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 150) 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 167) and `Munta_Certificate_Compile_Poly` (on page 169), 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 154. 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* $(v \# es)\ G$
⟨*proof*⟩

lemma (**in** *wf_digraph*) *reachable_vwalk_conv2*:
$u \rightarrow^*_G v \longleftrightarrow (u = v \wedge u \in verts\ G \vee (\exists\, vs.\ vwalk\ (u \# vs\ @\ [v])\ G))$
⟨*proof*⟩

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*
⟨*proof*⟩

lemmas *scc_digraphI* = *scc_digraph.loopfree_digraph_axioms*

end

context *wf_digraph*
begin

interpretation *scc_digraph*: *loopfree_digraph scc_graph*
⟨*proof*⟩

lemma *scc_graph_edgeE*:
assumes $\langle x \rightarrow_{scc\_graph} y \rangle$ **obtains** *a b* **where**
$a \in x$ $b \in y$ $a \rightarrow b$ $x \in sccs\_verts$ $y \in sccs\_verts$ $x \neq y$
⟨*proof*⟩

lemma *same_sccI*:
assumes $S \in sccs\_verts$ $x \in S$ $x \rightarrow^* y$ $y \rightarrow^* x$
shows $y \in S$
⟨*proof*⟩

lemma *scc_graph_reachable1E*:
assumes $\langle x \rightarrow^+_{scc\_graph} y \rangle$ **obtains** *a b* **where**

$x \in sccs\_verts \;\; y \in sccs\_verts \;\; x \neq y \; a \in x \; b \in y \; a \rightarrow^+ b$
⟨*proof*⟩

end

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

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

context *wf_digraph*
begin

interpretation *scc_digraph*: *loopfree_digraph scc_graph*
⟨*proof*⟩

interpretation *scc_digraph*: *dag scc_graph*
⟨*proof*⟩

lemmas *scc_digraphI* = *scc_digraph.dag_axioms*

end

context *fin_digraph*
begin

interpretation *scc_digraph*: *dag scc_graph*
⟨*proof*⟩

interpretation *scc_digraph*: *fin_dag scc_graph*
⟨*proof*⟩

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*:
$(a, b) \in \{(x, y).\ P\ x\ y\} \longleftrightarrow P\ a\ b$

⟨*proof*⟩

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* ⟨*proof*⟩

lemma *dominates_iff*:
$u \rightarrow_G v \longleftrightarrow E\ u\ v$
⟨*proof*⟩

lemma *reachable1_iff*:
$u \rightarrow^+_G v \longleftrightarrow reaches1\ u\ v$
⟨*proof*⟩

end

context *wf_digraph*
begin

interpretation *Graph_Defs E* ⟨*proof*⟩

lemma *reachable_iff*:
$u \rightarrow^*_G v \longleftrightarrow reaches\ u\ v$ **if** $u \in verts\ G$
⟨*proof*⟩

lemma *vwalk_iff*:
$vwalk\ (u\ \#\ xs)\ G \longleftrightarrow steps\ (u\ \#\ xs)$ **if** $u \in verts\ G$
⟨*proof*⟩

lemmas *graph_convs1* = *reachable_iff reachable1_iff vwalk_iff*

end

context *dag*
begin

sublocale *graph*: *DAG E*
⟨*proof*⟩
lemma *topological_numbering*:
fixes *S* **assumes** *finite S*
shows $\exists f :: \_ \Rightarrow nat.\ inj\_on\ f\ S \wedge (\forall x \in S.\ \forall y \in S.\ x \rightarrow y \longrightarrow f\ x < f\ y)$
⟨*proof*⟩

end

sublocale *fin_digraph* ⊆ *graph*: *Finite_Graph E*

⟨*proof*⟩

sublocale *fin_dag* ⊆ *graph*: *Finite_DAG E* ⟨*proof*⟩

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 $G_p = G$
⟨*proof*⟩

sublocale *digraph*: *nomulti_digraph G*
⟨*proof*⟩

sublocale *pdigraph*: *pair_wf_digraph* G_p
⟨*proof*⟩

lemma *arc_to_ends_eq*[*simp*]:
arc_to_ends G = *id*
⟨*proof*⟩

lemma *arcs_ends_eq*[*simp*]:
arcs_ends G = $\{(a,\ b).\ E\ a\ b\}$
⟨*proof*⟩

lemma *dominates_iff*[*simp*]:
$u \rightarrow_G v \longleftrightarrow E\ u\ v$
⟨*proof*⟩

lemma *verts_eq*[*simp*]:
verts G = *UNIV*
⟨*proof*⟩

lemma *reachable_iff*:
$u \rightarrow^*_G v \longleftrightarrow u \rightarrow* v$
⟨*proof*⟩

lemma *reachable1_iff*:
$u \rightarrow^+_G v \longleftrightarrow u \rightarrow^+ v$
⟨*proof*⟩

lemma *vwalk_iff*:
vwalk xs G ⟷ *steps xs*
⟨*proof*⟩

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*:
$u \rightarrow* v \longleftrightarrow (\exists p.\ steps\ p \wedge distinct\ p \wedge hd\ p = u \wedge last\ p = v)$
⟨*proof*⟩

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*:
$with\_proj\ {G_p}' = G'$
⟨*proof*⟩

sublocale *digraph_sub*: *wf_digraph* G'
⟨*proof*⟩

sublocale *pdigraph_sub*: *pair_wf_digraph* ${G_p}'$
⟨*proof*⟩

lemma *verts_eq*[*simp*]:
$verts\ G' = UNIV$
⟨*proof*⟩

end

context *Subgraph*
begin

lemma *subgraph*:
$subgraph\ G'\ G.G$
⟨*proof*⟩

lemma *spanning*:
$spanning\ G'\ G.G$
⟨*proof*⟩

lemma *G'_eq*:
$G'.G = G'$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *verts_eq*[*simp*]:
$verts\ G_n = \{x.\ V\ x\}$
⟨*proof*⟩

lemma *arcs_eq*[*simp*]:
$arcs\ G_n = arcs\ G'$
⟨*proof*⟩

lemma *subgraph_dominatesI*:
$a \rightarrow_{G_n} b$ **if** $a \rightarrow b\ V\ a\ V\ b$
⟨*proof*⟩

lemma *arcs_ends_eq*:
$arcs\_ends\ G_n = arcs\_ends\ G'$
⟨*proof*⟩

lemma *subgraph_nodes'*:
$subgraph\ G_n\ G'$
⟨*proof*⟩

lemma *subgraph_nodes*:
$subgraph\ G_n\ G$
⟨*proof*⟩

lemma *induced_subgraph*:
$induced\_subgraph\ G_n\ G$
⟨*proof*⟩
lemma *reachable1_iff*:
$u \rightarrow^+_{G_n} v \longleftrightarrow u \rightarrow^+_{G'} v$
⟨*proof*⟩

lemma *dominates_iff*:
$u \rightarrow_{G_n} v \longleftrightarrow E'\ u\ v$
⟨*proof*⟩

lemma *subgraph_vwalk_iff*:
$vwalk\ (v\ \#\ vs)\ G_n \longleftrightarrow G'.steps\ (v\ \#\ vs)$ **if** $V\ v$
⟨*proof*⟩

lemma *subgraph_reaches_iff*:
$u \rightarrow^*_{G_n} v \longleftrightarrow G'.reaches\ u\ v$ **if** $V\ u$
⟨*proof*⟩

lemma *subgraph_reaches1_iff*:
$u \rightarrow^+_{G_n} v \longleftrightarrow G'.reaches1\ u\ v$
⟨*proof*⟩

end

context *Graph_Invariant*
begin

lemma *reachable_iff*:
$u \rightarrow^*{}_{G_n} v \longleftrightarrow u \rightarrow^*{}_G v$ **if** $P\ u$
⟨*proof*⟩

lemma *reachable1_iff*:
$u \rightarrow^+{}_{G_n} v \longleftrightarrow u \rightarrow^+{}_G v$ **if** $P\ u$
⟨*proof*⟩

lemma *vwalk_iff*:
$vwalk\ (u\ \#\ vs)\ G_n \longleftrightarrow vwalk\ (u\ \#\ vs)\ G$ **if** $P\ u$
⟨*proof*⟩

end

2.4 Application: Topological Numberings on the SCCs of a Digraph

context *fin_digraph*
begin

interpretation *scc_digraph*: *fin_dag scc_graph*
⟨*proof*⟩

definition
$scc\_num \equiv SOME\ f :: (\_ \Rightarrow nat).$
$inj\_on\ f\ sccs\_verts \wedge (\forall x{\in}sccs\_verts.\ \forall y{\in}sccs\_verts.\ x \rightarrow_{scc\_graph} y \longrightarrow f\ x < f\ y)$

lemma
shows *scc_num_inj*: *inj_on scc_num sccs_verts* (**is** *?thesis1*)
and *scc_num_topological*:
$\forall x{\in}sccs\_verts.\ \forall y{\in}sccs\_verts.\ x \rightarrow_{scc\_graph} y \longrightarrow scc\_num\ x < scc\_num\ y$ (**is** *?thesis2*)
⟨*proof*⟩

end

context *Finite_Graph*
begin

interpretation *Graph_Invariant*
where $E = E$ **and** $P = \lambda x.\ x \in vertices$
⟨*proof*⟩

interpretation *digraph_nodes*: *fin_digraph* G_n
⟨*proof*⟩

definition
$is\_max\_scc\ S \equiv$
$S \subseteq vertices \wedge S \neq \{\} \wedge (\forall u \in S.\ \forall v \in S.\ u \rightarrow* v) \wedge (\forall u \in S.\ \forall v.\ v \notin S \longrightarrow \neg u \rightarrow* v \vee \neg v \rightarrow* u)$

lemma *is_max_scc_iff*:
is_max_scc S $\longleftrightarrow$ *S* $\in$ *digraph_nodes.sccs_verts*
⟨*proof*⟩

lemma *max_sccI*:
assumes *a* $\in$ *vertices*
obtains *A* **where** *is_max_scc A a* $\in$ *A*
⟨*proof*⟩

lemma *is_max_scc_disjoint*:
assumes *is_max_scc V is_max_scc W V* $\neq$ *W*
shows *V* $\cap$ *W* = {}
⟨*proof*⟩

definition
edge V W $\equiv$ $\exists$ *a* $\in$ *V*. $\exists$ *b* $\in$ *W*. *V* $\neq$ *W* $\wedge$ *E a b*

definition
scc_num $\equiv$ *SOME f* :: (_ $\Rightarrow$ *nat*).
inj_on f {*V*. *is_max_scc V*} $\wedge$ ($\forall$ *V*. $\forall$ *W*. *is_max_scc V* $\wedge$ *is_max_scc W* $\wedge$ *edge V W* $\longrightarrow$ *f V* < *f W*)

interpretation *scc_digraph*: *fin_dag digraph_nodes.scc_graph*
⟨*proof*⟩

lemma *edge_iff*:
edge x y $\longleftrightarrow$ *x* $\rightarrow_{digraph\_nodes.scc\_graph}$ *y*
if *x* $\in$ *digraph_nodes.sccs_verts y* $\in$ *digraph_nodes.sccs_verts*
⟨*proof*⟩

lemma
shows *scc_num_inj*: *inj_on scc_num* {*V*. *is_max_scc V*} (**is** *?thesis1*)
and *scc_num_topological*:
$\forall$ *V*. $\forall$ *W*. *is_max_scc V* $\wedge$ *is_max_scc W* $\wedge$ *edge V W* $\longrightarrow$ *scc_num V* < *scc_num W* (**is** *?thesis2*)
⟨*proof*⟩

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 $E :: 'a :: countable \Rightarrow 'a \Rightarrow bool$ **and** $F :: 'a \Rightarrow bool$
fixes $f :: 'a \Rightarrow nat$
assumes *f_topo*: $E\ a\ b \Longrightarrow$ *if* $F\ a$ *then* $f\ a > f\ b$ *else* $f\ a \geq f\ b$
begin

lemma *f_downward*:
assumes $E\ a\ b$
shows $f\ a \geq f\ b$
⟨*proof*⟩

lemma *f_strict*:
assumes $E\ a\ b\ F\ a$
shows $f\ a > f\ b$
⟨*proof*⟩

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes $F\ a\ E\ a\ a$
shows *False*
⟨*proof*⟩

lemma *reaches_f_mono*:
assumes $E^{**}\ a\ b$
shows $f\ a \geq f\ b$
⟨*proof*⟩

lemma *no_accepting_cycle*:
assumes $E^{++}\ x\ x$
shows $\neg\ F\ x$
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

sublocale *Graph_Defs* ⟨*proof*⟩

lemma *run_f_mono*:
assumes *run* $(x\ \#\#\ xs)\ c \geq f\ x$
shows *pred_stream* $(\lambda a.\ c \geq f\ a)\ xs$
⟨*proof*⟩

lemma *no_Buechi_run*:
assumes *run xs infs F xs*
shows *False*
⟨*proof*⟩

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 $f\ a \leq f\ b$
⟨*proof*⟩

lemma *f_strict*:
assumes $E\ a\ b\ F\ a$
shows $f\ a < f\ b$
⟨*proof*⟩

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes $F\ a\ E\ a\ a$
shows *False*
⟨*proof*⟩

lemma *reaches_f_mono*:
assumes $E^{**}\ a\ b$
shows $f\ a \leq f\ b$
⟨*proof*⟩

lemma *no_accepting_cycle*:
assumes $E^{++}\ x\ x$
shows $\neg\ F\ x$
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

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 $reach\ x \geq reach\ y$
⟨*proof*⟩

lemma *reach_strict*:
assumes $E\ x\ y\ F\ x$
shows $reach\ x > reach\ y$
⟨*proof*⟩

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*⟩

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 $\forall a \in S.$ *finite* $\{b.\ (a, b) \in R\}$
shows *finite* $(R\ ``\ S)$
⟨*proof*⟩

lemma (**in** *Graph_Defs*) *steps_two_iff*[*simp*]:
$steps\ [a, b] \longleftrightarrow E\ a\ b$
⟨*proof*⟩

lemma (**in** *Graph_Defs*) *steps_Cons_two_iff*:
$steps\ (a\ \#\ b\ \#\ as) \longleftrightarrow E\ a\ b \wedge steps\ (b\ \#\ as)$
⟨*proof*⟩

4.2 Certifying Cycle-Freeness in Graphs

locale *Contract1* =
fixes $A\ B :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $G :: 'a \Rightarrow bool$
begin

definition $E\ a\ b \equiv A\ a\ b \wedge G\ a \vee \neg\ G\ a \wedge B\ a\ b \wedge G\ b$

definition $E'\ a\ b \equiv G\ a \wedge G\ b \wedge (A\ a\ b \vee (\exists\ c.\ A\ a\ c \wedge \neg\ G\ c \wedge B\ c\ b))$

sublocale E: *Graph_Defs* E ⟨*proof*⟩
sublocale E': *Graph_Defs* E' ⟨*proof*⟩

lemma *steps_contract*:
assumes $E.steps\ (a\ \#\ xs\ @\ [b])\ G\ a\ G\ b$
shows $\exists\, as.\ E'.steps\ (a\ \#\ as\ @\ [b])$
⟨*proof*⟩

lemma E'_G[*dest*, *intro*]:
assumes $E'\ x\ y$
shows $G\ x\ G\ y$
⟨*proof*⟩

lemma E'_steps_G:
assumes $E'.steps\ (x\ \#\ xs)\ xs \neq []$
shows $G\ x$
⟨*proof*⟩

lemma $E\_E'\_cycle$:
assumes $E^{++}\ x\ x\ G\ x$
shows $E'^{++}\ x\ x$
⟨*proof*⟩
including *graph_automation* ⟨*proof*⟩

This can be proved by rotating the cycle if $\neg\ G\ x$

lemma
assumes $E^{**}\ s\ x\ E^{++}\ x\ x\ G\ s$
shows $E'^{**}\ s\ x \wedge E'^{++}\ x\ x$
⟨*proof*⟩
including *graph_automation* ⟨*proof*⟩
including *graph_automation* ⟨*proof*⟩

Certifying cycle-freeness with a topological ordering of graph components

context
fixes $f :: 'a \Rightarrow nat$ **and** $F :: 'a \Rightarrow bool$
assumes f_topo1: $\forall\, a\ b.\ G\ a \wedge A\ a\ b \wedge G\ b \longrightarrow (if\ F\ a\ then\ f\ a < f\ b\ else\ f\ a \leq f\ b)$
and f_topo2:
$\forall\, a\ b\ c.\ G\ a \wedge A\ a\ b \wedge \neg\ G\ b \wedge G\ c \wedge B\ b\ c \longrightarrow (if\ F\ a\ then\ f\ a < f\ c\ else\ f\ a \leq f\ c)$
begin

lemma *f_forward*:
assumes $E'\ a\ b$
shows $f\ a \leq f\ b$
⟨*proof*⟩

lemma *f_strict*:
assumes $E'\ a\ b\ F\ a$
shows $f\ a < f\ b$
⟨*proof*⟩

We do not even need this property any more.

lemma *no_trivial_lasso*:

assumes $F\ a\ G\ a\ E'\ a\ a$
shows *False*
⟨*proof*⟩

lemma *reaches_f_mono*:
assumes $E'^{**}\ a\ b$
shows $f\ a \leq f\ b$
⟨*proof*⟩

theorem *no_accepting_cycle*:
assumes $E'^{++}\ x\ x$
shows $\neg\ F\ x$
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma *f_forward*:
assumes $E\ a\ b$
shows $f\ a \leq f\ b$
⟨*proof*⟩

lemma *f_strict*:
assumes $E\ a\ b\ F\ a$
shows $f\ a < f\ b$
⟨*proof*⟩

We do not even need this property any more.

lemma *no_trivial_lasso*:
assumes $F\ a\ G\ a\ E\ a\ a$
shows *False*
⟨*proof*⟩

lemma *reaches_f_mono*:
assumes $E^{**}\ a\ b$
shows $f\ a \leq f\ b$
⟨*proof*⟩

theorem *no_accepting_cycle*:
assumes $E^{++}\ x\ x$
shows $\neg\ F\ x$
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

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*
  ⟨*proof*⟩

theorem *no_accepting_cycle*:
  assumes $E^{++}\ x\ x$
  shows $\neg\ F\ x$
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

interpretation *Simulation_Invariant*
  $E\ \lambda\ x\ y.\ \exists\ z.\ z \preceq y \wedge E\ x\ z \wedge P\ z\ (\preceq)\ P\ P$
  ⟨*proof*⟩

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** $\nexists\ s'.\ B.reaches\ b\ s' \wedge F\ s'\ a \preceq b\ P\ a\ P\ b$
  ⟨*proof*⟩

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** $\nexists\ s'.\ B.reaches\ b\ s' \wedge F'\ s'\ a \sim b\ PA\ a\ PB\ b$
⟨*proof*⟩

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' ⟨*proof*⟩

interpretation *Simulation_Invariant* $E\ E'\ (\preceq)\ \lambda s.\ True\ \lambda x.\ x \in S$
⟨*proof*⟩

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

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*⟩

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** $\forall\ s' \in S.\ \neg\ F\ s'$
⟨*proof*⟩

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*⟩

end

end

end

locale *Unreachability_Invariant1* = *Subsumption_Graph_Pre_Defs* + *preorder less_eq less* +
fixes I **and** $SE :: 'a \Rightarrow 'a \Rightarrow bool$ **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 $S = \{s.\ I\ s\}$
⟨*proof*⟩

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' ⟨*proof*⟩

interpretation *Simulation_Invariant* $E\ E'\ (\preceq)\ P\ I$
⟨*proof*⟩

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*⟩

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

begin

corollary *final_unreachable*:
$\nexists\ s'.\ reaches\ s_0\ s' \wedge F\ s'$ **if** $\forall\ s'.\ I\ s' \longrightarrow \neg\ F\ s'$
⟨*proof*⟩

