A Fully Verified Executable LTL Model Checker

Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow,
Alexander Schimpf, Jan-Georg Smaus

March 17, 2025

Abstract

We present an LTL model checker whose code has been completely verified using the Isabelle theorem prover. The checker consists of over 4000 lines of ML code. The code is produced using the Isabelle Refinement Framework, which allows us to split its correctness proof into (1) the proof of an abstract version of the checker, consisting of a few hundred lines of "formalized pseudocode", and (2) a verified refinement step in which mathematical sets and other abstract structures are replaced by implementations of efficient structures like red-black trees and functional arrays. This leads to a checker that, while still slower than unverified checkers, can already be used as a trusted reference implementation against which advanced implementations can be tested.

An early version of this model checker is described elsewhere [1].

Contents

1 Nested DFS using Standard Invariants Approach

theory *NDFS-SI*
imports
  CAVA-Automata.Automata-Impl
  CAVA-Automata.Lasso
  NDFS-SI-Statistics
  CAVA-Base.CAVA-Code-Target
begin

Implementation of a nested DFS algorithm for accepting cycle detection using the refinement framework. The algorithm uses the improvement of Holzmann et al. [2], i.e., it reports a cycle if the inner DFS finds a path back to the stack of the outer DFS. Moreover, an early cycle detection optimization is implemented [4], i.e., the outer DFS may already report a cycle on a back-edge involving an accepting node.

The algorithm returns a witness in case that an accepting cycle is detected.

The design approach to this algorithm is to first establish invariants of a generic DFS-Algorithm, which are then used to instantiate the concrete nested DFS-algorithm for Büchi emptiness check. This formalization can be seen as a predecessor of the formalization of Gabow's algorithm [3], where this technique has been further developed.

1.1 Tools for DFS Algorithms

1.1.1 Invariants

definition *gen-dfs-pre E U S V u0* ≡
  $E``U \subseteq U$ — Upper bound is closed under transitions
  ∧ *finite U* — Upper bound is finite
  ∧ $V \subseteq U$ — Visited set below upper bound
  ∧ $u0 \in U$ — Start node in upper bound
  ∧ $E``(V-S) \subseteq V$ — Visited nodes closed under reachability, or on stack
  ∧ $u0 \notin V$ — Start node not yet visited
  ∧ $S \subseteq V$ — Stack is visited
  ∧ $(\forall v \in S.\ (v,u0) \in (E \cap S \times UNIV)^*)$ — *u0* reachable from stack, only over stack

definition *gen-dfs-var U* ≡ *finite-psupset U*

definition *gen-dfs-fe-inv E U S V0 u0 it V brk* ≡
  $(\neg brk \longrightarrow E``(V-S) \subseteq V)$ — Visited set closed under reachability
  ∧ $E``\{u0\} - it \subseteq V$ — Successors of *u0* visited
  ∧ $V0 \subseteq V$ — Visited set increasing
  ∧ $V \subseteq V0 \cup (E - UNIV \times S)^* `` (E``\{u0\} - it - S)$ — All visited nodes reachable

definition *gen-dfs-post E U S V0 u0 V brk* ≡
(¬*brk* ⟶ *E*‘‘(*V*−*S*) ⊆ *V*) — Visited set closed under reachability
∧ *u0* ∈ *V* — *u0* visited
∧ *V0* ⊆ *V* — Visited set increasing
∧ *V* ⊆ *V0* ∪ (*E* − *UNIV*×*S*)* ‘‘ {*u0*} — All visited nodes reachable

definition *gen-dfs-outer E U V0 it V brk* ≡
V0 ⊆ *U* — Start nodes below upper bound
∧ *E*‘‘*U* ⊆ *U* — Upper bound is closed under transitions
∧ *finite U* — Upper bound is finite
∧ *V* ⊆ *U* — Visited set below upper bound
∧ (¬*brk* ⟶ *E*‘‘*V* ⊆ *V*) — Visited set closed under reachability
∧ (*V0* − *it* ⊆ *V*) — Start nodes already iterated over are visited

1.1.2 Invariant Preservation

lemma *gen-dfs-outer-initial*:
assumes *finite* (*E** ‘‘*V0*)
shows *gen-dfs-outer E* (*E** ‘‘*V0*) *V0 V0* {} *brk*
using *assms* **unfolding** *gen-dfs-outer-def*
by (*auto intro*: *rev-ImageI*)

lemma *gen-dfs-pre-initial*:
assumes *gen-dfs-outer E U V0 it V False*
assumes *v0*∈*U* − *V*
shows *gen-dfs-pre E U* {} *V v0*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-outer-def*
apply *auto*
done

lemma *fin-U-imp-wf*:
assumes *finite U*
shows *wf* (*gen-dfs-var U*)
using *assms* **unfolding** *gen-dfs-var-def* **by** *auto*

lemma *gen-dfs-pre-imp-wf*:
assumes *gen-dfs-pre E U S V u0*
shows *wf* (*gen-dfs-var U*)
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-var-def* **by** *auto*

lemma *gen-dfs-pre-imp-fin*:
assumes *gen-dfs-pre E U S V u0*
shows *finite* (*E* ‘‘ {*u0*})
apply (*rule finite-subset*[**where** *B*=*U*])
using *assms* **unfolding** *gen-dfs-pre-def*
by *auto*

Inserted *u0* on stack and to visited set

lemma *gen-dfs-pre-imp-fe*:
assumes *gen-dfs-pre E U S V u0*
shows *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0*
($E``\{u0\}$) (*insert u0 V*) *False*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-fe-inv-def*
apply *auto*
done

lemma *gen-dfs-fe-inv-pres-visited*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \in V'$
shows *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0* ($it - \{t\}$) *V′ False*
using *assms* **unfolding** *gen-dfs-fe-inv-def*
apply *auto*
done

lemma *gen-dfs-upper-aux*:
assumes $(x,y) \in E'^*$
assumes $(u0,x) \in E$
assumes $u0 \in U$
assumes $E' \subseteq E$
assumes $E``U \subseteq U$
shows $y \in U$
using *assms*
by *induct auto*

lemma *gen-dfs-fe-inv-imp-var*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \notin V'$
shows $(V',V) \in$ *gen-dfs-var U*
using *assms* **unfolding** *gen-dfs-fe-inv-def gen-dfs-pre-def gen-dfs-var-def*
apply (*clarsimp simp add*: *finite-psupset-def*)
apply (*blast dest*: *gen-dfs-upper-aux*)
done

lemma *gen-dfs-fe-inv-imp-pre*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0 it V′ False*
assumes $t \in it$ $it \subseteq E``\{u0\}$ $t \notin V'$
shows *gen-dfs-pre E U* (*insert u0 S*) *V′ t*
using *assms* **unfolding** *gen-dfs-fe-inv-def gen-dfs-pre-def*
apply *clarsimp*
apply (*intro conjI*)
apply (*blast dest*: *gen-dfs-upper-aux*)
apply *blast*

```
  apply blast
  apply blast
  apply clarsimp
  apply (rule rtrancl-into-rtrancl[where b=u0])
  apply (auto intro: rev-subsetD[OF - rtrancl-mono[where r=E ∩ S × UNIV]])
[]
  apply blast
  done

lemma gen-dfs-post-imp-fe-inv:
  assumes gen-dfs-pre E U S V u0
  assumes gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 it V′ False
  assumes t∈it   it⊆E``{u0}   t∉V′
  assumes gen-dfs-post E U (insert u0 S) V′ t V″ cyc
  shows gen-dfs-fe-inv E U (insert u0 S) (insert u0 V) u0 (it−{t}) V″ cyc
  using assms unfolding gen-dfs-fe-inv-def gen-dfs-post-def gen-dfs-pre-def
  apply clarsimp
  apply (intro conjI)
  apply blast
  apply blast
  apply blast
  apply (erule order-trans)
  apply simp
  apply (rule conjI)
    apply (erule order-trans[
      where y=insert u0 (V ∪ (E − UNIV × insert u0 S)*
        `` (E `` {u0} − it − insert u0 S))])
    apply blast

    apply (cases cyc)
      apply simp
      apply blast

      apply simp
      apply blast
  done

lemma gen-dfs-post-aux:
  assumes 1: (u0,x)∈E
  assumes 2: (x,y)∈(E − UNIV × insert u0 S)*
  assumes 3: S⊆V   x∉V
  shows (u0, y) ∈ (E − UNIV × S)*
proof −
  from 1 3 have (u0,x)∈(E − UNIV × S) by blast
  also have (x,y)∈(E − UNIV × S)*
    apply (rule-tac rev-subsetD[OF 2 rtrancl-mono])
    by auto
  finally show ?thesis .
qed
```

lemma *gen-dfs-fe-imp-post-brk*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0 it V′ True*
assumes $it \subseteq E``\{u0\}$
shows *gen-dfs-post E U S V u0 V′ True*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-fe-inv-def gen-dfs-post-def*
apply *clarify*
apply (*intro conjI*)
apply *simp*
apply *simp*
apply *simp*
apply *clarsimp*
apply (*blast intro*: *gen-dfs-post-aux*)
done

lemma *gen-dfs-fe-inv-imp-post*:
assumes *gen-dfs-pre E U S V u0*
assumes *gen-dfs-fe-inv E U* (*insert u0 S*) (*insert u0 V*) *u0* {} *V′ cyc*
assumes $cyc \longrightarrow cyc'$
shows *gen-dfs-post E U S V u0 V′ cyc′*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-fe-inv-def gen-dfs-post-def*
apply *clarsimp*
apply (*intro conjI*)
apply *blast*
apply (*auto intro*: *gen-dfs-post-aux*) []
done

lemma *gen-dfs-pre-imp-post-brk*:
assumes *gen-dfs-pre E U S V u0*
shows *gen-dfs-post E U S V u0* (*insert u0 V*) *True*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-post-def*
apply *auto*
done

1.1.3 Consequences of Postcondition

lemma *gen-dfs-post-imp-reachable*:
assumes *gen-dfs-pre E U S V0 u0*
assumes *gen-dfs-post E U S V0 u0 V brk*
shows $V \subseteq V0 \cup E^{*}``\{u0\}$
using *assms* **unfolding** *gen-dfs-post-def gen-dfs-pre-def*
apply *clarsimp*
apply (*blast intro*: *rev-subsetD*[*OF - rtrancl-mono*])
done

lemma *gen-dfs-post-imp-complete*:
assumes *gen-dfs-pre E U* {} *V0 u0*

assumes *gen-dfs-post E U* {} *V0 u0 V False*
shows *V0* ∪ E^* ``{*u0*} ⊆ *V*
using *assms* **unfolding** *gen-dfs-post-def gen-dfs-pre-def*
apply *clarsimp*
apply (*blast dest*: *Image-closed-trancl*)
done

lemma *gen-dfs-post-imp-eq*:
assumes *gen-dfs-pre E U* {} *V0 u0*
assumes *gen-dfs-post E U* {} *V0 u0 V False*
shows *V* = *V0* ∪ E^* ``{*u0*}
using *gen-dfs-post-imp-reachable*[*OF assms*] *gen-dfs-post-imp-complete*[*OF assms*]
by *blast*

lemma *gen-dfs-post-imp-below-U*:
assumes *gen-dfs-pre E U S V0 u0*
assumes *gen-dfs-post E U S V0 u0 V False*
shows *V* ⊆ *U*
using *assms* **unfolding** *gen-dfs-pre-def gen-dfs-post-def*
apply *clarsimp*
apply (*blast intro*: *rev-subsetD*[*OF - rtrancl-mono*] *dest*: *Image-closed-trancl*)
done

lemma *gen-dfs-post-imp-outer*:
assumes *gen-dfs-outer E U V0 it Vis0 False*
assumes *gen-dfs-post E U* {} *Vis0 v0 Vis False*
assumes *v0* ∈ *it* *it* ⊆ *V0* *v0* ∉ *Vis0*
shows *gen-dfs-outer E U V0* (*it* − {*v0*}) *Vis False*
proof −
{
assume *v0* ∈ *it* *it* ⊆ *V0* *V0* ⊆ *U* *E* `` *U* ⊆ *U*
hence E^* `` {*v0*} ⊆ *U*
by (*metis* (*full-types*) *empty-subsetI insert-subset rtrancl-reachable-induct subset-trans*)
} **note** *AUX*=*this*

show *?thesis*
using *assms*
unfolding *gen-dfs-outer-def gen-dfs-post-def*
using *AUX*
by *auto*
qed

lemma *gen-dfs-outer-already-vis*:
assumes *v0* ∈ *it* *it* ⊆ *V0* *v0* ∈ *V*
assumes *gen-dfs-outer E U V0 it V False*
shows *gen-dfs-outer E U V0* (*it* − {*v0*}) *V False*
using *assms*
unfolding *gen-dfs-outer-def*

by *auto*

1.2 Abstract Algorithm

1.2.1 Inner (red) DFS

A witness of the red algorithm is a node on the stack and a path to this node

type-synonym *'v red-witness = ('v list × 'v) option*

Prepend node to red witness

fun *prep-wit-red* :: *'v ⇒ 'v red-witness ⇒ 'v red-witness* **where**
 prep-wit-red v None = None
| *prep-wit-red v (Some (p,u)) = Some (v#p,u)*

Initial witness for node u with onstack successor v

definition *red-init-witness* :: *'v ⇒ 'v ⇒ 'v red-witness* **where**
 red-init-witness u v = Some ([u],v)

```
definition red-dfs where
  red-dfs E onstack V u ≡
    REC_T (λD (V,u). do {
      let V=insert u V;
      NDFS-SI-Statistics.vis-red-nres;

      — Check whether we have a successor on stack
      brk ← FOREACH_C (E``{u}) (λbrk. brk=None)
        (λt -.
          if t∈onstack then
            RETURN (red-init-witness u t)
          else
            RETURN None
        )
        None;

      — Recurse for successors
      case brk of
        None ⇒
          FOREACH_C (E``{u}) (λ(V,brk). brk=None)
            (λt (V,-).
              if t∉V then do {
                (V,brk) ← D (V,t);
                RETURN (V,prep-wit-red u brk)
              } else RETURN (V,None))
            (V,None)
      | - ⇒ RETURN (V,brk)
    }) (V,u)
```

datatype *'v blue-witness* =
  NO-CYC — No cycle detected
| *REACH 'v 'v list* — Path from current start node to node on stack, path contains accepting node.

| *CIRC 'v list 'v list* — *CIRI pr pl*: Lasso found from current start node.

