

A Fully Verified Executable LTL Model Checker

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

May 10, 2026

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	3
1.1	Tools for DFS Algorithms	3
1.1.1	Invariants	3
1.1.2	Invariant Preservation	4
1.1.3	Consequences of Postcondition	6
1.2	Abstract Algorithm	7
1.2.1	Inner (red) DFS	7
1.2.2	Outer (Blue) DFS	9
1.3	Correctness	10
1.4	Refinement	11
1.4.1	Setup for Custom Datatypes	11
1.4.2	Actual Refinement	12
2	Abstract Model-Checker	17
2.1	Specification of an LTL Model-Checker	17
2.2	Generic Implementation	19
3	Boolean Programs	21
3.1	Syntax and Semantics	21
3.2	Finiteness of reachable configurations	24

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* $E U S V u0 \equiv$

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

definition *gen-dfs-var* $U \equiv$ *finite-psupset* U

definition *gen-dfs-fe-inv* $E U S V0 u0 it V brk \equiv$

- $(\neg brk \longrightarrow E \text{''} (V - S) \subseteq V)$ — Visited set closed under reachability
- $\wedge E \text{''} \{u0\} - it \subseteq V$ — Successors of $u0$ visited
- $\wedge V0 \subseteq V$ — Visited set increasing
- $\wedge V \subseteq V0 \cup (E - UNIV \times S)^* \text{''} (E \text{''} \{u0\} - it - S)$ — All visited nodes reachable

definition *gen-dfs-post* $E U S V0 u0 V brk \equiv$
 $(\neg brk \longrightarrow E^*(V-S) \subseteq V)$ — Visited set closed under reachability
 $\wedge u0 \in V$ — $u0$ visited
 $\wedge V0 \subseteq V$ — Visited set increasing
 $\wedge V \subseteq V0 \cup (E - UNIV \times S)^* \{u0\}$ — All visited nodes reachable

definition *gen-dfs-outer* $E U V0 it V brk \equiv$
 $V0 \subseteq U$ — Start nodes below upper bound
 $\wedge E^*U \subseteq U$ — Upper bound is closed under transitions
 $\wedge finite U$ — Upper bound is finite
 $\wedge V \subseteq U$ — Visited set below upper bound
 $\wedge (\neg brk \longrightarrow E^*V \subseteq V)$ — Visited set closed under reachability
 $\wedge (V0 - it \subseteq V)$ — 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*
 $\langle proof \rangle$

lemma *gen-dfs-pre-initial*:
assumes *gen-dfs-outer* $E U V0 it V False$
assumes $v0 \in U - V$
shows *gen-dfs-pre* $E U \{\}$ $V v0$
 $\langle proof \rangle$

lemma *fin-U-imp-wf*:
assumes *finite* U
shows *wf* $(gen-dfs-var U)$
 $\langle proof \rangle$

lemma *gen-dfs-pre-imp-wf*:
assumes *gen-dfs-pre* $E U S V u0$
shows *wf* $(gen-dfs-var U)$
 $\langle proof \rangle$

lemma *gen-dfs-pre-imp-fin*:
assumes *gen-dfs-pre* $E U S V u0$
shows *finite* $(E \{u0\})$
 $\langle proof \rangle$

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* $E U (insert u0 S) (insert u0 V) u0$

$(E^{\{u0\}}) (insert\ u0\ V)\ False$
 $\langle proof \rangle$

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* $E\ U\ (insert\ u0\ S)\ (insert\ u0\ V)\ u0\ (it - \{t})\ V'\ False$
 $\langle proof \rangle$

lemma *gen-dfs-upper-aux:*

assumes $(x,y) \in E'^*$
assumes $(u0,x) \in E$
assumes $u0 \in U$
assumes $E' \subseteq E$
assumes $E^{\{u0\}} \subseteq U$
shows $y \in U$
 $\langle proof \rangle$

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 $(V', V) \in gen-dfs-var\ U$
 $\langle proof \rangle$

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* $E\ U\ (insert\ u0\ S)\ V'\ t$
 $\langle proof \rangle$

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 $t \in it\ it \subseteq E^{\{u0\}}\ t \notin V'$
assumes *gen-dfs-post* $E\ U\ (insert\ u0\ S)\ V'\ t\ V''\ cyc$
shows *gen-dfs-fe-inv* $E\ U\ (insert\ u0\ S)\ (insert\ u0\ V)\ u0\ (it - \{t})\ V''\ cyc$
 $\langle proof \rangle$

lemma *gen-dfs-post-aux:*

assumes 1: $(u0,x) \in E$
assumes 2: $(x,y) \in (E - UNIV \times insert\ u0\ S)^*$
assumes 3: $S \subseteq V\ x \notin V$
shows $(u0, y) \in (E - UNIV \times S)^*$
 $\langle proof \rangle$

lemma *gen-dfs-fe-imp-post-brk*:
assumes *gen-dfs-pre* $E U S V u0$
assumes *gen-dfs-fe-inv* $E U (\text{insert } u0 S) (\text{insert } u0 V) u0 \text{ it } V' \text{ True}$
assumes $it \subseteq E^* \{u0\}$
shows *gen-dfs-post* $E U S V u0 V' \text{ True}$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-fe-inv-imp-post*:
assumes *gen-dfs-pre* $E U S V u0$
assumes *gen-dfs-fe-inv* $E U (\text{insert } u0 S) (\text{insert } u0 V) u0 \{ \} V' \text{ cyc}$
assumes $\text{cyc} \longrightarrow \text{cyc}'$
shows *gen-dfs-post* $E U S V u0 V' \text{ cyc}'$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-pre-imp-post-brk*:
assumes *gen-dfs-pre* $E U S V u0$
shows *gen-dfs-post* $E U S V u0 (\text{insert } u0 V) \text{ True}$
 $\langle \text{proof} \rangle$