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*⟩

end

end

end

locale *Unreachability_Invariant* = *Subsumption_Graph_Pre_Defs* + *preorder less_eq less* +
fixes s_0
fixes $S :: 'a\ set$ **and** $SE :: 'a \Rightarrow 'a \Rightarrow bool$
assumes *mono*:
$s \preceq s' \Longrightarrow s \rightarrow t \Longrightarrow s \in S \vee reaches\ s_0\ s \Longrightarrow s' \in S \vee reaches\ s_0\ s'$
$\Longrightarrow \exists\ t'.\ t \preceq t' \wedge s' \rightarrow t'$
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

interpretation *Unreachability_Invariant1*
where $P = \lambda s.\ s \in S \vee reaches\ s_0\ s$
and $I = \lambda s.\ s \in S$
⟨*proof*⟩

lemma *reachable_S_subsumed*:
$\exists\ s'.\ s' \in S \wedge s \preceq s'$ **if** $s' \in S\ s_0 \preceq s'\ reaches\ s_0\ s$
⟨*proof*⟩

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' ⟨*proof*⟩

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*⟩

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*⟩

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** $\forall\ s' \in S.\ \neg\ F\ s'$
  ⟨*proof*⟩

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*⟩

end

end

end

context *Unreachability_Invariant1*
begin

interpretation *inv*: *Graph_Invariant*
  ⟨*proof*⟩

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

interpretation *E*: *Unreachability_Invariant*
  where $S = \{s.\ I\ s\}$
  ⟨*proof*⟩

end

end

locale *Unreachability_Invariant_Contract1* =
  Unreachability_Invariant1 +
  fixes $f :: 'a \Rightarrow nat$ **and** $F :: 'a \Rightarrow bool$
  assumes *f_topo*:
      $I\ a \Longrightarrow E\ a\ b \Longrightarrow SE\ b\ c \Longrightarrow$ (*if* $F\ a$ *then* $f\ a < f\ c$ *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'
  ⟨*proof*⟩

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

interpretation *c*: *Contract E1*
⟨*proof*⟩

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*⟩

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 ⟨*proof*⟩

interpretation *inv*: *Graph_Invariant* $E\ P$
⟨*proof*⟩

interpretation *unreach1*: *Unreachability_Invariant1*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'\ E$
$\lambda(l,\ s).\ l \in L \wedge s \in M\ l$
⟨*proof*⟩

interpretation *Unreachability_Invariant*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'\ E\ (l_0,\ s_0)$
$\{(l,\ s) \mid l\ s.\ l \in L \wedge s \in M\ l\}$
⟨*proof*⟩

lemma *E_L*:
assumes $l \in L\ s \in M\ l\ E\ (l,\ s)\ (l',\ s')$
shows $l' \in L$
⟨*proof*⟩

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*:
$s' \in M\ l_0 \wedge s_0 \preceq s'$
⟨*proof*⟩ **lemma** *s'_1*:
$(l_0,\ s') \in \{(l,\ s)\ |l\ s.\ l \in L \wedge s \in M\ l\}$
⟨*proof*⟩ **lemma** *s'_2*:
$(case\ (l_0,\ s_0)\ of\ (l,\ s) \Rightarrow \lambda(l',\ s').\ l' = l \wedge s \preceq s')\ (l_0,\ s')$
⟨*proof*⟩

theorem *final_unreachable*:
assumes $\forall\, s' \in \{(l,\ s)\ |l\ s.\ l \in L \wedge s \in M\ l\}.\ \neg\ F\ s'$
shows $\nexists\, s'.\ (l_0,\ s_0) \rightarrow\!* s' \wedge F\ s'$
⟨*proof*⟩

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**
$G.reaches\ (l_0,\ {s_0}')\ (l,\ s)\ G.reaches1\ (l,\ s)\ (l,\ s)\ F\ (l,\ s)\ {s_0}' \in M\ l_0\ s_0 \preceq {s_0}'$
⟨*proof*⟩

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* $f\ (l,\ s) \leq f\ (l2,\ s2)$
begin

definition E1 where E1 (l, s) (l', s') = E (l, s) (l', s') ∧ s ∈ M l ∧ s' ∈ M l' ∧ SE (l, s) (l', s') ∧ l ∈ L ∧ l' ∈ L ∧ s ∈ M 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). E (l, s) (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 ∧ s ∈ M l ∧ s' ∈ M l unfolding E'_def E1_def by (standard, auto) obtain s₀' l s where s₀ ≠ s₀' s₀' ∈ M l 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 − (rule 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 $G = \lambda(l,\ s).\ l \in L \wedge s \in M\ l$
⟨*proof*⟩

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*⟩

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 $\nexists\ x.\ reachable\ x \wedge x \rightarrow^{+}\ x \wedge F\ x$
⟨*proof*⟩

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 ⟨*proof*⟩

sublocale *P_invariant*: *Graph_Invariant E P*
⟨*proof*⟩

sublocale *G_invariant*: *Graph_Invariant E_mix P*
⟨*proof*⟩

lemma *mix_reachable_G*[*intro*]:
P x **if** *G_mix.reachable x*
⟨*proof*⟩

lemma *P_reachable*[*intro*]:
P a **if** *reachable a*
⟨*proof*⟩

lemma *mix_finite_reachable*: *finite* {*a*. *G_mix.reachable a*}
⟨*proof*⟩

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* $\preceq a'$
⟨*proof*⟩

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*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

theorem *reachability_complete'*:
$\exists\ s'.\ s \preceq s' \wedge G\_mix.reachable\ s' \wedge G\ s'$ **if** $a \rightarrow* s\ G\_mix.reachable\ a\ G\ a$
⟨*proof*⟩

lemma *steps_G_mix_steps*:
$\exists\ ys\ ns.\ list\_all2\ (\preceq)\ xs\ (nths\ ys\ ns) \wedge G\_mix.steps\ (b \#\ ys)$ **if**
$steps\ (a \#\ xs)\ P\ a\ a \preceq b\ G\ b$
⟨*proof*⟩

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*⟩

lemma *cycle_G_mix_cycle'*:
assumes $steps\ (s_0 \#\ ws\ @\ x \#\ xs\ @\ [x])$
shows $\exists\ y\ ys.\ x \preceq y \wedge G\_mix.steps\ (y \#\ ys\ @\ [y]) \wedge G\_mix.reachable\ y$
⟨*proof*⟩

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*⟩

theorem *no_accepting_cycleI*:
assumes $\nexists\ x.\ G\_mix.reachable\ x \wedge G\_mix.reaches1\ x\ x \wedge F\ x$
shows $\nexists\ x.\ reachable\ x \wedge x \rightarrow^+ x \wedge F\ x$
⟨*proof*⟩

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 ⟨*proof*⟩

sublocale *P_invariant*: *Graph_Invariant E P*

⟨*proof*⟩

sublocale *G_invariant*: *Graph_Invariant E_mix G*
⟨*proof*⟩

lemma *mix_reachable_G*[*intro*]:
G x **if** *G_mix.reachable x*
⟨*proof*⟩

lemma *P_reachable*[*intro*]:
P a **if** *reachable a*
⟨*proof*⟩

lemma *mix_finite_reachable*: *finite* {*a*. *G_mix.reachable a*}
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *G_mix_reaches*:
assumes *G a G b a* ⪯ *b*
shows *G_mix.reaches a b*
⟨*proof*⟩

lemma *subsumption_step'*:
∃ *b'*. *b* ⪯ *b'* ∧ *G_mix.reaches1 a' b'* **if** *P a E a b G a' a* ⪯ *a'*
⟨*proof*⟩

theorem *reachability_complete'*:
∃ *s'*. *s* ⪯ *s'* ∧ *G_mix.reachable s'* ∧ *G s'* **if** *a* →∗ *s G_mix.reachable a G a*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *cycle_G_mix_cycle''*:
assumes *steps* $(s_0 \# ws \mathbin{@} x \# xs \mathbin{@} [x])$
shows $\exists\ x'\ xs'\ ys'.\ x \preceq x' \wedge$ *G_mix.steps* $(s_0 \# xs' \mathbin{@} x' \# ys' \mathbin{@} [x'])$
⟨*proof*⟩

lemma *cycle_G_mix_cycle'*:
assumes *steps* $(s_0 \# ws \mathbin{@} x \# xs \mathbin{@} [x])$
shows $\exists y\ ys.\ x \preceq y \wedge$ *G_mix.steps* $(y \# ys \mathbin{@} [y]) \wedge$ *G_mix.reachable* y
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

end

sublocale *Reachability_Compatible_Subsumption_Graph_Final''*
⟨*proof*⟩

theorem *no_accepting_cycleI*:
assumes $\nexists\ x.$ *G_mix.reachable* $x \wedge$ *G_mix.reaches1* $x\ x \wedge F\ x$
shows $\nexists\ x.$ *reachable* $x \wedge x \rightarrow^+ x \wedge F\ x$
⟨*proof*⟩

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*⟩

end

sublocale *Reachability_Compatible_Subsumption_Graph_Final''*
⟨*proof*⟩

theorem *no_accepting_cycleI*:
assumes $\nexists\ x.$ *G_mix.reachable* $x \wedge$ *G_mix.reaches1* $x\ x \wedge F\ x$
shows $\nexists\ x.$ *reachable* $x \wedge x \rightarrow^+ x \wedge F\ x$
⟨*proof*⟩

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** $RE^{**}\ (l_0, s_0)\ (l, s)$
⟨*proof*⟩

interpretation *Reachability_Compatible_Subsumption_Graph_View*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'$
$E\ (l_0,\ s_0)\ RE$
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'\ covered$
⟨*proof*⟩

context
assumes *no_subsumption_cycle*: $G'.reachable\ x \Longrightarrow x\ {\rightarrow_G}^{+\prime}\ x \Longrightarrow x\ {\rightarrow_G}^{+}\ x$
fixes $F :: 'l \times 's \Rightarrow bool$ — Final states
assumes *F_mono*[*intro*]: $F\ (l,\ a) \Longrightarrow a \preceq b \Longrightarrow F\ (l,\ b)$
begin

interpretation *Liveness_Compatible_Subsumption_Graph*
$\lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \preceq s'\ \lambda\ (l,\ s)\ (l',\ s').\ l' = l \wedge s \prec s'$
$E\ (l_0,\ s_0)\ RE\ F$
⟨*proof*⟩

lemma
$\nexists\ x.\ reachable\ x \wedge F\ x \wedge x \rightarrow^{+} x$ **if** $\nexists\ x.\ G.reachable\ x \wedge F\ x \wedge x\ {\rightarrow_G}^{+}\ x$
⟨*proof*⟩

lemma *no_reachable_cycleI*:
$\nexists\ x.\ reachable\ x \wedge F\ x \wedge x \rightarrow^{+} x$ **if** $\nexists\ x.\ G'.reachable\ x \wedge F\ x \wedge x\ {\rightarrow_G}^{+}\ x$
⟨*proof*⟩

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 ⟨*proof*⟩
interpretation *E2*: *Graph_Start_Defs E2* s_0 ⟨*proof*⟩
interpretation *E*: *Graph_Start_Defs E* s_0 ⟨*proof*⟩
interpretation *C*: *Graph_Start_Defs C* a_0 ⟨*proof*⟩

lemma *E_closed*:
$\forall\ a \in S.\ \forall\ x \in a.\ \forall\ y.\ E\ x\ y \longrightarrow (\exists\ b \in S.\ y \in b)$
⟨*proof*⟩

interpretation *E_invariant*: *Graph_Invariant E* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
⟨*proof*⟩

interpretation *E1_invariant*: *Graph_Invariant E1* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
⟨*proof*⟩

interpretation *E2_invariant*: *Graph_Invariant E2* $\lambda\ x.\ \exists\ a \in S.\ x \in a$
⟨*proof*⟩

interpretation *C_invariant*: *Graph_Invariant C* $\lambda\ a.\ a \in S \wedge a \neq \{\}$
⟨*proof*⟩

interpretation *Simulation_Invariant E C* $(\in)$ $\lambda\ x.\ \exists\ a \in S.\ x \in a\ \lambda\ a.\ a \in S \wedge a \neq \{\}$
⟨*proof*⟩

interpretation *Subgraph E E1*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *certify_no_E1_cycle*:
assumes *E.reachable x E.reaches1 x x*
shows *E1.reaches1 x x*
⟨*proof*⟩

lemma *certify_no_accepting_cycle*:
assumes
$\forall\ a \in S.\ card\ a > 1 \longrightarrow (\forall\ x \in a.\ \neg\ F\ x)$
$\forall\ a \in S.\ \forall\ x.\ a = \{x\} \longrightarrow (\neg\ F\ x \vee \neg\ E1\ x\ x)$
assumes *E.reachable x E1.reaches1 x x*
shows $\neg\ F\ x$
⟨*proof*⟩
including *graph_automation_aggressive* ⟨*proof*⟩

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*⟩

Misc *heap* **lemma** *hoare_triple_success*:
assumes $<P>\ c\ <Q>$ **and** $(h,\ as) \models P$
shows *success c h*
⟨*proof*⟩

lemma *run_return*: *run* (*return x*) (*Some h*) (*Some h*) *x* **for** *h*
⟨*proof*⟩

lemma *return_htD*:
assumes $<Q\ x>$ *return x* $<PP>$
shows $Q\ x \Longrightarrow_A PP\ x$
⟨*proof*⟩

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 $<emp>\ c\ <\lambda r.\ \uparrow(P\ r)>_t$
shows *P* (*fst* (*the* (*execute c h*)))
⟨*proof*⟩

lemma *hoare_triple_run_heapD*:
assumes $<emp>\ c\ <\lambda r.\ \uparrow(P\ r)>_t$
shows *P* (*run_heap c*)
⟨*proof*⟩

lemma *list_all2_conjI*:

assumes *list_all2 P as bs list_all2 Q as bs*
shows *list_all2* ($\lambda a\ b.\ P\ a\ b \wedge Q\ a\ b$) *as bs*
⟨*proof*⟩

lemma *hoare_triple_run_map_heapD*:
assumes *list_all* ($\lambda x.$ <*emp*> *c x* <$\lambda r.$ $\uparrow$(*P x r*)>$_t$) *xs*
shows *list_all2 P xs* (*run_map_heap c xs*)
⟨*proof*⟩

lemma *hoare_triple_run_map_heapD'*:
assumes *list_all2* ($\lambda x\ xi.$ <*emp*> *c xi* <$\lambda r.$ $\uparrow$(*P x r*)>$_t$) *xs xsi*
shows *list_all2 P xs* (*run_map_heap c xsi*)
⟨*proof*⟩

definition
parallel_fold_map = *Heap_Monad.fold_map*

Misc *nres* **lemma** *no_fail_RES_bindI*:
assumes $\bigwedge x.\ x \in S \Longrightarrow$ *nofail* (*f x*)
shows *nofail* (*RES S* $\gg\!=$ *f*)
⟨*proof*⟩

lemma *nfoldli_ub_RES_rule*:
assumes $\bigwedge x\ s.\ x \in$ *set xs* $\Longrightarrow$ $s \in S \Longrightarrow$ *f x s* $\leq$ *RES S s* $\in S$
shows *nfoldli xs c f s* $\leq$ *RES S*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *SUCCEED_lt_RES_iff*[*simp*]:
SUCCEED $<$ *RES S* $\longleftrightarrow$ $S \neq \{\}$
⟨*proof*⟩

lemma *SUCCEED_lt_RETURN*[*intro*, *simp*]:
SUCCEED $<$ *RETURN x*
⟨*proof*⟩

lemma *SUCCEED_lt_FAIL*[*intro*, *simp*]:
SUCCEED $<$ *FAIL*
⟨*proof*⟩

lemma *bind_RES_gt_SUCCEED_I*:
assumes $\bigwedge s.$ *f s* $>$ *SUCCEED* $S \neq \{\}$
shows *do* {$x \leftarrow$ *RES S*; *f x*} $>$ *SUCCEED*
⟨*proof*⟩

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
⟨*proof*⟩

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 (λx. x = None) (λx _. do {
r ← P x; case r of None ⇒ RETURN None | Some r ⇒ RETURN (Some r)})
None
⟨*proof*⟩

lemma *monadic_list_all_fail_monadic_list_all_fail′*:
monadic_list_all_fail P xs =
monadic_list_all_fail′ (λx. do {b ← P x; RETURN (if b then None else Some x)}) xs
⟨*proof*⟩

lemma *monadic_list_all_rule*:
assumes ⋀*x. Pi x ≤ SPEC (λr. r = P x)*
shows *monadic_list_all Pi xs ≤ SPEC (λr. r ⟷ list_all P xs)*
⟨*proof*⟩

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

lemma *monadic_list_ex_rule*:
assumes ⋀*x. Pi x ≤ SPEC (λr. r = P x)*
shows *monadic_list_ex Pi xs ≤ SPEC (λr. r ⟷ list_ex P xs)*
⟨*proof*⟩

lemma *monadic_list_ex_empty*[*simp*]:
monadic_list_ex P [] = RETURN False
⟨*proof*⟩

lemma *monadic_list_all_empty*[*simp*]:
monadic_list_all P [] = RETURN True
⟨*proof*⟩

lemma *monadic_list_all_False*: *monadic_list_all* (λ*x*. *RETURN False*) *xs* = *RETURN* (*xs* = [])
⟨*proof*⟩

lemma *monadic_list_all_RETURN*:
monadic_list_all (λ*x*. *RETURN* (*P x*)) *xs* = *RETURN* (*list_all P xs*)
⟨*proof*⟩

lemma *monadic_list_ex_RETURN*:
monadic_list_ex (λ*x*. *RETURN* (*P x*)) *xs* = *RETURN* (*list_ex P xs*)
⟨*proof*⟩

lemma *monadic_list_ex_RETURN_mono*:
assumes *set xs* = *set ys*
shows *monadic_list_ex* (λ*s*. *RETURN* (*P s*)) *xs* ≤ *monadic_list_ex* (λ*s*. *RETURN* (*P s*)) *ys*
⟨*proof*⟩

lemma *monadic_list_ex_nofailI*:
assumes ⋀ *x*. *x* ∈ *set xs* ⟹ *nofail* (*f x*)
shows *nofail* (*monadic_list_ex f xs*)
⟨*proof*⟩

lemma *monadic_list_all_nofailI*:
assumes ⋀ *x*. *x* ∈ *set xs* ⟹ *nofail* (*f x*)
shows *nofail* (*monadic_list_all f xs*)
⟨*proof*⟩

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* (λ*x* _. *g x*) *a* > *SUCCEED* **for** *a c*
⟨*proof*⟩

lemma *monadic_list_all_gt_SUCCEED*:
monadic_list_all g xs > *SUCCEED*
⟨*proof*⟩

lemma *monadic_list_ex_gt_SUCCEED*:
monadic_list_ex g xs > *SUCCEED*
⟨*proof*⟩

end

lemma *monadic_list_ex_is_RETURN*:
∃ *r*. *monadic_list_ex* (λ*x*. *RETURN* (*P x*)) *xs* = *RETURN r*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)))
⟨*proof*⟩

