs.states'_memi broadcast bounds automata
map fst M_list M_list
]
unfolding *buechi_certificate_checker_def certificate_checker_pre1_def*

```
  unfolding Let_def
  unfolding states'_memi_alt_def[unfolded Let_def, symmetric, of automata bounds broadcast]
  by (clarsimp split: formula.split_asm simp: no_buechi_run_checker.refine[symmetric])

definition rename_check where
  rename_check num_split dc broadcast bounds' automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
    state_space
≡
do {
  let r = do_rename_mc (
      λ(show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ char list)).
      certificate_checker num_split dc state_space)
    dc broadcast bounds' automata k STR ''_urge'' L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
    (λ_ _. '' '') (λ_. '' '') (λ_. '' '');
  case r of Some r ⇒ do {
    r ← r;
    case r of
      None ⇒ Heap_Monad.return Preconds_Unsat
    | Some False ⇒ Heap_Monad.return Unsat
    | Some True ⇒ Heap_Monad.return Sat
  }
  | None ⇒ Heap_Monad.return Renaming_Failed
}


definition rename_check2 where
  rename_check2 num_split dc broadcast bounds' automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    state_space
≡
  let r = do_rename_mc (
      λ(show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ char list)).
      certificate_checker2 num_split dc state_space)
    dc broadcast bounds' automata k STR ''_urge'' L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    (λ_ _. '' '') (λ_. '' '') (λ_. '' '')
  in case r of
    Some False ⇒ Unsat
  | Some True ⇒ Sat
  | None ⇒ Renaming_Failed


definition rename_check3 where
  rename_check3 num_split dc broadcast bounds' automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    state_space
≡
  let r = do_rename_mc (
      λ(show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ string)).
      certificate_checker3 num_split dc state_space)
```

```
    dc broadcast bounds' automata k STR ''_urge'' L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    (λ_ _. '' '') (λ_. '' '') (λ_. '' '')
  in case r of
    Some False ⇒ Unsat
  | Some True ⇒ Sat
  | None ⇒ Renaming_Failed


definition rename_check_dbg where
  rename_check_dbg num_split dc broadcast bounds' automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
    state_space
≡
do {
  let r = do_rename_mc (
      λ(show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ string)).
      certificate_checker_dbg num_split dc show_clock show_state state_space)
    dc broadcast bounds' automata k STR ''_urge'' L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks;
  case r of Some r ⇒ do {
    r ← r;
    case r of
      None ⇒ Heap_Monad.return Preconds_Unsat
    | Some False ⇒ Heap_Monad.return Unsat
    | Some True ⇒ Heap_Monad.return Sat
  }
  | None ⇒ Heap_Monad.return Renaming_Failed
}


definition rename_check_buechi where
  rename_check_buechi num_split broadcast bounds' automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    state_space
≡
  let r = do_rename_mc (
      λ(show_clock :: (nat ⇒ string)) (show_state :: (nat list × int list ⇒ string)).
      buechi_certificate_checker num_split state_space)
    False broadcast bounds' automata k STR ''_urge'' L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    (λ_ _. '' '') (λ_. '' '') (λ_. '' '')
  in case r of
    Some False ⇒ Unsat
  | Some True ⇒ Sat
  | None ⇒ Renaming_Failed


theorem certificate_check_rename:
  <emp> rename_check num_split False broadcast bounds automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    inv_renum_states inv_renum_vars inv_renum_clocks
```

```
    state_space
    <λ Sat ⇒ ↑(
        (¬ N broadcast automata bounds,(L₀, map_of s₀, λ_ . 0) ⊨ formula))
     | Renaming_Failed ⇒ ↑(¬ Simple_Network_Rename_Formula
        broadcast bounds
        renum_acts renum_vars renum_clocks renum_states STR ''_urge''
        s₀ L₀ automata formula)
     | Unsat ⇒ true
     | Preconds_Unsat ⇒ true
    >_t (is ?A)
and certificate_check_rename2:
  case rename_check2 num_split False broadcast bounds automata k L₀ s₀ formula
      m num_states num_actions renum_acts renum_vars renum_clocks renum_states
      state_space
    of
      Sat ⇒ ¬ N broadcast automata bounds,(L₀, map_of s₀, λ_ . 0) ⊨ formula
    | Renaming_Failed ⇒ ¬ Simple_Network_Rename_Formula
          broadcast bounds
        renum_acts renum_vars renum_clocks renum_states STR ''_urge''
        s₀ L₀ automata formula
    | Unsat ⇒ True
    | Preconds_Unsat ⇒ True (is ?B)
and certificate_check_rename3:
  case rename_check3 num_split False broadcast bounds automata k L₀ s₀ formula
      m num_states num_actions renum_acts renum_vars renum_clocks renum_states
      state_space
    of
      Sat ⇒ ¬ N broadcast automata bounds,(L₀, map_of s₀, λ_ . 0) ⊨ formula
    | Renaming_Failed ⇒ ¬ Simple_Network_Rename_Formula
        broadcast bounds
        renum_acts renum_vars renum_clocks renum_states STR ''_urge''
        s₀ L₀ automata formula
    | Unsat ⇒ True
    | Preconds_Unsat ⇒ True (is ?C)
and buechi_check_rename:
  case rename_check_buechi num_split broadcast bounds automata k L₀ s₀ formula
      m num_states num_actions renum_acts renum_vars renum_clocks renum_states
      certificate
    of
      Sat ⇒
        ¬ has_deadlock (N broadcast automata bounds) (L₀, map_of s₀, λ_. 0) ⟶
        ¬ Graph_Defs.alw_ev
            (λ(L, s, u) (L', x, y). (N broadcast automata bounds) ⊢ ⟨L, s, u⟩ → ⟨L', x, y⟩)
            (λ(L, s, _). check_sexp (sexp_of_Ex formula) L (the ∘ s))
            (L₀, map_of s₀, λ_. 0)
    | Renaming_Failed ⇒ ¬ Simple_Network_Rename_Formula
        broadcast bounds
        renum_acts renum_vars renum_clocks renum_states STR ''_urge''
        s₀ L₀ automata formula
    | Unsat ⇒ True
    | Preconds_Unsat ⇒ True (is ?D)
proof −
  let ?urge = STR ''_urge''
  let ?automata = map (conv_urge ?urge) automata
```

have $*$:
Simple_Network_Rename_Formula_String
broadcast bounds
renum_acts renum_vars renum_clocks renum_states
automata ?urge formula s_0 L_0
= *Simple_Network_Rename_Formula*
broadcast bounds
renum_acts renum_vars renum_clocks renum_states
?urge s_0 L_0 *automata formula*