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* $E U S V0 u0 V \text{ brk}$
shows $V \subseteq V0 \cup E^* \{u0\}$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-post-imp-complete*:
assumes *gen-dfs-pre* $E U \{ \} V0 u0$
assumes *gen-dfs-post* $E U \{ \} V0 u0 V \text{ False}$
shows $V0 \cup E^* \{u0\} \subseteq V$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-post-imp-eq*:
assumes *gen-dfs-pre* $E U \{ \} V0 u0$
assumes *gen-dfs-post* $E U \{ \} V0 u0 V \text{ False}$
shows $V = V0 \cup E^* \{u0\}$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-post-imp-below-U*:
assumes *gen-dfs-pre* $E U S V0 u0$
assumes *gen-dfs-post* $E U S V0 u0 V \text{ False}$
shows $V \subseteq U$
 $\langle \text{proof} \rangle$

lemma *gen-dfs-post-imp-outer*:
assumes *gen-dfs-outer* $E U V0 \text{ it } \text{Vis0} \text{ False}$
assumes *gen-dfs-post* $E U \{ \} \text{Vis0 } v0 \text{ Vis} \text{ False}$

assumes $v0 \in it \quad it \subseteq V0 \quad v0 \notin Vis0$
shows $gen\text{-}dfs\text{-}outer \ E \ U \ V0 \ (it - \{v0\}) \ Vis \ False$
<proof>

lemma *gen-dfs-outer-already-vis*:
assumes $v0 \in it \quad it \subseteq V0 \quad v0 \in V$
assumes $gen\text{-}dfs\text{-}outer \ E \ U \ V0 \ it \ V \ False$
shows $gen\text{-}dfs\text{-}outer \ E \ U \ V0 \ (it - \{v0\}) \ V \ False$
<proof>

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 $'v \ red\text{-}witness = ('v \ list \times 'v) \ option$

Prepend node to red witness

fun *prep-wit-red* :: $'v \Rightarrow 'v \ red\text{-}witness \Rightarrow 'v \ red\text{-}witness$ **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* :: $'v \Rightarrow 'v \Rightarrow 'v \ red\text{-}witness$ **where**
red-init-witness $u \ v = Some \ ([u],v)$

definition *red-dfs* **where**

red-dfs $E \ onstack \ V \ u \equiv$
 $REC_T \ (\lambda D \ (V,u). \ do \ \{$
 $\quad let \ V = insert \ u \ V;$
 $\quad \mathit{NDFS-SI-Statistics.vis-red-nres};$

— Check whether we have a successor on stack
 $brk \leftarrow FOREACH_C \ (E''\{u\}) \ (\lambda brk. \ brk = None)$
 $(\lambda t \ .$
 $\quad if \ t \in onstack \ then$
 $\quad \quad RETURN \ (red\text{-}init\text{-}witness \ u \ t)$
 $\quad else$
 $\quad \quad RETURN \ None$
 $\quad)$
 $None;$

— Recurse for successors

case brk of
 $None \Rightarrow$
 $FOREACH_C \ (E''\{u\}) \ (\lambda (V,brk). \ brk = None)$
 $(\lambda t \ (V,-).$

```

      if  $t \notin V$  then do {
         $(V, brk) \leftarrow D(V, t)$ ;
        RETURN  $(V, prep-wit-red\ u\ brk)$ 
      } else RETURN  $(V, None)$ 
    )  $(V, None)$ 
  | -  $\Rightarrow$  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* — CIRC *pr pl*: Lasso found from current start node.

Prepend node to witness

primrec *prep-wit-blue* :: *'v \Rightarrow 'v blue-witness \Rightarrow '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 \Rightarrow 'v red-witness \Rightarrow '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 \Rightarrow 'v \Rightarrow 'v blue-witness*
where *init-wit-blue-early s t \equiv if $s = t$ then CIRC [] [s] else REACH t [s]*

Extract result from witness

term *lasso-ext*

definition *extract-res cyc*
 \equiv (case *cyc* of
 CIRC *pr pl* \Rightarrow Some (pr, pl)
 | - \Rightarrow None)

1.2.2 Outer (Blue) DFS

definition *blue-dfs*

$:: ('a,-) b\text{-graph-rec-scheme} \Rightarrow ('a \text{ list} \times 'a \text{ list}) \text{ option nres}$

where