Prepend node to witness

```
primrec prep-wit-blue :: 'v ⇒ 'v blue-witness ⇒ 'v blue-witness where
  prep-wit-blue u0 NO-CYC = NO-CYC
| prep-wit-blue u0 (REACH u p) = (
  if u0=u then
    CIRC [] (u0#p)
  else
    REACH u (u0#p)
  )
| prep-wit-blue u0 (CIRC pr pl) = CIRC (u0#pr) pl
```

Initialize blue witness

```
fun init-wit-blue :: 'v ⇒ 'v red-witness ⇒ 'v blue-witness where
  init-wit-blue u0 None = NO-CYC
| init-wit-blue u0 (Some (p,u)) = (
  if u=u0 then
    CIRC [] p
  else REACH u p)
```

```
definition init-wit-blue-early :: 'v ⇒ 'v ⇒ 'v blue-witness
  where init-wit-blue-early s t ≡ if s=t then CIRC [] [s] else REACH t [s]
```

Extract result from witness

```
term lasso-ext
```

```
definition extract-res cyc
  ≡ (case cyc of
      CIRC pr pl ⇒ Some (pr,pl)
    | - ⇒ None)
```

1.2.2 Outer (Blue) DFS

```
definition blue-dfs
  :: ('a,-) b-graph-rec-scheme ⇒ ('a list × 'a list) option nres
  where
  blue-dfs G ≡ do {
    NDFS-SI-Statistics.start-nres;
    (-,-,cyc) ← FOREACHc (g-V0 G) (λ(-,-,cyc). cyc=NO-CYC)
      (λv0 (blues,reds,-). do {
        if v0 ∉ blues then do {
```

```
      (blues,reds,-,cyc) ← REC_T (λD (blues,reds,onstack,s). do {
        let blues=insert s blues;
        let onstack=insert s onstack;
        let s-acc = s ∈ bg-F G;
        NDFS-SI-Statistics.vis-blue-nres;
        (blues,reds,onstack,cyc) ←
        FOREACH_C ((g-E G)``{s}) (λ(-,-,-,cyc). cyc=NO-CYC)
          (λt (blues,reds,onstack,cyc).
            if t ∈ onstack ∧ (s-acc ∨ t ∈ bg-F G) then (
              RETURN (blues,reds,onstack, init-wit-blue-early s t)
            ) else if t∉blues then do {
              (blues,reds,onstack,cyc) ← D (blues,reds,onstack,t);
              RETURN (blues,reds,onstack,(prep-wit-blue s cyc))
            } else do {
              NDFS-SI-Statistics.match-blue-nres;
              RETURN (blues,reds,onstack,cyc)
            })
          (blues,reds,onstack,NO-CYC);

        (reds,cyc) ←
        if cyc=NO-CYC ∧ s-acc then do {
          (reds,rcyc) ← red-dfs (g-E G) onstack reds s;
          RETURN (reds, init-wit-blue s rcyc)
        } else RETURN (reds,cyc);

        let onstack=onstack − {s};
        RETURN (blues,reds,onstack,cyc)
      }) (blues,reds,{},v0);
      RETURN (blues, reds, cyc)
    } else do {
      RETURN (blues, reds, NO-CYC)
    }
  }) ({},{},NO-CYC);
  NDFS-SI-Statistics.stop-nres;
  RETURN (extract-res cyc)
}
```

concrete-definition *blue-dfs-fe* **uses** *blue-dfs-def*

```
is blue-dfs G ≡ do {
  NDFS-SI-Statistics.start-nres;
  (-,-,cyc) ← ?FE;
  NDFS-SI-Statistics.stop-nres;
  RETURN (extract-res cyc)
}
```

concrete-definition *blue-dfs-body* **uses** *blue-dfs-fe-def*

```
is - ≡ FOREACHc (g-V0 G) (λ(-,-,cyc). cyc=NO-CYC)
  (λv0 (blues,reds,-). do {
```

if v0∉blues then do {
  (*blues*,*reds*,-,*cyc*) ← REC_T *?B* (*blues*,*reds*,{},*v0*);
  RETURN (*blues*, *reds*, *cyc*)
} *else do* {*RETURN* (*blues*, *reds*, *NO-CYC*)}
}) ({},{},*NO-CYC*)

thm *blue-dfs-body-def*

1.3 Correctness

Additional invariant to be maintained between calls of red dfs

definition *red-dfs-inv E U reds onstack* ≡
  E‘‘*U* ⊆ *U* — Upper bound is closed under transitions
  ∧ *finite U* — Upper bound is finite
  ∧ *reds* ⊆ *U* — Red set below upper bound
  ∧ *E*‘‘*reds* ⊆ *reds* — Red nodes closed under reachability
  ∧ *E*‘‘*reds* ∩ *onstack* = {} — No red node with edge to stack

lemma *red-dfs-inv-initial*:
  assumes *finite* (E^* ‘‘*V0*)
  shows *red-dfs-inv E* (E^* ‘‘*V0*) {} {}
  using *assms* **unfolding** *red-dfs-inv-def*
  apply (*auto intro*: *rev-ImageI*)
  done

Correctness of the red DFS.

theorem *red-dfs-correct*:
  fixes *v0 u0* :: ′*v*
  assumes *PRE*:
    red-dfs-inv E U reds onstack
    u0∈*U*
    u0∉*reds*
  shows *red-dfs E onstack reds u0*
    ≤ *SPEC* (λ(*reds*′,*cyc*). *case cyc of*
      Some (*p*,*v*) ⇒ *v*∈*onstack* ∧ *p*≠[] ∧ *path E u0 p v*
    | *None* ⇒
        red-dfs-inv E U reds′ *onstack*
        ∧ *u0*∈*reds*′
        ∧ *reds*′ ⊆ *reds* ∪ E^* ‘‘ {*u0*}
)
proof −
  let *?dfs-red* =
    REC_T (λ*D* (*V*,*u*). *do* {
      let V=*insert u V*;
      NDFS-SI-Statistics.vis-red-nres;

      — Check whether we have a successor on stack
      brk ← $FOREACH_C$ (*E*‘‘{*u*}) (λ*brk*. *brk*=*None*)

```
      (λt -. if t∈onstack then
          RETURN (red-init-witness u t)
        else RETURN None)
      None;

    — Recurse for successors
    case brk of
      None ⇒
        FOREACH_C (E‘‘{u}) (λ(V,brk). brk=None)
          (λt (V,-).
            if t∉V then do {
              (V,brk) ← D (V,t);
              RETURN (V,prep-wit-red u brk)
            } else RETURN (V,None))
          (V,None)
    | - ⇒ RETURN (V,brk)
  }) (V,u)

let REC_T ?body ?init =     ?dfs-red

define pre where pre = (λS (V,u0). gen-dfs-pre E U S V u0 ∧ E‘‘V ∩ onstack
= {})
 define post where post = (λS (V0,u0) (V,cyc). gen-dfs-post E U S V0 u0 V
(cyc≠None)
  ∧ (case cyc of None ⇒ E‘‘V ∩ onstack = {}
    | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v))


define fe-inv where fe-inv = (λS V0 u0 it (V,cyc).
  gen-dfs-fe-inv E U S V0 u0 it V (cyc≠None)
  ∧ (case cyc of None ⇒ E‘‘V ∩ onstack = {}
    | Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v))



from PRE have GENPRE: gen-dfs-pre E U {} reds u0
  unfolding red-dfs-inv-def gen-dfs-pre-def
  by auto
with PRE have PRE′: pre {} (reds,u0)
  unfolding pre-def red-dfs-inv-def
  by auto

have IMP-POST: SPEC (post {} (reds,u0))
  ≤ SPEC (λ(reds′,cyc). case cyc of
    Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path E u0 p v
  | None ⇒
      red-dfs-inv E U reds′ onstack
      ∧ u0∈reds′
      ∧ reds′ ⊆ reds ∪ E* ‘‘ {u0})
```

```
  apply (clarsimp split: option.split)
  apply (intro impI conjI allI)
  apply simp-all
proof -
  fix reds' p v
  assume post {} (reds,u0) (reds',Some (p,v))
  thus v∈onstack and p≠[] and path E u0 p v
    unfolding post-def by auto
next
  fix reds'
  assume post {} (reds, u0) (reds', None)
  hence GPOST: gen-dfs-post E U {} reds u0 reds' False
    and NS: E``reds' ∩ onstack = {}
    unfolding post-def by auto

  from GPOST show u0∈reds' unfolding gen-dfs-post-def by auto

  show red-dfs-inv E U reds' onstack
    unfolding red-dfs-inv-def
    apply (intro conjI)
    using GENPRE[unfolded gen-dfs-pre-def]
    apply (simp-all) [2]
    apply (rule gen-dfs-post-imp-below-U[OF GENPRE GPOST])
    using GPOST[unfolded gen-dfs-post-def] apply simp
    apply fact
    done

  from GPOST show reds' ⊆ reds ∪ E* `` {u0}
    unfolding gen-dfs-post-def by auto
qed

{
  fix σ S
  assume INV0: pre S σ
  have REC_T ?body σ
    ≤ SPEC (post S σ)

    apply (rule RECT-rule-arb[where
      pre=pre and
      V=gen-dfs-var U <*lex*> {} and
      arb=S
      ])

    apply refine-mono

    using INV0[unfolded pre-def] apply (auto intro: gen-dfs-pre-imp-wf) []

    apply fact
```

apply (*rename-tac D S u*)
apply (*intro refine-vcg*)

apply (*rule-tac I*=*λit cyc*.
  (*case cyc of None* ⇒ (*E``*{*b*} − *it*) ∩ *onstack* = {}
    | *Some* (*p*,*v*) ⇒ (*v*∈*onstack* ∧ *p*≠[] ∧ *path E b p v*))
  in *FOREACHc-rule*)
apply (*auto simp add*: *pre-def gen-dfs-pre-imp-fin*) []
apply *auto* []
apply (*auto*
  split: *option*.*split*
  simp: *red-init-witness-def intro*: *path1*) []

apply (*intro refine-vcg*)

apply (*rule-tac I*=*fe-inv* (*insert b S*) (*insert b a*) *b* **in**
  FOREACHc-rule
)
apply (*auto simp add*: *pre-def gen-dfs-pre-imp-fin*) []

apply (*auto simp add*: *pre-def fe-inv-def gen-dfs-pre-imp-fe*) []

apply (*intro refine-vcg*)

apply (*rule order-trans*)
apply (*rprems*)
apply (*clarsimp simp add*: *pre-def fe-inv-def*)
apply (*rule gen-dfs-fe-inv-imp-pre*, *assumption*+) []
apply (*auto simp add*: *pre-def fe-inv-def intro*: *gen-dfs-fe-inv-imp-var*) []

apply (*clarsimp simp add*: *pre-def post-def fe-inv-def*
  split: *option*.*split-asm prod*.*split-asm*
) []
  apply (*blast intro*: *gen-dfs-post-imp-fe-inv*)
  apply (*blast intro*: *gen-dfs-post-imp-fe-inv path-prepend*)

apply (*auto simp add*: *pre-def post-def fe-inv-def*
  intro: *gen-dfs-fe-inv-pres-visited*) []

apply (*auto simp add*: *pre-def post-def fe-inv-def*
  intro: *gen-dfs-fe-inv-imp-post*) []

apply (*auto simp add*: *pre-def post-def fe-inv-def*
  intro: *gen-dfs-fe-imp-post-brk*) []

apply (*auto simp add*: *pre-def post-def fe-inv-def* *intro*: *gen-dfs-pre-imp-post-brk*) []

apply (*auto simp add*: *pre-def post-def fe-inv-def*
intro: *gen-dfs-pre-imp-post-brk*) []

done
} **note** *GEN=this*

note *GEN*[*OF PRE'*]
also note *IMP-POST*
finally show *?thesis*
unfolding *red-dfs-def* **.**
qed