unfolding
Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
using *infinite_literal* **by** *auto*
define A **where** $A \equiv N$ *broadcast automata bounds*
define *check* **where** *check* $\equiv A$,(L_0, *map_of* s_0, $\lambda\_ .\ 0$) $\models$ *formula*
define A' **where** $A' \equiv N$
(*map renum_acts broadcast*)
(*map_index* (*renum_automaton renum_acts renum_vars renum_clocks renum_states*) *?automata*)
(*map* (λ(a,p). (*renum_vars* a, p)) *bounds*)
define *check'* **where** *check'* $\equiv$
A',(*map_index renum_states* L_0, *map_of* (*map* ($\lambda(x,\ v)$. (*renum_vars* x, v)) s_0), $\lambda\_ .\ 0$) $\models$
map_formula renum_states renum_vars id formula
define *deadlock_check* **where** *deadlock_check* $\equiv$ *has_deadlock* A (L_0, *map_of* s_0, $\lambda\_ .\ 0$)
define *deadlock_check'* **where** *deadlock_check'* $\equiv$
has_deadlock A' (*map_index renum_states* L_0, *map_of* (*map* ($\lambda(x,\ v)$. (*renum_vars* x, v)) s_0), $\lambda\_ .\ 0$)
define *buechi_check* **where** *buechi_check* $\equiv$
$\neg$ *Graph_Defs.alw_ev*
($\lambda(L,\ s,\ u)$ (L', x, y). $A \vdash \langle L,\ s,\ u\rangle \rightarrow \langle L',\ x,\ y\rangle$)
($\lambda(L,\ s,\ \_)$. *check_sexp* (*sexp_of_Ex formula*) L (*the* $\circ$ s))
(L_0, *map_of* s_0, $\lambda\_ .\ 0$)
define *buechi_check'* **where** *buechi_check'* $\equiv$
$\neg$ *Graph_Defs.alw_ev*
($\lambda(L,\ s,\ u)$ (L', x, y). $A' \vdash \langle L,\ s,\ u\rangle \rightarrow \langle L',\ x,\ y\rangle$)
($\lambda(L,\ s,\ \_)$. *check_sexp* (*sexp_of_Ex* (*map_formula renum_states renum_vars id formula*)) L (*the* $\circ$ s))
(*map_index renum_states* L_0, *map_of* (*map* ($\lambda(x,\ v)$. (*renum_vars* x, v)) s_0), $\lambda\_ .\ 0$)
define *renaming_valid* **where** *renaming_valid* $\equiv$
Simple_Network_Rename_Formula
broadcast bounds
renum_acts renum_vars renum_clocks renum_states STR ''_urge'' s_0 L_0 *automata formula*
define *formula_is_ex* **where**
formula_is_ex $\equiv \exists\varphi$. *map_formula renum_states renum_vars id formula* = *formula.EX* φ
have [*simp*]: *check* $\longleftrightarrow$ *check'* **if** *renaming_valid*
using *that* **unfolding** *check_def check'_def A_def A'_def renaming_valid_def*
by (*rule Simple_Network_Rename_Formula.models_iff'*[*symmetric*])
have [*simp*]: *buechi_check* $\longleftrightarrow$ *buechi_check'* **if** *renaming_valid formula_is_ex*
proof $-$
from *that*(*2*) **obtain** φ **where** *formula* = *formula.EX* φ
unfolding *formula_is_ex_def* **by** (*cases formula*) *auto*
with *that*(*1*) **show** *?thesis*
unfolding

```
    buechi_check_def buechi_check'_def check_def check'_def A_def A'_def renaming_valid_def
    by (simp add: Simple_Network_Rename_Formula.buechi_compatible'[symmetric])
 qed
 have **[simp]: deadlock_check ⟷ deadlock_check' if renaming_valid
  using that unfolding deadlock_check_def deadlock_check'_def A_def A'_def renaming_valid_def
   unfolding Simple_Network_Rename_Formula_def
   by − (elim conjE,
   rule Simple_Network_Language_Renaming.Simple_Network_Rename_Start.has_deadlock_iff'[symmetric])
 note [sep_heap_rules] =
   certificate_check[
   of num_split state_space
   map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
  map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states) ?automata
   m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x, v))
s₀
   map_formula renum_states renum_vars id formula,
   folded A'_def renaming_valid_def, folded check'_def, simplified
   ]
 show ?A ?B ?C ?D
   using certificate_check2[
     of num_split state_space
     map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states) ?automata
      m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x,
v)) s₀
     map_formula renum_states renum_vars id formula,
     folded A'_def renaming_valid_def, folded check'_def
   ]
   using certificate_check3[
     of num_split state_space
     map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states) ?automata
      m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x,
v)) s₀
     map_formula renum_states renum_vars id formula,
     folded A'_def renaming_valid_def, folded check'_def
   ]
   using buechi_certificate_check[
     of num_split certificate
     map renum_acts broadcast map (λ(a,p). (renum_vars a, p)) bounds
    map_index (renum_automaton renum_acts renum_vars renum_clocks renum_states) ?automata
      m num_states num_actions k map_index renum_states L₀ map (λ(x, v). (renum_vars x,
v)) s₀
     map_formula renum_states renum_vars id formula,
     folded A'_def renaming_valid_def,
     folded buechi_check'_def deadlock_check'_def formula_is_ex_def
   ]
   unfolding
     rename_check_def rename_check2_def rename_check3_def rename_check_buechi_def
     do_rename_mc_def rename_network_def
   unfolding if_False
    unfolding Simple_Network_Rename_Formula_String_Defs.check_renaming[symmetric] *
Let_def
   unfolding
```

```
    A_def[symmetric] renaming_valid_def[symmetric]
    check_def[symmetric] buechi_check_def[symmetric] deadlock_check_def[symmetric]
  by (sep_auto split: bool.splits)+
qed

theorem certificate_deadlock_check_rename:
  <emp> rename_check num_split True broadcast bounds automata k L₀ s₀ formula
    m num_states num_actions renum_acts renum_vars renum_clocks renum_states
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    state_space
    <λ Sat ⇒ ↑(¬ has_deadlock (N broadcast automata bounds) (L₀, map_of s₀, λ_ . 0))
     | Renaming_Failed ⇒ ↑(¬ Simple_Network_Rename_Formula
         broadcast bounds renum_acts renum_vars renum_clocks renum_states STR ''_urge'' s₀
L₀
         automata (formula.EX (sexp.not sexp.true)))
     | Unsat ⇒ true
     | Preconds_Unsat ⇒ true
    >t (is ?A)
and certificate_deadlock_check_rename2:
  case rename_check2 num_split True broadcast bounds automata k L₀ s₀ formula
      m num_states num_actions renum_acts renum_vars renum_clocks renum_states
      state_space
    of
      Sat ⇒ ¬ has_deadlock (N broadcast automata bounds) (L₀, map_of s₀, λ_ . 0)
    | Renaming_Failed ⇒ ¬ Simple_Network_Rename_Formula
         broadcast bounds renum_acts renum_vars renum_clocks renum_states STR ''_urge'' s₀
L₀
         automata (formula.EX (sexp.not sexp.true))
    | Unsat ⇒ True
    | Preconds_Unsat ⇒ True (is ?B)
and certificate_deadlock_check_rename3:
  case rename_check3 num_split True broadcast bounds automata k L₀ s₀ formula
      m num_states num_actions renum_acts renum_vars renum_clocks renum_states
      state_space
    of
      Sat ⇒ ¬ has_deadlock (N broadcast automata bounds) (L₀, map_of s₀, λ_ . 0)
    | Renaming_Failed ⇒ ¬ Simple_Network_Rename_Formula
         broadcast bounds renum_acts renum_vars renum_clocks renum_states STR ''_urge'' s₀
L₀
         automata (formula.EX (sexp.not sexp.true))
    | Unsat ⇒ True
    | Preconds_Unsat ⇒ True (is ?C)
proof −
  let ?formula = formula.EX (sexp.not sexp.true)
  let ?urge = STR ''_urge''
  let ?automata = map (conv_urge ?urge) automata
  have ∗:
    Simple_Network_Rename_Formula_String
        broadcast bounds
        renum_acts renum_vars renum_clocks renum_states
        automata ?urge ?formula s₀ L₀
  = Simple_Network_Rename_Formula
        broadcast bounds
        renum_acts renum_vars renum_clocks renum_states
```

?urge s_0 L_0 *automata ?formula*

unfolding
Simple_Network_Rename_Formula_String_def Simple_Network_Rename_Formula_def
Simple_Network_Rename_def Simple_Network_Rename_Formula_axioms_def
Simple_Network_Rename_Start_def Simple_Network_Rename_Start_axioms_def
using *infinite_literal* **by** *auto*
define *A* **where** *A* ≡ *N broadcast automata bounds*
define *check* **where** *check* ≡ *has_deadlock A* (L_0, *map_of* s_0, λ_ . *0*)
define *A'* **where** *A'* ≡ *N*
(*map renum_acts broadcast*)
(*map_index* (*renum_automaton renum_acts renum_vars renum_clocks renum_states*) *?automata*)
(*map* (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds*)
define *check'* **where** *check'* ≡
has_deadlock A' (*map_index renum_states* L_0, *map_of* (*map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0), λ_ . *0*)
define *renaming_valid* **where** *renaming_valid* ≡
Simple_Network_Rename_Formula
broadcast bounds renum_acts renum_vars renum_clocks renum_states ?urge s_0 L_0 *automata ?formula*

have ∗∗[*simp*]: *check* ⟷ *check'* **if** *renaming_valid*
using *that* **unfolding** *check_def check'_def A_def A'_def renaming_valid_def*
unfolding *Simple_Network_Rename_Formula_def*
by − (*elim conjE*,
rule Simple_Network_Language_Renaming.Simple_Network_Rename_Start.has_deadlock_iff'[*symmetric*])
note [*sep_heap_rules*] =
certificate_deadlock_check[
of num_split state_space
map renum_acts broadcast map (λ(*a*,*p*). (*renum_vars a*, *p*)) *bounds*
map_index (*renum_automaton renum_acts renum_vars renum_clocks renum_states*) *?automata*
m num_states num_actions k map_index renum_states L_0 *map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0
map_formula renum_states renum_vars id ?formula,
folded A'_def renaming_valid_def, *folded check'_def*, *simplified*
]
show *?A ?B ?C*
using *certificate_deadlock_check2*[
of num_split state_space
map renum_acts broadcast map (λ(*a*, *p*). (*renum_vars a*, *p*)) *bounds*
map_index (*renum_automaton renum_acts renum_vars renum_clocks renum_states*) *?automata*
m num_states num_actions k map_index renum_states L_0 *map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0
map_formula renum_states renum_vars id ?formula,
folded A'_def renaming_valid_def, *folded check'_def*, *simplified*
]
using *certificate_deadlock_check3*[
of num_split state_space
map renum_acts broadcast map (λ(*a*, *p*). (*renum_vars a*, *p*)) *bounds*
map_index (*renum_automaton renum_acts renum_vars renum_clocks renum_states*) *?automata*
m num_states num_actions k map_index renum_states L_0 *map* (λ(*x*, *v*). (*renum_vars x*, *v*)) s_0
map_formula renum_states renum_vars id ?formula,
folded A'_def renaming_valid_def, *folded check'_def*, *simplified*

]
unfolding
rename_check_def rename_check2_def rename_check3_def do_rename_mc_def rename_network_def
unfolding *if_True*
unfolding *Simple_Network_Rename_Formula_String_Defs.check_renaming*[*symmetric*] ∗
Let_def
unfolding
A_def[*symmetric*] *check_def*[*symmetric*] *renaming_valid_def*[*symmetric*]
by (*sep_auto split*: *bool.splits*)+
qed

lemmas [*code*] = *Simple_Network_Impl_nat_defs.states_i_def*

export_code *rename_check rename_check2 rename_check3 rename_check_buechi* **in** *SML* **module_name** *Test*

export_code *rename_check_dbg* **in** *SML* **module_name** *Test*

end

12 Assembling the Checker and Generating Code

We provide some optimized code equations.
Then we assemble the certificate checker:

- A parser is added to parse JSON files
- The resulting JSON data is validated and converted to the simple networks language
- The (binary) certificate is read
- There is an additional "renaming" that maps identifiers between the input and the certificate
- The certficiate, the renaming, and the model are passed to the checker
- and the output is printed
  Then the code is exported:
- We add some logging utilities (via code printings)
- Code generation is setup
- We export this to SML via reflection
- And test it on a few small examples

theory *Simple_Network_Language_Certificate_Code*
imports
Simple_Network_Language_Certificate
Simple_Network_Language_Certificate_Checking
begin

Optimized code equations **lemmas** [*code_unfold*] = *imp_for_imp_for'*

definition *dbm_subset'_impl'_int*
:: *nat* ⇒ *int DBMEntry Heap.array* ⇒ *int DBMEntry Heap.array* ⇒ *bool Heap*
where [*symmetric*, *int_folds*]:
dbm_subset'_impl'_int = *dbm_subset'_impl'*

lemma *less_eq_dbm_le_int*[*int_folds*]:
(≤) = *dbm_le_int*
unfolding *dbm_le_dbm_le_int less_eq* **..**

schematic_goal *dbm_subset'_impl'_int_code*[*code*]:
dbm_subset'_impl'_int ≡ λ*m a b*.
do {
imp_for 0 ((*m* + *1*) ∗ (*m* + *1*)) *Heap_Monad.return*
(λ*i* _. *do* {
x ← *Array.nth a i*; *y* ← *Array.nth b i*; *Heap_Monad.return* (*dbm_le_int x y*)
})
True
}

unfolding *dbm_subset'_impl'_int_def*[*symmetric*] *dbm_subset'_impl'_def int_folds* **.**

definition *dbm_subset_impl_int*
:: *nat* ⇒ *int DBMEntry Heap.array* ⇒ *int DBMEntry Heap.array* ⇒ *bool Heap*
where [*symmetric*, *int_folds*]:
dbm_subset_impl_int = *dbm_subset_impl*

schematic_goal *dbm_subset_impl_int_code*[*code*]:
dbm_subset_impl_int ≡ *?i*
unfolding *dbm_subset_impl_int_def*[*symmetric*] *dbm_subset_impl_def*
unfolding *int_folds*
.