```

blue-dfs G  $\equiv$  do {
  NDFS-SI-Statistics.start-nres;
   $(-, -, cyc) \leftarrow \text{FOREACH}_c (g\text{-}V0\ G) (\lambda(-, -, cyc). cyc = \text{NO-CYC})$ 
   $(\lambda v0 (blues, reds, -). \text{do} \{$ 
    if  $v0 \notin blues$  then do {
       $(blues, reds, -, cyc) \leftarrow \text{RECT} (\lambda D (blues, reds, onstack, s). \text{do} \{$ 
        let  $blues = \text{insert } s \text{ blues}$ ;
        let  $onstack = \text{insert } s \text{ onstack}$ ;
        let  $s\text{-acc} = s \in \text{bg-F } G$ ;
        NDFS-SI-Statistics.vis-blue-nres;
         $(blues, reds, onstack, cyc) \leftarrow$ 
        FOREACHC  $((g\text{-}E\ G) \text{ ``}\{s\}\text{ }) (\lambda(-, -, -, cyc). cyc = \text{NO-CYC})$ 
         $(\lambda t (blues, reds, onstack, cyc).$ 
          if  $t \in onstack \wedge (s\text{-acc} \vee t \in \text{bg-F } G)$  then (
            RETURN  $(blues, reds, onstack, \text{init-wit-blue-early } s\ t)$ 
          ) else if  $t \notin blues$  then do {
             $(blues, reds, onstack, cyc) \leftarrow D (blues, reds, onstack, t)$ ;
            RETURN  $(blues, reds, onstack, (\text{prep-wit-blue } s\ cyc))$ 
          } else do {
            NDFS-SI-Statistics.match-blue-nres;
            RETURN  $(blues, reds, onstack, cyc)$ 
          }
        )
      }
       $(blues, reds, onstack, \text{NO-CYC})$ ;