Main theorem: Correctness of the blue DFS

theorem *blue-dfs-correct*:
fixes *G* :: (*'v*,-) *b-graph-rec-scheme*
assumes *b-graph G*
assumes *finitely-reachable*: *finite* ((*g-E G*)* `` *g-V0 G*)
shows *blue-dfs G* ≤ *SPEC* (λ*r*.
case r of None ⇒ (∀ *L*. ¬*b-graph.is-lasso-prpl G L*)
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*)
proof −
interpret *b-graph G* **by** *fact*

let *?A* = *bg-F G*
let *?E* = *g-E G*
let *?V0* = *g-V0 G*

let *?U* = *?E** ``*?V0*

define *add-inv* **where** *add-inv* = (λ*blues reds onstack*.
¬(∃ *v*∈(*blues*−*onstack*)∩*?A*. (*v*,*v*)∈*?E*$^+$) — No cycles over finished, accepting states
∧ *reds* ⊆ *blues* — Red nodes are also blue
∧ *reds* ∩ *onstack* = {} — No red nodes on stack
∧ *red-dfs-inv ?E ?U reds onstack*)

define *cyc-post* **where** *cyc-post* = (λ*blues reds onstack u0 cyc*. (*case cyc of*
NO-CYC ⇒ *add-inv blues reds onstack*
| *REACH u p* ⇒
path ?E u0 p u
∧ *u* ∈ *onstack*−{*u0*}
∧ *insert u* (*set p*) ∩ *?A* ≠ {}
| *CIRC pr pl* ⇒ ∃ *v*.
pl≠[]
∧ *path ?E v pl v*
∧ *path ?E u0 pr v*
∧ *set pl* ∩ *?A* ≠ {}

))

define *pre* **where** *pre* = (λ(*blues*,*reds*,*onstack*,*u*::′*v*).
gen-dfs-pre ?E ?U onstack blues u ∧ *add-inv blues reds onstack*)

define *post* **where** *post* = (λ(*blues0*,*reds0*::′*v set*,*onstack0*,*u0*) (*blues*,*reds*,*onstack*,*cyc*).

onstack = *onstack0*
∧ *gen-dfs-post ?E ?U onstack0 blues0 u0 blues* (*cyc*≠*NO-CYC*)
∧ *cyc-post blues reds onstack u0 cyc*)

define *fe-inv* **where** *fe-inv* = (λ*blues0 u0 onstack0 it* (*blues*,*reds*,*onstack*,*cyc*).
onstack=*onstack0*
∧ *gen-dfs-fe-inv ?E ?U onstack0 blues0 u0 it blues* (*cyc*≠*NO-CYC*)
∧ *cyc-post blues reds onstack u0 cyc*)

define *outer-inv* **where** *outer-inv* = (λ*it* (*blues*,*reds*,*cyc*).
case cyc of
NO-CYC ⇒
add-inv blues reds {}
∧ *gen-dfs-outer ?E ?U ?V0 it blues False*
| *CIRC pr pl* ⇒ ∃ *v0*∈*?V0*. ∃ *v*.
pl≠[]
∧ *path ?E v pl v*
∧ *path ?E v0 pr v*
∧ *set pl* ∩ *?A* ≠ {}
| - ⇒ *False*)

have *OUTER-INITIAL*: *outer-inv V0* ({}, {}, *NO-CYC*)
unfolding *outer-inv-def add-inv-def*
using *finitely-reachable*
apply (*auto intro*: *red-dfs-inv-initial gen-dfs-outer-initial*)
done

{
fix *onstack blues u0 reds*
assume *pre* (*blues*,*reds*,*onstack*,*u0*)
hence *fe-inv* (*insert u0 blues*) *u0* (*insert u0 onstack*) (*?E*‘‘{*u0*})
(*insert u0 blues*,*reds*,*insert u0 onstack*,*NO-CYC*)
unfolding *fe-inv-def add-inv-def cyc-post-def*
apply *clarsimp*
apply (*intro conjI*)
apply (*simp add*: *pre-def gen-dfs-pre-imp-fe*)
apply (*auto simp*: *pre-def add-inv-def*) []
apply (*auto simp*: *pre-def add-inv-def*) []
apply (*auto simp*: *pre-def add-inv-def gen-dfs-pre-def*) []
apply (*auto simp*: *pre-def add-inv-def*) []

apply (*unfold pre-def add-inv-def red-dfs-inv-def gen-dfs-pre-def*) []

```
    apply clarsimp
    apply blast
    done
} note PRE-IMP-FE = this

have [simp]: ⋀u cyc. prep-wit-blue u cyc = NO-CYC ⟷ cyc=NO-CYC
  by (case-tac cyc) auto

{
  fix blues0 reds0 onstack0 and u0::'v and
    blues reds onstack blues' reds' onstack'
    cyc it t
  assume PRE: pre (blues0,reds0,onstack0,u0)
  assume FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,NO-CYC)
  assume IT: t∈it    it⊆?E``{u0}    t∉blues
  assume POST: post (blues,reds,onstack, t) (blues',reds',onstack',cyc)
  note [simp del] = path-simps
  have fe-inv (insert u0 blues0) u0 (insert u0 onstack0) (it−{t})
    (blues',reds',onstack',prep-wit-blue u0 cyc)
    unfolding fe-inv-def
    using PRE FEI IT POST
    unfolding fe-inv-def post-def pre-def
    apply (clarsimp)
    apply (intro allI impI conjI)
    apply (blast intro: gen-dfs-post-imp-fe-inv)
    unfolding cyc-post-def
    apply (auto split: blue-witness.split-asm simp: path-simps)
    done
} note FE-INV-PRES=this


{
  fix blues reds onstack u0
  assume pre (blues,reds,onstack,u0)
  hence u0∈?E* ``?V0
    unfolding pre-def gen-dfs-pre-def by auto
} note PRE-IMP-REACH = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack
  assume A: pre (blues0,reds0,onstack0,u0)
     fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO-CYC)
     u0∈?A
  have u0∉reds using A
    unfolding fe-inv-def add-inv-def pre-def cyc-post-def
    apply auto
    done
```

```
} note FE-IMP-RED-PRE = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack rcyc reds'
  assume PRE: pre (blues0,reds0,onstack0,u0)
  assume FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO-CYC)
  assume ACC: u0∈?A
  assume SPECR: case rcyc of
    Some (p,v) ⇒ v∈onstack ∧ p≠[] ∧ path ?E u0 p v
  | None ⇒
      red-dfs-inv ?E ?U reds' onstack
      ∧ u0∈reds'
      ∧ reds' ⊆ reds ∪ ?E* `` {u0}
  have post (blues0,reds0,onstack0,u0)
    (blues,reds',onstack − {u0},init-wit-blue u0 rcyc)
    unfolding post-def add-inv-def cyc-post-def
    apply (clarsimp)
    apply (intro conjI)
  proof goal-cases
    from PRE FEI show OS0[symmetric]: onstack − {u0} = onstack0
      by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def)

    from PRE FEI have u0∈onstack
      unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
      by auto

    from PRE FEI
    show POST: gen-dfs-post ?E (?E* `` ?V0) onstack0 blues0 u0 blues
      (init-wit-blue u0 rcyc ≠ NO-CYC)
      by (auto simp: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-post)

    case 3

    from FEI have [simp]: onstack=insert u0 onstack0
      unfolding fe-inv-def by auto
    from FEI have u0∈blues unfolding fe-inv-def gen-dfs-fe-inv-def by auto

    show ?case
      apply (cases rcyc)
      apply (simp-all add: split-paired-all)
    proof −
      assume [simp]: rcyc=None
      show (∀ v∈(blues − (onstack0 − {u0})) ∩ ?A. (v, v) ∉ ?E+) ∧
        reds' ⊆ blues ∧
        reds' ∩ (onstack0 − {u0}) = {} ∧
        red-dfs-inv ?E (?E* `` ?V0) reds' (onstack0 − {u0})
      proof (intro conjI)
        from SPECR have RINV: red-dfs-inv ?E ?U reds' onstack
```

```
  and u0∈reds'
  and REDS'R: reds' ⊆ reds ∪ ?E* ``{u0}
  by auto

from RINV show
  RINV': red-dfs-inv ?E (?E* `` ?V0) reds' (onstack0 − {u0})
  unfolding red-dfs-inv-def by auto

from RINV'[unfolded red-dfs-inv-def] have
  REDS'CL: ?E``reds' ⊆ reds'
  and DJ': ?E `` reds' ∩ (onstack0 − {u0}) = {} by auto

from RINV[unfolded red-dfs-inv-def] have
  DJ: ?E `` reds' ∩ (onstack) = {} by auto

show reds' ⊆ blues
proof
  fix v assume v∈reds'
  with REDS'R have v∈reds ∨ (u0,v)∈?E* by blast
  thus v∈blues proof
    assume v∈reds
    moreover with FEI have reds⊆blues
      unfolding fe-inv-def add-inv-def cyc-post-def by auto
    ultimately show ?thesis ..
  next
    from POST[unfolded gen-dfs-post-def OS0] have
      CL: ?E `` (blues − (onstack0 − {u0})) ⊆ blues and u0∈blues
      by auto
    from PRE FEI have onstack0 ⊆ blues
      unfolding pre-def fe-inv-def gen-dfs-pre-def gen-dfs-fe-inv-def
      by auto

    assume (u0,v)∈?E*
    thus v∈blues
    proof (cases rule: rtrancl-last-visit[where S=onstack − {u0}])
      case no-visit
      thus v∈blues using ‹u0∈blues› CL
        by induct (auto elim: rtranclE)
    next
      case (last-visit-point u)
      then obtain uh where (u0,uh)∈?E* and (uh,u)∈?E
        by (metis tranclD2)
      with REDS'CL DJ' ‹u0∈reds'› have uh∈reds'
        by (auto dest: Image-closed-trancl)
      with DJ' ‹(uh,u)∈?E› ‹u ∈ onstack − {u0}› have False
        by simp blast
      thus ?thesis ..
    qed
  qed
```

```
    qed

    show ∀ v∈(blues − (onstack0 − {u0})) ∩ ?A. (v, v) ∉ ?E⁺
    proof
      fix v
      assume A: v ∈ (blues − (onstack0 − {u0})) ∩ ?A
      show (v,v)∉?E⁺ proof (cases v=u0)
        assume v≠u0
        with A have v∈(blues − (insert u0 onstack)) ∩ ?A by auto
        with FEI show ?thesis
          unfolding fe-inv-def add-inv-def cyc-post-def by auto
      next
        assume [simp]: v=u0
        show ?thesis proof
          assume (v,v)∈?E⁺
          then obtain uh where (u0,uh)∈?E* and (uh,u0)∈?E
            by (auto dest: tranclD2)
          with REDS'CL DJ ‹u0∈reds'› have uh∈reds'
            by (auto dest: Image-closed-trancl)
          with DJ ‹(uh,u0)∈?E› ‹u0 ∈ onstack› show False by blast
        qed
      qed
    qed

    show reds' ∩ (onstack0 − {u0}) = {}
    proof (rule ccontr)
      assume reds' ∩ (onstack0 − {u0}) ≠ {}
      then obtain v where v∈reds' and v∈onstack0 and v≠u0 by auto

      from ‹v∈reds'› REDS'R have v∈reds ∨ (u0,v)∈?E*
        by auto
      thus False proof
        assume v∈reds
        with FEI[unfolded fe-inv-def add-inv-def cyc-post-def]
          ‹v∈onstack0›
        show False by auto
      next
        assume (u0,v)∈?E*
        with ‹v≠u0› obtain uh where (u0,uh)∈?E* and (uh,v)∈?E
          by (auto elim: rtranclE)
        with REDS'CL DJ ‹u0∈reds'› have uh∈reds'
          by (auto dest: Image-closed-trancl)
        with DJ ‹(uh,v)∈?E› ‹v ∈ onstack0› show False by simp blast
      qed
    qed
  qed
next
  fix u p
  assume [simp]: rcyc = Some (p,u)
```

```
      show
        (u = u0 ⟶ p ≠ [] ∧ path ?E u0 p u0 ∧ set p ∩ ?A ≠ {}) ∧
        (u ≠ u0 ⟶
            path ?E u0 p u ∧ u ∈ onstack0 ∧ (u ∈ ?A ∨ set p ∩ ?A ≠ {}))
      proof (intro conjI impI)
        from SPECR ‹u0∈?A› show
          u≠u0 ⟹ u∈onstack0
          p≠[]
          path ?E u0 p u
          u=u0 ⟹ path ?E u0 p u0
          set p ∩ F ≠ {}
          u ∈ F ∨ set p ∩ F ≠ {}
          by (auto simp: neq-Nil-conv path-simps)
      qed
    qed
  qed
} note RED-IMP-POST = this

{
  fix blues0 reds0 onstack0 u0 blues reds onstack and cyc :: 'v blue-witness
  assume PRE: pre (blues0,reds0,onstack0,u0)
  and FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      {} (blues,reds,onstack,NO-CYC)
  and FC[simp]: cyc=NO-CYC
  and NCOND: u0∉?A

  from PRE FEI have OS0: onstack0 = onstack − {u0}
    by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def) []

  from PRE FEI have u0∈onstack
    unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
    by auto
  with OS0 have OS1: onstack = insert u0 onstack0 by auto

  have post (blues0,reds0,onstack0,u0) (blues,reds,onstack − {u0},NO-CYC)
    apply (clarsimp simp: post-def cyc-post-def) []
    apply (intro conjI impI)
    apply (simp add: OS0)
    using PRE FEI apply (auto
      simp: pre-def fe-inv-def intro: gen-dfs-fe-inv-imp-post) []

    using FEI[unfolded fe-inv-def cyc-post-def] unfolding add-inv-def
    apply clarsimp
    apply (intro conjI)
    using NCOND apply auto []
    apply auto []
    apply (clarsimp simp: red-dfs-inv-def, blast) []
    done
} note NCOND-IMP-POST=this
```

```
{
  fix blues0 reds0 onstack0 u0 blues reds onstack it
    and cyc :: 'v blue-witness
  assume PRE: pre (blues0,reds0,onstack0,u0)
  and FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,cyc)
  and NC: cyc≠NO-CYC
  and IT: it⊆?E``{u0}
  from PRE FEI have OS0: onstack0 = onstack − {u0}
    by (auto simp: pre-def fe-inv-def add-inv-def gen-dfs-pre-def) []

  from PRE FEI have u0∈onstack
    unfolding pre-def gen-dfs-pre-def fe-inv-def gen-dfs-fe-inv-def
    by auto
  with OS0 have OS1: onstack = insert u0 onstack0 by auto

  have post (blues0,reds0,onstack0,u0) (blues,reds,onstack − {u0},cyc)
    apply (clarsimp simp: post-def) []
    apply (intro conjI impI)
    apply (simp add: OS0)
    using PRE FEI IT NC apply (auto
      simp: pre-def fe-inv-def intro: gen-dfs-fe-imp-post-brk) []
    using FEI[unfolded fe-inv-def] NC
    unfolding cyc-post-def
    apply (auto split: blue-witness.split simp: OS1) []
    done
} note BREAK-IMP-POST = this


{
  fix blues0 reds0 onstack0 and u0::'v and
    blues reds onstack cyc it t
  assume PRE: pre (blues0,reds0,onstack0,u0)
  assume FEI: fe-inv (insert u0 blues0) u0 (insert u0 onstack0)
      it (blues,reds,onstack,NO-CYC)
  assume IT: it⊆?E``{u0}     t∈it
  assume T-OS: t ∈ onstack
  assume U0ACC: u0∈F ∨ t∈F


  from T-OS have TIB: t ∈ blues using PRE FEI
    by (auto simp add: fe-inv-def pre-def gen-dfs-fe-inv-def gen-dfs-pre-def)

  have fe-inv (insert u0 blues0) u0 (insert u0 onstack0) (it − {t})
    (blues,reds,onstack,init-wit-blue-early u0 t)
    unfolding fe-inv-def
    apply (clarsimp simp: it-step-insert-iff[OF IT])
    apply (intro conjI)
```

```
    using PRE FEI apply (simp add: fe-inv-def pre-def)


    using FEI TIB apply (auto simp add: fe-inv-def gen-dfs-fe-inv-def) []

    unfolding cyc-post-def init-wit-blue-early-def
    using IT T-OS U0ACC apply (auto simp: path-simps) []
    done
} note EARLY-DET-OPT = this


{
  fix σ
  assume INV0: pre σ

  have REC_T (blue-dfs-body G) σ ≤ SPEC (post σ)
    apply (intro refine-vcg
      RECT-rule[where pre=pre
      and V=gen-dfs-var ?U <*lex*> {}]
    )
    apply (unfold blue-dfs-body-def, refine-mono) []
    apply (blast intro!: fin-U-imp-wf finitely-reachable)
    apply (rule INV0)


    apply (simp (no-asm) only: blue-dfs-body-def)
    apply (refine-rcg refine-vcg)




    apply (rule-tac
      I=fe-inv (insert bb a) bb (insert bb ab)
      in FOREACHc-rule')

    apply (auto simp add: pre-def gen-dfs-pre-imp-fin) []

    apply (blast intro: PRE-IMP-FE)

    apply (intro refine-vcg)


    apply (blast intro: EARLY-DET-OPT)


    apply (rule order-trans)
    apply (rprems)
    apply (clarsimp simp add: pre-def fe-inv-def cyc-post-def)
    apply (rule gen-dfs-fe-inv-imp-pre, assumption+) []
```