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 (λxs'. set xs' = xs);
      if xs = [] then RETURN True else do {
        b1 ← RETURN (l' ∈ L);
        ys ← SPEC (λxs. set xs = M l');
        b2 ← monadic_list_all (λx.
          monadic_list_ex (λy. RETURN (x ≼ y)) ys
        ) xs;
        RETURN (b1 ∧ b2)
      }
    }
    ) succs
  }
  ) L'
}
```

definition
check_invariant_spec $L' \equiv$
$\forall l \in L'. \forall s \in M\ l. \forall l'\ s'.\ E\ (l, s)\ (l', s') \longrightarrow l' \in L \wedge (\exists s'' \in M\ l'.\ s' \preceq s'')$

definition
check_invariant_spec_pre $L' \equiv$
$\forall l \in set\ L'. \forall (l',ys) \in set\ (succs\ l\ (M\ l)). \forall s' \in ys.\ l' \in L \wedge (\exists s'' \in M\ l'.\ s' \preceq s'')$

lemma *check_invariant_correct_pre*:
check_invariant $L' \leq$ *RETURN* (*check_invariant_spec_pre* L')
⟨*proof*⟩

context
fixes L'
assumes *pre*: $\forall l \in L. \forall s \in M\ l.\ P\ (l, s)$ *set* $L' \subseteq L$
begin

lemma *check_invariant_spec_pre_eq_check_invariant_spec*:
check_invariant_spec_pre $L' =$ *check_invariant_spec* (*set* L')
⟨*proof*⟩

lemma *check_invariant_correct_strong*:
check_invariant $L' \leq$ *RETURN* (*check_invariant_spec* (*set* L'))
⟨*proof*⟩

lemma *check_invariant_correct*:
check_invariant $L' \leq$ *SPEC* $(\lambda r.\ r \longrightarrow$ *check_invariant_spec* (*set* L'))
⟨*proof*⟩

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 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*$)$
⟨*proof*⟩

lemma *check_final_nofail*:
nofail check_final
⟨*proof*⟩

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 ← *check_prop P′*;
RETURN (*b1* ∧ *b2*)
}

lemma *check_all_pre_def*:
check_all_pre l_0 s_0 = *do* {
b1 ← *RETURN* ($l_0 \in L$);
b2 ← *RETURN* (P' (l_0, s_0));
xs ← *SPEC* (λ*xs*. *set xs* = *M* l_0);
b3 ← *monadic_list_ex* (λ*s*. *RETURN* ($s_0 \preceq s$)) *xs*;
b4 ← *check_prop P′*;
RETURN (*b1* ∧ *b2* ∧ *b3* ∧ *b4*)
}
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *check_init_correct*:
check_init l_0 $s_0 \leq$ *RETURN* (*check_init_spec* l_0 s_0)
⟨*proof*⟩

end

locale *Reachability_Impl_pre_start* =
Reachability_Impl_pre **where** $E = E$ **for** $E :: 'l \times 's \Rightarrow \_ +$
fixes $l_0 :: 'l$ **and** $s_0 :: 's$
begin

definition
check_all ≡ *do* {
b ← *check_all_pre* l_0 s_0;
if b then SPEC (λ*r*. *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* l_0 s_0;

$b2 \leftarrow SPEC\ (\lambda r.\ r \longrightarrow check\_invariant\_spec\ L)$;
$b3 \leftarrow check\_final$;
$RETURN\ (b1 \wedge b2 \wedge b3)$
}
⟨*proof*⟩

definition
$check\_all\_spec \equiv check\_all\_pre\_spec\ l_0\ s_0 \wedge check\_invariant\_spec\ L$

lemma *check_all_correct*:
$check\_all \leq SPEC\ (\lambda r.\ r \longrightarrow check\_all\_spec)$
⟨*proof*⟩

end

locale *Reachability_Impl_correct* =
Reachability_Impl_pre_start **where** $E = E$ +
Unreachability_Invariant_paired_pre **where** $E = E$ **for** $E :: 'l \times 's \Rightarrow \_$
begin

lemma *Unreachability_Invariant_pairedI*[*rule_format*]:
$check\_all\_spec$
$\longrightarrow Unreachability\_Invariant\_paired\ (\preceq)\ (\prec)\ M\ L\ E\ P\ l_0\ s_0\ (\lambda(l,\ u)\ (l',\ u').\ l' = l \wedge u \preceq u')$
⟨*proof*⟩

lemma *check_all_correct'*:
$check\_all \leq SPEC\ (\lambda r.\ r \longrightarrow$
$Unreachability\_Invariant\_paired\ (\preceq)\ (\prec)\ M\ L\ E\ P\ l_0\ s_0\ (\lambda(l,\ u)\ (l',\ u').\ l' = l \wedge u \preceq u'))$
⟨*proof*⟩

lemma *certify_unreachableI*:
$check\_all\_spec \wedge check\_final\_spec \longrightarrow (\nexists s'.\ E^{**}\ (l_0,\ s_0)\ s' \wedge F\ s')$
⟨*proof*⟩

lemma *certify_unreachable_correct*:
$certify\_unreachable \leq SPEC\ (\lambda r.\ r \longrightarrow check\_all\_spec \wedge check\_final\_spec)$
⟨*proof*⟩

lemma *certify_unreachable_correct'*:
$certify\_unreachable \leq SPEC\ (\lambda r.\ r \longrightarrow (\nexists s'.\ E^{**}\ (l_0,\ s_0)\ s' \wedge F\ s'))$
⟨*proof*⟩

end

locale *Buechi_Impl_invariant* =
Reachability_Impl_base **where** $E = E$ **for** $E :: 'l \times 's \Rightarrow \_$ +
fixes $L :: 'l\ set$ **and** $M :: 'l \Rightarrow ('s \times nat)\ set$
begin

definition *check_invariant_buechi* $R\ L' \equiv$
monadic_list_all $(\lambda l.$
do {

```
    let as = M l;
    as ← SPEC (λxs'. set xs' = as);
    monadic_list_all (λ(x, i). do {
      let succs = succs l {x};
      monadic_list_all (λ(l', xs). do {
        xs ← SPEC (λxs'. set xs' = xs);
        b1 ← RETURN (l' ∈ L);
        if xs = [] then RETURN True else do {
            ys ← SPEC (λxs. set xs = M l');
            b2 ← monadic_list_all (λy.
              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*⟩

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 ≡ λs 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** $\exists j.$ *is_arg_max ?f ?P* (s', j))
⟨*proof*⟩

lemma *SE_deterministic*:
assumes $\bigwedge s.$ *s1* $\preceq s \longleftrightarrow$ *s2* $\preceq s$
assumes *SE* $(l,$ *s1*$)$ $(l',$ *s1'*$)$ *SE* $(l,$ *s2*$)$ $(l',$ *s2'*$)$
shows *s2'* $=$ *s1'*
⟨*proof*⟩

lemma *SE_I*:
assumes $(s'', j) \in M\ l'$ *buechi_prop l l' i j s s' s''*
shows $\exists (s'', j) \in M\ l'.$ *SE* (l', s') (l', s'')
⟨*proof*⟩

definition
check_all_pre_spec1 inits $\equiv$
$(\forall l \in L.\ \forall s \in$ *fst* ‘ *M l. P'* $(l, s)) \wedge$
$(\forall (l_0, s_0) \in$ *inits.* $l_0 \in L \wedge (\exists\ (s', \_) \in M\ l_0.\ s_0 \preceq s') \wedge P'\ (l_0, s_0))$

definition
check_buechi inits $\equiv$ *do* {
$b \leftarrow$ *SPEC* $(\lambda r.\ r \longrightarrow$ *check_all_pre_spec1 inits*$)$;
if b then do {
ASSERT (*check_all_pre_spec1 inits*);
SPEC $(\lambda r.\ r \longrightarrow$ *check_invariant_buechi_spec* (*buechi_prop*) *L*)
} *else RETURN False*
}

definition
check_buechi_spec inits $\equiv$
check_all_pre_spec1 inits
$\wedge\ (\forall l \in L.\ \forall (s, i) \in M\ l.\ \forall l'\ s'.\ E\ (l, s)\ (l', s')$
$\longrightarrow (\exists (s'', j) \in M\ l'.$ *buechi_prop l l' i j s s' s''*$))$

definition
check_buechi_spec' *inits* $\equiv$
$(\forall (l_0, s_0) \in$ *inits. Unreachability_Invariant_paired* $(\preceq)$ $(\prec)$ $(\lambda l.$ *fst* ‘ *M l*$)$ *L E P* l_0 s_0 *SE*$)$
$\wedge\ (\forall l \in L.\ \forall (s, i) \in M\ l.\ \forall l'\ s'.\ E\ (l, s)\ (l', s')$
$\longrightarrow (\exists (s'', j) \in M\ l'.$ *buechi_prop l l' i j s s' s''*$))$

lemma *check_buechi_correct*:
check_buechi inits $\leq$ *SPEC* $(\lambda r.\ r \longrightarrow$ *check_buechi_spec inits*$)$
⟨*proof*⟩

end

locale *Buechi_Impl_correct* $=$
Buechi_Impl_pre **where** $M = M$ **and** $E = E+$
Unreachability_Invariant_paired_pre **where** $E = E$ **for** E **and** $M :: 'l \Rightarrow ('s \times nat)\ set$
begin

lemma *check_buechi_correct'*:
check_buechi inits $\leq$ *SPEC* $(\lambda r.\ r \longrightarrow$ *check_buechi_spec'* *inits*$)$

⟨*proof*⟩

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*⟩

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*⟩

lemma *no_buechi_run*:
assumes *check*: *check_buechi_spec'* *inits*
assumes *accepting_run*:
(*l*$_0$, *s*$_0$) ∈ *inits Graph_Defs.run E* ((*l*$_0$, *s*$_0$) ## *xs*) *alw* (*ev* (*holds F*)) ((*l*$_0$, *s*$_0$) ## *xs*)
shows *False*
⟨*proof*⟩

end

locale *Reachability_Impl_common* =
Reachability_Impl_pre **where** *less_eq* = *less_eq* **and** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S*
for *less_eq* :: '*b* ⇒ '*b* ⇒ *bool* (**infix** ≼ *50*) **and** *M* :: '*k* ⇒ '*b set option* +
assumes *L_finite*: *finite L*
and *M_ran_finite*: ∀ *S* ∈ *ran M*. *finite S*
and *succs_finite*: ∀ *l S*. ∀ (*l'*, *S'*) ∈ *set* (*succs l S*). *finite S* ⟶ *finite S'*
and *succs_empty*: ⋀*l*. *succs l* {} = []

begin

lemma *M_listD*:
assumes *M l* = *Some S*
shows ∃ *xs*. *set xs* = *S*
⟨*proof*⟩

lemma *L_listD*:
shows ∃ *xs*. *set xs* = *L*
⟨*proof*⟩

lemma *check_prop_gt_SUCCEED*:
check_prop P' > SUCCEED
⟨*proof*⟩

definition
check_final' 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);
monadic_list_all (λs.
RETURN (¬ PR_CONST F (l, s))
) xs
}
}
) l
}

lemma *check_final_alt_def*:
check_final' L M = check_final
⟨*proof*⟩

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 (λs.
RETURN (PR_CONST P' (l, s))
) xs;
RETURN r
}
}
) l
}

lemma *check_prop_alt_def*:
check_prop' L M = check_prop P'
⟨*proof*⟩

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
}
⟨proof⟩

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*$_0$*i* :: *'ki Heap* **and** *s*$_0$*i* :: *'ai Heap*
assumes *l*$_0$*i_l*$_0$[*sepref_fr_rules*]:
(*uncurry0 l*$_0$*i*, *uncurry0* (*RETURN* (*PR_CONST l*$_0$))) ∈ *unit_assn*^{k} →$_a$ *K*
assumes *s*$_0$*i_s*$_0$[*sepref_fr_rules*]:
(*uncurry0 s*$_0$*i*, *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*
⟨*proof*⟩

sepref_definition *print_check_impl* **is**

uncurry PRINT_CHECK :: $id\_assn^k *_a id\_assn^k \to_a id\_assn$
⟨*proof*⟩

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*)
:: $K \to \langle K,V\rangle map\_rel \to \langle V\rangle option\_rel \times_r \langle K,V\rangle map\_rel$
where *single_valued K single_valued* (K^{-1})
⟨*proof*⟩

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*)
⟨*proof*⟩ **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′*
⟨*proof*⟩ **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′*
⟨*proof*⟩

lemma *list_rev_aux*: *list_assn A a c* ⟹$_A$ *list_assn A* (*rev a*) (*rev c*)
⟨*proof*⟩

theorem *copy_list_refine*:
assumes
copy: (*copy*, *RETURN o COPY*) ∈ $A^k \to_a A$
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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *copy_list_COPY*:
monadic_map (*RETURN o COPY*) = *RETURN o COPY* (**is** *?l* = *?r*)
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⇒ (*'ki* :: {*heap*}) ⇒ *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) →$_a$ *bool_assn*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is_member_correct*:
(*uncurry is_member*, *uncurry* (*RETURN* ∘∘ *op_set_member*)) ∈ *Id* ×$_r$ *Id* → ⟨*bool_rel*⟩*nres_rel*
⟨*proof*⟩

lemmas [*sepref_fr_rules*] = *lso_id_hnr*

sepref_definition *is_member_impl* **is**
uncurry is_member :: $K^k *_a (lso\_assn\ K)^k \to_a bool\_assn$
⟨*proof*⟩

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'*
⟨*proof*⟩

lemma *check_invariant'_refine*:
check_invariant' *L_part M* ≤ *check_invariant L_part* **if** *L* = *dom M set L_part* ⊆ *L*
⟨*proof*⟩

lemma *check_invariant'_no_fail*:
nofail (*check_invariant'* L' M')
⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*] ⟨*proof*⟩

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*
⟨*proof*⟩

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} →$_a$ *id_assn*
⟨*proof*⟩

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}) →$_a$ *bool_assn*
⟨*proof*⟩

sepref_definition *is_K_eq_impl* **is**
uncurry (*RETURN oo* (=)) :: (*K*^{k} ∗$_a$ *K*^{k}) →$_a$ *bool_assn*
⟨*proof*⟩

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'*) :: $(list\_assn\ K)^k *_a table\_assn^k \rightarrow_a id\_assn$
⟨*proof*⟩

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*)
⟨*proof*⟩

definition
check_all_pre' L' M' ≡ *do* {
b1 ← *RETURN* (*PR_CONST* l_0 ∈ *L'*);
b2 ← *RETURN* (*PR_CONST P'* (*PR_CONST* l_0, *PR_CONST* s_0));
let S = *op_map_lookup* (*PR_CONST* l_0) *M'*;
case S of None ⇒ *RETURN False* | *Some S* ⇒ *do* {
xs ← *SPEC* (λ*xs*. *set xs* = *S*);
b3 ← *monadic_list_ex* (λ*s*. *RETURN* (*PR_CONST less_eq* (*PR_CONST* s_0) *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* (λ*r*. *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* (λ*xs*. *set xs* = *S*);
monadic_list_ex (λ*s*. *RETURN* (*PR_CONST less_eq* (*PR_CONST* s_0) *s*)) *xs*
}

lemma *check_all_pre'_refine*:
check_all_pre' L M $\leq$ *check_all_pre* l_0 s_0
⟨*proof*⟩

lemma *check_all'_refine*:
check_all' L M $\leq$ *check_all*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *L_member_fold*:
PR_CONST l_0 ∈ *L'* ≡ *PR_CONST L_member L'*
⟨*proof*⟩

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

lemma *looukp_fold*:
op_map_lookup (*PR_CONST* l_0) = *PR_CONST lookup*
⟨*proof*⟩

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*
⟨*proof*⟩

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)
⟨*proof*⟩

sepref_register *check1*

sepref_thm *check1_impl* **is**
uncurry0 (*RETURN* (*PR_CONST check1*)) :: *id_assn*^{k} →$_a$ *id_assn*
⟨*proof*⟩

sepref_thm *check2_impl* **is**
PR_CONST check2 :: *table_assn*^{k} →$_a$ *id_assn*
⟨*proof*⟩

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* = (⋃ *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* (λ*L*. *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*
⟨*proof*⟩

sepref_definition *check_all_pre_impl* **is**
uncurry (*PR_CONST check_all_pre′*) :: (*lso_assn K*)^k ∗$_a$ *table_assn*^{k} →$_a$ *id_assn*
⟨*proof*⟩

lemma *check_all_pre_impl_ht*:

```
  <table_assn b bi * lso_assn K a ai>
    check_all_pre_impl ai bi
   <λr. table_assn b bi * lso_assn K a ai * ↑ (RETURN r ≤ check_all_pre' a b)>t
  if nofail (check_all_pre' a b)
  ⟨proof⟩

definition
  check_all'' L' M' ≡ do {
  b ← PR_CONST check_all_pre' L' M';
  if b
  then do {
    xs ← SPEC (λxs. 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
  ⟨proof⟩

lemma split_spec_correct:
  split_spec L' M ≤ SPEC (λr. r ⟶ check_invariant_spec_pre L')
  if assms: dom M = L set L' ⊆ L
⟨proof⟩

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⟩

sepref_thm check_all_impl is
  uncurry (PR_CONST check_all') :: (lso_assn K)^k *a table_assn^k →a id_assn
  ⟨proof⟩

sepref_thm check_all_impl is
  uncurry (PR_CONST check_all') :: (lso_assn K)^k *a table_assn^k →a id_assn
  ⟨proof⟩
```

lemma *certify_unreachable_alt_def*:
certify_unreachable ≡ *do* {
b1 ← *PR_CONST check_all*;
b2 ← *PR_CONST check_final*;
RETURN (*b1* ∧ *b2*)
}
⟨*proof*⟩

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*
⟨*proof*⟩

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} *→$_a$ id_assn*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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

lemma (**in** −) [*sepref_import_param*]:
(*println*, *println*) ∈ *Id* → *Id*
⟨*proof*⟩

sepref_definition (**in** −) *print_line_impl* **is**
PRINTLN :: *id_assn*^{k} →$_a$ *id_assn*
⟨*proof*⟩

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*: $(show\_dbm\_impl, show\_dbm) \in A^d \to_a id\_assn$
assumes *show_loc_impl*: $(show\_loc\_impl, show\_loc) \in K^d \to_a id\_assn$
begin

lemma [*sepref_fr_rules*]: $(show\_dbm\_impl, PR\_CONST\ show\_dbm) \in A^d \to_a id\_assn$
⟨*proof*⟩

lemma [*sepref_fr_rules*]: $(show\_loc\_impl, PR\_CONST\ show\_loc) \in K^d \to_a id\_assn$
⟨*proof*⟩

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 :: $(K \times_a A)^d \to_a id\_assn$
⟨*proof*⟩

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*))
}
}
) *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 :: $(lso\_assn\ K)^k *_a table\_assn^d \to_a option\_assn\ (K \times_a A)$
⟨*proof*⟩

end

definition
check_invariant_fail L′ M′ ≡ *do* {
l ← *SPEC* (λ*xs*. *set xs* = *L′*);
monadic_list_all_fail′ (λ*l*.
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* (λ*xs′*. *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* (λ*xs*. *set xs* = *ys*);
b2 ← *monadic_list_all_fail* (λ*x′*.
monadic_list_ex (λ*y*. *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* (λ*xs′*. *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
:: $(lso\_assn\ K)^k *_a table\_assn^k \to_a$
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*)
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *list_set_rel_Nil*[*simp*, *intro*]:
([], {}) ∈ ⟨*Id*⟩*list_set_rel*
⟨*proof*⟩

lemma *specify_right*:
$c \le$ *SPEC* $P \gg\!= c'$ **if** $P\ x\ c \le c'\ x$
⟨*proof*⟩

lemma *res_right*:
$c \le$ *RES* $S \gg\!= c'$ **if** $x \in S\ c \le c'\ x$
⟨*proof*⟩

lemma *nres_relD*:
$c \le \Downarrow R\ a$ **if** $(c, a) \in \langle R\rangle nres\_rel$
⟨*proof*⟩

lemma *list_rel_setE1*:
assumes $x \in set\ xs$ $(xs, ys) \in \langle R\rangle list\_rel$
obtains y **where** $y \in set\ ys$ $(x, y) \in R$
⟨*proof*⟩

lemma *list_rel_setE2*:
assumes $y \in set\ ys$ $(xs, ys) \in \langle R\rangle list\_rel$
obtains x **where** $x \in set\ xs$ $(x, y) \in R$
⟨*proof*⟩

lemma *list_of_set_impl*[*autoref_rules*]:
$(\lambda xs.\ RETURN\ xs, list\_of\_set) \in \langle R\rangle list\_set\_rel \to \langle\langle R\rangle list\_rel\rangle nres\_rel$
⟨*proof*⟩

lemma *case_option_mono*:
(*case* x *of None* ⇒ a | *Some* x' ⇒ $f\ x'$, *case* y *of None* ⇒ b | *Some* x' ⇒ $g\ x'$) ∈ R
if $(x, y) \in \langle S\rangle option\_rel$ $(a, b) \in R$ $(f, g) \in S \to R$
⟨*proof*⟩

lemmas *case_option_mono′* =
case_option_mono[**where** $R = \langle R\rangle nres\_rel$ **for** R, *THEN nres_relD*, *THEN refine_IdD*]

lemma *bind_mono*:
assumes $m \le \Downarrow\ R\ m'$
and $\bigwedge x\ y.\ (x, y) \in R \Longrightarrow f\ x \le f'\ y$
shows *Refine_Basic.bind* $m\ f \le m' \gg\!= f'$
⟨*proof*⟩

lemma *list_all_split*:
assumes $set\ xs = (\bigcup xs \in set\ split.\ set\ xs)$
shows $list\_all\ P\ xs = list\_all\ id\ (map\ (list\_all\ P)\ split)$
⟨*proof*⟩

lemma *list_all_default_split*:
$list\_all\ P\ xs = list\_all\ id\ (map\ P\ xs)$
⟨*proof*⟩

locale *Reachability_Impl_pure_base* =
  Reachability_Impl_base **where** *less_eq* = *less_eq*
  for *less_eq* :: *'a* ⇒ *'a* ⇒ *bool* (**infix** ≼ *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*: (*Li*, *L*) ∈ ⟨*K*⟩*list_set_rel*
  assumes *lei_less_eq*: (*lei*, *less_eq*) ∈ *A* → *A* → *bool_rel*
  assumes *get_succs_succs*[*param*]:
    (*get_succs*, *succs*) ∈ *K* → ⟨*A*⟩*list_set_rel* → ⟨*K* ×$_r$ ⟨*A*⟩*list_set_rel*⟩*list_rel*
  assumes *full_split*: *set Li* = (⋃ *xs* ∈ *set Li_split*. *set xs*)
begin

lemma *lei_refine*[*refine_mono*]:
  ‹*RETURN* (*lei a b*) ≤ *RETURN* (*a'* ≼ *b'*)› **if** ‹(*a*, *a'*) ∈ *A*› ‹(*b*, *b'*) ∈ *A*›
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

