Model Checking and Theorem Proving in PVS

N. Shankar
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/

Computer Science Laboratory
SRI International
Menlo Park, CA

Main Points

Model checking is a powerful automated procedure for checking temporal (graph-theoretic) properties of small finite-state systems (Kripke models).

Model checking is "automatic" and can generate useful counterexamples.

But cannot cope effectively with large and unbounded state spaces.

It is therefore sensible to integrate model checking as a decision procedure within a theorem prover.

We present one such approach for integrating these two approaches based on symbolic model checking for Park's mu-calculus.

What is Model Checking?

Verification of modal/propositional properties of finite-state systems by means of state exploration or explicit fixpoint computations.

CTL model checking verifies branching-time properties of finite Kripke models.

Fairness cannot be expressed in CTL, but can be added as an extension.

LTL model checking verifies linear-time properties of finite Kripke models.

Bisimulation checks statewise isomorphism between two finite Kripke models.

Language containment checks if every infinite string accepted by one Büchi automaton is accepted by another.

Model Checking Mutual Exclusion

```
mutex  : THEORY

  BEGIN

   PC : TYPE = {sleeping, trying, critical}

   state : TYPE =
             [# pcp : PC,
                turn: bool,
             pcq : PC #]

   IMPORTING compose[state], ctlops[state],
             fairctlops[state],
             Fairctlops[state]

   s, s0, s1: VAR state
   ⋮

  END mutex
```

Peterson's Algorithm

```
I_P(s) : bool = (sleeping?(pcp(s)))

G_P(s0, s1): bool =
  (  (s1 = s0)                   %stutter
   OR (sleeping?(pcp(s0)) AND     %try
         s1 = s0 WITH [pcp := trying,
                       turn := FALSE])
   OR (trying?(pcp(s0)) AND   %enter critical
       (turn(s0) OR sleeping?(pcq(s0))) AND
      s1 = s0 WITH [pcp := critical])
   OR (critical?(pcp(s0)) AND  %exit critical
       s1 = s0 WITH [pcp := sleeping,
                     turn := FALSE ]))

I_Q(s) : bool = (sleeping?(pcq(s)))

G_Q(s0, s1): bool = ...

I(s) : bool = (I_P(s) AND I_Q(s))

G(s0, s1) : bool = (G_P(s0, s1)
                    OR G_Q(s0, s1))
```


Peterson: By Theorem Proving

```
safe(s) : bool = NOT (critical?(pcp(s))
                      AND critical?(pcq(s)))

safety_proved: CONJECTURE
   (FORALL (aa: sequence[state]):
     Init(I)(aa) AND Inv(G)(aa)
     IMPLIES Inv(safe)(aa))

strong_safe(s) : bool =
  ((critical?(pcp(s))
    IMPLIES (turn(s) OR sleeping?(pcq(s))))
  AND
  (critical?(pcq(s))
    IMPLIES (NOT turn(s)
             OR sleeping?(pcp(s)))))

strong_safety_proved: THEOREM
   (FORALL (aa: sequence[state]):
       Init(I)(aa) AND Inv(G)(aa)
     IMPLIES Inv(strong_safe)(aa))

strong_safe_implies_safe: THEOREM
   (strong_safe IMPLIES safe)(s)
```


Peterson: Proof

The proof of `safety_proved` fails because the proposed invariant is not inductive.

The invariant has to be *strengthened* to `strong_safe`, and proved by:

```
("" (SKOSIMP*)
    (EXPAND "Inv")
    (INDUCT-AND-SIMPLIFY "n"))
```

in 267 seconds.

The desired *safe* follows from the theorem `strong_safe_implies_safe` which is proved by `(GRIND)`.