apply (*auto simp add*: *pre-def fe-inv-def intro*: *gen-dfs-fe-inv-imp-var*) []

apply (*auto intro*: *FE-INV-PRES*) []

apply (*auto simp add*: *pre-def post-def fe-inv-def*
intro: *gen-dfs-fe-inv-pres-visited*) []

apply (*intro refine-vcg*)

apply (*rule order-trans*)
apply (*rule red-dfs-correct*[**where** U=?E^* ‘‘ ?$V0$])
apply (*auto simp add*: *fe-inv-def add-inv-def cyc-post-def*) []
apply (*auto intro*: *PRE-IMP-REACH*) []
apply (*auto dest*: *FE-IMP-RED-PRE*) []

apply (*intro refine-vcg*)
apply *clarsimp*
apply (*rule RED-IMP-POST*, *assumption+*) []

apply (*clarsimp*, *blast intro*: *NCOND-IMP-POST*) []

apply (*intro refine-vcg*)
apply *simp*

apply (*clarsimp*, *blast intro*: *BREAK-IMP-POST*) []
done
} **note** *GEN=this*

{
fix *v0 it blues reds*
assume $v0 \in it$ $it \subseteq V0$ $v0 \notin blues$
outer-inv it (*blues*, *reds*, *NO-CYC*)
hence *pre* (*blues*, *reds*, {}, *v0*)
unfolding *pre-def outer-inv-def*
by (*auto intro*: *gen-dfs-pre-initial*)
} **note** *OUTER-IMP-PRE* = *this*

{
fix *v0 it blues0 reds0 blues reds onstack cyc*
assume $v0 \in it$ $it \subseteq V0$ $v0 \notin blues0$
outer-inv it (*blues0*, *reds0*, *NO-CYC*)
post (*blues0*, *reds0*, {}, *v0*) (*blues*, *reds*, *onstack*, *cyc*)
hence *outer-inv* ($it - \{v0\}$) (*blues*, *reds*, *cyc*)
unfolding *post-def outer-inv-def cyc-post-def*
by (*fastforce split*: *blue-witness.split intro*: *gen-dfs-post-imp-outer*)
} **note** *POST-IMP-OUTER* = *this*

{

```
  fix v0 it blues reds
  assume v0 ∈ it    it ⊆ V0    outer-inv it (blues, reds, NO-CYC)
    v0 ∈ blues
  hence outer-inv (it − {v0}) (blues, reds, NO-CYC)
    unfolding outer-inv-def
    by (auto intro: gen-dfs-outer-already-vis)
} note OUTER-ALREX = this


{
  fix it blues reds cyc
  assume outer-inv it (blues, reds, cyc)    cyc ≠ NO-CYC
  hence case extract-res cyc of
      None ⇒ ∀ L. ¬ is-lasso-prpl L
    | Some x ⇒ is-lasso-prpl x
    unfolding outer-inv-def extract-res-def is-lasso-prpl-def
      is-lasso-prpl-pre-def
    apply (cases cyc)
    apply auto
    done
} note IMP-POST-CYC = this

{ fix pr pl blues reds
  assume ADD-INV: add-inv blues reds {}
  assume GEN-INV: gen-dfs-outer E (E* `` V0) V0 {} blues False
  assume LASSO: is-lasso-prpl (pr, pl)

  from LASSO[unfolded is-lasso-prpl-def is-lasso-prpl-pre-def]
  obtain v0 va where
    v0∈V0    pl≠[] and
    PR: path E v0 pr va and PL: path E va pl va and
    F: set pl ∩ F ≠ {}
    by auto

  from F obtain pl1 vf pl2 where [simp]: pl=pl1@vf#pl2 and vf∈F
    by (fastforce simp: in-set-conv-decomp)

  from PR PL have path E v0 (pr@pl1) vf    path E vf (vf#pl2@pl1) vf
    by (auto simp: path-simps)
  hence (v0,vf)∈E* and (vf,vf)∈E+
    by (auto dest: path-is-rtrancl path-is-trancl)

  from GEN-INV ‹v0∈V0› ‹(v0,vf)∈E*› have vf∈blues
    unfolding gen-dfs-outer-def
    apply (clarsimp)
    by (metis Image-closed-trancl rev-ImageI rev-subsetD)

  from ADD-INV[unfolded add-inv-def] ‹vf∈blues› ‹vf∈F› ‹(vf,vf)∈E+›
  have False by auto
```

} **note** *IMP-POST-NOCYC-AUX* = *this*

{
  fix *blues reds cyc*
  assume *outer-inv* {} (*blues*, *reds*, *cyc*)
  hence *case extract-res cyc of*
      None ⇒ ∀ *L*. ¬ *is-lasso-prpl L*
    | *Some x* ⇒ *is-lasso-prpl x*
    apply (*cases cyc*)
    apply (*simp-all add*: *IMP-POST-CYC*)
    unfolding *outer-inv-def extract-res-def*
    apply (*auto intro*: *IMP-POST-NOCYC-AUX*)
    done
} **note** *IMP-POST-NOCYC* = *this*

show *?thesis*
  unfolding *blue-dfs-fe.refine blue-dfs-body.refine*
  apply (*refine-rcg*
    FOREACHc-rule[**where** *I*=*outer-inv*]
    refine-vcg
)

  apply (*simp add*: *finitely-reachable finite-V0*)

  apply (*rule OUTER-INITIAL*)

  apply (*rule order-trans*[*OF GEN*])
  apply (*clarsimp*, *blast intro*: *OUTER-IMP-PRE*)

  apply (*clarsimp*, *blast intro*: *POST-IMP-OUTER*)

  apply (*clarsimp*, *blast intro*: *OUTER-ALREX*)

  apply (*clarsimp*, *blast intro*: *IMP-POST-NOCYC*)

  apply (*clarsimp*, *blast intro*: *IMP-POST-CYC*)
  done
qed

1.4 Refinement

1.4.1 Setup for Custom Datatypes

This effort can be automated, but currently, such an automation is not yet implemented

abbreviation *red-wit-rel R* ≡ ⟨⟨⟨*R*⟩*list-rel*,*R*⟩*prod-rel*⟩*option-rel*
abbreviation *i-red-wit I* ≡ $\langle\langle\langle I\rangle_i$*i-list*,$I\rangle_i$*i-prod*$\rangle_i$*i-option*

abbreviation *blue-wit-rel* ≡ (*Id*::(- *blue-witness* × -) *set*)
consts *i-blue-wit* :: *interface*

lemmas [*autoref-rel-intf*] = *REL-INTFI*[*of blue-wit-rel i-blue-wit*]

term *init-wit-blue-early*

lemma [*autoref-itype*]:
 NO-CYC $::_i$ *i-blue-wit*
 (=) $::_i$ *i-blue-wit* $\rightarrow_i$ *i-blue-wit* $\rightarrow_i$ *i-bool*
 init-wit-blue $::_i$ *I* $\rightarrow_i$ *i-red-wit I* $\rightarrow_i$ *i-blue-wit*
 init-wit-blue-early $::_i$ *I* $\rightarrow_i$ *I* $\rightarrow_i$ *i-blue-wit*
 prep-wit-blue $::_i$ *I* $\rightarrow_i$ *i-blue-wit* $\rightarrow_i$ *i-blue-wit*
 red-init-witness $::_i$ *I* $\rightarrow_i$ *I* $\rightarrow_i$ *i-red-wit I*
 prep-wit-red $::_i$ *I* $\rightarrow_i$ *i-red-wit I* $\rightarrow_i$ *i-red-wit I*
 extract-res $::_i$ *i-blue-wit* $\rightarrow_i$ $\langle\langle\langle I\rangle_i$*i-list*, $\langle I\rangle_i$*i-list*$\rangle_i$*i-prod*$\rangle_i$*i-option*
 red-dfs $::_i$ $\langle I\rangle_i$*i-slg* $\rightarrow_i$ $\langle I\rangle_i$*i-set* $\rightarrow_i$ $\langle I\rangle_i$*i-set* $\rightarrow_i$ *I*
  $\rightarrow_i$ $\langle\langle\langle I\rangle_i$*i-set*, *i-red-wit I*$\rangle_i$*i-prod*$\rangle_i$*i-nres*
 blue-dfs $::_i$ *i-bg i-unit I*
  $\rightarrow_i$ $\langle\langle\langle\langle I\rangle_i$*i-list*, $\langle I\rangle_i$*i-list*$\rangle_i$*i-prod*$\rangle_i$*i-option*$\rangle_i$*i-nres*
 by *auto*