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_pure where less_eq## 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 → A → bool_rel assumes Fi_F'[refine]: (Fi, F) ∈ K → A → bool_rel

locale *Reachability_Impl_pure* =
  Reachability_Impl_common **where** $M = M$ +
  Reachability_Impl_pure_base2 +
  Reachability_Impl_pre_start **where** $M = \lambda x.$ *case* $M\ x$ *of None* ⇒ {} | *Some* S ⇒ S
  for $M :: 'k \Rightarrow 'a$ *set option* +
  fixes *Mi* $l_0 i\ s_0 i$
  assumes *Mi_M*[*param*]: $(Mi,\ M) \in K \to \langle\langle A\rangle list\_set\_rel\rangle option\_rel$
  assumes $l_0 i\_l_0$[*refine*,*param*]: $(l_0 i,\ l_0) \in K$
    and $s_0 i\_s_0$[*refine*,*param*,*refine_mono*]: $(s_0 i,\ s_0) \in A$
begin

Refinement definition *check_invariant1* $L' \equiv$
do {
  monadic_list_all_split ($\lambda l.$
    case Mi l of
      None ⇒ *RETURN True*
    | *Some as* ⇒ *do* {
        let succs = get_succs l as;
        monadic_list_all ($\lambda(l',\ xs).$
        do {
          if xs = [] *then RETURN True*
          else do {
            case Mi l' of
              None ⇒ *RETURN False*
            | *Some ys* ⇒ *monadic_list_all* ($\lambda x.$
              monadic_list_ex ($\lambda y.$ *RETURN* (*lei x y*)) *ys*
) *xs*
          }
        }
) *succs*
      }
) L'
}

lemma *Mi_M_None_iff*[*simp*]:
  $M\ l$ = *None* ⟷ *Mi li* = *None* **if** $(li,\ l) \in K$
⟨*proof*⟩

lemma *check_invariant1_refine*[*refine*]:
  check_invariant1 L1 ≤ *check_invariant* L'
  if $(L1,\ L') \in \langle K\rangle list\_rel$ L = *dom M set* $L' \subseteq L$
  ⟨*proof*⟩

definition *check_prop1* **where**
  check_prop1 $L'\ M'$ = *do* {
  $l \leftarrow$ *RETURN* L';
  monadic_list_all ($\lambda l.$ *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
  ⟨proof⟩

lemma [refine]:
  (Mi l₀i ≠ None, l₀ ∈ L) ∈ bool_rel if L = dom M
  ⟨proof⟩

lemma [refine]:
  (Pi (l₀i, s₀i), P' (l₀, s₀)) ∈ bool_rel
  ⟨proof⟩

definition
  check_all_pre1 ≡ 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 (λs. RETURN (lei s₀i s)) xs;
    b4 ← check_prop1 Li Mi;
    RETURN (b1 ∧ b2 ∧ b3 ∧ b4)
  }
  }

lemma check_all_pre1_refine[refine]:
  (check_all_pre1, check_all_pre l₀ s₀) ∈ ⟨bool_rel⟩nres_rel if L = dom M
⟨proof⟩

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 $\leq$ *SPEC* ($\lambda r.\ r \longrightarrow$ *check_all_pre_spec* $l_0\ s_0 \wedge$ *check_invariant_spec* L) **if** $L =$ *dom M*
⟨*proof*⟩

definition
check_final1 $L' =$ *do* {
monadic_list_all ($\lambda l.$ *do* {
case op_map_lookup l Mi of None ⇒ *RETURN True* | *Some xs* ⇒ *do* {
monadic_list_all ($\lambda s.$
RETURN ($\neg$ *PR_CONST Fi* (l, s))
) *xs*
}
}
) L'
}

lemma *check_final1_refine*:
check_final1 Li $\leq$ *check_final'* L M
⟨*proof*⟩

definition
certify_unreachable1 = *do* {
b1 ← *check_all1*;
b2 ← *check_final1 Li*;
RETURN (*b1* $\wedge$ *b2*)
}

lemma *certify_unreachable1_correct*:
certify_unreachable1 $\leq$ *SPEC* ($\lambda r.\ r \longrightarrow$ *check_all_spec* $\wedge$ *check_final_spec*) **if** $L =$ *dom M*
⟨*proof*⟩

Synthesizing a pure program via rewriting **lemma** *check_final1_alt_def*:
check_final1 $L' =$ *RETURN* (*list_all_split*
($\lambda l.$ *case op_map_lookup l Mi of None* ⇒ *True* | *Some xs* ⇒ *list_all* ($\lambda s.\ \neg$ *Fi* (l, s)) *xs*) L')
⟨*proof*⟩

concrete_definition *check_final_impl*
uses *check_final1_alt_def* **is** _ = *RETURN ?f*

schematic_goal *check_prop1_alt_def*:
check_prop1 L' $M' \equiv$ *RETURN ?f*
⟨*proof*⟩

concrete_definition *check_prop_impl* **uses** *check_prop1_alt_def* **is** _ $\equiv$ *RETURN ?f*

schematic_goal *check_all_pre1_alt_def*:
check_all_pre1 $\equiv$ *RETURN ?f*
⟨*proof*⟩

concrete_definition *check_all_pre_impl* **uses** *check_all_pre1_alt_def* **is** _ $\equiv$ *RETURN ?f*

schematic_goal *check_invariant1_alt_def*:
check_invariant1 Li $\equiv$ *RETURN ?f*

⟨*proof*⟩

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

schematic_goal *certify_unreachable1_alt_def*:
certify_unreachable1 ≡ *RETURN ?f*
⟨*proof*⟩

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*
⟨*proof*⟩

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**
$M = \lambda x.$ *case M x of None* ⇒ {} | *Some S* ⇒ *abs_s ‘ S*
⟨*proof*⟩

end

locale *Reachability_Impl_pure_correct* =
Reachability_Impl_pure **where** $M = M$ +
Reachability_Impl_correct **where** $M = \lambda x.$ *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 $l_0 i$ $s_0 i$
$\longrightarrow (\nexists s'.\ E^{**}\ (l_0,\ s_0)\ s' \wedge F\ s')$
if $L = dom\ M$
⟨*proof*⟩

end

~~locale Reachability_Impl_simulation = Reachability_Impl_pure where E = E + Unreachability_Invariant_paired_p
where E = E' and less_eq = less_eq'' and less = less'' and P = P'' for E :: 'l × 's ⇒ 'l × 's
⇒ bool + and E' :: 'l' × 's' ⇒ 'l' × 's' ⇒ bool and E' :: 'l × 's'' ⇒ 'l × 's'' ⇒ bool and
less' :: 's'' ⇒ 's' ⇒ bool (infix ≺ 50) and less_eq' :: 's'' ⇒ 's' ⇒ bool (infix ≼ 50) +
and P' :: ('l' × 's') ⇒ bool and abs_l :: 'l ⇒ 'l' and abs_s :: 's ⇒ 's' and P'' :: ('l × 's'')
⇒ bool + and abs_l :: 'l ⇒ 'l and abs_s :: 's ⇒ 's' and F' :: ('l × 's') ⇒ boolbeginsublocale
correct: Reachability_Impl_correct where M = λx. case M x of None ⇒ {} | Some S ⇒ abs_s
‘ S and E = λ(l, s) (l', s'). True and P' = P'' and P = P'' and less_eq = less_eq'' and
less = less'' and succs = λl S. map (λ(l, S). (l, abs_s ‘ S)) (succs l {s. abs_s s ∈ S}) and F
= F' and s0 = abs_s s0'' apply standard subgoal for xs l apply (auto simp: succs_empty)
oopslemma certify_unreachable1_correct: certify_unreachable1 ≤ SPEC (λr. r ⟶ (∄s'. E'**
(abs_l l0, abs_s s0) s' ∧ F' s')) if L = dom Mproof - 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 by (refine_vcg~~

~~thm connect certify_unreachable[THEN mp, simp, code]~~

6 Certificates for Büchi Properties

locale *Buechi_Impl_pure* =
  Buechi_Impl_pre **where** $M = \lambda x.\ case\ M\ x\ of\ None \Rightarrow \{\} \mid Some\ S \Rightarrow S$ +
  Reachability_Impl_pure_base2
  for $M :: 'k \Rightarrow ('a \times nat)\ set\ option$ +
  fixes $Mi :: 'ki \Rightarrow ('ai \times nat)\ list\ option$
  assumes *Mi_M*[*param*]: $(Mi, M) \in K \to \langle\langle A \times_r Id\rangle list\_set\_rel\rangle option\_rel$
  assumes *F_mono*:
    $\bigwedge a\ b.\ F\ a \Longrightarrow P\ a \Longrightarrow (\lambda(l, s)\ (l', s').\ l' = l \wedge less\_eq\ s\ s')\ a\ b \Longrightarrow P\ b \Longrightarrow F\ b$
  fixes *inits* :: $('k \times 'a)\ set$ **and** *initsi* :: $('ki \times 'ai)\ list$
  assumes *initsi_inits*[*param*]: $(initsi, inits) \in \langle K \times_r A\rangle list\_set\_rel$
begin

Refinement definition *check_invariant_buechi'* $L' \equiv$
  monadic_list_all_split $(\lambda l.$
    case Mi l of
      None $\Rightarrow$ *RETURN True*
    | *Some as* $\Rightarrow$ *do* {
      monadic_list_all $(\lambda(x, i).$ *do* {
        let succs = get_succs l $[x]$;
        let is_accepting = Fi (l, x);
        let cmp = $(if\ is\_accepting\ then\ (\lambda j.\ i < j)\ else\ (\lambda j.\ i \leq j))$;
        monadic_list_all $(\lambda(l', xs).$
          if xs = $[]$ *then*
            RETURN True
          else
            case Mi l' of
              None $\Rightarrow$ *RETURN False*
            | *Some ys* $\Rightarrow$
              monadic_list_all $(\lambda y.$
                monadic_list_ex $(\lambda(z, j).$ *RETURN* $(lei\ y\ z \wedge cmp\ j))$ *ys*
) *xs*
) *succs*
      }) *as*
    }) L'

lemma *Mi_M_None_iff*[*simp*]:
  $M\ l = None \longleftrightarrow Mi\ li = None$ **if** $(li, l) \in K$
⟨*proof*⟩

lemma (**in** −) *list_set_rel_singletonI*[*param*]:
  assumes $(ai, a) \in A$
  shows $([ai], \{a\}) \in \langle A\rangle list\_set\_rel$
  ⟨*proof*⟩

lemma *check_invariant1_refine*[*refine*]:
  check_invariant_buechi' $L1 \leq$ *check_invariant_buechi buechi_prop* L'
  if $(L1, L') \in \langle K\rangle list\_rel\ L = dom\ M\ set\ L' \subseteq L$
  ⟨*proof*⟩

```
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
```

⟨*proof*⟩

sublocale *reachability*:
Reachability_Impl_pre **where** M = *image fst o* (λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *S*)
⟨*proof*⟩

lemma *check_prop_gt_SUCCEED*:
reachability.check_prop P' > *SUCCEED* **if** L = *dom M*
⟨*proof*⟩

lemma *check_prop_alt_def*:
check_prop' L M = *reachability.check_prop P'* **if** L = *dom M*
⟨*proof*⟩

lemma *check_init1_refine*:
(*check_init1*, *reachability.check_init*) ∈ $K \to A \to \langle$*bool_rel*⟩*nres_rel* **if** L = *dom M*
⟨*proof*⟩

lemma *check_all_pre1_correct*[*refine*]:
check_all_pre1 ≤ *SPEC* (λ*r*. *r* ⟶ *check_all_pre_spec1 inits*) **if** L = *dom M*
⟨*proof*⟩

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*⟩

Synthesizing a pure program via rewriting **schematic_goal** *check_prop1_alt_def*:
check_prop1 L' M' ≡ *RETURN ?f*
⟨*proof*⟩

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*)

}
⟨*proof*⟩

schematic_goal *check_all_pre1_alt_def*:
check_all_pre1 ≡ *RETURN ?f*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *PRINT_CHECK_unfold*:
PRINT_CHECK s x = *RETURN* (*print_check s x*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

end

locale *Reachability_Impl_imp_to_pure_base* = *Certification_Impl_base*
where $K = K$ **and** $A = A$
for $K :: 'k \Rightarrow ('ki :: \{hashable,heap\}) \Rightarrow assn$ **and** $A :: 's \Rightarrow ('si :: heap) \Rightarrow assn$
+
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*: $(L\_list, L') \in \langle the\_pure\ K\rangle list\_rel\ (Li, L') \in \langle K\_rel\rangle list\_rel\ set\ L' = L$
assumes *to_state_ht*: $(s1, s) \in A\_rel \Longrightarrow <emp>\ to\_state\ s1\ <\lambda si.\ A\ s\ si>$
assumes *from_state_ht*: $<A\ s\ si>\ from\_state\ si\ <\lambda s'.\ \uparrow((s', s) \in A\_rel)>_t$
assumes *from_loc*: $(li, l) \in the\_pure\ K \Longrightarrow (from\_loc\ li, l) \in K\_rel$
assumes *to_loc*: $(l1, l) \in K\_rel \Longrightarrow (to\_loc\ l1, l) \in the\_pure\ K$
assumes *K_rel*: *single_valued K_rel single_valued* (K_rel^{-1})
assumes *lei_less_eq*: $(lei, (\preceq)) \in A\_rel \to A\_rel \to bool\_rel$
assumes *full_split*: $set\ Li = (\bigcup xs \in set\ Li\_split.\ set\ xs)$
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*⟩

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* ×$_a$ *A*) *a ai*> **if** (*a1*, *a*) ∈ *K_rel* ×$_r$ *A_rel*
⟨*proof*⟩

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*

~~$s_0 i = run\_heap\ (do\ \{s \leftarrow s_0 i;\ from\_state\ s\})$~~
  ⟨*proof*⟩

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 = 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**
    $l_0 i$ = *from_loc* (*run_heap* $l_0 i$) **and**
    $s_0 i$ = *run_heap* (*do* {$s \leftarrow s_0 i$; *from_state s*})
  ⟨*proof*⟩

end

locale *Reachability_Impl_imp_to_pure_correct* =
  Reachability_Impl_imp_to_pure **where** *M* = *M*
  + *Reachability_Impl_correct* **where** *M* = λ*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 = 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**
    $l_0 i$ = *from_loc* (*run_heap* $l_0 i$) **and**
    $s_0 i$ = *run_heap* (*do* {$s \leftarrow s_0 i$; *from_state s*})
  ⟨*proof*⟩

end

locale *Buechi_Impl_imp_to_pure* = *Buechi_Impl_pre* **where**
M = λ*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* ⇒ (*'a* × *nat*) *set option*
and *K_rel* :: (*'ki* × *'k*) *set* **and** *A_rel* :: (*'ai* × *'a*) *set* +
fixes *inits initsi*
assumes *initsi_inits*: (*uncurry0 initsi*, *uncurry0* (*RETURN* (*PR_CONST inits*)))
∈ *unit_assn*^{k} →$_a$ *list_assn* (*K* ×$_a$ *A*)
fixes *Mi* :: *'ki* ⇒ (*'ai* × *nat*) *list option*
assumes *Mi_M*: (*Mi*, *M*) ∈ *K_rel* → ⟨⟨*A_rel* ×$_r$ *Id*⟩*list_set_rel*⟩*option_rel*
begin

lemma (**in** −) *list_all2_flip*:
list_all2 P xs ys **if** *list_all2 Q ys xs* (⋀*x y*. *Q y x* ⟹ *P x y*)
⟨*proof*⟩

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 = 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**
inits = *set inits* **and**
~~*initsi* = *map* (λ(*l*, *s*). *do* {*s* ← *from_state s*; *return* (*from_loc l*, *s*)}) (*run_heap initsi*)~~
initsi =
run_heap (*do* {
xs ← *initsi*;
Heap_Monad.fold_map (λ(*l*, *s*). *do* {*s* ← *from_state s*; *return* (*from_loc l*, *s*)}) *xs*})
⟨*proof*⟩

end

locale *Buechi_Impl_imp_to_pure_correct* =
Buechi_Impl_imp_to_pure **where** *M* = *M* +
Buechi_Impl_correct **where** *M* = λ*x*. *case M x of None* ⇒ {} | *Some S* ⇒ *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 = 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**
inits = *set inits* **and**
~~*initsi* = *map* (λ(*l*, *s*). (*from_loc l*, *from_state s*)) *run_heap initsi*~~
initsi =
run_heap (*do* {
xs ← *initsi*;
Heap_Monad.fold_map (λ(*l*, *s*). *do* {*s* ← *from_state s*; *return* (*from_loc l*, *s*)}) *xs*})
⟨*proof*⟩

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*) = (∃ *as bs*. *xs* = *as* @ *bs* ∧ *map f as* = *ys* ∧ *map f bs* = *zs*)
⟨*proof*⟩

lemma *Cons_subseq_iff*:
subseq (*x* # *xs*) *ys* ⟷ (∃ *as bs*. *ys* = *as* @ *x* # *bs* ∧ *subseq xs bs*)
⟨*proof*⟩

lemma *append_subseq_iff*:
subseq (*as* @ *bs*) *xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *zs* ∧ *subseq as ys* ∧ *subseq bs zs*)
⟨*proof*⟩

context *Graph_Defs*
begin

lemma *steps_append_singleE*:
assumes *steps* (*xs* @ [*x*])
obtains *xs* = [] | *ys y* **where** *xs* = *ys* @ [*y*] *steps xs y* → *x*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *steps_singleI*:
steps [*a*,*b*] **if** $a \rightarrow b$
⟨*proof*⟩

end

7.2 Backward Simulations

locale *Backward_Simulation* = *Simulation* **where** $A = E$ **and** $B = E$ **and** *sim* = *sim*
for $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** *sim* (**infix** $\preceq$ *60*) +
fixes $G :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$
assumes *simulation*: $b \in B \Longrightarrow G\ A\ B \Longrightarrow \exists\, a \in A.\ \exists\, b'.\ a \rightarrow b' \wedge b \preceq b'$
and *refl*[*intro*, *simp*]: $a \preceq a$ **and** *trans*: *transp* ($\preceq$)
begin

lemmas *A_simulation_steps* = *simulation_steps*

sublocale *Graph_Defs G* ⟨*proof*⟩

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

lemma *backward_simulation_reaches*:
$\exists\, a \in A.\ \exists\, b'.\ E^{**}\ a\ b' \wedge b \preceq b'$ **if** $G^{**}\ A\ B\ b \in B$
⟨*proof*⟩

lemma *backward_simulation_steps*:
$\exists\, a \in A.\ \exists\, as\ b'.\ A.steps\ (a\ \#\ as\ @\ [b'])\ \wedge\ b \preceq b'$ **if** $steps\ (A\ \#\ As\ @\ [B])\ b \in B$
⟨*proof*⟩

lemma *backward_simulation_reaches1*:
$\exists\, a \in A.\ \exists\, b'.\ E^{++}\ a\ b' \wedge b \preceq b'$ **if** $G^{++}\ A\ B\ b \in B$
⟨*proof*⟩

