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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *fin-U-imp-wf*:
assumes *finite U*
shows *wf* (*gen-dfs-var U*)
⟨*proof*⟩

lemma *gen-dfs-pre-imp-wf*:
assumes *gen-dfs-pre E U S V u0*
shows *wf* (*gen-dfs-var U*)
⟨*proof*⟩

lemma *gen-dfs-pre-imp-fin*:
assumes *gen-dfs-pre E U S V u0*
shows *finite* (*E* ‘‘ {*u0*})
⟨*proof*⟩

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*

$(E``\{u0\})$ $(insert\ u0\ V)$ $False$
⟨*proof*⟩

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 $t{\in}it \quad it{\subseteq}E``\{u0\} \quad t{\in}V'$
shows *gen-dfs-fe-inv E U* $(insert\ u0\ S)$ $(insert\ u0\ V)$ *u0* $(it{-}\{t\})$ *V' False*
⟨*proof*⟩

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 $y{\in}U$
⟨*proof*⟩

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 $t{\in}it \quad it{\subseteq}E``\{u0\} \quad t{\notin}V'$
shows $(V',V) \in$ *gen-dfs-var U*
⟨*proof*⟩

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 $t{\in}it \quad it{\subseteq}E``\{u0\} \quad t{\notin}V'$
shows *gen-dfs-pre E U* $(insert\ u0\ S)$ *V' t*
⟨*proof*⟩

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 $t{\in}it \quad it{\subseteq}E``\{u0\} \quad t{\notin}V'$
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*
⟨*proof*⟩

lemma *gen-dfs-post-aux*:
assumes *1*: $(u0,x){\in}E$
assumes *2*: $(x,y){\in}(E - UNIV \times insert\ u0\ S)^*$
assumes *3*: $S{\subseteq}V \quad x{\notin}V$
shows $(u0,\ y) \in (E - UNIV \times S)^*$
⟨*proof*⟩

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⊆E``{u0}*
shows *gen-dfs-post E U S V u0 V' True*
⟨*proof*⟩

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⟶cyc'*
shows *gen-dfs-post E U S V u0 V' cyc'*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⊆ V0 ∪ E* ``{u0}*
⟨*proof*⟩

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*
⟨*proof*⟩

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}*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*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-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*)
(λ*v0* (*blues*,*reds*,-). *do* {
if v0 ∉ *blues then do* {
(*blues*,*reds*,-,*cyc*) ← REC_T (λ*D* (*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*)
(λ*t* (*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) {} {}
  ⟨proof⟩
```

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⟩
```

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*⟩

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* ≡ ⟨⟨⟨*I*⟩$_i$*i-list*,*I*⟩$_i$*i-prod*⟩$_i$*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* →$_i$ *i-blue-wit* →$_i$ *i-bool*
  init-wit-blue ::$_i$ *I* →$_i$ *i-red-wit I* →$_i$ *i-blue-wit*
  init-wit-blue-early ::$_i$ *I* →$_i$ *I* →$_i$ *i-blue-wit*
  prep-wit-blue ::$_i$ *I* →$_i$ *i-blue-wit* →$_i$ *i-blue-wit*
  red-init-witness ::$_i$ *I* →$_i$ *I* →$_i$ *i-red-wit I*
  prep-wit-red ::$_i$ *I* →$_i$ *i-red-wit I* →$_i$ *i-red-wit I*
  extract-res ::$_i$ *i-blue-wit* →$_i$ ⟨⟨⟨*I*⟩$_i$*i-list*, ⟨*I*⟩$_i$*i-list*⟩$_i$*i-prod*⟩$_i$*i-option*
  red-dfs ::$_i$ ⟨*I*⟩$_i$*i-slg* →$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ ⟨*I*⟩$_i$*i-set* →$_i$ *I*
    →$_i$ ⟨⟨⟨*I*⟩$_i$*i-set*, *i-red-wit I*⟩$_i$*i-prod*⟩$_i$*i-nres*
  blue-dfs ::$_i$ *i-bg i-unit I*
    →$_i$ ⟨⟨⟨⟨*I*⟩$_i$*i-list*, ⟨*I*⟩$_i$*i-list*⟩$_i$*i-prod*⟩$_i$*i-option*⟩$_i$*i-nres*
  ⟨*proof*⟩

context begin interpretation *autoref-syn* ⟨*proof*⟩
lemma [*autoref-op-pat*]: *NO-CYC* ≡ *OP NO-CYC* :::$_i$ *i-blue-wit* ⟨*proof*⟩
end

term *lasso-rel-ext*

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

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

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*
⟨*proof*⟩
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* ×$_r$ *red-wit-rel R*⟩*nres-rel*
⟨*proof*⟩

thm *autoref-itype*(*1*−*10*)

schematic-goal *code-red-dfs-aux*:
shows *RETURN ?c* ≤ *impl-red-dfs E onstack V u*
⟨*proof*⟩
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*
⟨*proof*⟩
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*
⟨*proof*⟩

schematic-goal *code-red-dfs-hash-aux*:
shows *RETURN ?c* ≤ *impl-red-dfs-hash E onstack V u*
⟨*proof*⟩
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*
⟨*proof*⟩
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*
⟨*proof*⟩

schematic-goal *code-red-dfs-ahs-aux*:
shows *RETURN ?c* ≤ *impl-red-dfs-ahs E onstack V u*
⟨*proof*⟩
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*⇒-))