context begin interpretation *autoref-syn* **.**
lemma [*autoref-op-pat*]: *NO-CYC* ≡ *OP NO-CYC* $:::_i$ *i-blue-wit* **by** *simp*
end

term *lasso-rel-ext*

lemma *autoref-wit*[*autoref-rules-raw*]:
 (*NO-CYC*,*NO-CYC*)∈*blue-wit-rel*
 ((=), (=)) ∈ *blue-wit-rel* → *blue-wit-rel* → *bool-rel*
 ⋀*R*. *PREFER-id R*
  ⟹ (*init-wit-blue*, *init-wit-blue*) ∈ *R* → *red-wit-rel R* → *blue-wit-rel*
 ⋀*R*. *PREFER-id R*
  ⟹ (*init-wit-blue-early*, *init-wit-blue-early*) ∈ *R* → *R* → *blue-wit-rel*
 ⋀*R*. *PREFER-id R*
  ⟹ (*prep-wit-blue*,*prep-wit-blue*)∈*R* → *blue-wit-rel* → *blue-wit-rel*
 ⋀*R*. *PREFER-id R*
  ⟹ (*red-init-witness*, *red-init-witness*) ∈ *R*→*R*→*red-wit-rel R*
 ⋀*R*. *PREFER-id R*
  ⟹ (*prep-wit-red*,*prep-wit-red*) ∈ *R* → *red-wit-rel R* → *red-wit-rel R*
 ⋀*R*. *PREFER-id R*
  ⟹ (*extract-res*,*extract-res*)
   ∈ *blue-wit-rel* → $\langle\langle R\rangle$*list-rel*$\times_r\langle R\rangle$*list-rel*$\rangle$*option-rel*
 by (*simp-all*)

1.4.2 Actual Refinement

term *red-dfs*

term *map2set-rel* (*rbt-map-rel ord*)

term *rbt-set-rel*

schematic-goal *red-dfs-refine-aux*: (*?f*::*?'c*, *red-dfs*::(('*a*::*linorder* × -) *set*⇒-)) ∈ *?R*
  supply [*autoref-tyrel*] = *ty-REL*[**where** '*a*='*a set* **and** *R*=⟨*Id*⟩*dflt-rs-rel*]
  unfolding *red-dfs-def*[*abs-def*]
  apply (*autoref* (*trace*,*keep-goal*))
  done
concrete-definition *impl-red-dfs* **uses** *red-dfs-refine-aux*

lemma *impl-red-dfs-autoref*[*autoref-rules*]:
  fixes *R* :: ('*a*×'*a*::*linorder*) *set*
  assumes *PREFER-id R*
  shows (*impl-red-dfs*, *red-dfs*) ∈
    ⟨*R*⟩*slg-rel* → ⟨*R*⟩*dflt-rs-rel* → ⟨*R*⟩*dflt-rs-rel* → *R*
    → ⟨⟨*R*⟩*dflt-rs-rel* $\times_r$ *red-wit-rel R*⟩*nres-rel*
  using *assms impl-red-dfs.refine* **by** *simp*

thm *autoref-itype*(*1*−*10*)

schematic-goal *code-red-dfs-aux*:
  shows *RETURN ?c* ≤ *impl-red-dfs E onstack V u*
  unfolding *impl-red-dfs-def*
  by (*refine-transfer* (*post*) *the-resI*)
concrete-definition *code-red-dfs* **uses** *code-red-dfs-aux*
prepare-code-thms *code-red-dfs-def*
declare *code-red-dfs.refine*[*refine-transfer*]

export-code *code-red-dfs* **checking** *SML*

schematic-goal *red-dfs-hash-refine-aux*: (*?f*::*?'c*, *red-dfs*::(('*a*::*hashable* × -) *set*⇒-)) ∈ *?R*
  supply [*autoref-tyrel*] = *ty-REL*[**where** '*a*='*a set* **and** *R*=⟨*Id*⟩*hs.rel*]
  unfolding *red-dfs-def*[*abs-def*]
  apply (*autoref* (*trace*,*keep-goal*))
  done
concrete-definition *impl-red-dfs-hash* **uses** *red-dfs-hash-refine-aux*

thm *impl-red-dfs-hash.refine*

lemma *impl-red-dfs-hash-autoref*[*autoref-rules*]:
  fixes *R* :: ('*a*×'*a*::*hashable*) *set*
  assumes *PREFER-id R*
  shows (*impl-red-dfs-hash*, *red-dfs*) ∈

⟨R⟩slg-rel → ⟨R⟩hs.rel → ⟨R⟩hs.rel → R
→ ⟨⟨R⟩hs.rel ×$_r$ red-wit-rel R⟩nres-rel
using *assms impl-red-dfs-hash.refine* **by** *simp*

schematic-goal *code-red-dfs-hash-aux*:
shows *RETURN ?c ≤ impl-red-dfs-hash E onstack V u*
unfolding *impl-red-dfs-hash-def*
by (*refine-transfer* (*post*) *the-resI*)
concrete-definition *code-red-dfs-hash* **uses** *code-red-dfs-hash-aux*
prepare-code-thms *code-red-dfs-hash-def*
declare *code-red-dfs-hash.refine*[*refine-transfer*]

export-code *code-red-dfs-hash* **checking** *SML*

schematic-goal *red-dfs-ahs-refine-aux*: (*?f*::*?'c*, *red-dfs*::(('a::hashable × -) set⇒-))
∈ *?R*
supply [*autoref-tyrel*] = *ty-REL*[**where** 'a='a::hashable set **and** R=⟨Id⟩ahs.rel]
unfolding *red-dfs-def*[*abs-def*]
apply (*autoref* (*trace*,*keep-goal*))
done
concrete-definition *impl-red-dfs-ahs* **uses** *red-dfs-ahs-refine-aux*

lemma *impl-red-dfs-ahs-autoref*[*autoref-rules*]:
fixes *R* :: ('a×'a::hashable) set
assumes *PREFER-id R*
shows (*impl-red-dfs-ahs*, *red-dfs*) ∈
⟨R⟩slg-rel → ⟨R⟩ahs.rel → ⟨R⟩ahs.rel → R
→ ⟨⟨R⟩ahs.rel ×$_r$ red-wit-rel R⟩nres-rel
using *assms impl-red-dfs-ahs.refine* **by** *simp*

schematic-goal *code-red-dfs-ahs-aux*:
shows *RETURN ?c ≤ impl-red-dfs-ahs E onstack V u*
unfolding *impl-red-dfs-ahs-def*
by (*refine-transfer the-resI*)
concrete-definition *code-red-dfs-ahs* **uses** *code-red-dfs-ahs-aux*
prepare-code-thms *code-red-dfs-ahs-def*
declare *code-red-dfs-ahs.refine*[*refine-transfer*]

export-code *code-red-dfs-ahs* **checking** *SML*

schematic-goal *blue-dfs-refine-aux*: (*?f*::*?'c*, *blue-dfs*::('a::linorder b-graph-rec⇒-))
∈ *?R*
supply [*autoref-tyrel*] =
ty-REL[**where** 'a='a **and** R=Id]
ty-REL[**where** 'a='a set **and** R=⟨Id⟩dflt-rs-rel]
unfolding *blue-dfs-def*[*abs-def*]
apply (*autoref* (*trace*,*keep-goal*))

done
concrete-definition *impl-blue-dfs* **uses** *blue-dfs-refine-aux*

thm *impl-blue-dfs.refine*

lemma *impl-blue-dfs-autoref*[*autoref-rules*]:
fixes R :: (*'a* × *'a*::*linorder*) *set*
assumes *PREFER-id R*
shows (*impl-blue-dfs*, *blue-dfs*)
∈ *bg-impl-rel-ext unit-rel R*
→ ⟨⟨⟨*R*⟩*list-rel* $\times_r$ ⟨*R*⟩*list-rel*⟩*Relators.option-rel*⟩*nres-rel*
using *assms impl-blue-dfs.refine* **by** *simp*

schematic-goal *code-blue-dfs-aux*:
shows *RETURN ?c* ≤ *impl-blue-dfs G*
unfolding *impl-blue-dfs-def*
apply (*refine-transfer* (*post*) *the-resI*
order-trans[*OF det-RETURN code-red-dfs.refine*])
done
concrete-definition *code-blue-dfs* **uses** *code-blue-dfs-aux*
prepare-code-thms *code-blue-dfs-def*
declare *code-blue-dfs.refine*[*refine-transfer*]

export-code *code-blue-dfs* **checking** *SML*

schematic-goal *blue-dfs-hash-refine-aux*: (*?f*::*?'c*, *blue-dfs*::(*'a*::*hashable b-graph-rec*⇒-))
∈ *?R*
supply [*autoref-tyrel*] =
ty-REL[**where** *'a*=*'a* **and** *R*=*Id*]
ty-REL[**where** *'a*=*'a*::*hashable set* **and** *R*=⟨*Id*⟩*hs.rel*]
unfolding *blue-dfs-def*[*abs-def*]
using [[*autoref-trace-failed-id*]]
apply (*autoref* (*trace*,*keep-goal*))
done
concrete-definition *impl-blue-dfs-hash* **uses** *blue-dfs-hash-refine-aux*

lemma *impl-blue-dfs-hash-autoref*[*autoref-rules*]:
fixes R :: (*'a* × *'a*::*hashable*) *set*
assumes *PREFER-id R*
shows (*impl-blue-dfs-hash*, *blue-dfs*) ∈ *bg-impl-rel-ext unit-rel R*
→ ⟨⟨⟨*R*⟩*list-rel* $\times_r$ ⟨*R*⟩*list-rel*⟩*Relators.option-rel*⟩*nres-rel*
using *assms impl-blue-dfs-hash.refine* **by** *simp*

schematic-goal *code-blue-dfs-hash-aux*:
shows *RETURN ?c* ≤ *impl-blue-dfs-hash G*
unfolding *impl-blue-dfs-hash-def*
apply (*refine-transfer the-resI*
order-trans[*OF det-RETURN code-red-dfs-hash.refine*])
done

concrete-definition *code-blue-dfs-hash* **uses** *code-blue-dfs-hash-aux*
prepare-code-thms *code-blue-dfs-hash-def*
declare *code-blue-dfs-hash.refine*[*refine-transfer*]

export-code *code-blue-dfs-hash* **checking** *SML*

schematic-goal *blue-dfs-ahs-refine-aux*: (*?f*::*?'c*, *blue-dfs*::(*'a*::*hashable b-graph-rec*⇒-)) ∈ *?R*
  supply [*autoref-tyrel*] =
    ty-REL[**where** *'a*=*'a* **and** *R*=*Id*]
    ty-REL[**where** *'a*=*'a*::*hashable set* **and** *R*=⟨*Id*⟩*ahs.rel*]
  unfolding *blue-dfs-def*[*abs-def*]
  apply (*autoref* (*trace*,*keep-goal*))
  done
concrete-definition *impl-blue-dfs-ahs* **uses** *blue-dfs-ahs-refine-aux*

lemma *impl-blue-dfs-ahs-autoref*[*autoref-rules*]:
  fixes *R* :: (*'a* × *'a*::*hashable*) *set*
  assumes *MINOR-PRIO-TAG 5*
  assumes *PREFER-id R*
  shows (*impl-blue-dfs-ahs*, *blue-dfs*) ∈ *bg-impl-rel-ext unit-rel R*
    → ⟨⟨⟨*R*⟩*list-rel* $\times_r$ ⟨*R*⟩*list-rel*⟩*Relators.option-rel*⟩*nres-rel*
  using *assms impl-blue-dfs-ahs.refine* **by** *simp*

thm *impl-blue-dfs-ahs-def*

schematic-goal *code-blue-dfs-ahs-aux*:
  shows *RETURN ?c* ≤ *impl-blue-dfs-ahs G*
  unfolding *impl-blue-dfs-ahs-def*
  apply (*refine-transfer the-resI*
    order-trans[*OF det-RETURN code-red-dfs-ahs.refine*])
  done
concrete-definition *code-blue-dfs-ahs* **uses** *code-blue-dfs-ahs-aux*
prepare-code-thms *code-blue-dfs-ahs-def*
declare *code-blue-dfs-ahs.refine*[*refine-transfer*]