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* $= n\ a \preceq y\ x \in A$
⟨*proof*⟩

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*]:
$x \preceq x$
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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 $E :: 'a \Rightarrow 'a \Rightarrow bool$ **and** *sim* (**infix** $\preceq$ *60*) **and** *sim'* (**infix** $\preceq''$ *60*)
begin

Corresponds to lemma 12 of [1].

lemma *cycle_Buechi_run*:
assumes *steps* (A # *As* @ [A]) $a \in A \forall a \in A.\ P\ a \forall a \in A.\ \varphi\ a$
$\forall x\ y.\ x \preceq y \wedge x \in A \wedge \varphi\ x \longrightarrow \varphi\ y \forall x\ y.\ x \preceq' y \wedge \varphi\ x \longrightarrow \varphi\ y$
obtains x *xs* **where** *A.run* (x ## *xs*) *infs* φ *xs* $x \in A$
⟨*proof*⟩

end

lemma (**in** *Simulation_Invariant*) *simulation_run'*:
assumes *A.run* (*x ## xs*) *x* $\sim$ *y PA x PB y*
shows $\exists$ *ys*. *B.run* (*y ## ys*) $\wedge$ *stream_all2* (λ*a b*. *a* $\sim$ *b* $\wedge$ *PA a* $\wedge$ *PB b*) *xs ys*
⟨*proof*⟩

context *Graph_Invariant_Start*
begin

lemma *reachable_reaches_equiv*: *G'.reaches x y* $\longleftrightarrow$ *x* $\rightarrow*$ *y* **if** *reachable x* **for** *x y*
⟨*proof*⟩

lemma *reachable_reaches1_equiv*: *G'.reaches1 x y* $\longleftrightarrow$ *x* $\rightarrow^+$ *y* **if** *reachable x* **for** *x y*
⟨*proof*⟩

lemma *reachable_steps_equiv*:
G'.steps (*x # xs*) $\longleftrightarrow$ *steps* (*x # xs*) **if** *reachable x*
⟨*proof*⟩

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*
⟨*proof*⟩

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* λ*x y*. *G x y* $\wedge$ *Q x* $\wedge$ *Q y* +
backward: *Backward_Double_Simulation* **where** *E* = *E* **and** *G* = λ*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
⟨*proof*⟩

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* $\wedge$ φ *x* $\longrightarrow$ φ *y*
$\forall$ *x y*. *x* $\preceq$ *y* $\wedge$ φ *x* $\longrightarrow$ φ *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*⟩

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

lemma *step_trans_z'_sound*:
$A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z'\rangle \Longrightarrow \forall u' \in Z'.\ \exists u \in Z.\ \exists d.\ \ A \vdash' \langle l, u\rangle \rightarrow^t \langle l', u'\rangle$
⟨*proof*⟩

lemma *step_trans_z'_exact_strong*:
assumes $A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z'\rangle$
shows $Z' = \{u'.\ \exists u \in Z.\ A \vdash' \langle l, u\rangle \rightarrow^t \langle l', u'\rangle\}$
⟨*proof*⟩

lemma *step_a_step_trans_iff*:
$A \vdash \langle l, u\rangle \rightarrow_a \langle l', u'\rangle \longleftrightarrow (\exists g\ r.\ A \vdash_t \langle l, u\rangle \rightarrow_{(g,a,r)} \langle l', u'\rangle)$
⟨*proof*⟩

lemma *step_trans'_step_trans_iff*:
$(\exists t.\ A \vdash' \langle l, u\rangle \rightarrow^t \langle l', u'\rangle) \longleftrightarrow A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle$
⟨*proof*⟩

8.2 Time-Abstract Simulation

locale *Time_Abstract_Simulation* =
  fixes $A :: ('a, 'c, 't :: time, 'l)\ ta$
  fixes $sim :: 'l \times ('c \Rightarrow 't :: time) \Rightarrow 'l \times ('c \Rightarrow 't) \Rightarrow bool$ (**infix** $\preceq$ *60*)
  assumes *sim*:
  $\bigwedge l\ l'\ l_1\ u\ u'\ u_1\ t.\ (l, u) \preceq (l', u') \Longrightarrow A \vdash' \langle l, u\rangle \rightarrow^t \langle l_1, u_1\rangle$
    $\Longrightarrow \exists\, u_1'.\ A \vdash' \langle l', u'\rangle \rightarrow^t \langle l_1, u_1'\rangle \wedge (l_1, u_1) \preceq (l_1, u_1')$
  assumes *refl*: $\bigwedge u.\ u \preceq u$ **and** *trans*: $\bigwedge u\ v\ w.\ u \preceq v \Longrightarrow v \preceq w \Longrightarrow u \preceq w$
begin

lemma *simE*:
  assumes $(l, u) \preceq (l', u')\ A \vdash' \langle l, u\rangle \rightarrow^t \langle l_1, u_1\rangle$
  obtains u_1' **where** $A \vdash' \langle l', u'\rangle \rightarrow^t \langle l_1, u_1'\rangle\ (l_1, u_1) \preceq (l_1, u_1')$
  ⟨*proof*⟩

definition *abs* :: $'l \Rightarrow ('c, 't)\ zone \Rightarrow ('c, 't)\ zone$ (α _ _ [*71*,*71*] *71*) **where**
  $\alpha\ l\ W = \{v.\ \exists\, v' \in W.\ (l, v) \preceq (l, v')\}$

lemma *simulation_mono*:
  assumes $\alpha\ l\ Z \subseteq \alpha\ l\ Z'\ A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l_1, Z_1\rangle\ A \vdash' \langle l, Z'\rangle \rightsquigarrow^t \langle l_1, Z_1'\rangle$
  shows $\alpha\ l_1\ Z_1 \subseteq \alpha\ l_1\ Z_1'$
⟨*proof*⟩

lemma *simulation*:
  assumes $\alpha\ l\ Z = \alpha\ l\ Z'\ A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z_1\rangle\ A \vdash' \langle l, Z'\rangle \rightsquigarrow^t \langle l', Z_1'\rangle$
  shows $\alpha\ l'\ Z_1 = \alpha\ l'\ Z_1'$
  ⟨*proof*⟩

lemma *simulation'*:
  assumes $\alpha\ l\ Z = \alpha\ l\ Z'\ A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z_1\rangle$
  shows $\exists\, Z_1'.\ A \vdash' \langle l, Z'\rangle \rightsquigarrow^t \langle l', Z_1'\rangle \wedge \alpha\ l'\ Z_1 = \alpha\ l'\ Z_1'$
⟨*proof*⟩

lemma *abs_involutive*:
  $\alpha\ l\ (\alpha\ l\ Z) = \alpha\ l\ Z$
  ⟨*proof*⟩

lemma *abs_widens*:
  $Z \subseteq \alpha\ l\ Z$
  ⟨*proof*⟩

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

corollary *transition_compatibility*:
  assumes $A \vdash' \langle l, \alpha\ l\ Z\rangle \rightsquigarrow^t \langle l', W\rangle\ A \vdash' \langle l, Z\rangle \rightsquigarrow^t \langle l', Z'\rangle$
  shows $\alpha\ l'\ W = \alpha\ l'\ Z'$
  ⟨*proof*⟩

inductive *step_abs* ::
  $('a, 'c, 't, 'l)\ ta \Rightarrow 'l \Rightarrow ('c, 't)\ zone \Rightarrow 'a \Rightarrow 'l \Rightarrow ('c, 't)\ zone \Rightarrow bool$
(_ ⊢ ⟨_, _⟩ $\rightsquigarrow_{\alpha(\_)}$ ⟨_, _⟩ [*61*,*61*,*61*] *61*)
where
  step_alpha:

$A \vdash \langle l, Z\rangle \rightsquigarrow_\tau \langle l', Z'\rangle \Longrightarrow A \vdash \langle l', Z'\rangle \rightsquigarrow_{\restriction a} \langle l'', Z''\rangle$
$\Longrightarrow A \vdash \langle l, \alpha\ l\ Z\rangle \rightsquigarrow_{\alpha(a)} \langle l'', \alpha\ l''\ Z''\rangle$

interpretation *sim1*: *Simulation* **where**
$A = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \to \langle l', u'\rangle$ **and**
$B = \lambda(l, Z)\ (l', Z').\ \exists\, a.\ A \vdash \langle l, Z\rangle \rightsquigarrow_{\alpha(a)} \langle l', Z'\rangle$ **and**
$sim = \lambda(l, u)\ (l', Z).\ l' = l \wedge u \in Z \wedge \alpha\ l\ Z = Z$
⟨*proof*⟩

interpretation *sim2*: *Simulation* **where**
$A = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \to \langle l', u'\rangle$ **and**
$B = \lambda(l, Z)\ (l', Z').\ \exists\, a.\ A \vdash \langle l, \alpha\ l\ Z\rangle \rightsquigarrow_{\alpha(a)} \langle l', \alpha\ l'\ Z'\rangle$ **and**
$sim = \lambda(l, u)\ (l', Z).\ l' = l \wedge u \in Z$
⟨*proof*⟩

sublocale *self_simulation*: *Self_Simulation* **where**
$E = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \to \langle l', u'\rangle$ **and** $P = \lambda\_.\ True$
⟨*proof*⟩

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 $\exists\, R \in \mathcal{R}\ l.\ u \in R$
⟨*proof*⟩

lemma *regions_finite*:
finite $(\mathcal{R}\ l)$
⟨*proof*⟩

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

lemma *refl*:
$(l, u) \equiv_M (l, u)$
⟨*proof*⟩

lemma *sym*:
$(l, u) \equiv_M (l', u') \longleftrightarrow (l', u') \equiv_M (l, u)$
⟨*proof*⟩

lemma *trans*:
$(l, u) \equiv_M (l'', u'')$ **if** $(l, u) \equiv_M (l', u')$ $(l', u') \equiv_M (l'', u'')$
⟨*proof*⟩

lemma *equiv*:
equivp $(\equiv_M)$
⟨*proof*⟩

lemma *same_loc*:
$l' = l$ **if** $(l, u) \equiv_M (l', u')$ *valid* (l, u)
⟨*proof*⟩

lemma *regions_simI*:
$(l, u) \equiv_M (l, u')$ **if** $l \in$ *state_set* A $R \in \mathcal{R}$ l $u \in R$ $u' \in R$
⟨*proof*⟩

lemma *regions_simD*:
$u' \in R$ **if** $l \in$ *state_set* A $R \in \mathcal{R}$ l $u \in R$ $(l, u) \equiv_M (l', u')$
⟨*proof*⟩

lemma *finite_quotient*:
finite (*UNIV* // $\{(x, y).\ x \equiv_M y\}$)
⟨*proof*⟩

sublocale *region_self_simulation*: *Self_Simulation* **where**
$E = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle$ **and** $sim = (\equiv_M)$ **and** $P = valid$
⟨*proof*⟩

end

8.3 LU-Simulation

definition
constraints_of A $l = \bigcup$ (*set* ‘ *insert* (*inv_of* A l) $\{g.\ \exists a\ r\ l'.\ (l, g, a, r, l') \in$ *trans_of* $A\}$)

definition
is_lower A $L \equiv$
$\forall l.\ \forall ac \in$ *constraints_of* A l. *case ac of*
GT c $x \Rightarrow L$ l $c \geq x$ |
GE c $x \Rightarrow L$ l $c \geq x$ |
EQ c $x \Rightarrow L$ l $c \geq x$ |
_ $\Rightarrow$ *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 $c \geq x$ |
LE c $x \Rightarrow U$ l $c \geq x$ |
EQ c $x \Rightarrow U$ l $c \geq x$ |
_ $\Rightarrow$ *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 $\forall x \in clk\_set\ A - set\ r.\ k\ l\ x \geq k\ l_1\ x$
⟨*proof*⟩

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 $l' = l\ v\ x = v'\ x$
$\mid l' = l\ v\ x > v'\ x\ v'\ x > L\ l\ x$
$\mid l' = l\ v\ x < v'\ x\ v\ x > U\ l\ x$
⟨*proof*⟩

lemma *sim_locD*:
$l' = l$ **if** $(l,\ v) \preceq (l',\ v')$
⟨*proof*⟩

lemma *sim_nonneg*:
$u\ x \geq 0$ **if** $(l,\ u) \preceq (l',\ u')\ u'\ x \geq 0\ x \in clk\_set\ A\ U\ l\ x \geq 0$
⟨*proof*⟩

lemma *sim_time_shift*:
$(l,\ v \oplus d) \preceq (l',\ v' \oplus d)$ **if** $(l,\ v) \preceq (l',\ v')\ d \geq 0$
⟨*proof*⟩

lemma *constraints_of_clk_set*:
assumes $g \in constraints\_of\ A\ l$
shows
$g = LT\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = LE\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = EQ\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = GE\ c\ x \Longrightarrow c \in clk\_set\ A$
$g = GT\ c\ x \Longrightarrow c \in clk\_set\ A$
⟨*proof*⟩

lemma *constraint_simulation*:
assumes $g \in constraints\_of\ A\ l\ (l,\ v) \preceq (l',\ v')\ v \vdash_a g$
shows $v' \vdash_a g$
⟨*proof*⟩

lemma *inv_simulation*:
assumes $v \vdash inv\_of\ A\ l\ (l,\ v) \preceq (l',\ v')$

shows $v' \vdash$ *inv_of A l*
⟨*proof*⟩

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*⟩

lemma *sim_delay*:
assumes $(l,\ v) \preceq (l',\ v')$ $d \geq 0$
shows $(l,\ v \oplus d) \preceq (l',\ v' \oplus d)$
⟨*proof*⟩

lemma *clock_set_iff*:
$([r{\rightarrow}0]v)\ c = (\mathit{if}\ c \in \mathit{set}\ r\ \mathit{then}\ 0\ \mathit{else}\ v\ c)$
⟨*proof*⟩

lemma *sim_reset*:
assumes $A \vdash l \longrightarrow^{g,a,r} l_1$ $v \vdash g\ (l,\ v) \preceq (l',\ v')$
shows $(l_1,\ [r{\rightarrow}0]v) \preceq (l_1,\ [r{\rightarrow}0]v')$
⟨*proof*⟩

lemma *step_t_simulation*:
$(l,\ u) \preceq (l',\ u') \Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow^{d} \langle l_1,\ u_1\rangle$
$\Longrightarrow \exists\, u_1{}'.\ A \vdash \langle l_1,\ u'\rangle \rightarrow^{d} \langle l_1,\ u_1{}'\rangle \wedge (l_1,\ u_1) \preceq (l,\ u_1{}')$
⟨*proof*⟩

lemma *step_a_simulation*:
$(l,\ u) \preceq (l',\ u') \Longrightarrow A \vdash \langle l,\ u\rangle \rightarrow_a \langle l_1,\ u_1\rangle$
$\Longrightarrow \exists\, u_1{}'.\ A \vdash \langle l,\ u'\rangle \rightarrow_a \langle l_1,\ u_1{}'\rangle \wedge (l_1,\ u_1) \preceq (l_1,\ u_1{}')$
⟨*proof*⟩

lemma *step_trans_simulation*:
$(l,\ u) \preceq (l',\ u') \Longrightarrow A \vdash_t \langle l,\ u\rangle \rightarrow_t \langle l_1,\ u_1\rangle$
$\Longrightarrow \exists\, u_1{}'.\ A \vdash_t \langle l,\ u'\rangle \rightarrow_t \langle l_1,\ u_1{}'\rangle \wedge (l_1,\ u_1) \preceq (l_1,\ u_1{}')$
⟨*proof*⟩

sublocale *Time_Abstract_Simulation A sim*
⟨*proof*⟩

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
$\forall\, l \in L.\ \forall\, s \in M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s'' \in M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
⟨*proof*⟩

interpretation *Simulation* **where**
$A = E'$ **and**
$B = E$ **and**
$sim = \lambda(l,\ s)\ (l',\ s').\ l' = l \wedge sim\ (l,\ s')\ (l',\ s)$
⟨*proof*⟩

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*: $\bigwedge l\ s\ l1\ s1\ l2\ s2.$
$l \in L \Longrightarrow s \in M'\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M'\ l2 \Longrightarrow E'\ (l,\ s)\ (l1,\ s1) \Longrightarrow SE'\ (l1,\ s1)\ (l2,\ s2)$
$\Longrightarrow$
$f'\ (l,\ s) \leq f'\ (l2,\ s2)$
⟨*proof*⟩

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**
$E' = E'$
⟨*proof*⟩

interpretation *sim2*: *Simulation* **where**
$sim = \lambda(l,\ x)\ (l',\ y).\ l' = l \wedge y = \alpha\ l\ x$ **and**
$A = E$ **and**
$B = E'$
⟨*proof*⟩

interpretation *bisim*: *Bisimulation* **where**
$sim = \lambda(l,\ x)\ (l',\ y).\ l' = l \wedge y = \alpha\ l\ x$ **and**
$A = E$ **and**
$B = E'$
⟨*proof*⟩

lemma *simulationE*:
assumes $\alpha\ l\ a = \alpha\ l\ a'\ E\ (l,\ a)\ (l',\ b)$
obtains b' **where** $E\ (l,\ a')\ (l',\ b')\ \alpha\ l'\ b = \alpha\ l'\ b'$
⟨*proof*⟩

lemma *M'_eq*:
$sim.M' = M'$
⟨*proof*⟩

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
$\forall\, l{\in}L.\ \forall\, s{\in}M'\ l.\ \forall\, l'\ s'.\ E'\ (l,\ s)\ (l',\ s') \longrightarrow l' \in L \wedge (\exists\, s''{\in}M'\ l'.\ SE'\ (l',\ s')\ (l',\ s''))$
⟨*proof*⟩

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*⟩

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' $f = f'$
⟨*proof*⟩

lemma *topo_simulation*: $\bigwedge l\ s\ l1\ s1\ l2\ s2.$
$l \in L \Longrightarrow s \in M'\ l \Longrightarrow l2 \in L \Longrightarrow s2 \in M'\ l2 \Longrightarrow E'\ (l,\ s)\ (l1,\ s1) \Longrightarrow SE'\ (l1,\ s1)\ (l2,\ s2) \Longrightarrow$
$f'\ (l,\ s) \leq f'\ (l2,\ s2)$
⟨*proof*⟩

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*:
$A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z''\rangle \longleftrightarrow (\exists t.\ A \vdash' \langle l,\ Z\rangle \rightsquigarrow^t \langle l',\ Z''\rangle)$
⟨*proof*⟩

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**
$\alpha = abs$
⟨*proof*⟩

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**
$sim = \lambda(l,\ Z)\ (l',\ Z').\ l' = l \wedge Z' = \alpha\ l\ Z$
⟨*proof*⟩