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

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

```

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

is *blue-dfs* *G* \equiv 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 - \equiv *FOREACHc* (*g-V0 G*) ($\lambda(-,-,cyc). cyc=NO-CYC$)
 ($\lambda v0 (blues,reds,-). do$ {
 if $v0 \notin blues$ then do {
 $(blues,reds,-,cyc) \leftarrow RECT ?B (blues,reds,\{ \},v0)$;
 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* $E U reds onstack \equiv$
 $E^*U \subseteq U$ — Upper bound is closed under transitions
 \wedge *finite* U — Upper bound is finite
 $\wedge reds \subseteq U$ — Red set below upper bound
 $\wedge E^*reds \subseteq reds$ — Red nodes closed under reachability
 $\wedge E^*reds \cap onstack = \{ \}$ — No red node with edge to stack

lemma *red-dfs-inv-initial*:
assumes *finite* (E^*V0)
shows *red-dfs-inv* $E (E^*V0) \{ \} \{ \}$
 <proof>

Correctness of the red DFS.

theorem *red-dfs-correct*:

fixes $v0 u0 :: 'v$
assumes *PRE*:
 red-dfs-inv $E U reds onstack$
 $u0 \in U$
 $u0 \notin reds$
shows *red-dfs* $E onstack reds u0$
 $\leq SPEC (\lambda(reds',cyc). case cyc of$
 Some (p,v) $\Rightarrow v \in onstack \wedge p \neq [] \wedge path E u0 p v$
 | *None* \Rightarrow
 red-dfs-inv $E U reds' onstack$
 $\wedge u0 \in reds'$
 $\wedge reds' \subseteq reds \cup E^* \{ u0 \}$
)
 <proof>

Main theorem: Correctness of the blue DFS

theorem *blue-dfs-correct*:

fixes $G :: ('v, -) \text{ b-graph-rec-scheme}$

assumes *b-graph* G

assumes *finitely-reachable*: *finite* $((g-E \ G)^* \text{ “ } g-V0 \ G)$

shows *blue-dfs* $G \leq \text{SPEC } (\lambda r.$

case r *of* *None* $\Rightarrow (\forall L. \neg \text{b-graph.is-lasso-prpl } G \ L)$

 | *Some* $L \Rightarrow \text{b-graph.is-lasso-prpl } G \ L)$

$\langle \text{proof} \rangle$

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* $R \equiv \langle \langle \langle R \rangle \text{list-rel}, R \rangle \text{prod-rel} \rangle \text{option-rel}$

abbreviation *i-red-wit* $I \equiv \langle \langle \langle I \rangle_i \text{i-list}, I \rangle_i \text{i-prod} \rangle_i \text{i-option}$

abbreviation *blue-wit-rel* $\equiv (Id :: (- \text{blue-witness} \times -) \text{set})$

consts *i-blue-wit* $:: \text{interface}$

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

term *init-wit-blue-early*

lemma [*autoref-itype*]:

NO-CYC $::_i \text{i-blue-wit}$

$(=) ::_i \text{i-blue-wit} \rightarrow_i \text{i-blue-wit} \rightarrow_i \text{i-bool}$

init-wit-blue $::_i I \rightarrow_i \text{i-red-wit } I \rightarrow_i \text{i-blue-wit}$

init-wit-blue-early $::_i I \rightarrow_i I \rightarrow_i \text{i-blue-wit}$

prep-wit-blue $::_i I \rightarrow_i \text{i-blue-wit} \rightarrow_i \text{i-blue-wit}$

red-init-witness $::_i I \rightarrow_i I \rightarrow_i \text{i-red-wit } I$

prep-wit-red $::_i I \rightarrow_i \text{i-red-wit } I \rightarrow_i \text{i-red-wit } I$

extract-res $::_i \text{i-blue-wit} \rightarrow_i \langle \langle I \rangle_i \text{i-list}, \langle I \rangle_i \text{i-list} \rangle_i \text{i-prod} \rangle_i \text{i-option}$

red-dfs $::_i \langle I \rangle_i \text{i-slg} \rightarrow_i \langle I \rangle_i \text{i-set} \rightarrow_i \langle I \rangle_i \text{i-set} \rightarrow_i I$

$\rightarrow_i \langle \langle \langle I \rangle_i \text{i-set}, \text{i-red-wit } I \rangle_i \text{i-prod} \rangle_i \text{i-nres}$

blue-dfs $::_i \text{i-bg } \text{i-unit } I$

$\rightarrow_i \langle \langle \langle \langle I \rangle_i \text{i-list}, \langle I \rangle_i \text{i-list} \rangle_i \text{i-prod} \rangle_i \text{i-option} \rangle_i \text{i-nres}$

$\langle \text{proof} \rangle$

context **begin** *interpretation* *autoref-syn* $\langle \text{proof} \rangle$

lemma [*autoref-op-pat*]: *NO-CYC* $\equiv \text{OP } \text{NO-CYC} ::_i \text{i-blue-wit} \langle \text{proof} \rangle$

end

term *lasso-rel-ext*

lemma *autoref-wit*[*autoref-rules-raw*]:

$(NO-CYC, NO-CYC) \in \text{blue-wit-rel}$
 $((=), (=)) \in \text{blue-wit-rel} \rightarrow \text{blue-wit-rel} \rightarrow \text{bool-rel}$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{init-wit-blue}, \text{init-wit-blue}) \in R \rightarrow \text{red-wit-rel } R \rightarrow \text{blue-wit-rel}$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{init-wit-blue-early}, \text{init-wit-blue-early}) \in R \rightarrow R \rightarrow \text{blue-wit-rel}$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{prep-wit-blue}, \text{prep-wit-blue}) \in R \rightarrow \text{blue-wit-rel} \rightarrow \text{blue-wit-rel}$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{red-init-witness}, \text{red-init-witness}) \in R \rightarrow R \rightarrow \text{red-wit-rel } R$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{prep-wit-red}, \text{prep-wit-red}) \in R \rightarrow \text{red-wit-rel } R \rightarrow \text{red-wit-rel } R$
 $\bigwedge R. \text{PREFER-id } R$
 $\implies (\text{extract-res}, \text{extract-res})$
 $\in \text{blue-wit-rel} \rightarrow \langle \langle R \rangle \text{list-rel} \times_r \langle R \rangle \text{list-rel} \rangle \text{option-rel}$
 $\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

concrete-definition *impl-red-dfs uses red-dfs-refine-aux*

lemma *impl-red-dfs-autoref[autoref-rules]:*

fixes $R :: ('a \times 'a :: \text{linorder}) \text{ set}$

assumes *PREFER-id R*

shows $(\text{impl-red-dfs}, \text{red-dfs}) \in$

$\langle R \rangle \text{slg-rel} \rightarrow \langle R \rangle \text{dflt-rs-rel} \rightarrow \langle R \rangle \text{dflt-rs-rel} \rightarrow R$

$\rightarrow \langle \langle R \rangle \text{dflt-rs-rel} \times_r \text{red-wit-rel } R \rangle \text{nres-rel}$

$\langle \text{proof} \rangle$

thm *autoref-itype(1–10)*

schematic-goal *code-red-dfs-aux:*

shows $\text{RETURN } ?c \leq \text{impl-red-dfs } E \text{ onstack } V u$

$\langle \text{proof} \rangle$

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*: ($?f::?'c$, $red-dfs::('a::hashable \times -) set \Rightarrow -$)
 $\in ?R$

$\langle proof \rangle$

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 $R :: ('a \times 'a::hashable) set$

assumes *PREFER-id* R

shows $(impl-red-dfs-hash, red-dfs) \in$
 $\langle R \rangle slg-rel \rightarrow \langle R \rangle hs.rel \rightarrow \langle R \rangle hs.rel \rightarrow R$
 $\rightarrow \langle \langle R \rangle hs.rel \times_r red-wit-rel R \rangle nres-rel$
 $\langle proof \rangle$

schematic-goal *code-red-dfs-hash-aux*:

shows $RETURN ?c \leq impl-red-dfs-hash E onstack V u$
 $\langle proof \rangle$

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*: ($?f::?'c$, $red-dfs::('a::hashable \times -) set \Rightarrow -$)
 $\in ?R$

$\langle proof \rangle$

concrete-definition *impl-red-dfs-ahs* **uses** *red-dfs-ahs-refine-aux*

lemma *impl-red-dfs-ahs-autoref*[*autoref-rules*]:

fixes $R :: ('a \times 'a::hashable) set$

assumes *PREFER-id* R

shows $(impl-red-dfs-ahs, red-dfs) \in$
 $\langle R \rangle slg-rel \rightarrow \langle R \rangle ahs.rel \rightarrow \langle R \rangle ahs.rel \rightarrow R$
 $\rightarrow \langle \langle R \rangle ahs.rel \times_r red-wit-rel R \rangle nres-rel$
 $\langle proof \rangle$

schematic-goal *code-red-dfs-ahs-aux*:

shows $RETURN ?c \leq impl-red-dfs-ahs E onstack V u$
 $\langle proof \rangle$

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*: ($?f::?'c$, $blue-dfs::('a::linorder b-graph-rec \Rightarrow -)$)

```

∈ ?R
  ⟨proof⟩
concrete-definition impl-blue-dfs uses blue-dfs-refine-aux

thm impl-blue-dfs.refine

lemma impl-blue-dfs-autoref[autoref-rules]:
  fixes  $R :: ('a \times 'a::linorder)$  set
  assumes PREFER-id R
  shows (impl-blue-dfs, blue-dfs)
  ∈ bg-impl-rel-ext unit-rel R
  → ⟨⟨⟨ $R$ ⟩list-rel ×r ⟨ $R$ ⟩list-rel⟩Relators.option-rel⟩nres-rel
  ⟨proof⟩

schematic-goal code-blue-dfs-aux:
  shows RETURN ?c ≤ impl-blue-dfs G
  ⟨proof⟩
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
  ⟨proof⟩
concrete-definition impl-blue-dfs-hash uses blue-dfs-hash-refine-aux