export-code *code-blue-dfs-ahs* **checking** *SML*

Correctness theorem

theorem *code-blue-dfs-correct*:
  assumes *G*: *b-graph G*   *finite* ((*g-E G*)* `` *g-V0 G*)
  assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
  shows *RETURN* (*code-blue-dfs Gi*) ≤ *SPEC* (λ*r*.
    case r of None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
  | *Some L* ⇒ *b-graph.is-lasso-prpl G L*)
proof −
  note *code-blue-dfs.refine*
  also note *impl-blue-dfs.refine*[*param-fo*, *OF REL*, *THEN nres-relD*]
  also note *blue-dfs-correct*[*OF G*]

finally show *?thesis* **by** (*simp cong*: *option.case-cong*)
qed

theorem *code-blue-dfs-correct'*:
assumes *G*: *b-graph G* *finite* $((g\text{-}E\ G)^*$ *“ g-V0 G*)
assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
shows *case code-blue-dfs Gi of*
None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*
using *code-blue-dfs-correct*[*OF G REL*]
by *simp*

theorem *code-blue-dfs-hash-correct*:
assumes *G*: *b-graph G* *finite* $((g\text{-}E\ G)^*$ *“ g-V0 G*)
assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
shows *RETURN* (*code-blue-dfs-hash Gi*) ≤ *SPEC* (λ*r*.
case r of None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*)
proof −
note *code-blue-dfs-hash.refine*
also note *impl-blue-dfs-hash.refine*[*param-fo*, *OF REL*, *THEN nres-relD*]
also note *blue-dfs-correct*[*OF G*]
finally show *?thesis* **by** (*simp cong*: *option.case-cong*)
qed

theorem *code-blue-dfs-hash-correct'*:
assumes *G*: *b-graph G* *finite* $((g\text{-}E\ G)^*$ *“ g-V0 G*)
assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
shows *case code-blue-dfs-hash Gi of*
None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*
using *code-blue-dfs-hash-correct*[*OF G REL*]
by *simp*

theorem *code-blue-dfs-ahs-correct*:
assumes *G*: *b-graph G* *finite* $((g\text{-}E\ G)^*$ *“ g-V0 G*)
assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
shows *RETURN* (*code-blue-dfs-ahs Gi*) ≤ *SPEC* (λ*r*.
case r of None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*)
proof −
note *code-blue-dfs-ahs.refine*
also note *impl-blue-dfs-ahs.refine*[*param-fo*, *OF REL*, *THEN nres-relD*]
also note *blue-dfs-correct*[*OF G*]
finally show *?thesis* **by** (*simp cong*: *option.case-cong*)
qed

theorem *code-blue-dfs-ahs-correct'*:
assumes *G*: *b-graph G* *finite* $((g\text{-}E\ G)^*$ *“ g-V0 G*)

assumes *REL*: (*Gi*,*G*)∈*bg-impl-rel-ext unit-rel Id*
shows *case code-blue-dfs-ahs Gi of*
None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*
using *code-blue-dfs-ahs-correct*[*OF G REL*]
by *simp*

Export for benchmarking

schematic-goal *acc-of-list-impl-hash*:
notes [*autoref-tyrel*] =
ty-REL[**where** *'a=nat set* **and** *R*=⟨*nat-rel*⟩*iam-set-rel*]

shows (*?f*::*?'c*,λ*l*::*nat list*.
let s=(*set l*):::$_r$⟨*nat-rel*⟩*iam-set-rel*
in (λ*x*::*nat*. *x*∈*s*)
) ∈ *?R*
apply (*autoref* (*keep-goal*))
done

concrete-definition *acc-of-list-impl-hash* **uses** *acc-of-list-impl-hash*
export-code *acc-of-list-impl-hash* **checking** *SML*

definition *code-blue-dfs-nat*
≡ *code-blue-dfs* :: - ⇒ (*nat list* × -) *option*
definition *code-blue-dfs-hash-nat*
≡ *code-blue-dfs-hash* :: - ⇒ (*nat list* × -) *option*
definition *code-blue-dfs-ahs-nat*
≡ *code-blue-dfs-ahs* :: - ⇒ (*nat list* × -) *option*

definition *succ-of-list-impl-int* ≡
succ-of-list-impl o map (λ(*u*,*v*). (*nat-of-integer u*, *nat-of-integer v*))

definition *acc-of-list-impl-hash-int* ≡
acc-of-list-impl-hash o map nat-of-integer

export-code
code-blue-dfs-nat
code-blue-dfs-hash-nat
code-blue-dfs-ahs-nat
succ-of-list-impl-int
acc-of-list-impl-hash-int
nat-of-integer
integer-of-nat
lasso-ext
in *SML* **module-name** *HPY-new-hash*
file ‹*nested-dfs-hash.sml*›

end

2 Abstract Model-Checker

theory *CAVA-Abstract*
imports
  CAVA-Base.CAVA-Base
  CAVA-Automata.Automata
  LTL.LTL
begin

This theory defines the abstract version of the cava model checker, as well as a generic implementation.

2.1 Specification of an LTL Model-Checker

Abstractly, an LTL model-checker consists of three components:

1. A conversion of LTL-formula to Indexed Generalized Buchi Automata (IGBA) over sets of atomic propositions.
2. An intersection construction, which takes a system and an IGBA, and creates an Indexed Generalized Buchi Graph (IGBG) and a projection function to project runs of the IGBG back to runs of the system.
3. An emptiness check for IGBGs.

Given an LTL formula, the LTL to Buchi conversion returns a Generalized Buchi Automaton that accepts the same language.

definition *ltl-to-gba-spec*
  :: *'prop ltlc* ⇒ (*'q*, *'prop set*, -) *igba-rec-scheme nres*
  — Conversion of LTL formula to generalized buchi automaton
  where *ltl-to-gba-spec* φ ≡ *SPEC* (λ*gba*.
    igba.lang gba = *language-ltlc* φ ∧ *igba gba* ∧ *finite* ((*g-E gba*)* `` *g-V0 gba*))

definition *inter-spec*
  :: (*'s*,*'prop set*,-) *sa-rec-scheme*
  ⇒ (*'q*,*'prop set*,-) *igba-rec-scheme*
  ⇒ ((*'prod-state*,-) *igb-graph-rec-scheme* × (*'prod-state* ⇒ *'s*)) *nres*
  — Intersection of system and IGBA
  where ⋀*sys ba*. *inter-spec sys ba* ≡ *do* {
    ASSERT (*sa sys*);
    ASSERT (*finite* ((*g-E sys*)* `` *g-V0 sys*));
    ASSERT (*igba ba*);
    ASSERT (*finite* ((*g-E ba*)* `` *g-V0 ba*));
    SPEC (λ(*G*,*project*). *igb-graph G* ∧ *finite* ((*g-E G*)* `` *g-V0 G*) ∧ (∀ *r*.
      (∃ *r'*. *igb-graph.is-acc-run G r'* ∧ *r* = *project o r'*)
        ⟷ (*graph-defs.is-run sys r* ∧ *sa-L sys o r* ∈ *igba.lang ba*)))
  }

```
definition find-ce-spec
  :: ('q,-) igb-graph-rec-scheme ⇒ 'q word option option nres
  — Check Generalized Buchi graph for emptiness, with optional counterexample
  where find-ce-spec G ≡ do {
    ASSERT (igb-graph G);
    ASSERT (finite ((g-E G)* `` g-V0 G));
    SPEC (λres. case res of
      None ⇒ (∀ r. ¬igb-graph.is-acc-run G r)
    | Some None ⇒ (∃ r. igb-graph.is-acc-run G r)
    | Some (Some r) ⇒ igb-graph.is-acc-run G r
    )}
```

Using the specifications from above, we can specify the essence of the model-checking algorithm: Convert the LTL-formula to a GBA, make an intersection with the system and check the result for emptiness.

```
definition abs-model-check
  :: 'ba-state itself ⇒ 'ba-more itself
  ⇒ 'prod-state itself ⇒ 'prod-more itself
  ⇒ ('s,'prop set,-) sa-rec-scheme ⇒ 'prop ltlc
  ⇒ 's word option option nres
  where
  abs-model-check - - - - sys φ ≡ do {
    gba :: ('ba-state,-,'ba-more) igba-rec-scheme
      ← ltl-to-gba-spec (Not-ltlc φ);
    ASSERT (igba gba);
    ASSERT (sa sys);
    (Gprod::('prod-state,'prod-more)igb-graph-rec-scheme, map-state)
      ← inter-spec sys gba;
    ASSERT (igb-graph Gprod);
    ce ← find-ce-spec Gprod;

    case ce of
      None ⇒ RETURN None
    | Some None ⇒ RETURN (Some None)
    | Some (Some r) ⇒ RETURN (Some (Some (map-state o r)))
  }
```

The main correctness theorem states that our abstract model checker really checks whether the system satisfies the formula, and a correct counterexample is returned (if any). Note that, if the model does not satisfy the formula, returning a counterexample is optional.

```
theorem abs-model-check-correct:
  abs-model-check T1 T2 T3 T4 sys φ ≤ do {
    ASSERT (sa sys);
    ASSERT (finite ((g-E sys)* `` g-V0 sys));
    SPEC (λres. case res of
      None ⇒ sa.lang sys ⊆ language-ltlc φ
```

| *Some None* ⇒ ¬ *sa.lang sys* ⊆ *language-ltlc* φ
| *Some* (*Some r*) ⇒ *graph-defs.is-run sys r* ∧ *sa-L sys* ∘ *r* ∉ *language-ltlc* φ)
}
unfolding *abs-model-check-def ltl-to-gba-spec-def inter-spec-def find-ce-spec-def*
apply (*refine-rcg refine-vcg ASSERT-leI le-ASSERTI*)
apply (*auto simp*: *sa.lang-def sa.accept-def*[*THEN meta-eq-to-obj-eq*, *THEN ext*[*of sa.accept sys*]])

done

2.2 Generic Implementation

In this section, we define a generic implementation of an LTL model checker, that is parameterized with implementations of its components.

abbreviation *ltl-rel* ≡ *Id* :: (*'a ltlc* × -) *set*

locale *impl-model-checker* =
— Assembly of a generic model-checker
fixes *sa-rel* :: (*'sai* × (*'s*,*'prop set*,*'sa-more*) *sa-rec-scheme*) *set*
fixes *igba-rel* :: (*'igbai* × (*'q*, *'prop set*, *'igba-more*) *igba-rec-scheme*) *set*
fixes *igbg-rel* :: (*'igbgi* × (*'sq*, *'igbg-more*) *igb-graph-rec-scheme*) *set*
fixes *ce-rel* :: (*'cei* × *'sq word*) *set*
fixes *mce-rel* :: (*'mcei* × *'s word*) *set*

fixes *ltl-to-gba-impl* :: *'cfg-l2b* ⇒ *'prop ltlc* ⇒ *'igbai*
fixes *inter-impl* :: *'cfg-int* ⇒ *'sai* ⇒ *'igbai* ⇒ *'igbgi* × (*'sq* ⇒ *'s*)
fixes *find-ce-impl* :: *'cfg-ce* ⇒ *'igbgi* ⇒ *'cei option option*
fixes *map-run-impl* :: (*'sq* ⇒ *'s*) ⇒ *'cei* ⇒ *'mcei*

assumes [*relator-props*, *simp*, *intro*!]: *single-valued mce-rel*

assumes *ltl-to-gba-refine*:
⋀*cfg*. (*ltl-to-gba-impl cfg*,*ltl-to-gba-spec*)
∈ *ltl-rel* → ⟨*igba-rel*⟩*plain-nres-rel*
assumes *inter-refine*:
⋀*cfg*. (*inter-impl cfg*,*inter-spec*)
∈ *sa-rel* → *igba-rel* → ⟨*igbg-rel* $\times_r$ (*Id* → *Id*)⟩*plain-nres-rel*
assumes *find-ce-refine*:
⋀*cfg*. (*find-ce-impl cfg*,*find-ce-spec*)
∈ *igbg-rel* → ⟨⟨⟨*ce-rel*⟩*option-rel*⟩*option-rel*⟩*plain-nres-rel*

assumes *map-run-refine*: (*map-run-impl*,(∘)) ∈ (*Id* → *Id*) → *ce-rel* → *mce-rel*