lemmas [*code_unfold*] = *int_folds*

export_code *state_space* **in** *SML* **module_name** *Test*

hide_const *Parser_Combinator.return*
hide_const *Error_Monad.return*

definition
show_lit = *String.implode o show*

definition *rename_state_space* ≡ λ*dc ids_to_names* (*broadcast*, *automata*, *bounds*) L_0 s_0 *formula*.
let _ = *println* (*STR ''Make renaming''*) *in*
do {
(*m*, *num_states*, *num_actions*, *renum_acts*, *renum_vars*, *renum_clocks*, *renum_states*,
inv_renum_states, *inv_renum_vars*, *inv_renum_clocks*)
← *make_renaming broadcast automata bounds*;
let _ = *println* (*STR ''Renaming''*);
let (*broadcast'*, *automata'*, *bounds'*) = *rename_network*

```
    broadcast bounds automata renum_acts renum_vars renum_clocks renum_states;
   let _ = println (STR ''Calculating ceiling'');
  let k = Simple_Network_Impl_nat_defs.local_ceiling broadcast' bounds' automata' m num_states;
   let _ = println (STR ''Running model checker'');
   let inv_renum_states' = (λi. ids_to_names i o inv_renum_states i);
   let f = (λshow_clock
     show_state broadcast bounds' automata m num_states num_actions k L₀ s₀ formula.
     state_space broadcast bounds' automata m num_states num_actions k L₀ s₀ formula
       show_clock show_state ()
   );
   let r = do_rename_mc f dc broadcast bounds automata k STR ''_urge'' L₀ s₀ formula
     m num_states num_actions renum_acts renum_vars renum_clocks renum_states
     inv_renum_states' inv_renum_vars inv_renum_clocks;
   let show_clock = show o inv_renum_clocks;
  let show_state = (show_state :: _ ⇒ _ ⇒ _ × int list ⇒ _) inv_renum_states inv_renum_vars;
   let renamings =
     (m, num_states, num_actions, renum_acts, renum_vars, renum_clocks, renum_states,
      inv_renum_states, inv_renum_vars, inv_renum_clocks
     );
   Result (r, show_clock, show_state, renamings, k)
 }

definition
  check_subsumed n xs (i :: int) M ≡
  do {
   (_, b) ← imp_nfoldli xs (λ(_, b). return (¬ b)) (λM' (j, b).
     if i = j then return (j + 1, b) else do {
       b ← dbm_subset'_impl n M M';
       if b then return (j, True) else return (j + 1, False)
     }
   ) (0, False);
   return b
  }


definition
  imp_filter_index P xs = do {
  (_, xs) ← imp_nfoldli xs (λ_. return True) (λx (i :: nat, xs).
    do {
      b ← P i x;
      return (i + 1, if b then (x # xs) else xs)
    }
  ) (0, []);
  return (rev xs)
  }

definition
  filter_dbm_list n xs =
    imp_filter_index (λi M. do {b ← check_subsumed n xs i M; return (¬ b)}) xs

partial_function (heap) imp_map :: ('a ⇒ 'b Heap) ⇒ 'a list ⇒ 'b list Heap where
  imp_map f xs =
  (if xs = [] then return [] else do {y ← f (hd xs); ys ← imp_map f (tl xs); return (y # ys)})
```

```
lemma imp_map_simps[code, simp]:
  imp_map f [] = return []
  imp_map f (x # xs) = do {y ← f x; ys ← imp_map f xs; return (y # ys)}
  by (simp add: imp_map.simps)+

definition trace_state where
  trace_state n show_clock show_state ≡
  λ (l, M). do {
      let st = show_state l;
      m ← show_dbm_impl n show_clock show M;
      let s = ''('' @ st @ '', ['' @ m @ ''])'';
      let s = String.implode s;
      let _ = println s;
      return ()
  }
 for show_clock show_state

definition
  show_str = String.implode o show

definition parse_convert_run_print where
  parse_convert_run_print dc s ≡
   case parse json s ⤜ convert of
     Error es ⇒ do {let _ = map println es; return ()}
   | Result (ids_to_names, _, broadcast, automata, bounds, formula, L₀, s₀) ⇒ do {
     let r = rename_state_space dc ids_to_names (broadcast, automata, bounds) L₀ s₀ formula;
     case r of
       Error es ⇒ do {let _ = map println es; return ()}
     | Result (r, show_clk, show_st, renamings, k) ⇒
       case r of None ⇒ return () | Some r ⇒
       do {
         r ← r;
         let _ = STR ''Number of discrete states: '' + (length r |> show_str) |> println;
         let _ =
           STR ''Size of passed list: '' + show_str (sum_list (map (length o snd) r)) |> println;
         let n = Simple_Network_Impl.clk_set' automata |> list_of_set |> length;
         r ← imp_map (λ (a, b). do {
             b ← imp_map (return o snd) b; b ← filter_dbm_list n b; return (a, b)
           }) r;
         let _ = STR ''Number of discrete states: '' + show_str (length r) |> println;
         let _ = STR ''Size of passed list after removing subsumed states: ''
           + show_str (sum_list (map (length o snd) r)) |> println;
         let show_dbm = (λM. do {
           s ← show_dbm_impl_all n show_clk show M;
           return (''<'' @ s @ ''>'')
         });
         _ ← imp_map (λ (s, xs).
         do {
           let s = show_st s;
           xs ← imp_map show_dbm xs;
           let _ = s @ '': '' @ show xs |> String.implode |> println;
           return ()
         }
         ) r;
```

```
      return ()
    }
  }
```

ML ‹
```
structure Timing : sig
  val start_timer: unit -> unit
  val save_time: string -> unit
  val get_timings: unit -> (string * Time.time) list
end = struct
  val t = Unsynchronized.ref Time.zeroTime;
  val timings = Unsynchronized.ref [];
  fun start_timer () = (t := Time.now ());
  fun get_elapsed () = Time.- (Time.now (), !t);
  fun save_time s = (timings := ((s, get_elapsed ()) :: !timings));
  fun get_timings () = !timings;
end
```
›

ML ‹
```
fun print_timings () =
  let
    val tab = Timing.get_timings ();
    fun print_time (s, t) = writeln(s ^ ": " ^ Time.toString t);
  in map print_time tab end;
```
›

code_printing
```
  constant Show_State_Defs.tracei ⇀
      (SML)  (fn n => fn show_state => fn show_clock => fn typ => fn x => ()) _ _ _
  and (OCaml) (fun n show_state show_clock ty x -> ()) _ _ _
```

datatype *mode* = *Impl1* | *Impl2* | *Impl3* | *Buechi* | *Debug*

definition
```
  distr xs ≡
  let (m, d) =
  fold
    (λx (m, d). case m x of None ⇒ (m(x ↦ 1:: nat), x # d) | Some y ⇒ (m(x ↦ (y + 1)), d))
    xs (Map.empty, [])
  in map (λx. (x, the (m x))) (sort d)
```

definition *split_k''* :: *nat* ⇒ (*'a* × *'b list*) *list* ⇒ (*'a* × *'b list*) *list list* **where**
```
  split_k'' k xs ≡ let
    width = sum_list (map (length o snd) xs) div k;
    width = (if length xs mod k = 0 then width else width + 1)
  in split_size (length o snd) width 0 [] xs
```

definition
```
  print_errors es = do {Heap_Monad.fold_map print_line_impl es; return ()}
```

We can actually let Munta produce reachability certificates for Muntac. This is what the following code does. A few test cases are run below right in the Isabelle/ML environment.