Peterson: By Model Checking

```
safety: LEMMA
  I(s) IMPLIES
       AG(G, safe)(s)

live(s) : bool = (critical?(pcp(s))
                  OR critical?(pcq(s)))

try(s)  : bool = (trying?(pcp(s))
                  OR trying?(pcq(s)))

fairness1: CONJECTURE
  I(s) IMPLIES fairAF(G, live)(try)(s)

fairness2: LEMMA
  I(s) IMPLIES
       FairAF(G, live)((: try, NOT try :))(s)

sanity: LEMMA
  I(s) IMPLIES
       FairEF(G, live)((: try, NOT try :))(s)
```

Peterson: Model Checking

```
safety :

  |-------
{1}   (FORALL (s: state): I(s)
                   IMPLIES AG(G, safe)(s))

Rule? (model-check)
Rewriting with I_P
Rewriting with I_Q
Rewriting with I
Rewriting with EU
Rewriting with EF
Rewriting with AG
Rewriting with safe
Rewriting with G_P
Rewriting with G_Q
Rewriting with G
By rewriting and mu-simplifying,
Q.E.D.


Run time  = 7.97 secs.
Real time = 12.94 secs.
```


PVS and Model Checking

Such an integration is desirable since any proof-oriented approach is likely to do poorly on control-oriented problems with small finite states.

Model checking fails to handle large (realistic) state spaces, high-level specifications, and data-oriented problems.

Integrating theorem proving and model checking is simple and only requires:

1. A formalism for describing models
2. A formalism for stating temporal properties
3. A definition of the satisfaction relation

PVS and mu-calculus

Burch *et al* have shown how Park's mu-calculus can be used to capture a number of finite-state verification paradigms.

PVS integrates model checking by means of:

- A definitional presentation of Park's mu-calculus
- A BDD-based validity checker for the Boolean fragment of this mu-calculus (due to Geert Janssen), and
- A translation between the general mu-calculus (on finite states) to the Boolean fragment.

Thus, a *model* is given by means of an *initial-state* predicate and a *next-state* relation.

A *temporal property* is stated by means of fixpoints of monotone predicate transformers.

Park's Mu-Calculus

Quantified Boolean formulas with least/greatest fixpoint operators.

Formulas A in signature Σ:

- propositional variable z in Σ
- negation: $\neg A$, conjunction: $A_1 \wedge A_2$, disjunction: $A_1 \vee A_2$, implication: $A_1 \supset A_2$, equivalence: $A_1 = A_2$
- application: $p(A_1, \ldots, A_n)$, where p is an n-ary relational term
- existential quantification: $\exists z\colon A$, universal quantification $\forall z\colon A$

Park's Mu-Calculus

n-ary relational terms in Σ:

- n-ary relation variable Z in Σ
- $\lambda z_1, z_2, \ldots, z_n.\ A$, where $z_1, z_2, \ldots, z_n$ are propositional variables in Σ
- $\mu Z.\ P[Z]$, least fixpoint of monotone predicate transformer P
- $\nu Z.\ P[Z]$, greatest fixpoint: $\neg\mu Z.\ \neg P[\neg Z]$.

Intuition: Let $\top$ be $\lambda z_1 \ldots, z_n$: $\mathbf{true}$ and $\perp$ be $\lambda z_1, \ldots, z_n$: $\mathbf{false}$, then if P is $\cup$-continuous, and Q is $\cap$-continuous, then

$$(\mu Z.P[Z])(z_1, \ldots, z_n) = \bigvee_i P^i[\perp](z_1, \ldots, z_n)$$
$$(\nu Z.Q[Z])(z_1, \ldots, z_n) = \bigwedge_i Q^i[\top](z_1, \ldots, z_n)$$

The Mu-Calculus

Model checking: Check validity of mu-calculus formula A with finite state type and explicit next-state relations.

The Mu-Calculus: Take an arbitrary *state* type — not just n-tuples of booleans.