begin

fun *cfg-l2b* **where** *cfg-l2b* (*c1*,*c2*,*c3*) = *c1*

```
fun cfg-int where cfg-int (c1,c2,c3) = c2
fun cfg-ce where cfg-ce (c1,c2,c3) = c3

definition impl-model-check
  :: ('cfg-l2b×'cfg-int×'cfg-ce)
    ⇒ 'sai ⇒ 'prop ltlc ⇒ 'mcei option option
  where
  impl-model-check cfg sys φ ≡ let
    ba = ltl-to-gba-impl (cfg-l2b cfg) (Not-ltlc φ);
    (G,map-q) = inter-impl (cfg-int cfg) sys ba;
    ce = find-ce-impl (cfg-ce cfg) G
  in
    case ce of
      None ⇒ None
    | Some None ⇒ Some None
    | Some (Some ce) ⇒ Some (Some (map-run-impl map-q ce))

lemma impl-model-check-refine:
  (impl-model-check cfg,abs-model-check
      TYPE('q) TYPE('igba-more) TYPE('sq) TYPE('igbg-more))
    ∈ sa-rel → ltl-rel → ⟨⟨⟨mce-rel⟩option-rel⟩option-rel⟩plain-nres-rel
  apply (intro fun-relI plain-nres-relI)
  unfolding abs-model-check-def impl-model-check-def

  apply (simp only: let-to-bind-conv pull-out-let-conv
    pull-out-RETURN-case-option)

  apply (refine-rcg
    ltl-to-gba-refine[param-fo, THEN plain-nres-relD]
    rel-arg-cong[where f=Not-ltlc]
    inter-refine[param-fo, THEN plain-nres-relD]
    find-ce-refine[param-fo, THEN plain-nres-relD]
  )

  apply (simp-all split: option.split)
  apply (auto elim: option-relE)
  apply (parametricity add: map-run-refine)
  apply simp
  done

theorem impl-model-check-correct:
  assumes R: (sysi,sys)∈sa-rel
  assumes [simp]: sa sys   finite ((g-E sys)* `` g-V0 sys)
  shows case impl-model-check cfg sysi φ of
    None
      ⇒ sa.lang sys ⊆ language-ltlc φ
  | Some None
      ⇒ ¬ sa.lang sys ⊆ language-ltlc φ
  | Some (Some ri)
```

```
      ⇒ (∃ r. (ri,r)∈mce-rel
        ∧ graph-defs.is-run sys r ∧ sa-L sys o r ∉ language-ltlc φ)
  proof −
    note impl-model-check-refine[
      where cfg=cfg,
      param-fo,
      THEN plain-nres-relD,
      OF R IdI[of φ]]
    also note abs-model-check-correct
    finally show ?thesis
      apply (simp split: option.split)
      apply (simp add: refine-pw-simps pw-le-iff)
      apply (auto elim!: option-relE) []
      done
  qed

  theorem impl-model-check-correct-no-ce:
    assumes (sysi,sys)∈sa-rel
    assumes SA: sa sys    finite ((g-E sys)* `` g-V0 sys)
    shows impl-model-check cfg sysi φ = None
    ⟷ sa.lang sys ⊆ language-ltlc φ
    using impl-model-check-correct[where cfg=cfg, OF assms, of φ]
    by (auto
      split: option.splits
      simp: sa.lang-def[OF SA(1)] sa.accept-def[OF SA(1), abs-def])

end


end
```

3 Boolean Programs

```
theory BoolProgs
imports
  CAVA-Base.CAVA-Base
  Word-Lib.Generic-set-bit
begin
```

3.1 Syntax and Semantics

```
datatype bexp = TT | FF | V nat | Not bexp | And bexp bexp | Or bexp bexp

type-synonym state = bitset

fun bval :: bexp ⇒ state ⇒ bool where
bval TT s = True |
bval FF s = False |
bval (V n) s = bs-mem n s |
```

```
bval (Not b) s = (¬ bval b s) |
bval (And b_1 b_2) s = (bval b_1 s & bval b_2 s) |
bval (Or b_1 b_2) s = (bval b_1 s | bval b_2 s)
```

```
datatype instr =
  AssI nat list    bexp list |
  TestI bexp int |
  ChoiceI (bexp ∗ int) list |
  GotoI int
```

```
type-synonym config = nat ∗ state
type-synonym bprog = instr array
```

Semantics Notice: To be equivalent in semantics with SPIN, there is no such thing as a finite run:

- Deadlocks (i.e. empty Choice) are self-loops
- program termination is self-loop

```
fun exec :: instr ⇒ config ⇒ config list where
exec instr (pc,s) = (case instr of
  AssI ns bs ⇒ let bvs = zip ns (map (λb. bval b s) bs) in
              [(pc + 1, foldl (λs (n,bv). set-bit s n bv) s bvs)] |
  TestI b d ⇒ [if bval b s then (pc+1, s) else (nat(int(pc+1)+d), s)] |
  ChoiceI bis ⇒ let succs = [(nat(int(pc+1)+i), s) . (b,i) <− bis, bval b s]
               in if succs = [] then [(pc,s)] else succs |
  GotoI d ⇒ [(nat(int(pc+1)+d),s)])
```

```
function exec' :: bprog ⇒ state ⇒ nat ⇒ nat list where
exec' ins s pc = (
  if pc < array-length ins then (
    case (array-get ins pc) of
      AssI ns bs ⇒ [pc] |
      TestI b d ⇒ (
        if bval b s then exec' ins s (pc+1)
        else let pc'=(nat(int(pc+1)+d)) in if pc'>pc then exec' ins s pc'
        else [pc']
      ) |
      ChoiceI bis ⇒ let succs = [(nat(int(pc+1)+i)) . (b,i) <− bis, bval b s]
                    in if succs = [] then [pc] else concat (map (λpc'. if pc'>pc then
exec' ins s pc' else [pc']) succs) |
      GotoI d ⇒ let pc' = nat(int(pc+1)+d) in (if pc'>pc then exec' ins s pc' else
[pc'])
  ) else [pc]
)
by pat-completeness auto
termination
  apply (relation measure (%(ins,s,pc). array-length ins − pc))
```

```
  apply auto
  done

fun nexts1 :: bprog ⇒ config ⇒ config list where
nexts1 ins (pc,s) = (
  if pc < array-length ins then
    exec (array-get ins pc) (pc,s)
  else
    [(pc,s)])

fun nexts :: bprog ⇒ config ⇒ config list where
  nexts ins (pc,s) = concat (
    map
      (λ(pc,s). map (λpc. (pc,s)) (exec′ ins s pc))
      (nexts1 ins (pc,s))
  )

declare nexts.simps [simp del]

datatype
  com = SKIP
      | Assign nat list    bexp list
      | Seq    com  com
      | GC  (bexp ∗ com)list
      | IfTE  bexp com com
      | While  bexp com

locale BoolProg-Syntax begin
  notation
      Assign       (‹- ::= -› [999, 61] 61)
    and Seq          (‹-;/ -›  [60, 61] 60)
    and GC           (‹IF - FI›)
    and IfTE         (‹(IF -/ THEN -/ ELSE -)›  [0, 61, 61] 61)
    and While        (‹(WHILE -/ DO -)›  [0, 61] 61)
end

context begin interpretation BoolProg-Syntax .
fun comp′ :: com ⇒ instr list where
comp′ SKIP = [] |
comp′ (Assign n b) = [AssI n b] |
comp′ (c1;c2) = comp′ c1 @ comp′ c2 |
comp′ (IF gcs FI) =
  (let cgcs = map (λ(b,c). (b,comp′ c)) gcs in
   let addbc = (λ(b,cc) (bis,ins).
         let cc′ = cc @ (if ins = [] then [] else [GotoI (int(length ins))]) in
         let bis′ = map (λ(b,i). (b, i + int(length cc′))) bis
         in ((b,0)#bis′, cc′ @ ins)) in
   let (bis,ins) = foldr addbc cgcs ([],[])
   in ChoiceI bis # ins) |
```

comp′ (*IF b THEN c1 ELSE c2*) =
  (*let ins1* = *comp′ c1 in let ins2* = *comp′ c2 in*
   let i1 = *int*(*length ins1* + *1*) *in let i2* = *int*(*length ins2*)
   in TestI b i1 # *ins1* @ *GotoI i2* # *ins2*) |
comp′ (*WHILE b DO c*) =
  (*let ins* = *comp′ c in*
   let i = *int*(*length ins* + *1*)
   in TestI b i # *ins* @ [*GotoI* (−(*i*+*1*))])

value *comp′* (*IF* [(*V 0*, [*1*,*0*] ::= [*TT*, *FF*]), (*V 1*, [*0*] ::= [*TT*])] *FI*)

end

definition *comp* :: *com* ⇒ *bprog* **where**
  comp = *array-of-list* ∘ *comp′*

fun *opt′* **where**
  opt′ (*GotoI d*) *ys* = (*let next* = *λi*. (*case i of GotoI d* ⇒ *d* + *1* | - ⇒ *0*)
                          in if d < *0* ∨ *nat d* ≥ *length ys then* (*GotoI d*)#*ys*
                             else let d′ = *d* + *next* (*ys* ! *nat d*)
                             in (*GotoI d′* # *ys*))
| *opt′ x ys* = *x*#*ys*

definition *opt* :: *instr list* ⇒ *instr list* **where**
  opt instr = *foldr opt′ instr* []

definition *optcomp* :: *com* ⇒ *bprog* **where**
  optcomp ≡ *array-of-list* ∘ *opt* ∘ *comp′*

3.2 Finiteness of reachable configurations

inductive-set *reachable-configs*
  for *bp* :: *bprog*
  and c_s :: *config* — start configuration
where
c_s ∈ *reachable-configs bp* c_s |
c ∈ *reachable-configs bp* c_s ⟹ *x* ∈ *set* (*nexts bp c*) ⟹ *x* ∈ *reachable-configs bp* c_s

lemmas *reachable-configs-induct* = *reachable-configs.induct*[*split-format*(*complete*),*case-names 0 1*]

fun *offsets* :: *instr* ⇒ *int set* **where**
offsets (*AssI* - -) = {*0*} |
offsets (*TestI* - *i*) = {*0*,*i*} |
offsets (*ChoiceI bis*) = *set*(*map snd bis*) ∪ {*0*} |

offsets (*GotoI i*) = {*i*}

definition *offsets-is* :: *instr list* ⇒ *int set* **where**
offsets-is ins = (*UN instr* : *set ins*. *offsets instr*)

definition *max-next-pcs* :: *instr list* ⇒ *nat set* **where**
max-next-pcs ins = {*nat*(*int*(*length ins* + *1*) + *i*) |*i*. *i* : *offsets-is ins*}

lemma *finite-max-next-pcs*: *finite*(*max-next-pcs bp*)
proof−
  { fix *instr* **have** *finite* (*offsets instr*) **by**(*cases instr*) *auto* **}**
  moreover
  { fix *ins* **have** *max-next-pcs ins* = (*UN i* : *offsets-is ins*. {*nat*(*int*(*length ins* + *1*) + *i*)})
    by(*auto simp add*: *max-next-pcs-def*) **}**
  ultimately show *?thesis* **by**(*auto simp add*: *offsets-is-def*)
qed

lemma (**in** *linorder*) *le-Max-insertI1*: ⟦ *finite A*; $x \leq b$ ⟧ ⟹ $x \leq$ *Max* (*insert b A*)
by (*metis Max-ge finite.insertI insert-iff order-trans*)
lemma (**in** *linorder*) *le-Max-insertI2*: ⟦ *finite A*; $A \neq \{\}$; $x \leq$ *Max A* ⟧ ⟹ $x \leq$ *Max* (*insert b A*)
by(*auto simp add*: *max-def not-le simp del*: *Max-less-iff*)

lemma *max-next-pcs-not-empty*:
  pc<*length bp* ⟹ *x* : *set* (*exec* (*bp*!*pc*) (*pc*,*s*)) ⟹ *max-next-pcs bp* ≠ {}
apply(*drule nth-mem*)
apply(*fastforce simp*: *max-next-pcs-def offsets-is-def split*: *instr.splits*)
done

lemma *Max-lem2*:
  assumes *pc* < *length bp*
  and (*pc'*, *s'*) ∈ *set* (*exec* (*bp*!*pc*) (*pc*, *s*))
  shows *pc'* ≤ *Max* (*max-next-pcs bp*)
using *assms*
proof (*cases bp* ! *pc*)
  case (*ChoiceI l*)
  show *?thesis*
  proof (*cases pc'* = *pc*)
    case *True* **with** *assms ChoiceI* **show** *?thesis*
      by (*auto simp*: *Max-ge-iff max-next-pcs-not-empty finite-max-next-pcs*)
        (*force simp add*: *max-next-pcs-def offsets-is-def dest*: *nth-mem*)
  next
    case *False* **with** *ChoiceI assms* **obtain** *b i* **where**
      bi: *bval b s*   (*b*,*i*) ∈ *set l*   *pc'* = *nat*(*int*(*pc*+*1*)+*i*)
      by (*auto split*: *if-split-asm*)

with *ChoiceI assms* **have** $i \in \bigcup(\mathit{offsets}\ `\ (\mathit{set}\ bp))$ **by** (*force dest*: *nth-mem*)
with *bi assms* **have** $\exists\, a.\ (a \in \mathit{max\text{-}next\text{-}pcs}\ bp \wedge pc' \le a)$
unfolding *max-next-pcs-def offsets-is-def* **by** *force*
thus *?thesis*
by (*auto simp*: *Max-ge-iff max-next-pcs-not-empty*[*OF assms*] *finite-max-next-pcs*)