$\in$ *?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* × *'a*::*linorder*) *set*
assumes *PREFER-id R*
shows (*impl-blue-dfs*, *blue-dfs*)
$\in$ *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*⇒-))
$\in$ *?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* × *'a*::*hashable*) *set*
assumes *PREFER-id R*
shows (*impl-blue-dfs-hash*, *blue-dfs*) $\in$ *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*⇒-))
$\in$ *?R*
⟨*proof*⟩
concrete-definition *impl-blue-dfs-ahs* **uses** *blue-dfs-ahs-refine-aux*

lemma *impl-blue-dfs-ahs-autoref*[*autoref-rules*]:
fixes $R :: ('a \times 'a{::}hashable)\ set$
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* ×$_r$ ⟨*R*⟩*list-rel*⟩*Relators.option-rel*⟩*nres-rel*
⟨*proof*⟩

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

schematic-goal *code-blue-dfs-ahs-aux*:
shows *RETURN ?c* ≤ *impl-blue-dfs-ahs G*
⟨*proof*⟩
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*⟩

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 *case code-blue-dfs Gi of*
None ⇒ ∀ *prpl*. ¬*b-graph.is-lasso-prpl G prpl*
| *Some L* ⇒ *b-graph.is-lasso-prpl G L*
⟨*proof*⟩

theorem *code-blue-dfs-hash-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-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*⟩

theorem *code-blue-dfs-hash-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-hash Gi 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 *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*⟩

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*,λ*l*::*nat list*.
let s=(*set l*):::$_r$⟨*nat-rel*⟩*iam-set-rel*
in (λ*x*::*nat*. *x*∈*s*)
) ∈ *?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*
≡ *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* $\varphi \equiv$ *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)
}
⟨*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* ≡ *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* ×$_r$ (*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*,(*o*)) ∈ (*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*
⟨*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* ⇒ *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* (λ*b*. *bval b s*) *bs*) *in*
[(*pc* + *1*, *foldl* (λ*s* (*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]
)
⟨proof⟩
termination
  ⟨proof⟩

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 ⟨proof⟩
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*⟩

lemma (**in** *linorder*) *le-Max-insertI1*: ⟦ *finite A*; *x* ≤ *b* ⟧ ⟹ *x* ≤ *Max* (*insert b A*)
⟨*proof*⟩
lemma (**in** *linorder*) *le-Max-insertI2*: ⟦ *finite A*; *A* ≠ {}; *x* ≤ *Max A* ⟧ ⟹ *x* ≤ *Max* (*insert b A*)
⟨*proof*⟩

lemma *max-next-pcs-not-empty*:
pc<*length bp* ⟹ *x* : *set* (*exec* (*bp*!*pc*) (*pc*,*s*)) ⟹ *max-next-pcs bp* ≠ {}
⟨*proof*⟩

lemma *Max-lem2*:
assumes *pc* < *length bp*
and (*pc′*, *s′*) ∈ *set* (*exec* (*bp*!*pc*) (*pc*, *s*))
shows *pc′* ≤ *Max* (*max-next-pcs bp*)

⟨*proof*⟩

lemma *Max-lem1*: ⟦ *pc* < *length bp*; (*pc'*, *s'*) ∈ *set* (*exec* (*bp* ! *pc*) (*pc*, *s*))⟧
⟹ *pc'* ≤ *Max* (*insert x* (*max-next-pcs bp*))
⟨*proof*⟩

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* ⟨*proof*⟩

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*⟩

lemma *array-idx-in-set*:
⟦ *pc* < *array-length ins*; *array-get ins pc* = *x* ⟧
⟹ *x* ∈ *set* (*list-of-array ins*)
⟨*proof*⟩

lemma *rcs-aux*:
assumes *pc* < *pc-bound bp*
assumes *pc'*∈*set* (*exec' bp s pc*)
shows *pc'* < *pc-bound bp*
⟨*proof*⟩

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 = $\bigcup$(*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*⟩

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*
⟨*proof*⟩

lemma *reachable-configs-in-bound*:
c ∈ *config-bound bp s0* ⟹ *reachable-configs bp c* ⊆ *config-bound bp s0*
⟨*proof*⟩

lemma *reachable-configs-out-of-bound*: (*pc*′,*s*′)∈*reachable-configs bp* (*pc*,*s*)
⟹ ¬ *pc* < *pc-bound bp* ⟹ (*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* ≡ *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.