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

schematic-goal code-blue-dfs-hash-aux:
  shows RETURN ?c ≤ impl-blue-dfs-hash G
  ⟨proof⟩
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
  ⟨proof⟩
concrete-definition impl-blue-dfs-ahs uses blue-dfs-ahs-refine-aux

```

lemma *impl-blue-dfs-ahs-autoref*[*autoref-rules*]:
fixes $R :: ('a \times 'a::hashable) \text{ set}$
assumes *MINOR-PRIO-TAG 5*
assumes *PREFER-id R*
shows $(\text{impl-blue-dfs-ahs}, \text{blue-dfs}) \in \text{bg-impl-rel-ext unit-rel } R$
 $\rightarrow \langle \langle R \rangle \text{list-rel} \times_r \langle R \rangle \text{list-rel} \rangle \text{Relators.option-rel} \rangle \text{nres-rel}$
 $\langle \text{proof} \rangle$

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

schematic-goal *code-blue-dfs-ahs-aux*:
shows $\text{RETURN } ?c \leq \text{impl-blue-dfs-ahs } G$
 $\langle \text{proof} \rangle$
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 $G: \text{b-graph } G \quad \text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G)$
assumes $REL: (Gi, G) \in \text{bg-impl-rel-ext unit-rel } Id$
shows $\text{RETURN } (\text{code-blue-dfs } Gi) \leq \text{SPEC } (\lambda r.$
 $\text{case } r \text{ of } None \Rightarrow \forall \text{prpl. } \neg \text{b-graph.is-lasso-prpl } G \ \text{prpl}$
 $| \text{Some } L \Rightarrow \text{b-graph.is-lasso-prpl } G \ L)$
 $\langle \text{proof} \rangle$

theorem *code-blue-dfs-correct'*:
assumes $G: \text{b-graph } G \quad \text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G)$
assumes $REL: (Gi, G) \in \text{bg-impl-rel-ext unit-rel } Id$
shows *case* *code-blue-dfs* Gi *of*
 $None \Rightarrow \forall \text{prpl. } \neg \text{b-graph.is-lasso-prpl } G \ \text{prpl}$
 $| \text{Some } L \Rightarrow \text{b-graph.is-lasso-prpl } G \ L$
 $\langle \text{proof} \rangle$

theorem *code-blue-dfs-hash-correct*:
assumes $G: \text{b-graph } G \quad \text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G)$
assumes $REL: (Gi, G) \in \text{bg-impl-rel-ext unit-rel } Id$
shows $\text{RETURN } (\text{code-blue-dfs-hash } Gi) \leq \text{SPEC } (\lambda r.$
 $\text{case } r \text{ of } None \Rightarrow \forall \text{prpl. } \neg \text{b-graph.is-lasso-prpl } G \ \text{prpl}$
 $| \text{Some } L \Rightarrow \text{b-graph.is-lasso-prpl } G \ L)$
 $\langle \text{proof} \rangle$

theorem *code-blue-dfs-hash-correct'*:
assumes $G: \text{b-graph } G \quad \text{finite } ((g-E \ G)^* \text{ `` } g-V0 \ G)$
assumes $REL: (Gi, G) \in \text{bg-impl-rel-ext unit-rel } Id$
shows *case* *code-blue-dfs-hash* Gi *of*
 $None \Rightarrow \forall \text{prpl. } \neg \text{b-graph.is-lasso-prpl } G \ \text{prpl}$

| *Some L* \Rightarrow *b-graph.is-lasso-prpl G L*
 <proof>

theorem *code-blue-dfs-ahs-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-ahs Gi) ≤ SPEC (λr.*
case r of None ⇒ ∀ prpl. ¬b-graph.is-lasso-prpl G prpl
 | *Some L ⇒ b-graph.is-lasso-prpl G L)*
 <proof>

theorem *code-blue-dfs-ahs-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 *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*
 <proof>

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, λl::nat list.*
let s=(set l):::ₙ(nat-rel)iam-set-rel
in (λx::nat. x ∈ s)
) \in *?R*
 <proof>

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*

\equiv *code-blue-dfs :: - \Rightarrow (nat list \times -) option*

definition *code-blue-dfs-hash-nat*

\equiv *code-blue-dfs-hash :: - \Rightarrow (nat list \times -) option*

definition *code-blue-dfs-ahs-nat*

\equiv *code-blue-dfs-ahs :: - \Rightarrow (nat list \times -) option*

definition *succ-of-list-impl-int* \equiv

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

definition *acc-of-list-impl-hash-int* \equiv

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*

$:: 'prop\ ltlc \Rightarrow ('q, 'prop\ set, -)\ igba\text{-}rec\text{-}scheme\ nres$

— Conversion of LTL formula to generalized buchi automaton

where *ltl-to-gba-spec* $\varphi \equiv SPEC (\lambda gba.$

$igba.lang\ gba = language\text{-}ltlc\ \varphi \wedge igba\ gba \wedge finite\ ((g\text{-}E\ gba)^* \text{ “ } g\text{-}V0\ gba))$

definition *inter-spec*

$:: ('s, 'prop\ set, -)\ sa\text{-}rec\text{-}scheme$

$\Rightarrow ('q, 'prop\ set, -)\ igba\text{-}rec\text{-}scheme$