definition *parse_convert_run_check* **where**

```
parse_convert_run_check mode num_split dc s ≡
 case parse json s ≫= convert of
   Error es ⇒ print_errors es
 | Result (ids_to_names, _, broadcast, automata, bounds, formula, L₀, s₀) ⇒ do {
    let r = rename_state_space dc ids_to_names (broadcast, automata, bounds) L₀ s₀ formula;
    case r of
      Error es ⇒ print_errors es
    | Result (r, show_clk, show_st, renamings, k) ⇒
      case r of None ⇒ return () | Some r ⇒ do {
      let t = now ();
      r ← r;
      let t = now () − t;
      print_line_impl
        (STR ''Time for model checking + certificate extraction: '' + time_to_string t);
      let (m,num_states,num_actions,renum_acts,renum_vars,renum_clocks,renum_states,
        inv_renum_states, inv_renum_vars, inv_renum_clocks
      ) = renamings;
      let _ = start_timer ();
      state_space ← Heap_Monad.fold_map (λ(s, xs).
        do {
          let xs = map snd xs;
          xs ← Heap_Monad.fold_map (dbm_to_list_impl m) xs;
          return (s, xs)
        }
      ) r;
      let _ = save_time STR ''Time for converting DBMs in certificate'';
      print_line_impl
        (STR ''Number of discrete states of state space: '' + show_lit (length state_space));
      let _ = STR ''Size of passed list: '' + show_str (sum_list (map (length o snd) r))
        |> println;
      STR ''DBM list length distribution: '' + show_str (distr (map (length o snd) state_space))
        |> print_line_impl;
      let split =
      (if mode = Impl3 then split_k'' num_split state_space else split_k num_split state_space);
      let split_distr = map (sum_list o map (length o snd)) split;
      STR ''Size of passed list distribution after split: '' + show_str split_distr
        |> print_line_impl;
      let t = now ();
      check ← case mode of
        Debug ⇒ rename_check_dbg num_split dc broadcast bounds automata k L₀ s₀ formula
          m num_states num_actions renum_acts renum_vars renum_clocks renum_states
          inv_renum_states inv_renum_vars inv_renum_clocks
          state_space
      | Impl1 ⇒ rename_check num_split dc broadcast bounds automata k L₀ s₀ formula
          m num_states num_actions renum_acts renum_vars renum_clocks renum_states
          state_space
      | Impl2 ⇒ rename_check2 num_split dc broadcast bounds automata k L₀ s₀ formula
          m num_states num_actions renum_acts renum_vars renum_clocks renum_states
          state_space |> return
      | Impl3 ⇒ rename_check3 num_split dc broadcast bounds automata k L₀ s₀ formula
          m num_states num_actions renum_acts renum_vars renum_clocks renum_states
          state_space |> return;
      let t = now () − t;
      print_line_impl (STR ''Time for certificate checking: '' + time_to_string t);
```

```
      case check of
        Renaming_Failed ⇒ print_line_impl (STR ''Renaming failed'')
      | Preconds_Unsat ⇒ print_line_impl (STR ''Preconditions were not met'')
      | Sat ⇒ print_line_impl (STR ''Certificate was accepted'')
      | Unsat ⇒ print_line_impl (STR ''Certificate was rejected'')
      }
  }

ML ‹
  fun do_test dc file =
  let
    val dir = @{master_dir} + @{path benchmarks/}
    val file = dir + Path.explode file
    val s = File.read file
  in
    @{code parse_convert_run_print} dc s end
›

ML_val ‹
  do_test true HDDI_02.muntax ()
›

ML_val ‹
  do_test true simple.muntax ()
›

ML_val ‹
  do_test true light_switch.muntax ()
›

ML_val ‹
  do_test true PM_test.muntax ()
›

ML_val ‹
  do_test true bridge.muntax ()
›

code_printing
  constant parallel_fold_map ⇀
    (SML)   (fn f => fn xs => fn () => Par'_List.map (fn x => f x ()) xs) _ _

definition
  num_split ≡ 4 :: nat

ML ‹
  fun do_check dc file =
  let
    val dir = @{master_dir} + @{path benchmarks/}
    val file = dir + Path.explode file
    val s = File.read file
  in
    @{code parse_convert_run_check} @{code Impl3} @{code num_split} dc s end
›
```

ML_val ‹
do_check false HDDI_02.muntax ()
›

ML_val ‹
do_check true HDDI_02.muntax ()
›

ML_val ‹
do_check true simple.muntax ()
›

ML_val ‹
do_check true light_switch.muntax ()
›

ML_val ‹
do_check false PM_test.muntax ()
›

ML_val ‹
do_check true PM_test.muntax ()
›

ML_val ‹
do_check true bridge.muntax ()
›

ML_val ‹
do_check false PM_all_3.muntax ()
›

ML_val ‹
do_check true PM_all_3.muntax ()
›

Executing *Heap_Monad.fold_map* in parallel in Isabelle/ML

definition
‹*Test* ≡ *Heap_Monad.fold_map* (λ*x*. *do* {*let* _ = *println STR* ''*x*''; *return x*}) ([*1*,*2*,*3*] :: *nat list*)›

ML_val ‹@{*code Test*} ()›

Checking external certificates **definition**
list_of_json_object obj ≡
case obj of
Object m ⇒ *Result m*
| _ ⇒ *Error* [*STR* ''*Not an object*'']

definition
map_of_debug prefix m ≡
let m = *map_of m in*

```
    (λx.
      case m x of
        None ⇒ let _ = println (STR ''Key error('' + prefix + STR ''): '' + show_lit x) in None
      | Some v ⇒ Some v) for prefix

definition
  renaming_of_json' opt prefix json ≡ do {
    vars ← list_of_json_object json;
    vars ← combine_map (λ (a, b). do {b ← of_nat b; Result (String.implode a, b)}) vars;
    let vars = vars @ (case opt of Some x ⇒ [(x, fold max (map snd vars) 0 + 1)] | _ ⇒ []);
    Result (the o map_of_debug prefix vars, the o map_of_debug prefix (map prod.swap vars))
  }
   for json prefix

definition renaming_of_json ≡ renaming_of_json' None

definition
  nat_renaming_of_json prefix max_id json ≡ do {
    vars ← list_of_json_object json;
    vars ← combine_map (λ(a, b). do {
      a ← parse lx_nat (String.implode a);
      b ← of_nat b;
      Result (a, b)
    }) vars;
    let ids = fst ‘ set vars;
    let missing = filter (λi. i ∉ ids) [0..<max_id];
    let m = the o map_of_debug prefix vars;
    let m = extend_domain m missing (length vars);
    Result (m, the o map_of_debug prefix (map prod.swap vars))
  }
   for json prefix

definition convert_renaming ::
  (nat ⇒ nat ⇒ String.literal) ⇒ (String.literal ⇒ nat) ⇒ JSON ⇒ _ where
  convert_renaming ids_to_names process_names_to_index json ≡ do {
    json ← of_object json;
    vars ← get json ''vars'';
    (var_renaming, var_inv) ← renaming_of_json STR ''var renaming'' vars;
    clocks ← get json ''clocks'';
    (clock_renaming, clock_inv)
      ← renaming_of_json' (Some STR ''_urge'') STR ''clock renaming'' clocks;
    processes ← get json ''processes'';
    (process_renaming, process_inv) ← renaming_of_json STR ''process renaming'' processes;
    locations ← get json ''locations'';
    locations ← list_of_json_object locations; — process name → json
    locations ← combine_map (λ(name, renaming). do {
        let p_num = process_names_to_index (String.implode name);
        assert
          (process_renaming (String.implode name) = p_num)
          (STR ''Process renamings do not agree on '' + String.implode name);
        let max_id = 1000;
       renaming ← nat_renaming_of_json (STR ''process'' + show_str p_num) max_id renaming;
          — location id → nat
```

```
      Result (p_num, renaming)
    }
    ) locations;
  — process id → location id → nat
   let location_renaming = the o map_of_debug STR ''location'' (map (λ(i, f, _). (i, f)) locations);
   let location_inv = the o map_of_debug STR ''location inv'' (map (λ(i, _, f). (i, f)) locations);
   Result (var_renaming, clock_renaming, location_renaming, var_inv, clock_inv, location_inv)
  }

  for json

definition
  load_renaming dc model renaming ≡
  case
  do {
    model ← parse json model;
    renaming ← parse json renaming;
    (ids_to_names, process_names_to_index, broadcast, automata, bounds, formula, L_0, s_0)
      ← convert model;
    convert_renaming ids_to_names process_names_to_index renaming
  }
  of
    Error e ⇒ return (Error e)
  | Result r ⇒ do {
    let (var_renaming, clock_renaming, location_renaming, _, _, _) = r;
    let _ = map (λp. map (λn. location_renaming p n |> show_lit |> println) [0..<8]) [0..<6];
    return (Result ())
  }


definition parse_compute where
  parse_compute model renaming ≡
   do {
    model ← parse json model;
    (ids_to_names, process_names_to_index, broadcast, automata, bounds, formula, L_0, s_0)
      ← convert model;
    renaming ← parse json renaming;
    (var_renaming, clock_renaming, location_renaming,
     inv_renum_vars, inv_renum_clocks, inv_renum_states)
      ← convert_renaming ids_to_names process_names_to_index renaming;
    (m, num_states, num_actions, renum_acts, renum_vars, renum_clocks, renum_states, _,
_, _)
      ← make_renaming broadcast automata bounds;
   assert (renum_clocks STR ''_urge'' = m) STR ''Computed renaming: _urge is not last clock!'';
    let renum_vars = var_renaming;
    let renum_clocks = clock_renaming;
    let renum_states = location_renaming;
    assert (renum_clocks STR ''_urge'' = m) STR ''Given renaming: _urge is not last clock!'';
    let _ = println (STR ''Renaming'');
    let (broadcast', automata', bounds') = rename_network
      broadcast bounds automata renum_acts renum_vars renum_clocks renum_states;
    let _ = println (STR ''Calculating ceiling'');
   let k = Simple_Network_Impl_nat_defs.local_ceiling broadcast' bounds' automata' m num_states;
```

```
    let urgent_locations = map (λ(_, urgent, _, _). urgent) automata';
    Result (broadcast, bounds, automata, urgent_locations, k, L_0, s_0, formula,
        m, num_states, num_actions, renum_acts, renum_vars, renum_clocks, renum_states,
        inv_renum_states, inv_renum_vars, inv_renum_clocks)
  } for num_split

datatype 'l state_space =
    Reachable_Set (reach_of: (('l list × int list) × int DBMEntry list list) list)
  | Buechi_Set    (buechi_of: (('l list × int list) × (int DBMEntry list × nat) list) list)

definition normalize_dbm :: nat ⇒ int DBMEntry list ⇒ int DBMEntry list where
  normalize_dbm m xs = do {
    dbm ← Array.of_list xs;
    dbm ← fw_impl_int m dbm;
    Array.freeze dbm
  } |> run_heap


definition insert_every_nth :: nat ⇒ 'a ⇒ 'a list ⇒ 'a list where
  insert_every_nth n a xs ≡
    fold (λx (i, xs). if i = n then (1, a # x # xs) else (i + 1, x # xs)) xs (1, []) |> snd |> rev

definition convert_dbm where
  convert_dbm urge m dbm =
    take m dbm @ Le 0 #
    insert_every_nth m DBMEntry.INF (drop m dbm)
    @ (if urge then Le 0 else DBMEntry.INF) # replicate (m − 1) DBMEntry.INF @ [Le 0]
  |> normalize_dbm m

fun convert_state_space :: _ ⇒ _ ⇒ int state_space ⇒ nat state_space where
  convert_state_space m is_urgent (Reachable_Set xs) = Reachable_Set (
    map (λ((locs, vars), dbms).
      ((map nat locs, vars), map (convert_dbm (is_urgent (locs, vars)) m) dbms))
    xs)
| convert_state_space m is_urgent (Buechi_Set xs) = Buechi_Set (
    map (λ((locs, vars), dbms).
      ((map nat locs, vars), map (λ(dbm, i). (convert_dbm (is_urgent (locs, vars)) m dbm, i))
dbms))
    xs)

fun len_of_state_space where
  len_of_state_space (Reachable_Set xs) = length xs
| len_of_state_space (Buechi_Set xs) = length xs

context
  fixes num_clocks :: nat
  fixes inv_renum_states :: nat ⇒ nat ⇒ nat
    and inv_renum_vars   :: nat ⇒ String.literal
    and inv_renum_clocks :: nat ⇒ String.literal
begin

private definition show_st where
  show_st = show_state inv_renum_states inv_renum_vars
```

```
private definition show_dbm :: int DBMEntry list ⇒ char list where
  show_dbm = dbm_list_to_string num_clocks (show_clock inv_renum_clocks) show

fun show_state_space where
  show_state_space (Reachable_Set xs) =
    map (λ(l, xs).
      map (λx. ''('' @ show_st l @ '', <'' @ show_dbm x @ ''>)'' |> String.implode |> println)
xs)
    xs
| show_state_space (Buechi_Set xs) =
    map (λ(l, xs).
      map (λ(x, i). show i @ '': ('' @ show_st l @ '', <'' @ show_dbm x @ ''>)''
            |> String.implode |> println)
      xs)
    xs

end

definition
  print_sep ≡ λ(). println (String.implode (replicate 100 CHR ''−''))

definition parse_convert_check where
  parse_convert_check mode num_split dc model renaming state_space show_cert ≡
   let
     r = parse_compute model renaming
   in case r of Error es ⇒ do {let _ = map println es; return ()}
   | Result r ⇒ do {
      let (broadcast, bounds, automata, urgent_locations, k, L_0, s_0, formula,
        m, num_states, num_actions, renum_acts, renum_vars, renum_clocks, renum_states,
        inv_renum_states, inv_renum_vars, inv_renum_clocks) = r;
      let is_urgent = (λ(L, _). list_ex (λ(l, urgent). l ∈ set urgent) (zip L urgent_locations));
      let inv_renum_clocks = (λi. if i = m then STR ''_urge'' else inv_renum_clocks i);
      let t = now ();
      let state_space = convert_state_space m is_urgent state_space;
      let t = now () − t;
      let _ = println (STR ''Time for converting state space: '' + time_to_string t);
      let _ = start_timer ();
      let _ = save_time STR ''Time for converting DBMs in certificate'';
      let _ =
        println (STR ''Number of discrete states: ''+ show_lit (len_of_state_space state_space));
      let _ = do {
        if show_cert then do {
          let _ = print_sep ();
          let _ = println (STR ''Certificate'');
          let _ = print_sep ();
            let _ = show_state_space m inv_renum_states inv_renum_vars inv_renum_clocks
state_space;
          let _ = print_sep ();
          return ()}
        else return ()
      };
      let t = now ();
      check ← case mode of
        Debug ⇒ rename_check_dbg num_split dc broadcast bounds automata k L_0 s_0 formula
```

```
        m num_states num_actions renum_acts renum_vars renum_clocks renum_states
        inv_renum_states inv_renum_vars inv_renum_clocks
        (reach_of state_space)
    | Impl1 ⇒ rename_check num_split dc broadcast bounds automata k L₀ s₀ formula
        m num_states num_actions renum_acts renum_vars renum_clocks renum_states
        (reach_of state_space)
    | Impl2 ⇒ rename_check2 num_split dc broadcast bounds automata k L₀ s₀ formula
        m num_states num_actions renum_acts renum_vars renum_clocks renum_states
        (reach_of state_space) |> return
    | Impl3 ⇒ rename_check3 num_split dc broadcast bounds automata k L₀ s₀ formula
        m num_states num_actions renum_acts renum_vars renum_clocks renum_states
        (reach_of state_space) |> return
    | Buechi ⇒ rename_check_buechi num_split broadcast bounds automata k L₀ s₀ formula
        m num_states num_actions renum_acts renum_vars renum_clocks renum_states
        (buechi_of state_space) |> return;
    let t = now () − t;
    let _ = println (STR ''Time for certificate checking: '' + time_to_string t);
    case check of
      Renaming_Failed ⇒ do {let _ = println STR ''Renaming failed''; return ()}
    | Preconds_Unsat ⇒ do {let _ = println STR ''Preconditions were not met''; return ()}
    | Sat ⇒ do {let _ = println STR ''Certificate was accepted''; return ()}
    | Unsat ⇒ do {let _ = println STR ''Certificate was rejected''; return ()}
  }
 for num_split and state_space :: int state_space


code_printing
  constant IArray.length' ⇀ (SML) (IntInf.fromInt o Vector.length)

code_printing
  constant Parallel.map ⇀ (SML) Par'_List.map

lemma [code]: run_map_heap f xs = Parallel.map (run_heap o f) xs
  unfolding run_map_heap_def Parallel.map_def ..

code_printing code_module Timing ⇀ (SML)
‹
structure Timing : sig
  val start_timer: unit -> unit
  val save_time: string -> unit
  val get_timings: unit -> (string * Time.time) list
  val set_cpu: bool -> unit
end = struct

  open Timer;

  val is_cpu = Unsynchronized.ref false;
  fun set_cpu b = is_cpu := b;

  val cpu_timer: cpu_timer option Unsynchronized.ref = Unsynchronized.ref NONE;
  val real_timer: real_timer option Unsynchronized.ref = Unsynchronized.ref NONE;

  val timings = Unsynchronized.ref [];
  fun start_timer () = (
```

```
    if !is_cpu then
      cpu_timer := SOME (startCPUTimer ())
    else
      real_timer := SOME (startRealTimer ()));
  fun get_elapsed () = (
    if !is_cpu then
      #usr (!cpu_timer |> the |> checkCPUTimer)
    else
      (!real_timer |> the |> checkRealTimer));
  fun save_time s = (timings := ((s, get_elapsed ()) :: !timings));
  fun get_timings () = !timings;
end
›
```