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*:
$(\exists\ a.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow_{\beta(a)} \langle l',\ W\rangle) \longleftrightarrow (\exists\ Z'.\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge W = \beta\ l'\ Z')$
⟨*proof*⟩

lemma *step_betaE*:
assumes $A \vdash \langle l,\ Z\rangle \rightsquigarrow_{\beta(a)} \langle l',\ W\rangle$
obtains Z' **where** $A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle\ W = \beta\ l'\ Z'$
⟨*proof*⟩

definition
$loc\_is\ l\ s \equiv \forall\ (l',\ \_) \in s.\ l' = l$

lemma α_V:
$\alpha\ l\ Z \subseteq V$ **if** $Z \subseteq V$
⟨*proof*⟩

lemma β_V:
$\beta\ l\ Z \subseteq V$ **if** $Z \subseteq V\ I\ Z\ l \in state\_set\ A$
⟨*proof*⟩

lemma *step_z'_V*:
$Z' \subseteq V$ **if** $A \vdash \langle l, Z\rangle \rightsquigarrow \langle l', Z'\rangle\ Z \subseteq V$
⟨*proof*⟩

Corresponds to lemma 6 of [1].

lemma *backward_simulation*:
assumes
$b \in S'\ loc\_is\ l\ S\ loc\_is\ l'\ S'\ A \vdash \langle l, R\_of\ S\rangle \rightsquigarrow_{\beta(a)} \langle l', R\_of\ S'\rangle$
$I\ (R\_of\ S)\ R\_of\ S \subseteq V$
shows $\exists\, a{\in}S.\ \exists\, b'.\ (case\ a\ of\ (l, u) \Rightarrow \lambda(l', u').\ A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle)\ b' \wedge b \preceq b'$
⟨*proof*⟩

lemma *step'_step_beta*:
assumes
$(l, u) \in a'\ A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle\ loc\_is\ l1\ a'\ R\_of\ a' \subseteq V\ I\ (R\_of\ a')$
shows
$\exists\, b'.\ (\exists\, a\ l\ l'.\ loc\_is\ l\ a' \wedge loc\_is\ l'\ b' \wedge a' \neq \{\} \wedge b' \neq \{\} \wedge$
$A \vdash \langle l, R\_of\ a'\rangle \rightsquigarrow_{\beta a} \langle l', R\_of\ b'\rangle) \wedge (l', u') \in b'$
⟨*proof*⟩

definition *beta_step* **where**
$beta\_step \equiv \lambda s\ s'.\ \exists\, a\ l\ l'.\ loc\_is\ l\ s \wedge loc\_is\ l'\ s' \wedge s \neq \{\} \wedge s' \neq \{\} \wedge$
$A \vdash \langle l, R\_of\ s\rangle \rightsquigarrow_{\beta(a)} \langle l', R\_of\ s'\rangle$

lemma *beta_step_inv*:
assumes $beta\_step\ a\ b\ \exists\, l{\in}state\_set\ A.\ loc\_is\ l\ a \wedge R\_of\ a \subseteq V \wedge I\ (R\_of\ a)$
shows $\exists\, l{\in}state\_set\ A.\ loc\_is\ l\ b \wedge R\_of\ b \subseteq V \wedge I\ (R\_of\ b)$
⟨*proof*⟩

lemma *from_R_R_of*:
assumes $loc\_is\ l\ S$
shows $from\_R\ l\ (R\_of\ S) = S$
⟨*proof*⟩

interpretation *backward_simulation*: *Backward_Double_Simulation_Complete* **where**
$E = \lambda(l, u)\ (l', u').\ A \vdash' \langle l, u\rangle \rightarrow \langle l', u'\rangle$ **and**
$G = beta\_step$ **and**
$sim' = (\equiv_M)$ **and**
$P = valid$ **and**
$Q = \lambda s.\ \exists\, l \in state\_set\ A.\ loc\_is\ l\ s \wedge R\_of\ s \subseteq V \wedge I\ (R\_of\ s)$ **and**
$a_0 = from\_R\ l_0\ Z_0$
⟨*proof*⟩

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'*)
  ⟨*proof*⟩

lemma *dbm_nonneg_abstra*[*dbm_nonneg*]:
  assumes *dbm_nonneg n M*
  shows *dbm_nonneg n* (*abstra ac M v*)
  ⟨*proof*⟩

lemma *dbm_nonneg_abstr*[*dbm_nonneg*]:
  assumes *dbm_nonneg n M*
  shows *dbm_nonneg n* (*abstr g M v*)
  ⟨*proof*⟩

lemma *dbm_nonneg_up*[*dbm_nonneg*]:
  dbm_nonneg n (*up M*) **if** *dbm_nonneg n M*
  ⟨*proof*⟩

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*)

⟨*proof*⟩

lemma *dbm_nonneg_reset'*[*dbm_nonneg*]:
fixes $M :: 't :: time\ DBM$
assumes $dbm\_nonneg\ n\ M\ \forall\, c \in set\ r.\ v\ c > 0$
shows $dbm\_nonneg\ n\ (reset'\ M\ n\ r\ v\ 0)$
⟨*proof*⟩

lemma *dbm_nonneg_fw_upd*[*dbm_nonneg*]:
$dbm\_nonneg\ n\ (fw\_upd\ M\ k'\ i\ j)$ **if** $dbm\_nonneg\ n\ M$
⟨*proof*⟩

lemma *dbm_nonneg_fwi*[*dbm_nonneg*]:
$dbm\_nonneg\ n\ (fwi\ M\ n\ k'\ i\ j)$ **if** $dbm\_nonneg\ n\ M$
⟨*proof*⟩

lemma *dbm_nonneg_fw*[*dbm_nonneg*]:
$dbm\_nonneg\ n\ (fw\ M\ n\ k)$ **if** $dbm\_nonneg\ n\ M$
⟨*proof*⟩

lemma *dbm_nonneg_FW*[*dbm_nonneg*]:
$dbm\_nonneg\ n\ (FW\ M\ n)$ **if** $dbm\_nonneg\ n\ M$
⟨*proof*⟩

definition
$empty\_dbm \equiv \lambda\_\ \_.\ Lt\ 0$

lemma *neg_diag_zero_empty_dbmI*:
assumes $M\ 0\ 0 < 0$
shows $[M]_{v,n} = \{\}$
⟨*proof*⟩

lemma *empty_dbm_empty_zone*:
$[empty\_dbm]_{v,n} = \{\}$
⟨*proof*⟩

lemma *canonical_empty_dbm*:
$canonical\ empty\_dbm\ n$
⟨*proof*⟩

lemma *dbm_int_empty_dbm*:
$dbm\_int\ empty\_dbm\ n$
⟨*proof*⟩

lemma *dbm_nonneg_empty_dbm*:
$dbm\_nonneg\ n\ empty\_dbm$
⟨*proof*⟩

lemmas [*simp*] = *any_le_inf*

lemma *DBM_val_boundedD1*:
assumes $u \vdash_{v,n} M\ v\ c \leq n$
shows $Le\ (-\ u\ c) \leq M\ 0\ (v\ c)$
⟨*proof*⟩

lemma *DBM_val_boundedD2*:
assumes $u \vdash_{v,n} M\ v\ c \leq n$
shows $Le\ (u\ c) \leq M\ (v\ c)\ 0$
⟨*proof*⟩

lemma *DBM_val_boundedD3*:
assumes $u \vdash_{v,n} M\ v\ c1 \leq n\ v\ c2 \leq n$
shows $Le\ (u\ c1 - u\ c2) \leq M\ (v\ c1)\ (v\ c2)$
⟨*proof*⟩

lemma *dbm_default_And*:
assumes *dbm_default M n dbm_default M′ n*
shows *dbm_default* $(And\ M\ M')\ n$
⟨*proof*⟩

lemma *dbm_default_abstra*:
assumes *dbm_default M n constraint_pair ac* $= (x,\ m)\ v\ x \leq n$
shows *dbm_default* $(abstra\ ac\ M\ v)\ n$
⟨*proof*⟩

lemma *dbm_default_abstr*:
assumes *dbm_default M n* $\forall (x,\ m) \in collect\_clock\_pairs\ g.\ v\ x \leq n$
shows *dbm_default* $(abstr\ g\ M\ v)\ n$
⟨*proof*⟩

lemma *dbm_entry_dense*:
fixes $a\ b :: 't :: time\ DBMEntry$
assumes $a + b \geq 0\ b > 0$
obtains x **where** $x > 0\ b \geq Le\ x\ Le\ (-x) \leq a$
⟨*proof*⟩
include *no_library_syntax*
⟨*proof*⟩

lemma *canonical_diag*:
fixes $M :: 't :: time\ DBM$
assumes $canonical\ M\ n\ i \leq n$
shows $M\ i\ i \geq Lt\ 0$
⟨*proof*⟩

lemma *canonical_diag_nonnegI*:
fixes $M :: 't :: time\ DBM$
assumes $canonical\ M\ n \forall i \leq n.\ M\ i\ i \neq Lt\ 0$
shows $\forall i \leq n.\ M\ i\ i \geq 0$
⟨*proof*⟩

context *Regions_common*
begin

lemma *canonical_non_emptyI*:
assumes $[M]_{v,n} \neq \{\}$
shows $canonical\ (FW\ M\ n)\ n$
⟨*proof*⟩

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 $[M]_{v,n} \subseteq V$
⟨*proof*⟩

lemma *canonical_dbm_valid*:
valid_dbm M **if** *canonical_dbm M*
⟨*proof*⟩

lemma *dbm_nonnegI*:
assumes *canonical M n* $[M]_{v,n} \subseteq V$ ∀ *i* ≤ *n*. *M i i* ≠ *Lt 0*
shows *dbm_nonneg n M*
⟨*proof*⟩

lemma *vabstr′_V*:
obtains *M* **where** *vabstr′ V M*
⟨*proof*⟩

lemma *canonical_dbm_empty_dbm*:
canonical_dbm empty_dbm
⟨*proof*⟩

lemma *vabstr′_empty_dbm*:
vabstr′ {} *empty_dbm*
⟨*proof*⟩

lemma *vabstr′I*:
assumes *dbm_int M′ n Z′* ⊆ *V Z′* = $[M']_{v,n}$
obtains *M′* **where** *vabstr′ Z′ M′*
⟨*proof*⟩

end

context *Regions_TA*
begin

The following does not hold:

lemma
fixes *M* :: *real DBM*
assumes *canonical M n*
and ∀ *i* ≤ *n*. *M 0 i* ≤ *0*
shows ∀ *i* ≤ *n*. *0* ≤ *M i i*
⟨*proof*⟩

lemma *global_clock_numbering*:
global_clock_numbering A v n
⟨*proof*⟩

sublocale *step_z'_bisim_step_z_dbm'*: *Bisimulation*
$\lambda(l,\ Z)\ (l',\ Z').\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle$
$\lambda(l,\ M)\ (l',\ M').\ \exists\, a.\ A \vdash' \langle l,M\rangle \rightsquigarrow_a \langle l',M'\rangle$
$\lambda(l,\ Z)\ (l',\ M).\ l' = l \wedge Z = [M]_{v,n}$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*: $u' \in V \Longrightarrow (l,\ u) \preceq (l',\ u') \Longrightarrow u \in V$
fixes *extra* :: $'l \Rightarrow$ *real DBM* $\Rightarrow$ *real DBM* **and** l_0 **and** M_0
assumes *extra_widens*: *vabstr' Z M* $\Longrightarrow Z \subseteq [$*extra l M*$]_{v,n}$
and *extra_α*: *vabstr' Z M* $\Longrightarrow [$*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* $\Longrightarrow$ *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* $\wedge$ *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*
⟨*proof*⟩

lemma *apx_abs*:
apx l $([M]_{v,n}) \subseteq \alpha\ l\ ([M]_{v,n})$ **if** *canonical_dbm M*
⟨*proof*⟩

lemma *α_V*:
$\alpha\ l\ Z \subseteq V$ **if** $Z \subseteq V$
⟨*proof*⟩

lemma *apx_V*:
apx l $([M]_{v,n}) \subseteq V$ **if** *canonical_dbm M*
⟨*proof*⟩

lemma *apx_empty*:
apx l {} = {}
⟨*proof*⟩

lemma *apx_ex*:
assumes *canonical_dbm M*
shows ∃ *M'*. *apx l* ([*M*]$_{v,n}$) = [*extra l M'*]$_{v,n}$ ∧ *canonical_dbm M'*
⟨*proof*⟩

lemma *vabstr'_apx*:
assumes *vabstr' Z M Z* ⊆ *V*
obtains *M* **where** *vabstr' (apx l Z) M*
⟨*proof*⟩

lemma *apx_finite*:
finite {*apx l Z* |*l Z*. ∃ *M*. *vabstr' Z M* ∧ *l* ∈ *state_set A*} (**is** *finite ?S*)
⟨*proof*⟩

sublocale *Time_Abstract_Simulation_Sandwich*
where *β* = *apx* **and** *I* = *λZ*. ∃ *M*. *vabstr' Z M* **and** *Z*$_0$ = [*M*$_0$]$_{v,n}$
⟨*proof*⟩

end

end
theory *Normalized_Zone_Semantics_Certification*
imports *Munta_Base.Normalized_Zone_Semantics_Impl_Semantic_Refinement*
begin

no_notation *TA_Start_Defs.step_impl'* (⟨_, _⟩ ⇝_ ⟨_, _⟩ [*61*,*61*,*61*] *61*)

context *TA_Start_Defs*
begin

definition
E_precise_op l r g l' M ≡
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 ≡
let
M1 = *abstr_repair* (*inv_of A l*) (*up_canonical_upd M n*);
M2 = *filter_diag* (*λ M*. *abstr_repair g M*) *M1*;
M3 = *filter_diag* (*λ 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 ≡
let
M' = *abstr_repair* (*inv_of A l*) (*up_canonical_upd M n*);
f1 = *λ 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'
⟨*proof*⟩

no_notation *step_impl'* (⟨_, _⟩ ⇝_ ⟨_, _⟩ [61,61,61] 61)

abbreviation *step_impl_precise* (⟨_, _⟩ ⇝_ ⟨_, _⟩ [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* ≡ (λ(*l*, *Z*) (*l''*, *Z''*). ∃ *a*. ⟨*l*, *Z*⟩ ⇝*a* ⟨*l''*, *Z''*⟩)

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*:
$E\_precise = (\lambda(l, M)\ (l', M'').\ \exists\ g\ a\ r.\ A \vdash l \longrightarrow^{g,a,r} l' \wedge M'' = E\_precise\_op\ l\ r\ g\ l'\ M)$
⟨*proof*⟩

lemma *E_E_from_op_step*:
∃ *c*. *E_from_op a c* ∧ *b* ∼ *c* **if** *E_precise a b wf_state a*
⟨*proof*⟩

lemma *E_from_op_E_step*:
∃ *c*. *E_precise a c* ∧ *b* ∼ *c* **if** *E_from_op a b wf_state a*
⟨*proof*⟩

lemma *E_from_op_wf_state*:
wf_state b **if** *wf_state a E_from_op a b*
⟨*proof*⟩

lemma *E_precise_wf_dbm*[*intro*]:
wf_dbm D' **if** *E_precise* (*l*, *D*) (*l'*, *D'*) *wf_dbm D*
⟨*proof*⟩

lemma *E_precise_wf_state*:
wf_state a ⟹ *E_precise a b* ⟹ *wf_state b*
⟨*proof*⟩

theorem *E_from_op_reachability_check*:
(∃ *l' D'*. *E_precise*** *a*0 (*l'*, *D'*) ∧ *F_rel* (*l'*, *D'*))

$\longleftrightarrow$ ($\exists$ l' D'. *E_from_op*** a_0 (l', D') $\wedge$ *F_rel* (l', D'))
⟨*proof*⟩

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*⟩

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'*

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *E_from_op_bisim*:
Bisimulation_Invariant E_precise E_from_op ($\sim$) *wf_state wf_state*
⟨*proof*⟩

lemma *E_from_op_bisim_empty*:
Bisimulation_Invariant
(λ(*l*, *M*) (*l'*, *M'*). *E_precise* (*l*, *M*) (*l'*, *M'*) $\wedge$ $\neg$ *check_diag n M'*)
(λ(*l*, *M*) (*l'*, *M'*). *E_from_op* (*l*, *M*) (*l'*, *M'*) $\wedge$ $\neg$ *check_diag n M'*)
($\sim$) *wf_state wf_state*
⟨*proof*⟩

end

definition *step_z_dbm'* ::
('*a*, '*c*, '*t*, '*s*) *ta* $\Rightarrow$ '*s* $\Rightarrow$ '*t* :: {*linordered_cancel_ab_monoid_add*,*uminus*} *DBM*
$\Rightarrow$ ('*c* $\Rightarrow$ *nat*) $\Rightarrow$ *nat* $\Rightarrow$ '*a* $\Rightarrow$ '*s* $\Rightarrow$ '*t DBM* $\Rightarrow$ *bool*
(_ $\vdash'$ ⟨_, _⟩ $\rightsquigarrow$_,_,_ ⟨_, _⟩ [61,61,61,61] 61) **where**
A $\vdash'$ ⟨*l*,*D*⟩ $\rightsquigarrow_{v,n,a}$ ⟨*l''*,*D''*⟩ $\equiv$
$\exists$ *l' D'*. *A* $\vdash$ ⟨*l*, *D*⟩ $\rightsquigarrow_{v,n,\tau}$ ⟨*l'*, *D'*⟩ $\wedge$ *A* $\vdash$ ⟨*l'*, *D'*⟩ $\rightsquigarrow_{v,n,\upharpoonright a}$ ⟨*l''*, *D''*⟩

context *TA_Start*
begin

lemma *E_precise_op'_bisim*:
E_precise_op' l r g l' M $\simeq$ *E_precise_op l r g l' M* **if** *A* $\vdash$ *l* $\longrightarrow^{g,a,r}$ *l' wf_dbm M*
⟨*proof*⟩

lemma *step_z_'_step_z_dbm'_equiv*:
Bisimulation_Invariant
(λ (*l*, *D*) (*l'*, *D'*). $\exists$ *a*. *step_z'* (*conv_A A*) *l D l' D'*)

$(\lambda\ (l,\ D)\ (l',\ D').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ D\ v\ n\ a\ l'\ D')$
$(\lambda(l,\ Z)\ (l',\ M).\ l = l' \wedge [M]_{v,n} = Z)$
$(\lambda(l,\ Z).\ True)$
$(\lambda(l,\ y).\ True)$
$\langle proof \rangle$

lemma *step_z_dbm'_step_impl_precise_equiv*:
Bisimulation_Invariant
$(\lambda\ (l,\ D)\ (l',\ D').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ D\ v\ n\ a\ l'\ D')$
$(\lambda(l,\ D)\ (l',\ D').\ \exists a.\ \langle l,\ D\rangle \rightsquigarrow_a \langle l',\ D'\rangle)$
$(\lambda(l,\ M)\ (l',\ D).\ l = l' \wedge [curry\ (conv\_M\ D)]_{v,n} = [M]_{v,n})$
$(\lambda(l,\ y).\ valid\_dbm\ y)$
wf_state
$\langle proof \rangle$

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*)
$\langle proof \rangle$

sublocale *E_precise_op'*: *E_Precise_Bisim* _ _ _ _ *E_precise_op'*
$\langle proof \rangle$

lemma *step_z_dbm'_final_bisim*:
Bisimulation_Invariant
$(\lambda\ (l,\ D)\ (l',\ D').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ D\ v\ n\ a\ l'\ D')$
E_precise_op'.E_from_op
$(\lambda\ (l,\ M)\ (l',\ D').\ l' = l \wedge [curry\ (conv\_M\ D')]_{v,n} = [M]_{v,n})$
$(\lambda(l,\ y).\ valid\_dbm\ y)$ *wf_state*
$\langle proof \rangle$

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* $\vdash' \langle l,\ D\rangle \rightsquigarrow_{v,n,a} \langle l',\ D'\rangle \wedge [M']_{v,n} \subseteq [D']_{v,n}$
$\langle proof \rangle$