```

⇒ (('prod-state,-) igb-graph-rec-scheme × ('prod-state ⇒ 's)) nres
— Intersection of system and IGBA
where  $\bigwedge_{sys\ ba} \text{inter-spec } sys\ ba \equiv \text{do } \{$ 
  ASSERT (sa sys);
  ASSERT (finite ((g-E sys)* “ g-V0 sys));
  ASSERT (igba ba);
  ASSERT (finite ((g-E ba)* “ g-V0 ba));
  SPEC ( $\lambda(G,project). \text{igb-graph } G \wedge \text{finite } ((g-E\ G)^* \text{ “ } g-V0\ G) \wedge (\forall r. (\exists r'. \text{igb-graph.is-acc-run } G\ r' \wedge r = \text{project } o\ r') \longleftrightarrow (\text{graph-defs.is-run } sys\ r \wedge \text{sa-L } sys\ o\ r \in \text{igba.lang } ba))$ )
}

```

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 ( $\lambda res. \text{case } res \text{ of}$ 
    None ⇒ ( $\forall r. \neg \text{igb-graph.is-acc-run } G\ r$ )
  | Some None ⇒ ( $\exists r. \text{igb-graph.is-acc-run } G\ r$ )
  | 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  $\varphi \equiv \text{do } \{$ 
  gba :: ('ba-state,-,'ba-more) igba-rec-scheme
    ← ltl-to-gba-spec (Not-ltlc  $\varphi$ );
  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  $\varphi \leq$  do {
  ASSERT (sa sys);
  ASSERT (finite ((g-E sys)* “ g-V0 sys));
  SPEC ( $\lambda$ res. case res of
    None  $\Rightarrow$  sa.lang sys  $\subseteq$  language-ltlc  $\varphi$ 
  | Some None  $\Rightarrow \neg$  sa.lang sys  $\subseteq$  language-ltlc  $\varphi$ 
  | Some (Some r)  $\Rightarrow$  graph-defs.is-run sys r  $\wedge$  sa-L sys  $\circ$  r  $\notin$  language-ltlc  $\varphi$ )
  }
<proof>

```

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* \equiv *Id* :: ('a ltlc \times -) set

locale *impl-model-checker* =

— Assembly of a generic model-checker

fixes *sa-rel* :: ('sai \times ('s, 'prop set, 'sa-more) sa-rec-scheme) set

fixes *igba-rel* :: ('igbai \times ('q, 'prop set, 'igba-more) igba-rec-scheme) set

fixes *igbg-rel* :: ('igbgi \times ('sq, 'igbg-more) igb-graph-rec-scheme) set

fixes *ce-rel* :: ('cei \times 'sq word) set

fixes *mce-rel* :: ('mcei \times 's word) set

fixes *ltl-to-gba-impl* :: 'cfg-l2b \Rightarrow 'prop ltlc \Rightarrow 'igbai

fixes *inter-impl* :: 'cfg-int \Rightarrow 'sai \Rightarrow 'igbai \Rightarrow 'igbgi \times ('sq \Rightarrow 's)

fixes *find-ce-impl* :: 'cfg-ce \Rightarrow 'igbgi \Rightarrow 'cei option option

fixes *map-run-impl* :: ('sq \Rightarrow 's) \Rightarrow 'cei \Rightarrow 'mcei

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

assumes *ltl-to-gba-refine*:

\wedge *cfg*. (*ltl-to-gba-impl* *cfg*, *ltl-to-gba-spec*)

\in *ltl-rel* \rightarrow <*igba-rel*>plain-nres-rel

assumes *inter-refine*:

\wedge *cfg*. (*inter-impl* *cfg*, *inter-spec*)

\in *sa-rel* \rightarrow *igba-rel* \rightarrow <*igbg-rel* \times_r (*Id* \rightarrow *Id*)>plain-nres-rel

assumes *find-ce-refine*:

\wedge *cfg*. (*find-ce-impl* *cfg*, *find-ce-spec*)

\in *igbg-rel* \rightarrow <<<*ce-rel*>option-rel>option-rel>plain-nres-rel

assumes *map-run-refine*: (*map-run-impl*, (*o*)) \in (*Id* \rightarrow *Id*) \rightarrow *ce-rel* \rightarrow *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*
⟨*proof*⟩

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*⟩

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*

⟨*proof*⟩

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  $\Rightarrow$  state  $\Rightarrow$  bool where
  bval TT s = True |
  bval FF s = False |
  bval (V n) s = bs-mem n s |
  bval (Not b) s = ( $\neg$  bval b s) |
  bval (And b1 b2) s = (bval b1 s & bval b2 s) |
  bval (Or b1 b2) s = (bval b1 s | bval b2 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  $\Rightarrow$  config  $\Rightarrow$  config list where
  exec instr (pc,s) = (case instr of
    AssI ns bs  $\Rightarrow$  let bvs = zip ns (map ( $\lambda$ b. bval b s) bs) in
      [(pc + 1, foldl ( $\lambda$ s (n,bv). set-bit s n bv) s bvs)] |
    TestI b d  $\Rightarrow$  [if bval b s then (pc+1, s) else (nat(int(pc+1)+d), s)] |
    ChoiceI bis  $\Rightarrow$  let succs = [(nat(int(pc+1)+i), s) . (b,i)  $\leftarrow$  bis, bval b s]
      in if succs = [] then [(pc,s)] else succs |
```

$GotoI\ d \Rightarrow [(nat(int(pc+1)+d),s)]$