Optimized code printings definition
array_freeze′ = array_freeze

lemma [*code*]: *array_freeze a = array_freeze′ a*
unfolding *array_freeze′_def* **..**

definition
array_unfreeze′ = array_unfreeze

lemma [*code*]: *array_unfreeze a = array_unfreeze′ a*
unfolding *array_unfreeze′_def* **..**

definition
array_copy′ = array_copy

lemma [*code*]: *array_copy a = array_copy′ a*
unfolding *array_copy′_def* **..**

code_printing constant *array_freeze′* ⇀ (*SML*) (*fn* () => *Array.vector* _)

code_printing constant *array_unfreeze′* ⇀ (*SML*)
(*fn a* => *fn* () => *Array.tabulate* (*Vector.length a*, *fn i* => *Vector.sub* (*a*, *i*))) _

code_printing constant *array_copy′* ⇀ (*SML*)
(*fn a* => *fn* () => *Array.tabulate* (*Array.length a*, *fn i* => *Array.sub* (*a*, *i*))) _

According to microbenchmarks, these versions are nearly two times faster.

code_printing constant *array_unfreeze′* ⇀ (*SML*)
```
(fn a => fn () =>
  if Vector.length a = 0
  then Array.fromList []
  else
    let
      val x = Vector.sub(a, 0)
      val n = Array.array (Vector.length a, x)
      val '_ = Array.copyVec {src=a,dst=n,di=0}
    in n end
) _
```

code_printing constant *array_copy′* ⇀ (*SML*)

```
(fn a => fn () =>
  if Array.length a = 0
  then Array.fromList []
  else
    let
      val x = Array.sub(a, 0)
      val n = Array.array (Array.length a, x)
      val '_ = Array.copy {src=a,dst=n,di=0}
    in n end
) _


partial_function (heap) imp_for_int_inner ::
  integer ⇒ integer ⇒ ('a ⇒ bool Heap) ⇒ (integer ⇒ 'a ⇒ 'a Heap) ⇒ 'a ⇒ 'a Heap where
  imp_for_int_inner i u c f s = (if i ≥ u then return s else
    do {ctn <- c s; if ctn then f i s ⤜ imp_for_int_inner (i + 1) u c f else return s})

lemma integer_of_nat_le_simp:
  integer_of_nat i ≤ integer_of_nat u ⟷ i ≤ u
  unfolding integer_of_nat_eq_of_nat by simp

lemma imp_for_imp_for_int_inner[code_unfold]:
  imp_for i u c f s
  = imp_for_int_inner (integer_of_nat i) (integer_of_nat u) c (f o nat_of_integer) s
  apply (induction u - i arbitrary: i u s)
   apply (simp add: integer_of_nat_le_simp imp_for_int_inner.simps; fail)
  apply (subst imp_for_int_inner.simps, simp add: integer_of_nat_le_simp)
  apply safe
  subgoal
    by auto
  apply (fo_rule arg_cong, rule ext)
  apply clarsimp
  apply (fo_rule arg_cong)
  apply (auto simp: algebra_simps integer_of_nat_eq_of_nat)
  done

definition
  imp_for_int i u c f s ≡
  imp_for_int_inner (integer_of_nat i) (integer_of_nat u) c (f o nat_of_integer) s

lemma imp_for_imp_for_int[code_unfold]:
  imp_for = imp_for_int
  by (intro ext, unfold imp_for_int_def imp_for_imp_for_int_inner, rule HOL.refl)

partial_function (heap) imp_for'_int_inner ::
  integer ⇒ integer ⇒ (integer ⇒ 'a ⇒ 'a Heap) ⇒ 'a ⇒ 'a Heap where
  imp_for'_int_inner i u f s =
    (if i ≥ u then return s else f i s ⤜ imp_for'_int_inner (i + 1) u f)

lemma imp_for'_imp_for_int_inner:
  imp_for' i u f s ≡
  imp_for'_int_inner (integer_of_nat i) (integer_of_nat u) (f o nat_of_integer) s
  apply (induction u - i arbitrary: i u s)
   apply (simp add: integer_of_nat_le_simp imp_for'_int_inner.simps; fail)
```

apply (*subst imp_for'_int_inner.simps*, *simp add*: *integer_of_nat_le_simp*)
apply *safe*
subgoal
by *auto*
apply (*fo_rule arg_cong*, *rule ext*)
apply (*auto simp*: *algebra_simps integer_of_nat_eq_of_nat*)
done