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
$\langle proof \rangle$

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
⟨*proof*⟩

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*⟩

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'*
⟨*proof*⟩

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 $\exists\ M'.$ *E_from_op_empty* (l,M) $(l',M') \wedge$ *dbm_subset n D' M'*
⟨*proof*⟩

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*⟩

lemma (**in** *Graph_Start_Defs*) *run_reachable*:
pred_stream reachable xs **if** *run* (*s_0 ## xs*)
⟨*proof*⟩

lemma *pred_stream_stream_all2_combine*:
assumes *pred_stream P xs stream_all2 Q xs ys* ⋀*x y*. *P x* ⟹ *Q x y* ⟹ *R x y*
shows *stream_all2 R xs ys*
⟨*proof*⟩

lemma *stream_all2_pred_stream_combine*:
assumes *stream_all2 Q xs ys pred_stream P ys* ⋀*x y*. *Q x y* ⟹ *P y* ⟹ *R x y*
shows *stream_all2 R xs ys*
⟨*proof*⟩

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*
⟨*proof*⟩

context *TA_Start*
begin

lemma *V_I*:
assumes ∀ *i* ∈ {*1..<Suc n*}. *M 0 i* ≤ *0*
shows $[M]_{v,n}$ ⊆ *V*
⟨*proof*⟩

end

lemma *Simulation_Composition*:
fixes *A B C*
assumes
Simulation A B sim1 Simulation B C sim2 ⋀*a c*. (∃ *b*. *sim1 a b* ∧ *sim2 b c*) ⟷ *sim a c*
shows *Simulation A C sim*
⟨*proof*⟩

lemma *fold_generalize_start*:
assumes ⋀*a*. *P a* ⟹ *Q* (*fold g xs a*) *P a*
shows *Q* (*fold g xs a*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *set_of_list_alt_def*:

set_of_list = RETURN o set
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *the_pure_Id*:
the_pure (λ*a c*. ↑ (*c* = *a*)) = *Id*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *array_unfreeze_ht*:
<emp> array_unfreeze a <amtx_assn n m id_assn c> **if** *iarray_mtx_rel n m c a*
⟨*proof*⟩

lemma *array_freeze_ht*:
<amtx_assn n m id_assn c a> array_freeze a <λa. ↑(iarray_mtx_rel n m c a)>$_t$
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

schematic_goal *M_impl*:
(*?M*::*?'c*, *map_of_list*:::$_r$⟨*Id*,*Id*⟩*dflt_ahm_rel*) ∈ *?R*
⟨*proof*⟩

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*⟩

theorem *frozen_hm_of_list_lookup*:
(λ*k*. *frozen_hm_lookup k frozen_hm_of_list*, *map_of_list*) ∈ *Id* → ⟨*Id*⟩*option_rel*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

context *TA_Impl*
begin

interpretation *DBM_Impl n* ⟨*proof*⟩

sepref_definition *E_precise_op′_impl* **is**
uncurry4 (λ *l r*. *RETURN ooo E_precise_op′ l r*) :: *op_impl_assn*
⟨*proof*⟩

end

locale *TA_Impl_Precise* =
  TA_Impl _ _ _ l_0 _ _ _ _ _ $l_0 i$
  + *op_precise*: *E_Precise_Bisim* _ l_0 **for** l_0 :: *'s* **and** $l_0 i$ :: *'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*, (λ*l*. *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*** (*l*$_0$, *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*)
  ⟨*proof*⟩

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
$\leq$ *RETURN* $\{D' \mid D'\ D.\ D \in S \wedge D' = f\ l\ r\ g\ l'\ D \wedge \neg\ check\_diag\ n\ D'\}$
⟨*proof*⟩

lemma *succs_precise'_refine*:
succs_precise' l S $\leq$ *RETURN* (*succs_precise l S*)
⟨*proof*⟩

lemma *succs_precise'_correct*:
(*uncurry succs_precise'*, *uncurry* (*RETURN oo PR_CONST succs_precise*)) $\in$ *Id* $\times_r$ *Id* $\rightarrow$ ⟨*Id*⟩*nres_rel*
⟨*proof*⟩

sepref_register *PR_CONST f* ::
's $\Rightarrow$ *nat list* $\Rightarrow$ (*nat*, *int*) *acconstraint list* $\Rightarrow$ *'s* $\Rightarrow$ *int DBMEntry i_mtx* $\Rightarrow$ *int DBMEntry i_mtx*

lemma *aux3*:
ID D x'a TYPE(*nat* $\times$ *nat* $\Rightarrow$ *int DBMEntry*) = *ID D x'a TYPE*(*int DBMEntry i_mtx*)
⟨*proof*⟩

lemmas [*sepref_fr_rules*] =
set_of_list_hnr Leadsto_Impl.lso_id_hnr
op_impl

interpretation *DBM_Impl n* ⟨*proof*⟩

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*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemmas *succs_precise_impl_refine* = *succs_precise'_impl.refine*[*FCOMP succs_precise'_correct*]

lemma *succs_precise_finite*:

$\forall\, l\ S.\ \forall\, (l',\ S') \in set\ (succs\_precise\ l\ S).\ finite\ S \longrightarrow finite\ S'$
⟨*proof*⟩

definition
$wf\_dbm'\ D \equiv (canonical'\ D \vee check\_diag\ n\ D) \wedge$
$(list\_all\ (\lambda i.\ D\ (i,\ i) \leq 0)\ [0..<n+1]) \wedge list\_all\ (\lambda i.\ D\ (0,\ i) \leq 0)\ [0..<n+1]$

theorem *wf_dbm'_wf_dbm*:
fixes $D :: nat \times nat \Rightarrow int\ DBMEntry$
assumes $wf\_dbm'\ D$
shows $wf\_dbm\ D$
⟨*proof*⟩

lemma *canonical'_compute*:
$canonical'\ M =$
$list\_all\ (\lambda i.$
$list\_all\ (\lambda j.$
$list\_all\ (\lambda k.$
$M\ (i,\ k) \leq M\ (i,\ j) + M\ (j,\ k)$
$)[0..<n+1])\ [0..<n+1])\ [0..<n+1]$

⟨*proof*⟩

sepref_definition *canonical'_impl* **is**
$RETURN\ o\ PR\_CONST\ canonical' :: mtx\_assn^k \rightarrow_a bool\_assn$
⟨*proof*⟩

sepref_thm *wf_dbm'_impl* **is**
$RETURN\ o\ PR\_CONST\ wf\_dbm' :: mtx\_assn^k \rightarrow_a bool\_assn$
⟨*proof*⟩

definition
$states\_mem\ l \equiv l \in states'$

definition
$P \equiv \lambda\ (l,\ M).\ PR\_CONST\ states\_mem\ l \wedge wf\_dbm'\ M$

lemma *P_correct*:
$l \in states' \wedge wf\_dbm\ M$ **if** $P\ (l,\ M)$
⟨*proof*⟩

lemmas [*sepref_import_param*] = *states_mem_impl*[*folded states_mem_def*]

sepref_register *states_mem*

sepref_register $wf\_dbm' :: 'c\ DBMEntry\ i\_mtx \Rightarrow bool$

lemmas [*sepref_fr_rules*] =

wf_dbm'_impl.refine_raw

sepref_definition *P_impl* **is**
RETURN o PR_CONST P :: $(prod\_assn\ (pure\ loc\_rel)\ mtx\_assn)^k \to_a bool\_assn$
⟨*proof*⟩

lemma *P_impl_refine*:
$(P\_impl, (RETURN \circ\circ PR\_CONST)\ P) \in (location\_assn \times_a mtx\_assn)^k \to_a bool\_assn$
⟨*proof*⟩

lemma *E_from_op_states*:
$l' \in states'$ **if** *op_precise.E_from_op* $(l, M)\ (l', M')\ l \in states'$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *L_list_rel*:
$(L\_list, L) \in \langle location\_rel \rangle list\_rel$
⟨*proof*⟩

lemma *L_list_hnr*:
(*uncurry0* (*return L_list*), *uncurry0* (*RETURN* (*PR_CONST* (*set L*))))
$\in unit\_assn^k \to_a lso\_assn\ location\_assn$
⟨*proof*⟩

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
$\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$
$\lambda\ (l,\ Z)\ (l',\ Z').\ \exists\ a.\ conv\_A\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge Z' \neq \{\}$
$\lambda\ (l,\ u)\ (l',\ Z).\ l' = l \wedge u \in Z$
⟨*proof*⟩

interpretation *B*:
Simulation
$\lambda\ (l,\ Z)\ (l',\ Z').\ \exists\ a.\ conv\_A\ A \vdash \langle l,\ Z\rangle \rightsquigarrow \langle l',\ Z'\rangle \wedge Z' \neq \{\}$
$\lambda\ (l,\ M)\ (l',\ M').\ \exists\ a.\ conv\_A\ A \vdash' \langle l,\ M\rangle \rightsquigarrow_{v,n,a} \langle l',\ M'\rangle \wedge \lceil M' \rceil_{v,n} \neq \{\}$
$\lambda\ (l,\ Z)\ (l',\ M).\ l' = l \wedge Z = [M]_{v,n}$
⟨*proof*⟩

interpretation
Simulation
$\lambda\ (l,\ u)\ (l',\ u').\ conv\_A\ A \vdash' \langle l,\ u\rangle \to \langle l',\ u'\rangle$
$\lambda\ (l,\ M)\ (l',\ M').\ \exists\ a.\ step\_z\_dbm'\ (conv\_A\ A)\ l\ M\ v\ n\ a\ l'\ M' \wedge \lceil M' \rceil_{v,n} \neq \{\}$
$\lambda\ (l,\ u)\ (l',\ M).\ l' = l \wedge u \in [M]_{v,n}$
⟨*proof*⟩

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*⟩

lemma *op_precise_unreachable_correct*:
assumes $\nexists\ s'.$ *op_precise.E_from_op_empty*** $(l_0,\ init\_dbm)\ s' \wedge F\ s'$
shows $\nexists\ u\ l'\ u'.\ (\forall\ c \leq n.\ u\ c = 0) \wedge conv\_A\ A \vdash' \langle l_0,\ u\rangle \to* \langle l',\ u'\rangle \wedge F'\ (l',\ u')$
⟨*proof*⟩

lemma *op_precise_unreachable_correct'*:
$(uncurry0\ (SPEC\ (\lambda r.\ r \longrightarrow$
$(\nexists\ s'.$ *op_precise.E_from_op_empty*** $(l_0,\ init\_dbm)\ s' \wedge F\ s'))),$
$uncurry0\ (SPEC\ (\lambda r.\ r \longrightarrow$
$(\nexists\ u\ l'\ u'.\ (\forall\ c \leq n.\ u\ c = 0) \wedge conv\_A\ A \vdash' \langle l_0,\ u\rangle \to* \langle l',\ u'\rangle \wedge F'\ (l',\ u')))))$
$\in Id \to \langle Id\rangle nres\_rel$
⟨*proof*⟩

lemma *IArray_list_to_dbm_rel*[*param*]:
$(IArray,\ list\_to\_dbm\ n)$
$\in \{(xs,\ ys).\ xs = ys \wedge length\ xs = Suc\ n * Suc\ n\} \to \{(a,\ b).\ iarray\_mtx\_rel\ (Suc\ n)\ (Suc\ n)$
$b\ a\}$
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *P_loc1*:
list_all
(λ(*l*, *xs*). *P_loc l* ∧ *states_mem_impl l* ∧ *list_all P xs*) *M_list*
⟨*proof*⟩

lemma *M_list_rel1*:
(*M_list*, *M_list1*) ∈ ⟨*location_rel* ×*r* ⟨*br id P*⟩*list_rel*⟩*list_rel*
⟨*proof*⟩

lemma *dom_M_eq1_aux*:
dom (*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*)
) *xs m*) = *dom m* ∪ *fst ‘ set xs* **for** *xs m*
⟨*proof*⟩

lemma *dom_M_eq1*:
dom M1 = fst ‘ set M_list1
⟨*proof*⟩

lemma *L_dom_M_eqI1*:
assumes *fst ‘ set M_list = set L_list*
shows *set L = dom M1*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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)
⟨*proof*⟩

lemma *M_finite*:
∀ S∈ran M. finite S
⟨*proof*⟩

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*⟩

sepref_register *PR_CONST M_list'*

interpretation *DBM_Impl n* ⟨*proof*⟩

sepref_definition *M_table* **is**
uncurry0 (*RETURN M*) :: *unit_assn*^{k} →$_a$ *hm.hms_assn' location_assn* (*lso_assn mtx_assn*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

concrete_definition *certify_unreachable_pure*
uses *pure.certify_unreachable_impl_pure_correct*[*unfolded to_pair_def get_succs_def*] **is** *?f* ⟶ _

lemma *certify_unreachable_pure_refine*:
assumes *fst* ‘ *set M_list* = *set L_list certify_unreachable_pure*
shows ♯ *u l' u'*. (∀ *c*≤*n*. *u c* = *0*) ∧ *conv_A A* ⊢' ⟨l_0, *u*⟩ →∗ ⟨*l'*, *u'*⟩ ∧ *F'* (*l'*, *u'*)
⟨*proof*⟩

end

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*:
$\bigwedge L\ Li.$
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*;
$l_0 i$ = *Heap_Monad.return* $l_0 i$;
$s_0 i$ = *init_dbm_impl*;
succsi = *succs_precise′_impl*;
M_table = *M_table*
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*;
certify_unreachable_impl_inner Fi Pi copyi Lei succsi $l_0 i$ $s_0 i$ *splitteri L_list M_table*
}
⟨*proof*⟩

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
  ⟨proof⟩

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 (λp 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
  ⟨proof⟩