```

function  $exec' :: bprog \Rightarrow state \Rightarrow nat \Rightarrow nat\ list$  where
 $exec' ins\ s\ pc =$ 
  ( $if\ pc < array-length\ ins\ then$  (
    ( $case\ (array-get\ ins\ pc)$  of
      ( $AssI\ ns\ bs \Rightarrow [pc]$  |
        ( $TestI\ b\ d \Rightarrow$  (
          ( $if\ bval\ b\ s\ then\ exec'\ ins\ s\ (pc+1)$ 
            ( $else\ let\ pc'=(nat(int(pc+1)+d))\ in\ if\ pc'>pc\ then\ exec'\ ins\ s\ pc'$ 
              ( $else\ [pc']$ 
            ) |
          ( $ChoiceI\ bis \Rightarrow let\ succs = [(nat(int(pc+1)+i)) . (b,i) <- bis, bval\ b\ s]$ 
            ( $in\ if\ succs = []\ then\ [pc]\ else\ concat\ (map\ (\lambda pc'.\ if\ pc'>pc\ then$ 
 $exec'\ ins\ s\ pc'\ else\ [pc'])\ succs)$  |
            ( $GotoI\ d \Rightarrow let\ pc' = nat(int(pc+1)+d)\ in\ (if\ pc'>pc\ then\ exec'\ ins\ s\ pc'\ else$ 
 $[pc']$ 
          )  $else\ [pc]$ 
        )
      )
    )
  )
  <proof>
termination
  <proof>

```