lemma *imp_for'_int_cong*:
imp_for' l u f a = *imp_for' l u g a*
if $\bigwedge$ *i x*. *l* ≤ *i* ⟹ *i* < *u* ⟹ *f i x* = *g i x*
using *that*
apply (*induction u* − *l arbitrary*: *l u a*)
apply (*simp add*: *imp_for'.simps*; *fail*)
apply (*subst imp_for'.simps*, *simp add*: *integer_of_nat_le_simp*)
apply *safe*
apply *clarsimp*
apply (*fo_rule arg_cong*)
apply *auto*
done

definition
imp_for'_int i u f s ≡
imp_for'_int_inner (*integer_of_nat i*) (*integer_of_nat u*) (*f o nat_of_integer*) *s*

lemma *imp_for'_imp_for_int*[*code_unfold*, *int_folds*]:
imp_for' = *imp_for'_int*
by (*intro ext*, *unfold imp_for'_int_def imp_for'_imp_for_int_inner*, *rule HOL.refl*)

code_printing code_module *Iterators* ⇀ (*SML*)
‹
fun imp_for_inner i u c f s =
let
fun imp_for1 i u f s =
if IntInf.<= (*u*, *i*) *then* (*fn* () => *s*)
else if c s () *then imp_for1* (*i* + *1*) *u f* (*f i s* ())
else (*fn* () => *s*)
in imp_for1 i u f s end;

fun imp_fora_inner i u f s =
let
fun imp_for1 i u f s =
if IntInf.<= (*u*, *i*) *then* (*fn* () => *s*)
else imp_for1 (*i* + *1*) *u f* (*f i s* ())
in imp_for1 i u f s end;
›

code_reserved (*SML*) *imp_for_inner imp_fora_inner*

definition

```
  nth_integer a n = Array.nth a (nat_of_integer n)

definition
  upd_integer n x a = Array.upd (nat_of_integer n) x a

code_printing
  constant upd_integer ⇀ (SML) (fn a => fn () => (Array.update (a, IntInf.toInt _, _); a))
_
| constant nth_integer ⇀ (SML) (fn () => Array.sub (_, IntInf.toInt _))

definition
  fw_upd_impl_integer n a k i j = do {
  let n = n + 1;
  let i' = i * n + j;
  y ← nth_integer a (i * n + k);
  z ← nth_integer a (k * n + j);
  x ← nth_integer a i';
  let m = dbm_add_int y z;
  if dbm_lt_int m x then upd_integer i' m a else return a
}


lemma nat_of_integer_add:
  nat_of_integer i + nat_of_integer j = nat_of_integer (i + j) if i ≥ 0 j ≥ 0
proof −
  have ∗: nat a + nat b = nat (a + b) if a ≥ 0 b ≥ 0 for a b
    using that by simp
  show ?thesis
    unfolding nat_of_integer.rep_eq
    apply (simp add: ∗)
    apply (subst ∗)
    subgoal
      using zero_integer.rep_eq less_eq_integer.rep_eq that by metis
    subgoal
      using zero_integer.rep_eq less_eq_integer.rep_eq that by metis
    ..
qed

lemma nat_of_integer_mult:
  nat_of_integer i * nat_of_integer j = nat_of_integer (i * j) if i ≥ 0 j ≥ 0
  using that
proof −
  have ∗: nat a * nat b = nat (a * b) if a ≥ 0 b ≥ 0 for a b
    using that by (simp add: nat_mult_distrib)
  show ?thesis
    unfolding nat_of_integer.rep_eq
    apply simp
    apply (subst ∗)
    subgoal
      using zero_integer.rep_eq less_eq_integer.rep_eq that by metis
    subgoal
      using zero_integer.rep_eq less_eq_integer.rep_eq that by metis
    ..
qed
```

lemma *fw_upd_impl_int_fw_upd_impl_integer*:
fw_upd_impl_int (*nat_of_integer n*) *a* (*nat_of_integer k*) (*nat_of_integer i*) (*nat_of_integer j*)
= *fw_upd_impl_integer n a k i j*
if *i* ≥ *0 j* ≥ *0 k* ≥ *0 n* ≥ *0*
unfolding
fw_upd_impl_integer_def fw_upd_impl_int_def[*symmetric*] *fw_upd_impl_def mtx_get_def mtx_set_def*
unfolding *int_folds less*
unfolding *nth_integer_def upd_integer_def fst_conv snd_conv*
using *that*
apply (*simp add*: *nat_of_integer_add nat_of_integer_mult algebra_simps*)
apply (*fo_rule arg_cong* | *rule ext*)+
apply (*simp add*: *nat_of_integer_add nat_of_integer_mult algebra_simps*)
done

lemma *integer_of_nat_nat_of_integer*:
integer_of_nat (*nat_of_integer n*) = *n* **if** *n* ≥ *0*
using *that* **by** (*simp add*: *integer_of_nat_eq_of_nat max_def*)

lemma *integer_of_nat_aux*:
integer_of_nat (*nat_of_integer n* + *1*) = *n* + *1* **if** *n* ≥ *0*
proof −
have *nat_of_integer n* + *1* = *nat_of_integer n* + *nat_of_integer 1*
by *simp*
also have ... = *nat_of_integer* (*n* + *1*)
using *that* **by** (*subst nat_of_integer_add*; *simp*)
finally show *?thesis*
using *that* **by** (*simp add*: *integer_of_nat_nat_of_integer*)
qed

definition
fwi_impl_integer n a k = *fwi_impl_int* (*nat_of_integer n*) *a* (*nat_of_integer k*)

lemma *fwi_impl_int_eq1*:
fwi_impl_int n a k ≡ *fwi_impl_integer* (*integer_of_nat n*) *a* (*integer_of_nat k*)
unfolding *fwi_impl_integer_def fwi_impl_int_def* **by** *simp*

partial_function (*heap*) *imp_for'_int_int* ::
int ⇒ *int* ⇒ (*int* ⇒ *'a* ⇒ *'a Heap*) ⇒ *'a* ⇒ *'a Heap* **where**
imp_for'_int_int i u f s =
(*if i* ≥ *u then return s else f i s* ⤜ *imp_for'_int_int* (*i* + *1*) *u f*)

lemma *int_bounds_up_induct*:
assumes ⋀*l*. *u* ≤ (*l* :: *int*) ⟹ *P l u*
and ⋀*l*. *P* (*l* + *1*) *u* ⟹ *l* < *u* ⟹ *P l u*
shows *P l u*
by (*induction nat* (*u* − *l*) *arbitrary*: *l*) (*auto intro*: *assms*)

lemma *imp_for'_int_int_cong*:
imp_for'_int_int l u f a = *imp_for'_int_int l u g a*
if ⋀ *i x*. *l* ≤ *i* ⟹ *i* < *u* ⟹ *f i x* = *g i x*
using *that*

apply (*induction arbitrary*: *a rule*: *int_bounds_up_induct*)
apply (*simp add*: *imp_for'_int_int.simps*; *fail*)
apply (*subst* (*2*) *imp_for'_int_int.simps*)
apply (*subst imp_for'_int_int.simps*)
apply *simp*
apply (*fo_rule arg_cong*, *rule ext*)
.

lemma *plus_int_int_of_integer_aux*:
(*plus_int* (*int_of_integer l*) *1*) = *int_of_integer* (*l* + *1*)
by *simp*

lemma *imp_for'_int_inner_imp_for'_int_int*:
imp_for'_int_inner l u f a
= *imp_for'_int_int* (*int_of_integer l*) (*int_of_integer u*) (*f o integer_of_int*) *a*
apply (*induction int_of_integer l int_of_integer u arbitrary*: *a l u rule*: *int_bounds_up_induct*)
apply (*simp add*: *imp_for'_int_int.simps imp_for'_int_inner.simps less_eq_integer.rep_eq*; *fail*)
apply (*subst imp_for'_int_int.simps*)
apply (*subst imp_for'_int_inner.simps*)
apply (*simp add*: *less_eq_integer.rep_eq*)
apply (*fo_rule arg_cong*, *rule ext*)
apply (*subst plus_int_int_of_integer_aux*)
apply *rprems*
using *plus_int_int_of_integer_aux* **by** *metis+*

lemma *imp_for'_int_inner_cong*:
imp_for'_int_inner l u f a = *imp_for'_int_inner l u g a*
if ⋀ *i x*. *l* ≤ *i* ⟹ *i* < *u* ⟹ *f i x* = *g i x*
unfolding *imp_for'_int_inner_imp_for'_int_int*
by (*rule imp_for'_int_int_cong*)(*auto simp*: *less_eq_integer.rep_eq less_integer.rep_eq intro*: *that*)

schematic_goal *fwi_impl_int_unfold1*:
fwi_impl_int (*nat_of_integer n*) *a* (*nat_of_integer k*) = *?i* **if** *0* ≤ *k 0* ≤ *n*
unfolding *fwi_impl_int_def*[*symmetric*] *fwi_impl_def*
unfolding *int_folds*
unfolding *imp_for'_int_def*
unfolding *comp_def integer_of_nat_0*
apply (*rule imp_for'_int_inner_cong*)
apply (*rule imp_for'_int_inner_cong*)
apply (*rule fw_upd_impl_int_fw_upd_impl_integer*)
by (*assumption* | *rule that*)+

lemma *integer_of_nat_add*:
integer_of_nat (*x* + *y*) = *integer_of_nat x* + *integer_of_nat y*
by (*simp add*: *integer_of_nat_eq_of_nat*)

schematic_goal *fwi_impl_int_code* [*code*]:
fwi_impl_int n a k ≡ *?x*
unfolding *fwi_impl_int_eq1*
unfolding *fwi_impl_integer_def*
apply (*subst fwi_impl_int_unfold1*)
apply (*simp add*: *integer_of_nat_eq_of_nat*; *fail*)

apply (*simp add*: *integer_of_nat_eq_of_nat*; *fail*)
unfolding *nat_of_integer_integer_of_nat*
unfolding *integer_of_nat_add integer_of_nat_1*
apply (*abstract_let integer_of_nat n + 1 n_plus_1*)
apply (*abstract_let integer_of_nat n n*)
.