Mu-calculus in PVS

```
mucalculus[T: type]: theory
   begin

   s: var T

   p, p1, p2: var pred[T]

   predtt: type = [pred[T] → pred[T]]

   pp: var predtt

   setofpred: var pred[pred[T]]

   importing orders[pred[T]]

   p1 ≤ p2: bool = ∀ s: p1(s) ⊃ p2(s)

   ismono(pp): bool =
     ∀ p1, p2:
       p1 ≤ p2 ⊃ pp(p1) ≤ pp(p2)

   isfix(pp, p): bool = (pp(p) = p)
```


Mu-calculus in PVS (contd.)

```
⋂setofpred: pred[T] =
  λ s:
    (∀ p: (p ∈ setofpred) ⊃ p(s))

lfp(pp): pred[T] = ⋂{p | pp(p) ≤ p}

μ(pp): pred[T] = lfp(pp)

islfp(pp, p1): bool =
  isfix(pp, p1)∧
    ∀p2:
      isfix(pp, p2) ⊃ p1 ≤ p2

⋃setofpred: pred[T] =
  λ s: ∃ p: (p ∈ setofpred) ∧ p(s)

gfp(pp): pred[T] = ⋃{p | p ≤ (pp(p))}

ν(pp): pred[T] = gfp(pp)

isgfp(pp, p1): bool =
  isfix(pp, p1)∧
    ∀p2:
      isfix(pp, p2) ⊃ p2 ≤ p1

end mucalculus
```

Connectives on Predicates

```
connectives[T: type]: theory
  begin

  f, g: var pred[T]

  u, v, w: var T

  ∧(f, g)(u): bool  =  f(u) ∧ g(u);

  ∨(f, g)(u): bool  =  f(u) ∨ g(u)

  ¬(f)(u): bool  =  ¬f(u);

  ⊃(f, g)(u): bool  =  f(u) ⊃ g(u);

  ⇔(f, g)(u): bool  =  f(u) ⇔ g(u)

  end connectives
```


CTL

A branching-time temporal logic. State formulas S and path formulas Q:

- propositions: p, q.
- propositional combinations of state formulas
- path formulas: $\mathbf{X}S$ ("next state S"), $\mathbf{F}S$ ("eventually S"), $\mathbf{G}S$ ("always S"), $S_1\mathbf{U}S_2$ (" S_1 until S_2")
- state formulas: $\mathbf{A}P$ ("all paths P"), $\mathbf{E}P$ ("some path P")

CTL only allows $\mathbf{AX}$, $\mathbf{EX}$, $\mathbf{AF}$, $\mathbf{AG}$, $\mathbf{AU}$, $\mathbf{EF}$, $\mathbf{EG}$, and $\mathbf{EU}$.

CTL in mu-calculus

CTL operators (indexed by a next-state relation) can be easily defined in the mu-calculus:

- $(\mathbf{EX}_N f)(u) \equiv (\exists v: N(u, v) \wedge f(v))$
- $(f\ \mathbf{EU}_N\ g)(u) \equiv (\mu Q: g \vee (f \wedge \mathbf{EX}_N Q)))(u)$
- $(\mathbf{EG}_N f)(u) \equiv (\nu Q: f \wedge \mathbf{EX}_N Q)(u)$

Fairness cannot be expressed in CTL but can be captured in the mu-calculus.

CTL in PVS

```
ctlops[state: type]: theory
  begin
  u, v, w: var state
  f, g, Q, P, p1, p2: var pred[state]
  Z: var pred[[state, state]]
  N: var [state, state → bool]

  EX(N, f)(u): bool  =  (∃ v: (f(v) ∧ N(u, v)))

  EG(N, f): pred[state]  =
    ν(λ Q: (f ∧ EX(N, Q)))

  EU(N, f, g): pred[state]  =
    μ(λ Q: (g ∨ (f ∧ EX(N, Q))))

  EF(N, f): pred[state]  =  EU(N, (λ u: true), f)

  AX(N, f): pred[state]  =  ¬EX(N, ¬f)

  AF(N, f): pred[state]  =  ¬EG(N, ¬f)

  AG(N, f): pred[state]  =  ¬EF(N, ¬f)

  end ctlops
```