lemma M2_alt_def:
  M2 = M1 TYPE(int DBMEntry list × nat) M_list (λ(M, i). (list_to_dbm n M, i))
```

⟨*proof*⟩

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*))
⟨*proof*⟩

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* = λ*x*. *case M2 x of None* ⇒ {} | *Some S* ⇒ *S*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *init_locsi_init_locs1*:
(*init_locsi*, *init_locs1*) ∈ ⟨*location_rel*⟩*list_rel*
⟨*proof*⟩

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′*

⟨*proof*⟩

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*⟩

definition
inits = *map* (λ*l*. (*l*, *init_dbm*)) *init_locs1*

interpretation *DBM_Impl n* ⟨*proof*⟩

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*)
⟨*proof*⟩

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* = λ(*l*, *M*). *l* ∈ *states'* ∧ *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* = λ*k*. *Impl_Array_Hash_Map.ahm_lookup* (=) *bounded_hashcode_nat k M2i*
and *inits* = *inits*
and *initsi* = *initsi*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *empty_steps_states′*:
l′ ∈ *states′* **if** *op_precise.E_from_op_empty*** (*l*, *D*) (*l′*, *D′*) *l* ∈ *states′*
⟨*proof*⟩

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*)}
⟨*proof*⟩

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
$\longrightarrow$ $(\forall\, u.\ (\forall\, c{\leq}n.\ u\ c = 0) \longrightarrow \neg\ deadlock\ (l_0,\ u))$
⟨*proof*⟩

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 $\bigwedge li.\ P\_loc\ li \Longrightarrow \exists\, l.\ (li,\ l) \in loc\_rel$
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* = $(\bigcup xs \in set\ Li\_split.\ set\ xs)$
shows
deadlock.certify_unreachable_pure L_list M_list Li_split
$\longrightarrow$ $(\forall\, u.\ (\forall\, c{\leq}n.\ u\ c = 0) \longrightarrow \neg\ deadlock\ (l_0,\ u))$
⟨*proof*⟩

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 $\bigwedge li.\ P\_loc\ li \Longrightarrow \exists\, l.\ (li,\ l) \in loc\_rel$
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*: $\bigwedge xs.\ set\ xs = (\bigcup xs \in set\ (splitter\ xs).\ set\ xs)$
and *same_split*:
$\bigwedge L\ Li.$
list_assn (*list_assn location_assn*) (*splitter L*) (*splitteri Li*) = *list_assn location_assn L Li*
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* $\longrightarrow$ $(\forall\, u.\ (\forall\, c{\leq}n.\ u\ c = 0) \longrightarrow \neg\ deadlock\ (l_0,\ u))$)))
∈ $unit\_assn^k \rightarrow_a bool\_assn$
⟨*proof*⟩

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*
⟨*proof*⟩

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_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 M_list*;
certify_unreachable_impl_inner Fi Pi copyi Lei succsi l_0i s_0i *splitteri L_list M_table*
}
⟨*proof*⟩

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 M_list*
in
certify_unreachable_impl2 Fi Pi copyi Lei succsi l_0i s_0i *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'*)
⟨*proof*⟩

lemma *unreachability_checker2'_is_unreachability_checker2*:
unreachability_checker2' = *unreachability_checker2 L_list M_list splitteri*
⟨*proof*⟩

end

end

end

context *TA_Impl_Precise*
begin

lemma (**in** *TA_Impl*) *dbm_subset_correct*:
assumes *wf_dbm D* **and** *wf_dbm M*
shows $[curry\ (conv\_M\ D)]_{v,n} \subseteq [curry\ (conv\_M\ M)]_{v,n} \longleftrightarrow dbm\_subset\ n\ D\ M$
⟨*proof*⟩

lemma *empty_steps_states'*:
$l' \in states'$ **if** *op_precise.E_from_op_empty*** $(l, D)\ (l', D')\ l \in states'$
⟨*proof*⟩

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**
$F\_impl = \lambda(l, M).\ do\ \{r \leftarrow check\_deadlock\_impl\ l\ M;\ return\ (\neg\ r)\}$
⟨*proof*⟩

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 $\bigwedge li.\ P\_loc\ li \Longrightarrow \exists\, l.\ (li, l) \in loc\_rel$
and *list_all* $(\lambda x.\ P\_loc\ x \wedge states\_mem\_impl\ x)\ L\_list$
and *list_all* $(\lambda(l, xs).\ list\_all\ (\lambda M.\ length\ M = Suc\ n * Suc\ n)\ xs)\ M\_list$
and *fst ‘ set M_list* = *set L_list*
assumes *full_split*: $\bigwedge xs.\ set\ xs = (\bigcup xs \in set\ (splitter\ xs).\ set\ xs)$
and *same_split*:
$\bigwedge L\ Li.$
list_assn (list_assn location_assn) (splitter L) (splitteri Li) = *list_assn location_assn L Li*
shows
deadlock.unreachability_checker2 L_list M_list splitteri
⟶ $(\forall\, u.\ (\forall\, c{\le}n.\ u\ c = 0) \longrightarrow \neg\ deadlock\ (l_0, u))$
⟨*proof*⟩

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 $\bigwedge li.\ P\_loc\ li \Longrightarrow \exists\, l.\ (li, l) \in loc\_rel$
and *list_all* $(\lambda x.\ P\_loc\ x \wedge states\_mem\_impl\ x)\ L\_list$
and *list_all* $(\lambda(l, xs).\ list\_all\ (\lambda M.\ length\ M = Suc\ n * Suc\ n)\ xs)\ M\_list$
and *fst ‘ set M_list* = *set L_list*
assumes *full_split*: *set L_list* = $(\bigcup xs \in set\ Li\_split.\ set\ xs)$
shows
deadlock.certify_unreachable_pure L_list M_list Li_split
⟶ $(\forall\, u.\ (\forall\, c{\le}n.\ u\ c = 0) \longrightarrow \neg\ deadlock\ (l_0, u))$
⟨*proof*⟩

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*
⟨*proof*⟩

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
  ⟨proof⟩ lemma insert_restrict_ran:
  insert v (ran (m |` (− {k}))) = ran m  if m k = Some v
  ⟨proof⟩ lemma 1:
  (m |` (− {x}))(x ↦ y) = m if m x = Some y
  ⟨proof⟩ lemma 2:
  (m1 ++ m2)(x ↦ y) = m1(x ↦ y) ++ m2 if x ∉ dom m2
  ⟨proof⟩

lemma list_of_map_correct[refine]:
  list_of_map m ≤ SPEC (λ r. map_of r = m) if finite (dom m)
  ⟨proof⟩

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))
  ⟨proof⟩

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*))
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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} →$_a$ (*list_assn* (*K* ×$_a$ *lso_assn A*))
  ⟨*proof*⟩

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.a₀_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*
⟨*proof*⟩

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** → *60*)
no_notation *Omega_Words_Fun.build* (**infixr** ‹##› *65*)
no_notation *Assertions.models* (**infix** ⊨ *50*)

Misc **lemma** *list_set_rel_singleton_iff*:
([*a*], {*b*}) ∈ ⟨*R*⟩*list_set_rel* ⟷ (*a*, *b*) ∈ *R*
⟨*proof*⟩

definition (**in** *Graph_Defs*)
alw_ev φ *x* ≡ ∀ *xs*. *run* (*x* ## *xs*) ⟶ *alw* (*ev* (*holds* φ)) (*x* ## *xs*)

lemma (**in** *Graph_Defs*) *alw_ev_to_ctl_star*:
alw_ev φ *x* ⟷ *models_state* (*All* (*G* (*F* (*State* (*PropS* φ))))) *x*
⟨*proof*⟩

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* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(*map_state_formula* (λφ (*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'
⟷ *Graph_Defs.models_state*
(λ (*L*, *s*, *u*) (*L'*, *s'*, *u'*). *sem* ⊢ ⟨*L*, *s*, *u*⟩ → ⟨*L'*, *s'*, *u'*⟩)
(*map_state_formula* (λφ (*L*, *s*, _). *check_sexp* φ *L* (*the o s*)) (*All* (*G* (*F* (*State* (*PropS* φ))))))
a_0 (**is** *Graph_Defs.models_state* _ ?φ _ ⟷ _)
⟨*proof*⟩

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* (λ*p*. *renum_states p*) *renum_vars id* φ) *L* (*the o s*)) a_0'
⟷ *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*)) a_0
⟨*proof*⟩

lemmas *buechi_compatible'* =
buechi_compatible[*unfolded rename_N_eq_sem*, *folded N_eq_sem*, *unfolded* a_0*_def* a_0*'_def* Φ*'_def*]

end

lemma *stream_all2_flip*:
assumes *stream_all2 R xs ys*
shows *stream_all2* (λ*y x*. *R x y*) *ys xs*
⟨*proof*⟩

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*
⟨*proof*⟩

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
  ⟨proof⟩

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)
  ⟨proof⟩

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
  ⟨proof⟩

lemma split_size_non_empty[simp]:
  split_size f k n acc xs = [] ⟷ False
  ⟨proof⟩

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'
  ⟨proof⟩

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)
  ⟨proof⟩

lemma length_split_size_1:
  sum_list (map length (split_size (λ_. 1 :: nat) k n acc xs)) = length xs + length acc
  ⟨proof⟩

lemma length_sum_list:
  sum_list (map length (split_eq_width k xs)) = length xs
  ⟨proof⟩

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*
⟨*proof*⟩

lemma *length_hd_split_eq_width*:
length (hd (split_eq_width width xs)) = (if length xs < width then length xs else width)
⟨*proof*⟩

lemma *list_all2_split_k*:
list_all2 R xs ys ⟷ list_all2 (list_all2 R) (split_k k xs) (split_k k ys)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *split_k_full_split*:
set xs = (⋃ x ∈ set (split_k n xs). set x)
⟨*proof*⟩

lemma *split_k′_full_split*:
fst ‘ set xs = (⋃ x ∈ set (split_k′ n xs). set x)
⟨*proof*⟩

lemma (**in** *Graph_Defs*) *Ex_ev_reaches*:
∃ y. x →∗ y ∧ φ y **if** *Ex_ev φ x*
⟨*proof*⟩
including *graph_automation* ⟨*proof*⟩

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
(λxc σ. mtx_get (Suc n) M (0, xc) ⤜ (λx′e. Heap_Monad.return (x′e ≤ Le 0))) True

definition
check_diag_nonpos n M ≡ imp_for 0 (n + 1) Heap_Monad.return
(λxc σ. mtx_get (Suc n) M (xc, xc) ⤜ (λx′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*
  ⟨*proof*⟩

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 →$_a$ *list_assn id_assn*
  ⟨*proof*⟩

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) (λxc 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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*

⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *E_op_impl_code*[*code*]:
E_op_impl ≡ *?i*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *and_entry_impl_code* [*code*]:
and_entry_impl ≡ *?impl*
⟨*proof*⟩

schematic_goal *and_entry_repair_impl_code* [*code*]:
and_entry_repair_impl ≡ *?impl*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *upd_entries_impl_code* [*code*]:
upd_entries_impl ≡ *?impl*
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *dbm_minus_canonical_impl_code* [*code*]:
dbm_minus_canonical_impl ≡ *?i*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

Extracting executable implementations **lemma** *hfkeep_hfdropI*:
assumes (*fi*, *f*) ∈ $A^k \rightarrow_a B$
shows (*fi*, *f*) ∈ $A^d \rightarrow_a B$
⟨*proof*⟩

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 $F = \lambda(l, \_).\ F\ l$
and $F1 = F\ o\ fst$
and $F' = F\ o\ fst$
and $F\_impl = impl.F\_impl$
⟨*proof*⟩

lemma *state_impl_abstract'*:
assumes *states'_memi li*
shows $\exists\, l.\ (li, l) \in loc\_rel$
⟨*proof*⟩

interpretation *Bisimulation_Invariant*
$(\lambda(l, u)\ (l', u').\ conv\_A\ prod\_ta \vdash' \langle l, u\rangle \to \langle l', u'\rangle)$
$(\lambda(L, s, u)\ (L', s', u').\ Simple\_Network\_Language.conv\ A \vdash \langle L, s, u\rangle \to \langle L', s', u'\rangle)$
$(\lambda((L, s), u)\ (L', s', u').\ L = L' \wedge u = u' \wedge s = s')$
$(\lambda((L, s), u).\ conv.all\_prop\ L\ s)$
$(\lambda(L, s, u).\ conv.all\_prop\ L\ s)$
⟨*proof*⟩

lemma *unreachability_prod*:
assumes
formula = formula.EX φ
$(\nexists\, u\ l'\ u'.\ (\forall\, c{\le}m.\ u\ c = 0) \wedge conv\_A\ prod\_ta \vdash' \langle l_0, u\rangle \to* \langle l', u'\rangle \wedge PR\_CONST\ F\ l')$
shows $\neg$ *Simple_Network_Language.conv A*,$(L_0, map\_of\ s_0, \lambda\_.\ 0) \models formula$
⟨*proof*⟩

lemma *no_buechi_run_prod*:
assumes
formula = formula.EX φ
$\neg$ *has_deadlock* (*Simple_Network_Language.conv A*) $(L_0, map\_of\ s_0, \lambda\_.\ 0)$
$\nexists\, u\ xs.\ (\forall\, c{\le}m.\ u\ c = 0) \wedge reach.run\ ((l_0, u)\ \#\#\ xs) \wedge alw\ (ev\ (holds\ (F \circ fst)))\ ((l_0, u)\ \#\#\ xs)$
shows $\neg$ *Graph_Defs.alw_ev*
$(\lambda\ (L, s, u)\ (L', s', u').\ Simple\_Network\_Language.conv\ A \vdash \langle L, s, u\rangle \to \langle L', s', u'\rangle)$
$(\lambda\ (L, s, \_).\ check\_sexp\ \varphi\ L\ (the\ o\ s))\ (L_0, map\_of\ s_0, \lambda\_.\ 0)$
⟨*proof*⟩

lemma *deadlock_prod*:
$\neg$ *reach.deadlock* $(l_0, \lambda\_.\ 0)$
$\longleftrightarrow \neg$ *has_deadlock* (*Simple_Network_Language.conv A*) $(L_0, map\_of\ s_0, \lambda\_.\ 0)$
⟨*proof*⟩

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*⟩

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*⟩

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*
⟨*proof*⟩

theorem *unreachability_checker3_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.certify_unreachable_pure L_list M_list (*split_k' num_split M_list*) ⟶
¬ *Simple_Network_Language.conv A*,(*L*$_0$, *map_of s*$_0$, λ_. *0*) ⊨ *formula*
⟨*proof*⟩

lemma *init_conds*: {*l*$_0$} ⊆ *states'* ([*l*$_0$*i*], {*l*$_0$}) ∈ ⟨*loc_rel*⟩*list_set_rel*
⟨*proof*⟩

theorem *no_buechi_run_checker_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
and ¬ *has_deadlock* (*Simple_Network_Language.conv A*) (*L*$_0$, *map_of s*$_0$, λ_. *0*)
shows
impl.certify_no_buechi_run_pure L_list M_list (*split_k' num_split M_list*) [*l*$_0$*i*] ⟶
¬ *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⟩

lemma abstr_repair_impl_refine:
  impl.abstr_repair_impl = abstr_repair_impl m
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

schematic_goal trans_impl_alt_def:
  trans_impl ≡ ?impl
  ⟨proof⟩

schematic_goal succs1_alt_def:
  succs1 ≡ ?impl
  ⟨proof⟩

schematic_goal succs_impl_alt_def:
  impl.succs_precise'_impl ≡ ?impl
  ⟨proof⟩

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*
⟨*proof*⟩

context
fixes *L_list*
assumes *L_list*: *list_all states'_memi L_list*
begin

private lemma *A*:
list_all (λ*x*. *states'_memi x* ∧ *states'_memi x*) *L_list*
⟨*proof*⟩

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* (λ*M*. *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
}
⟨*proof*⟩

schematic_goal *unreachability_checker_alt_def*:
impl.unreachability_checker L_list M_list (*split_k num_split*) ≡ *?x*
⟨*proof*⟩

definition *no_deadlock_certifier* **where**
no_deadlock_certifier ≡
Reachability_Problem_Impl_Precise.unreachability_checker
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*)))

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* ⤜ (λ*r*. *Heap_Monad.return* (¬ *r*)));
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 l0i s0i (split_k num_split) L_list M_table;
    Heap_Monad.return r
  }
  ⟨proof⟩

schematic_goal no_deadlock_certifier_alt_def:
  no_deadlock_certifier L_list M_list (split_k num_split) ≡ ?x
  ⟨proof⟩

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
  ⟨proof⟩

definition no_deadlock_certifier2 where
  no_deadlock_certifier2 ≡
  Reachability_Problem_Impl_Precise.unreachability_checker2
    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)))

schematic_goal no_deadlock_certifier2_alt_def:
  no_deadlock_certifier2 L_list M_list (split_k num_split) ≡ ?x
  ⟨proof⟩

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
  ⟨proof⟩


schematic_goal check_deadlock_impl_alt_def2:
  impl.check_deadlock_impl ≡ ?impl
  ⟨proof⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*) [$l_0 i$] ≡ *?x*
if *fst* ‘ *set M_list* = *set L_list*
⟨*proof*⟩

end

end

theorem *no_deadlock_certifier_hnr*:
assumes *list_all states'_memi 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*
shows (
uncurry0 (*no_deadlock_certifier L_list M_list* (*split_k num_split*)),
uncurry0 (*SPEC* (λ*r*. *r* ⟶
¬ *has_deadlock* (*Simple_Network_Language.conv A*) (L_0, *map_of* s_0, λ_. *0*))))
∈ *unit_assn*^{k} →$_a$ *bool_assn*
⟨*proof*⟩

theorem *no_deadlock_certifier2_refine*:
assumes *list_all states'_memi 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*

shows *no_deadlock_certifier2 L_list M_list (split_k num_split)* $\longrightarrow$
¬ has_deadlock (Simple_Network_Language.conv A) (L_0, map_of s_0, λ_. 0)
⟨*proof*⟩

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)* $\longrightarrow$
¬ has_deadlock (Simple_Network_Language.conv A) (L_0, map_of s_0, λ_. 0)
⟨*proof*⟩

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_table;//////

```
    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*
⟨*proof*⟩

schematic_goal *check_deadlock_fail_alt_def*:
check_deadlock_fail ≡ *?t*
⟨*proof*⟩

schematic_goal *check_invariant_fail_alt_def*:

check_invariant_fail ≡ *?t*
⟨*proof*⟩

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* ∧ (∀ *i*<*n_ps*. *L* ! *i* ∈ *states_i* ! *i*) ∧
length s = *n_vs* ∧ *Simple_Network_Impl_nat_defs.check_boundedi bounds' s*
)

⟨*proof*⟩

definition
certificate_checker_pre
L_list M_list broadcast bounds' automata m num_states num_actions k L_0 s_0 *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 (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
    ////////////////////////////////////////////////////////////
  ≡
  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
  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 L_0 s_0 L_list M_list num_split
    else
      unreachability_checker2
        broadcast bounds' automata m num_states num_actions L_0 s_0 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 L_0 s_0 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 L_0 s_0 formula
  in if check then
    if dc then
      no_deadlock_certifier3
        broadcast bounds' automata m num_states num_actions L_0 s_0 L_list M_list num_split
    else
      unreachability_checker3
        broadcast bounds' automata m num_states num_actions L_0 s_0 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 L_0 s_0 formula
  ≡
  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'';
    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 L_0 s_0 formula
~~inv_renum_states inv_renum_vars inv_renum_clocks~~
<λ Some r ⇒ ↑(r ⟶ ¬ N broadcast automata bounds,(L_0, map_of s_0, λ_ . 0) ⊨ formula)
| None ⇒ true>$_t$
⟨proof⟩

theorem *certificate_deadlock_check*:
<emp> certificate_checker num_split True
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 formula
~~inv_renum_states inv_renum_vars inv_renum_clocks~~
<λ Some r ⇒ ↑(r ⟶ ¬ has_deadlock (N broadcast automata bounds) (L_0, map_of s_0, λ_ . 0))
| None ⇒ true>$_t$
⟨proof⟩

theorem *certificate_check2*:
certificate_checker2 num_split False
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 formula
⟶ ¬ N broadcast automata bounds,(L_0, map_of s_0, λ_ . 0) ⊨ formula
⟨proof⟩

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
⟶ ¬ has_deadlock (N broadcast automata bounds) (L_0, map_of s_0, λ_ . 0)
⟨proof⟩

theorem *certificate_check3*:
certificate_checker3 num_split False
M_list broadcast bounds automata m num_states num_actions k L_0 s_0 formula
⟶ ¬ N broadcast automata bounds,(L_0, map_of s_0, λ_ . 0) ⊨ formula
⟨proof⟩

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
⟶ ¬ has_deadlock (N broadcast automata bounds) (L_0, map_of s_0, λ_ . 0)
⟨proof⟩

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
⟶ ¬ has_deadlock (N broadcast automata bounds) (L_0, map_of s_0, λ_. 0)
⟶ (∃φ. formula = formula.EX φ) ∧ ¬ 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))
(L_0, map_of s_0, λ_. 0)
⟨proof⟩

```
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''
s_0 L_0 automata formula)
| *Unsat* ⇒ *true*
| *Preconds_Unsat* ⇒ *true*
$>_t$ (**is** *?A*)
and *certificate_check_rename2*:
case rename_check2 num_split False broadcast bounds automata k L_0 s_0 formula
m num_states num_actions renum_acts renum_vars renum_clocks renum_states
state_space
of
Sat ⇒ ¬ *N broadcast automata bounds*,(*L_0*, *map_of s_0*, *λ_ . 0*) ⊨ *formula*
| *Renaming_Failed* ⇒ ¬ *Simple_Network_Rename_Formula*
broadcast bounds
renum_acts renum_vars renum_clocks renum_states STR ''_urge''
s_0 L_0 automata formula
| *Unsat* ⇒ *True*
| *Preconds_Unsat* ⇒ *True* (**is** *?B*)
and *certificate_check_rename3*:
case rename_check3 num_split False broadcast bounds automata k L_0 s_0 formula
m num_states num_actions renum_acts renum_vars renum_clocks renum_states
state_space
of
Sat ⇒ ¬ *N broadcast automata bounds*,(*L_0*, *map_of s_0*, *λ_ . 0*) ⊨ *formula*
| *Renaming_Failed* ⇒ ¬ *Simple_Network_Rename_Formula*
broadcast bounds
renum_acts renum_vars renum_clocks renum_states STR ''_urge''
s_0 L_0 automata formula
| *Unsat* ⇒ *True*
| *Preconds_Unsat* ⇒ *True* (**is** *?C*)
and *buechi_check_rename*:
case rename_check_buechi num_split broadcast bounds automata k L_0 s_0 formula
m num_states num_actions renum_acts renum_vars renum_clocks renum_states
certificate
of
Sat ⇒
¬ *has_deadlock* (*N broadcast automata bounds*) (*L_0*, *map_of s_0*, *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*))
(*L_0*, *map_of s_0*, *λ_. 0*)
| *Renaming_Failed* ⇒ ¬ *Simple_Network_Rename_Formula*
broadcast bounds
renum_acts renum_vars renum_clocks renum_states STR ''_urge''
s_0 L_0 automata formula
| *Unsat* ⇒ *True*
| *Preconds_Unsat* ⇒ *True* (**is** *?D*)
⟨*proof*⟩

theorem *certificate_deadlock_check_rename*:
<emp> rename_check num_split True broadcast bounds automata k L_0 s_0 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⟩

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*
  ⟨*proof*⟩

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
    }

  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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′ = (λ*i*. *ids_to_names i o inv_renum_states i*);
let f = (λ*show_clock*
show_state broadcast bounds′ *automata m num_states num_actions k* L_0 s_0 *formula*.
state_space broadcast bounds′ *automata m num_states num_actions k* L_0 s_0 *formula*
show_clock show_state ()
);
let r = *do_rename_mc f dc broadcast bounds automata k STR* ''*_urge*'' 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*;
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*)) (λ*M*′ (*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)}
  ⟨proof⟩

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*⟩

```
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, L_0, s_0) ⇒ do {
  let r = rename_state_space dc ids_to_names (broadcast, automata, bounds) L_0 s_0 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 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
        state_space
    | Impl1 ⇒ rename_check 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
        state_space
    | Impl2 ⇒ rename_check2 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
        state_space |> return
    | Impl3 ⇒ rename_check3 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
        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*⟩

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*⟩

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*⟩

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
    (λ*x*.
      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, L₀, s₀)
    ← 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, L₀, s₀)
      ← 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, 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)
   } 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, 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) = 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 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
          (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
  ⟨proof⟩

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
  ⟨proof⟩
```

definition
array_unfreeze′ = array_unfreeze

lemma [*code*]: *array_unfreeze a = array_unfreeze′ a*
⟨*proof*⟩

definition
array_copy′ = array_copy

lemma [*code*]: *array_copy a = array_copy′ a*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *imp_for'_int_cong*:
imp_for' l u f a = *imp_for' l u g a*
if ⋀ *i x*. *l* ≤ *i* ⟹ *i* < *u* ⟹ *f i x* = *g i x*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *nat_of_integer_mult*:
nat_of_integer i ∗ *nat_of_integer j* = *nat_of_integer* (*i* ∗ *j*) **if** *i* ≥ *0 j* ≥ *0*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *integer_of_nat_nat_of_integer*:
integer_of_nat (*nat_of_integer n*) = *n* **if** *n* ≥ *0*
⟨*proof*⟩

lemma *integer_of_nat_aux*:
integer_of_nat (*nat_of_integer n* + *1*) = *n* + *1* **if** *n* ≥ *0*
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *plus_int_int_of_integer_aux*:
(plus_int (int_of_integer l) 1) = int_of_integer (l + 1)
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

schematic_goal *fwi_impl_int_unfold1*:
fwi_impl_int (nat_of_integer n) a (nat_of_integer k) = ?i **if** *0 ≤ k 0 ≤ n*
⟨*proof*⟩

lemma *integer_of_nat_add*:
integer_of_nat (x + y) = integer_of_nat x + integer_of_nat y
⟨*proof*⟩

schematic_goal *fwi_impl_int_code* [*code*]:
fwi_impl_int n a k ≡ ?x
⟨*proof*⟩

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*⟩

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*⟩

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.