code_printing code_module *Integer* ⇀ (*Eval*)
‹
structure Integer: *sig*
val div_mod: *int* −> *int* −> *int* ∗ *int*
end = *struct*
fun div_mod i j = (*i div j*, *i mod j*)
end
›

Delete unused code module.

code_printing code_module *Bits_Integer* ⇀ (*SML*) ‹›

For agreement with SML

code_printing
type_constructor *Typerep.typerep* ⇀ (*Eval*)
| **constant** *Typerep.Typerep* ⇀ (*Eval*)

lemmas [*code*] = *imp_for'_int_inner.simps imp_for_int_inner.simps*

We eliminate this module because it uses constants that are only implemented by *IntInf* and not *Int*. When compiling we will use a hack to change *IntInf* with *Int* (for efficiency) and therefore this module would break the hack.

code_printing code_module *Bit_Shifts* ⇀ (*SML*) ‹›

Using the *Eval* code target instead of *SML* makes it easier to replace *IntInf* with *Int*.

export_code *parse_convert_check parse_convert_run_print parse_convert_run_check Result Error*
nat_of_integer int_of_integer DBMEntry.Le DBMEntry.Lt DBMEntry.INF
Impl1 Impl2 Impl3 Buechi Debug Reachable_Set Buechi_Set
E_op_impl
in *Eval* **module_name** *Model_Checker* **file_prefix** *Certificate*

export_code *parse_convert_check parse_convert_run_print parse_convert_run_check Result Error*
nat_of_integer int_of_integer DBMEntry.Le DBMEntry.Lt DBMEntry.INF
Impl1 Impl2 Impl3 Buechi Reachable_Set Buechi_Set
E_op_impl
in *SML* **module_name** *Model_Checker*

code_printing code_module *Printing* ⇀ (*Haskell*)
‹
import qualified Debug.Trace;

print s = Debug.Trace.trace s ();

printM s = Debug.Trace.traceM s;
›

code_printing
  constant *Printing.print* ⇀ (*Haskell*) *Printing.print _*

code_printing
  constant *print_line_impl* ⇀ (*Haskell*) *Printing.printM _*

code_printing
  type_constructor *time* ⇀ (*Haskell*) *Integer*
  | **constant** *now* ⇀ (*Haskell*) *Prelude.const 0*
  | **constant** *time_to_string* ⇀ (*Haskell*) *Prelude.show _*
  | **constant** (−) :: *time* ⇒ *time* ⇒ *time* ⇀ (*Haskell*) (−)

code_printing
  constant *list_of_set′* ⇀ (*Haskell*) (*case _ of Set xs −> xs*)

export_code *parse_convert_check parse_convert_run_print parse_convert_run_check Result Error*
  nat_of_integer int_of_integer DBMEntry.Le DBMEntry.Lt DBMEntry.INF
  Impl1 Impl2 Impl3 Buechi Reachable_Set Buechi_Set
  in *Haskell* **module_name** *Model_Checker*

end

13 Testing Infrastructure

theory *Munta_Certificate_Testing*
  imports *Main*
begin

— Produces commands for generating a certificate for a single benchmark with MLUNTA, e.g.
`mluntac-poly -certificate PM_all_5.cert -renaming PM_all_5.renaming -model PM_all_5.muntax`
checking it with MUNTA, and checking the result: `./check_benchmark.sh`
```
  muntac -certificate PM_all_5.cert -renaming PM_all_5.renaming -model PM_all_5.muntax
```
ML ‹
fun mk_cert mlunta_path name =
  let

```
    val benchmark = name ^ .muntax
    val gen_certificate = implode_space [
      mlunta_path,
      −certificate, name ^ .cert,
      −renaming, name ^ .renaming,
      −model, benchmark
    ]
  in
    gen_certificate
  end

val it1 = mk_cert mluntac−poly PM_all_5

fun check_cert muntac_path name =
  let
    val benchmark = name ^ .muntax
  in
    implode_space [
      ./check_benchmark.sh,
      muntac_path,
      −certificate, name ^ .cert,
      −renaming, name ^ .renaming,
      −model, benchmark
    ]
  end

val it2 = check_cert muntac PM_all_5

val mlunta_dir_proper = Path.append (Path.current |> File.absolute_path) (Path.explode mlunta)

val mlunta_dir = Path.explode mlunta |> File.absolute_path

val library_path =
  Path.append mlunta_dir (Path.explode src/isalib/library.sml) |> Path.implode
val basics_path =
  Path.append mlunta_dir (Path.explode src/isalib/basics.sml) |> Path.implode

val mlunta_certificate_path = mlunta/src/serialization/mlunta_certificate
›

end
```

14 Build and Test Exported Program With MLton

```
theory Munta_Certificate_Compile_MLton
  imports Simple_Network_Language_Certificate_Code Munta_Certificate_Testing
begin
```

Here is how to compile Munta Certifier with MLton and then run some benchmarks:

```
compile_generated_files code/Certificate.ML (in Simple_Network_Language_Certificate_Code)
  external_files
    ‹Unsynchronized.sml›
    ‹Writeln.sml›
```

```
    ‹Util.sml›
    ‹Muntac.sml›
    ‹Mlton_Main.sml›
    ‹sequential.sml›
    ‹muntac.mlb›
    ‹check_benchmark.sh›
    (in ML)
  and
    ‹HDDI_02.muntax›
    ‹HDDI_02_broadcast.muntax›
    ‹HDDI_08_broadcast.muntax›
    ‹PM_all_1.muntax›
    ‹PM_all_2.muntax›
    ‹PM_all_3.muntax›
    ‹PM_all_4.muntax›
    ‹PM_all_5.muntax›
    ‹PM_all_6.muntax›
    ‹PM_all_urgent.muntax›
    ‹bridge.muntax›
    ‹csma_05.muntax›
    ‹csma_06.muntax›
    ‹fischer.muntax›
    ‹fischer_05.muntax›
    ‹hddi_08.muntax›
    ‹light_switch.muntax› (in benchmarks)
  export_files ‹muntac› (exe)
  where ‹fn dir =>
    let
      val exec = Generated_Files.execute dir

      val _ = exec ‹Copy MLunta› (cp -r '^ Path.implode mlunta_dir ^' .)
     val _ = exec ‹Compile MLunta› cd mlunta && mlton=$ISABELLE_MLTON make build_checker
&& cd ..
      val mlunta_path = mlunta/build/mluntac-mlton

      val _ =
        exec ‹Copy Isabelle library files›
          (cp '^ library_path ^' library.ML && cp '^ basics_path ^' basics.ML)

      val _ =
        exec ‹Preparation›
          mv code/Certificate.ML Certificate.ML
      val _ =
        exec ‹Replace int type›
          sed -i -e 's/IntInf/Int/g' Certificate.ML
      val _ =
        exec ‹set›
          set -x
      val _ =
        — Efficient settings for ARM64 machines
        exec ‹Compilation›
          (verbatim‹$ISABELLE_MLTON $ISABELLE_MLTON_OPTIONS› ^
            — these additional settings have been copied from the AFP entry PAC_Checker
            -const 'MLton.safe false' -verbose 1 -inline 700 -cc-opt -O3 ^
```

```
      — this one does not work on ARM64 though
      ////codegen_native////
      — these used to be the defaults for Munta
      −default−type int64 ^
      −output muntac ^
      −mlb−path−var 'MLUNTA_CERT ^mlunta_certificate_path ^' ^
      muntac.mlb)
    val muntac_path = ./muntac;
    val muntac_path_dc = ./muntac −dc — For deadlock checking

    val _ = writeln Generating certificates.

    val _ = exec ‹Gen HDDI_02› (mk_cert mlunta_path HDDI_02)
    val _ = exec ‹Gen HDDI_02_broadcast› (mk_cert mlunta_path HDDI_02_broadcast)
    val _ = exec ‹Gen HDDI_08_broadcast› (mk_cert mlunta_path HDDI_08_broadcast)
    val _ = exec ‹Gen hddi_08› (mk_cert mlunta_path hddi_08)
    val _ = exec ‹Gen PM_all_1› (mk_cert mlunta_path PM_all_1)
    val _ = exec ‹Gen PM_all_2› (mk_cert mlunta_path PM_all_2)
    val _ = exec ‹Gen PM_all_3› (mk_cert mlunta_path PM_all_3)
    val _ = exec ‹Gen PM_all_4› (mk_cert mlunta_path PM_all_4)
    val _ = exec ‹Gen PM_all_5› (mk_cert mlunta_path PM_all_5)
    val _ = exec ‹Gen csma_05› (mk_cert mlunta_path csma_05)
    val _ = exec ‹Gen csma_06› (mk_cert mlunta_path csma_06)
    val _ = exec ‹Gen fischer_05› (mk_cert mlunta_path fischer_05)

    val _ = writeln Finished generating certificates. Now checking.;

    val _ = exec ‹Test HDDI_02› (check_cert muntac_path HDDI_02)
    val _ = exec ‹Test HDDI_02_broadcast› (check_cert muntac_path HDDI_02_broadcast)
    val _ = exec ‹Test HDDI_08_broadcast› (check_cert muntac_path HDDI_08_broadcast)
    val _ = exec ‹Test hddi_08› (check_cert muntac_path hddi_08)
    val _ = exec ‹Test PM_all_1› (check_cert muntac_path PM_all_1)
    val _ = exec ‹Test PM_all_2› (check_cert muntac_path PM_all_2)
    val _ = exec ‹Test PM_all_3› (check_cert muntac_path PM_all_3)
    val _ = exec ‹Test csma_05› (check_cert muntac_path csma_05)
    val _ = exec ‹Test csma_06› (check_cert muntac_path csma_06)
    val _ = exec ‹Test fischer_05› (check_cert muntac_path fischer_05)
    val _ = exec ‹Test PM_all_4› (check_cert muntac_path PM_all_4)
    val _ = exec ‹Test PM_all_5› (check_cert muntac_path PM_all_5)

    val _ = exec ‹Test deadlock HDDI_02› (check_cert muntac_path_dc HDDI_02)
  in () end›

end
```

15 Build and Test Exported Program With Poly/ML

```
theory Munta_Certificate_Compile_Poly
  imports Simple_Network_Language_Certificate_Code Munta_Certificate_Testing
begin
```

Mock Compilation Instead of using Poly/ML's *polyc*, we emulate compilation within Isabelle's Poly/ML environment. Below we also show how *polyc* could be used alternatively.