qed
qed (*auto simp*: *Max-ge-iff max-next-pcs-not-empty finite-max-next-pcs*,
(*force simp add*: *max-next-pcs-def offsets-is-def dest*: *nth-mem split*: *if-split-asm*)+)

lemma *Max-lem1*: $[\![\ pc < \mathit{length}\ bp;\ (pc', s') \in \mathit{set}\ (\mathit{exec}\ (bp\ !\ pc)\ (pc, s))]\!]$
$\Longrightarrow pc' \le \mathit{Max}\ (\mathit{insert}\ x\ (\mathit{max\text{-}next\text{-}pcs}\ bp))$
apply(*rule le-Max-insertI2*)
apply (*simp add*: *finite-max-next-pcs*)
apply(*simp add*: *max-next-pcs-not-empty*)
apply(*auto intro*!: *Max-lem2 simp del*:*exec.simps*)
done

definition *pc-bound bp* ≡ *max*
(*Max* (*max-next-pcs* (*list-of-array bp*)) + *1*)
(*array-length bp* + *1*)

declare *exec′.simps*[*simp del*]

lemma [*simp*]: *length* (*list-of-array a*) = *array-length a* **by** (*cases a*) *auto*

lemma *aux2*:
assumes *A*: *pc* < *array-length ins*
assumes *B*: *ofs* ∈ *offsets-is* (*list-of-array ins*)
shows *nat* (*1* + *int pc* + *ofs*) < *pc-bound ins*
proof −
have *nat* (*int* (*1* + *array-length ins*) + *ofs*)
∈ *max-next-pcs* (*list-of-array ins*)
using *B* **unfolding** *max-next-pcs-def*
by *auto*
with *A* **show** *?thesis*
unfolding *pc-bound-def*
apply −
apply (*rule max.strict-coboundedI1*)
apply *auto*
apply (*drule Max-ge*[*OF finite-max-next-pcs*])
apply *simp*
done
qed

lemma *array-idx-in-set*:
$[\![\ pc < \mathit{array\text{-}length}\ ins;\ \mathit{array\text{-}get}\ ins\ pc = x\ ]\!]$
$\Longrightarrow x \in \mathit{set}\ (\mathit{list\text{-}of\text{-}array}\ ins)$
by (*induct ins*) *auto*

lemma *rcs-aux*:
  assumes *pc* < *pc-bound bp*
  assumes *pc*′∈*set* (*exec*′ *bp s pc*)
  shows *pc*′ < *pc-bound bp*
  using *assms*
proof (*induction bp s pc arbitrary*: *pc*′ *rule*: *exec*′.*induct*[*case-names C*])
  case (*C ins s pc pc*′)
  from *C*.*prems* **show** *?case*
    apply (*subst* (*asm*) *exec*′.*simps*)
    apply (*split if-split-asm instr*.*split-asm*)+
    apply (*simp add*: *pc-bound-def*)

    apply (*simp split*: *if-split-asm add*: *Let-def*)
    apply (*frule* (*2*) *C*.*IH*(*1*), *auto*) []
    apply (*auto simp*: *pc-bound-def*) []
    apply (*frule* (*2*) *C*.*IH*(*2*), *auto*) []
    apply (*rename-tac bexp int*′)
    apply (*subgoal-tac int*′ ∈ *offsets-is* (*list-of-array ins*))
    apply (*blast intro*: *aux2*)
    apply (*auto simp*: *offsets-is-def*) []
    apply (*rule-tac x*=*TestI bexp int*′ **in** *bexI*, *auto simp*: *array-idx-in-set*) []

    apply (*rename-tac list*)
    apply (*auto split*: *if-split-asm*)[]
    apply (*frule* (*1*) *C*.*IH*(*3*), *auto*) []
    apply (*force*)
    apply (*force*)
    apply (*subgoal-tac ba* ∈ *offsets-is* (*list-of-array ins*))
    apply (*blast intro*: *aux2*)
    apply (*auto simp*: *offsets-is-def*) []
    apply (*rule-tac x*=*ChoiceI list* **in** *bexI*, *auto simp*: *array-idx-in-set*) []

    apply (*rename-tac int*′)
    apply (*simp split*: *if-split-asm add*: *Let-def*)
    apply (*frule* (*1*) *C*.*IH*(*4*), *auto*) []
    apply (*subgoal-tac int*′ ∈ *offsets-is* (*list-of-array ins*))
    apply (*blast intro*: *aux2*)
    apply (*auto simp*: *offsets-is-def*) []
    apply (*rule-tac x*=*GotoI int*′ **in** *bexI*, *auto simp*: *array-idx-in-set*) []

    apply *simp*
    done
qed

primrec *bexp-vars* :: *bexp* ⇒ *nat set* **where**
  bexp-vars TT = {}
| *bexp-vars FF* = {}

| *bexp-vars* (*V n*) = {*n*}
| *bexp-vars* (*Not b*) = *bexp-vars b*
| *bexp-vars* (*And b1 b2*) = *bexp-vars b1* ∪ *bexp-vars b2*
| *bexp-vars* (*Or b1 b2*) = *bexp-vars b1* ∪ *bexp-vars b2*

primrec *instr-vars* :: *instr* ⇒ *nat set* **where**
  instr-vars (*AssI xs bs*) = *set xs* ∪ ⋃(*bexp-vars'set bs*)
| *instr-vars* (*TestI b* -) = *bexp-vars b*
| *instr-vars* (*ChoiceI cs*) = ⋃(*bexp-vars'fst'set cs*)
| *instr-vars* (*GotoI* -) = {}

find-consts *'a array* ⇒ *'a list*

definition *bprog-vars* :: *bprog* ⇒ *nat set* **where**
  bprog-vars bp = ⋃(*instr-vars'set* (*list-of-array bp*))

definition *state-bound bp s0*
  ≡ {*s*. *bs-α s* − *bprog-vars bp* = *bs-α s0* − *bprog-vars bp*}
abbreviation *config-bound bp s0* ≡ {*0*..< *pc-bound bp*} × *state-bound bp s0*

lemma *exec-bound*:
  assumes *PCB*: *pc* < *array-length bp*
  assumes *SB*: *s* ∈ *state-bound bp s0*
  shows *set* (*exec* (*array-get bp pc*) (*pc*,*s*)) ⊆ *config-bound bp s0*
proof (*clarsimp simp del*: *exec.simps*, *intro conjI*)

  obtain *instrs* **where** *BP-eq*[*simp*]: *bp* = *Array instrs* **by** (*cases bp*)
  from *PCB* **have** *PCB'*[*simp*]: *pc* < *length instrs* **by** *simp*

  fix *pc' s'*
  assume *STEP*: (*pc'*,*s'*) ∈ *set* (*exec* (*array-get bp pc*) (*pc*,*s*))
  hence *STEP'*: (*pc'*,*s'*) ∈ *set* (*exec* (*instrs*!*pc*) (*pc*,*s*)) **by** *simp*

  show *pc'* < *pc-bound bp*
    using *Max-lem2*[*OF PCB' STEP'*]
    unfolding *pc-bound-def* **by** *simp*

  show *s'* ∈ *state-bound bp s0*
    using *STEP' SB*
  proof (*cases instrs*!*pc*)
    case (*AssI xs vs*)

    have *set xs* ⊆ *instr-vars* (*instrs*!*pc*)
      by (*simp add*: *AssI*)
    also have ... ⊆ *bprog-vars bp*
      apply (*simp add*: *bprog-vars-def*)
      by (*metis PCB' UN-upper nth-mem*)
    finally have *XSB*: *set xs* ⊆ *bprog-vars bp* **.**

```
  {
    fix x s v
    assume A: x ∈ bprog-vars bp    s ∈ state-bound bp s0

    have SB-CNV: bs-α (set-bit s x v)
      = (if v then (insert x (bs-α s)) else (bs-α s − {x}))
      by (cases v) (simp-all add: set-bit-eq flip: bs-insert-def bs-delete-def)

    from A have set-bit s x v ∈ state-bound bp s0
      unfolding state-bound-def
      by (auto simp: SB-CNV)
  } note aux=this

  {
    fix vs
    have foldl (λs (x, y). set-bit s x y) s (zip xs vs)
      ∈ state-bound (Array instrs) s0
      using SB XSB
      apply (induct xs arbitrary: vs s)
      apply simp
      apply (case-tac vs)
      apply simp
      using aux
      apply (auto)
      done
  } note aux2=this

  thus ?thesis using STEP'
    by (simp add: AssI)
 qed (auto split: if-split-asm)
qed

lemma in-bound-step:
 notes [simp del] = exec.simps
 assumes BOUND: c ∈ config-bound bp s0
 assumes STEP: c'∈set (nexts bp c)
 shows c' ∈ config-bound bp s0
 using BOUND STEP
 apply (cases c)
 apply (auto
   simp add: nexts.simps
   split: if-split-asm)
 apply (frule (2) exec-bound[THEN subsetD])
 apply clarsimp
 apply (frule (1) rcs-aux)
 apply simp
```

```
  apply (frule (2) exec-bound[THEN subsetD])
  apply clarsimp

  apply (frule (1) rcs-aux)
  apply simp
  done

lemma reachable-configs-in-bound:
  c ∈ config-bound bp s0 ⟹ reachable-configs bp c ⊆ config-bound bp s0
proof
  fix c'
  assume c' ∈ reachable-configs bp c    c ∈ config-bound bp s0
  thus c' ∈ config-bound bp s0
    apply induction
    apply simp
    by (rule in-bound-step)
qed

lemma reachable-configs-out-of-bound: (pc',s')∈reachable-configs bp (pc,s)
  ⟹ ¬ pc < pc-bound bp ⟹ (pc',s') = (pc,s)
proof (induct rule: reachable-configs-induct)
  case (1 pc' s' pc'' s'')
  hence [simp]: pc'=pc    s'=s by auto
  from 1(4) have ¬ pc < array-length bp unfolding pc-bound-def by auto
  with 1(3) show ?case
    by (auto simp add: nexts.simps exec'.simps)
qed auto

lemma finite-bexp-vars[simp, intro!]: finite (bexp-vars be)
  by (induction be) auto

lemma finite-instr-vars[simp, intro!]: finite (instr-vars ins)
  by (cases ins) auto

lemma finite-bprog-vars[simp, intro!]: finite (bprog-vars bp)
  unfolding bprog-vars-def by simp

lemma finite-state-bound[simp, intro!]: finite (state-bound bp s0)
  unfolding state-bound-def
  apply (rule finite-imageD[where f=bs-α])
  apply (rule finite-subset[where
    B = {s. s − bprog-vars bp = bs-α s0 − bprog-vars bp}])
  apply auto []
  apply (rule finite-if-eq-beyond-finite)
  apply simp

  apply (rule inj-onI)
  apply (fold bs-eq-def)
  apply (auto simp: bs-eq-correct)
```

done

lemma *finite-config-bound*[*simp*, *intro*!]: *finite* (*config-bound bp s0*)
by *blast*

lemma *reachable-configs-finite*[*simp*, *intro*!]:
finite (*reachable-configs bp c*)
proof (*cases c*, *clarsimp*)
fix *pc s*
show *finite* (*reachable-configs bp* (*pc*, *s*))
proof (*cases pc* < *pc-bound bp*)
case *False* **from** *reachable-configs-out-of-bound*[*OF* - *False*, **where** *s*=*s*]
have *reachable-configs bp* (*pc*, *s*) ⊆ {(*pc*,*s*)} **by** *auto*
thus *?thesis* **by** (*rule finite-subset*) *auto*
next
case *True*
hence (*pc*,*s*) ∈ *config-bound bp s*
by (*simp add*: *state-bound-def*)
thus *?thesis*
by (*rule finite-subset*[*OF reachable-configs-in-bound*]) *simp*
qed
qed

definition *bpc-is-run bpc r* ≡ *let* (*bp*,*c*)=*bpc in r 0* = *c* ∧ (∀ *i*. *r* (*Suc i*) ∈ *set* (*BoolProgs.nexts bp* (*r i*)))
definition *bpc-props c* ≡ *bs-α* (*snd c*)
definition *bpc-lang bpc* ≡ {*bpc-props o r* | *r*. *bpc-is-run bpc r*}

fun *print-config* ::
(*nat* ⇒ *string*) ⇒ (*bitset* ⇒ *string*) ⇒ *config* ⇒ *string* **where**
print-config f fx (*p*,*s*) = *f p* @ '' '' @ *fx s*

end

References

[1] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J.-G. Smaus. A fully verified executable ltl model checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer Berlin Heidelberg, 2013.

[2] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. of SPIN Workshop*, volume 32 of *Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1997.

[3] P. Lammich. Verified efficient implementation of Gabow's strongly connected component algorithm. In *Proc. of ITP*, 2014. to appear.

[4] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *LNCS*, pages 174–190. Springer, 2005.