```

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

```

```

fun  $nexts :: bprog \Rightarrow config \Rightarrow config\ list$  where
 $nexts ins\ (pc,s) = concat$  (
  ( $map$ 
    ( $(\lambda(pc,s).\ map\ (\lambda pc.\ (pc,s))\ (exec'\ ins\ s\ pc))$ 
    ( $nexts1\ ins\ (pc,s)$ 
  )

```

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

<i>Assign</i>	$\langle _ ::= _ \rangle [999, 61] 61$
and <i>Seq</i>	$\langle _ ; _ \rangle [60, 61] 60$
and <i>GC</i>	$\langle _ IF _ FI \rangle$
and <i>IfTE</i>	$\langle _ (IF _ / THEN _ / ELSE _) \rangle [0, 61, 61] 61$
and <i>While</i>	$\langle _ (WHILE _ / DO _) \rangle [0, 61] 61$

end

context begin interpretation *BoolProg-Syntax* $\langle proof \rangle$

fun *comp'* :: *com* \Rightarrow *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* ($\lambda(b,c). (b, comp' c)$) *gcs* *in*

let *addbc* = ($\lambda(b,cc) (bis,ins)$.

let *cc'* = *cc* @ (*if* *ins* = [] *then* [] *else* [*GotoI* (*int*(*length ins*))]) *in*

let *bis'* = *map* ($\lambda(b,i). (b, i + int(length cc'))$) *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* \Rightarrow *bprog* **where**

comp = *array-of-list* \circ *comp'*

fun *opt'* **where**

opt' (*GotoI* *d*) *ys* = (*let* *next* = $\lambda i. (case\ i\ of\ GotoI\ d \Rightarrow d + 1 \mid _ \Rightarrow 0)$

in *if* *d* < 0 \vee *nat* *d* \geq *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* \Rightarrow *instr list* **where**

opt *instr* = *foldr* *opt'* *instr* []

definition $optcomp :: com \Rightarrow bprog$ **where**

$optcomp \equiv array-of-list \circ opt \circ comp'$

3.2 Finiteness of reachable configurations

inductive-set $reachable-configs$

for $bp :: bprog$

and $c_s :: config$ — start configuration

where

$c_s \in reachable-configs\ bp\ c_s \mid$

$c \in reachable-configs\ bp\ c_s \implies x \in set\ (nexts\ bp\ c) \implies x \in reachable-configs\ bp$

c_s

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

fun $offsets :: instr \Rightarrow int\ set$ **where**

$offsets\ (AssI\ -\ -) = \{0\} \mid$

$offsets\ (TestI\ -\ i) = \{0, i\} \mid$

$offsets\ (ChoiceI\ bis) = set(map\ snd\ bis) \cup \{0\} \mid$

$offsets\ (GotoI\ i) = \{i\}$

definition $offsets-is :: instr\ list \Rightarrow int\ set$ **where**

$offsets-is\ ins = (UN\ instr : set\ ins.\ offsets\ instr)$

definition $max-next-pcs :: instr\ list \Rightarrow nat\ set$ **where**

$max-next-pcs\ ins = \{nat(int(length\ ins + 1) + i) \mid i.\ i : offsets-is\ ins\}$

lemma $finite-max-next-pcs: finite(max-next-pcs\ bp)$

$\langle proof \rangle$

lemma **(in** $linorder$) $le-Max-insertI1: \llbracket finite\ A; x \leq b \rrbracket \implies x \leq Max\ (insert\ b\ A)$

$\langle proof \rangle$

lemma **(in** $linorder$) $le-Max-insertI2: \llbracket finite\ A; A \neq \{\}; x \leq Max\ A \rrbracket \implies x \leq Max\ (insert\ b\ A)$

$\langle proof \rangle$

lemma $max-next-pcs-not-empty:$

$pc < length\ bp \implies x : set\ (exec\ (bp!pc)\ (pc, s)) \implies max-next-pcs\ bp \neq \{\}$

$\langle proof \rangle$

lemma $Max-lem2:$

assumes $pc < length\ bp$

and $(pc', s') \in set\ (exec\ (bp!pc)\ (pc, s))$

shows $pc' \leq Max\ (max-next-pcs\ bp)$

<proof>

lemma *Max-lem1*: $\llbracket pc < \text{length } bp; (pc', s') \in \text{set } (\text{exec } (bp ! pc) (pc, s)) \rrbracket$
 $\implies pc' \leq \text{Max } (\text{insert } x (\text{max-next-pcs } bp))$
<proof>

definition *pc-bound bp* $\equiv \text{max}$
 $(\text{Max } (\text{max-next-pcs } (\text{list-of-array } bp)) + 1)$
 $(\text{array-length } bp + 1)$

declare *exec'.simps*[*simp del*]

lemma [*simp*]: $\text{length } (\text{list-of-array } a) = \text{array-length } a$ *<proof>*

lemma *aux2*:
assumes *A*: $pc < \text{array-length } ins$
assumes *B*: $ofs \in \text{offsets-is } (\text{list-of-array } ins)$
shows $\text{nat } (1 + \text{int } pc + ofs) < \text{pc-bound } ins$
<proof>

lemma *array-idx-in-set*:
 $\llbracket pc < \text{array-length } ins; \text{array-get } ins \ pc = x \rrbracket$
 $\implies x \in \text{set } (\text{list-of-array } ins)$
<proof>

lemma *rcs-aux*:
assumes $pc < \text{pc-bound } bp$
assumes $pc' \in \text{set } (\text{exec}' bp \ s \ pc)$
shows $pc' < \text{pc-bound } bp$
<proof>

primrec *bexp-vars* :: $bexp \Rightarrow \text{nat set}$ **where**
 $bexp\text{-vars } TT = \{\}$
 $| bexp\text{-vars } FF = \{\}$
 $| bexp\text{-vars } (V \ n) = \{n\}$
 $| bexp\text{-vars } (\text{Not } b) = bexp\text{-vars } b$
 $| bexp\text{-vars } (\text{And } b1 \ b2) = bexp\text{-vars } b1 \cup bexp\text{-vars } b2$
 $| bexp\text{-vars } (\text{Or } b1 \ b2) = bexp\text{-vars } b1 \cup bexp\text{-vars } b2$

primrec *instr-vars* :: $\text{instr} \Rightarrow \text{nat set}$ **where**
 $\text{instr}\text{-vars } (\text{AssI } xs \ bs) = \text{set } xs \cup \bigcup (bexp\text{-vars}'\text{set } bs)$
 $| \text{instr}\text{-vars } (\text{TestI } b \ -) = bexp\text{-vars } b$
 $| \text{instr}\text{-vars } (\text{ChoiceI } cs) = \bigcup (bexp\text{-vars}'\text{fst}'\text{set } cs)$
 $| \text{instr}\text{-vars } (\text{GotoI } -) = \{\}$

find-consts '*a* *array* \Rightarrow '*a* *list*

definition *bprog-vars* :: $bprog \Rightarrow \text{nat set}$ **where**

$bprog\text{-vars } bp = \bigcup (instr\text{-vars}'set (list\text{-of-array } bp))$

definition *state-bound bp s0*

$\equiv \{s. bs\text{-}\alpha s - bprog\text{-vars } bp = bs\text{-}\alpha s0 - bprog\text{-vars } bp\}$

abbreviation *config-bound bp s0* $\equiv \{0..< pc\text{-bound } bp\} \times state\text{-bound } bp s0$

lemma *exec-bound:*

assumes *PCB*: $pc < array\text{-length } bp$

assumes *SB*: $s \in state\text{-bound } bp s0$

shows $set (exec (array\text{-get } bp pc) (pc,s)) \subseteq config\text{-bound } bp s0$

<proof>

lemma *in-bound-step:*

notes $[simp del] = exec.simps$

assumes *BOUND*: $c \in config\text{-bound } bp s0$

assumes *STEP*: $c' \in set (nexts bp c)$

shows $c' \in config\text{-bound } bp s0$

<proof>

lemma *reachable-configs-in-bound:*

$c \in config\text{-bound } bp s0 \implies reachable\text{-configs } bp c \subseteq config\text{-bound } bp s0$

<proof>

lemma *reachable-configs-out-of-bound*: $(pc',s') \in reachable\text{-configs } bp (pc,s)$

$\implies \neg pc < pc\text{-bound } bp \implies (pc',s') = (pc,s)$

<proof>

lemma *finite-bexp-vars*[*simp, intro!*]: *finite (bexp-vars be)*

<proof>

lemma *finite-instr-vars*[*simp, intro!*]: *finite (instr-vars ins)*

<proof>

lemma *finite-bprog-vars*[*simp, intro!*]: *finite (bprog-vars bp)*

<proof>

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

<proof>

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

<proof>

lemma *reachable-configs-finite*[*simp, intro!*]:

finite (reachable-configs bp c)

<proof>

definition *bpc-is-run bpc r* $\equiv let (bp,c)=bpc \text{ in } r 0 = c \wedge (\forall i. r (Suc i) \in set (BoolProgs.nexts bp (r i)))$

definition *bpc-props* $c \equiv \text{bs-}\alpha$ (*snd* c)

definition *bpc-lang* $\text{bpc} \equiv \{ \text{bpc-props } o \ r \mid r. \text{bpc-is-run } \text{bpc } r \}$

fun *print-config* ::

$(\text{nat} \Rightarrow \text{string}) \Rightarrow (\text{bitset} \Rightarrow \text{string}) \Rightarrow \text{config} \Rightarrow \text{string}$ **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.