We prepare a few pieces of code that will be passed to Poly/ML for mocking compilation. This will be used for evaluating the code:

ML ‹
— from *HOL/Library/code_test.ML*
val flags = {*environment* = *ML_Env.SML*, *redirect* = *false*, *verbose* = *false*, *catch_all* = *true*,
debug = *NONE*, *writeln* = *writeln*, *warning* = *warning*}
fun eval cmd = *ML_Context.eval flags Position.none* (*ML_Lex.read_text* (*cmd*, *Position.none*))
›

— Redirecting output to file:
ML ‹
val mock_print = ***verbatim*** ‹
val out_stream_ref = *ref* (*NONE* : *BinIO.outstream option*);
fun print s =
BinIO.output (*Option.valOf* (!*out_stream_ref*), *Byte.stringToBytes s*)
›
›

— Mocking command line arguments:
ML ‹
val mock_cmd = ***verbatim*** ‹
structure CommandLine =
struct

val mock_args = *ref* []
fun arguments () = !*mock_args*
fun set_mock_args s =
mock_args := *String.tokens* (*fn c* => *c* = #) *s*;

end
›
›

— Set command line arguments, run *main*, and capture output
ML ‹
fun run_cmd dir args =
let
val out_path = *dir* + *Path.basic out* |> *Path.expand* |> *Path.implode*
in ***verbatim*** ‹
CommandLine.set_mock_args › ^ *args* ^ ***verbatim*** ‹;
val out_stream = *BinIO.openOut* › ^ *out_path* ^ ***verbatim*** ‹;
val _ = *out_stream_ref* := *SOME out_stream*;
main ();
BinIO.flushOut out_stream;
BinIO.closeOut out_stream
› *end*
›

— This is essentially the content of *build_muntac.sml*:
ML ‹
val files = [
Unsynchronized.sml,
Writeln.sml,

```
  basics.ML,
  library.ML,
  parallel_task_queue.sml
]
val mlunta_files = [
  prelude.sml,
  serializable.sml,
  certificate.sml
]
val final_files = [
  Util.sml,
  Muntac.sml
]
›
```

— Mock compiling

```
ML ‹
fun mock_compile dir file_name =
  let
    val _ = eval mock_cmd
    val _ = eval mock_print
    fun mk_path file_name = Path.append dir (Path.explode file_name)
    fun mk_mlunta_path file_name =
      dir + Path.explode mlunta_certificate_path + Path.basic file_name |> Path.expand
    val _ = app (fn x => mk_path x |> ML_Context.eval_file flags) files
    val _ = ML_Context.eval_file flags (dir + Path.basic file_name)
    val _ = app (fn x => mk_mlunta_path x |> ML_Context.eval_file flags) mlunta_files
    val _ = app (fn x => mk_path x |> ML_Context.eval_file flags) final_files
  in () end
›
```

— Also need to mock *check_cert*: simply pass the captured output to *check_benchmark.sh*

```
ML ‹
fun mock_exec_check_cert dir title muntac_path name =
  let
    fun check () =
      Generated_Files.execute dir title verbatim‹./check_benchmark.sh cat out›
   fun mk_path name ext = dir + Path.basic name |> Path.ext ext |> Path.expand |> Path.implode
    val args = implode_space [
      muntac_path,
      −certificate, mk_path name cert,
      −renaming, mk_path name renaming,
      −model, mk_path name muntax
    ]
    val cmd = run_cmd dir args
    val _ = eval cmd
  in
    check ()
  end
›
```

Compilation and Testing Here is how to compile Munta Certifier with Poly/ML and then run some benchmarks:

```
compile_generated_files code/Certificate.ML (in Simple_Network_Language_Certificate_Code)
  external_files
    ‹Writeln.sml›
    ‹Util.sml›
    ‹Muntac.sml›
    ‹Mlton_Main.sml›
    ‹Unsynchronized.sml›
    ‹sequential.sml›
    ‹parallel_task_queue.sml›
    ‹build_muntac.sml›
    ‹check_benchmark.sh›
    (in ML)
  and
    ‹HDDI_02.muntax›
    ‹HDDI_02_broadcast.muntax›
    ‹HDDI_08_broadcast.muntax›
    ‹PM_all_1.muntax›
    ‹PM_all_2.muntax›
    ‹PM_all_3.muntax›
    ‹PM_all_4.muntax›
    ‹PM_all_5.muntax›
    ‹PM_all_6.muntax›
    ‹PM_all_urgent.muntax›
    ‹bridge.muntax›
    ‹csma_05.muntax›
    ‹csma_06.muntax›
    ‹fischer.muntax›
    ‹fischer_05.muntax›
    ‹hddi_08.muntax›
    ‹light_switch.muntax› (in benchmarks)
  export_files ‹muntac_poly› (exe)
  where ‹fn dir =>
    let
      val mock = true — whether to eval in Poly/ML instead of using polyc

      val exec = Generated_Files.execute dir

      val _ = exec ‹Copy MLunta› (cp -r ' ^ Path.implode mlunta_dir ^ ' .)
     val _ = exec ‹Compile MLunta› cd mlunta && mlton=$ISABELLE_MLTON make build_checker
&& cd ..
      val mlunta_path = mlunta/build/mluntac-mlton

      val _ =
        exec ‹Copy Isabelle library files›
          (cp ' ^ library_path ^ ' library.ML && cp ' ^ basics_path ^ ' basics.ML)

      val _ =
        exec ‹Preparation›
          mv code/Certificate.ML Certificate.ML
      val _ =
        exec ‹Replace int type›
          sed -i -e 's/IntInf/Int/g' Certificate.ML
      val _ =
        exec ‹set›
```

```
    set −x
val _ =
  if mock then
    (mock_compile dir Certificate.ML;
     exec ‹Compilation› touch muntac_poly)
  else
    exec ‹Compilation›
      (MLUNTA_CERT= ^ mlunta_certificate_path ^   ^
       verbatim‹$ML_HOME› ^
       /polyc −o muntac_poly build_muntac.sml)

val muntac_path = if mock then  else ./muntac_poly
val muntac_path_n6 = muntac_path ^ −n 6 — Parallel checking for larger certificates
val muntac_path_dc = muntac_path ^ −dc — For deadlock checking

val _ = writeln Generating certificates.

val _ = exec ‹Gen HDDI_02› (mk_cert mlunta_path HDDI_02)
val _ = exec ‹Gen HDDI_02_broadcast› (mk_cert mlunta_path HDDI_02_broadcast)
val _ = exec ‹Gen HDDI_08_broadcast› (mk_cert mlunta_path HDDI_08_broadcast)
val _ = exec ‹Gen hddi_08› (mk_cert mlunta_path hddi_08)
val _ = exec ‹Gen PM_all_1› (mk_cert mlunta_path PM_all_1)
val _ = exec ‹Gen PM_all_2› (mk_cert mlunta_path PM_all_2)
val _ = exec ‹Gen PM_all_3› (mk_cert mlunta_path PM_all_3)
val _ = exec ‹Gen PM_all_4› (mk_cert mlunta_path PM_all_4)
val _ = exec ‹Gen PM_all_5› (mk_cert mlunta_path PM_all_5)
val _ = exec ‹Gen csma_05› (mk_cert mlunta_path csma_05)
val _ = exec ‹Gen csma_06› (mk_cert mlunta_path csma_06)
val _ = exec ‹Gen fischer_05› (mk_cert mlunta_path fischer_05)

val _ = writeln Finished generating certificates. Now checking.;

fun exec_check_cert title muntac_path name =
  if mock then
    mock_exec_check_cert dir title muntac_path name
  else
    exec title (check_cert muntac_path name)

val _ = exec_check_cert ‹Test HDDI_02› muntac_path HDDI_02
val _ = exec_check_cert ‹Test HDDI_02_broadcast› muntac_path HDDI_02_broadcast
val _ = exec_check_cert ‹Test HDDI_08_broadcast› muntac_path HDDI_08_broadcast
val _ = exec_check_cert ‹Test PM_all_1› muntac_path PM_all_1
val _ = exec_check_cert ‹Test PM_all_2› muntac_path PM_all_2
val _ = exec_check_cert ‹Test PM_all_3› muntac_path PM_all_3
val _ = exec_check_cert ‹Test csma_05› muntac_path_n6 csma_05
— 30 s on an M1 Mac
val _ = exec_check_cert ‹Test csma_06› muntac_path_n6 csma_06
val _ = exec_check_cert ‹Test fischer_05› muntac_path_n6 fischer_05
val _ = exec_check_cert ‹Test hddi_08› muntac_path hddi_08
val _ = exec_check_cert ‹Test PM_all_4› muntac_path PM_all_4
— 60 s on an M1 Mac
val _ = exec_check_cert ‹Test PM_all_5› muntac_path_n6 PM_all_5

val _ = exec_check_cert ‹Test deadlock HDDI_02› muntac_path_dc HDDI_02
```

in () *end*›

end

References

[1] G. Li. Checking timed büchi automata emptiness using lu-abstractions. In J. Ouaknine and F. W. Vaandrager, editors, *Formal Modeling and Analysis of Timed Systems, 7th International Conference, FORMATS 2009, Budapest, Hungary, September 14-16, 2009. Proceedings*, volume 5813 of *Lecture Notes in Computer Science*, pages 228–242. Springer, 2009.

[2] S. Wimmer. *Trustworthy Verification of Realtime Systems.* PhD thesis, Technical University of Munich, Germany, 2020.

[3] S. Wimmer, F. Herbreteau, and J. van de Pol. Certifying emptiness of timed büchi automata. In N. Bertrand and N. Jansen, editors, *Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020, Vienna, Austria, September 1-3, 2020, Proceedings*, volume 12288 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2020.

[4] S. Wimmer and J. von Mutius. Verified certification of reachability checking for timed automata. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 425–443. Springer, 2020.