fairCTL

```
fairctlops[state: type]: theory
  begin

  u, v, w: var state

  f, g, Q, P, p1, p2: var pred[state]

  N: var [state, state → bool]

  Ff: var pred[state]

  EG_fair(N, f)(Ff): pred[state] =
    ν(λ P: EU(N, f, f ∧ Ff ∧ EX(N, P)))

  AF_fair(N, f)(Ff): pred[state] =
    ¬EG_fair(N, ¬f)(Ff)

  fair?(N, Ff): pred[state] =
    EG_fair(N, λ u: true)(Ff)

  EX_fair(N, f)(Ff): pred[state] =
    EX(N, f ∧ fair?(N, Ff))
```


fairCTL

```
EU_fair(N, f, g)(Ff): pred[state] =
  EU(N, f, g ∧ fair?(N, Ff))

EF_fair(N, f)(Ff): pred[state] =
  EF(N, f ∧ fair?(N, Ff))

AX_fair(N, f)(Ff): pred[state] =
  ¬EX_fair(N, ¬f)(Ff)

AG_fair(N, f)(Ff): pred[state] =
  ¬EF_fair(N, ¬f)(Ff)

AU_fair(N, f, g)(Ff): pred[state] =
  ¬EU_fair(N, ¬g,
        ¬f∧
          ¬g)(Ff)∧
    AF_fair(N, g)(Ff)

end fairctlops
```


Multiple fairness conditions

```
Fairctlops[state: type]: theory
  begin

  u, v, w: var state
  f, g, Q, P, p1, p2: var pred[state]
  N: var [state, state → bool]
  Fflist, Gflist: var list[pred[state]]

  CheckFair(Q, N, f, Fflist):
    recursive pred[state]
      = (cases Fflist of
            cons(Ff, Gflist):
              EU(N, f,
                    f∧
                      Ff∧
                        EX(N,
                          CheckFair(Q, N,
                            f, Gflist))),
            null: Q
            endcases)
      measure length(Fflist)

  FairEG(N, f)(Fflist): pred[state] =
    ν(λ P: CheckFair(P, N, f, Fflist))

  end Fairctlops
```


N-process Mutual Exclusion

N processes from 0 to $N-1$.

Subsystem of i processes $M(i)$ described by:

$$\begin{aligned} M(1) &= P(0) \\ M(i+1) &= P(i) \| M(i), i > 1 \end{aligned}$$

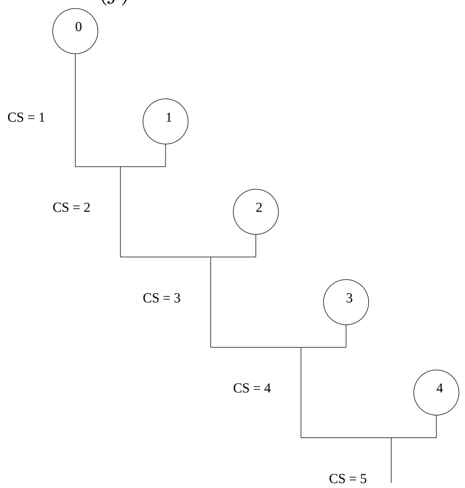
where each process $P(i)$ participates in a 2-process mutex with the 'winner' from the first i processes.

For j in $0..(i-1)$, $P(j)$ is a winner if $cs(j) = i$.

To prove: **AG**$(M(i), \text{mutex}(i))$.

N-process Mutual Exclusion

In round i, the winner j, $0 \leq j < i$ with $cs(j) = i$ competes with process i using 2-process mutual exclusion. For N processes $0 \ldots N-1$, the winner is the sole process j with $cs(j) = N$.



Proof by Abstraction + Induction

Base: Only one process.

Induction: By induction hypothesis and compositionality, at most one process j in $0..(i-1)$ s.t. $cs(j) \geq i$.

Then, processes $0..(i-1)$ can be abstracted as P and process $i+1$ as Q.

The induction conclusion follows by model checking.

Form of abstraction valid for ACTL formulas (with only universal path quantification).

Complete PVS verification including validity of abstraction.

Composing Specifications

Program P consists of initialization predicate I, internal action G, and external (allowed) action R: $\langle I, G, R \rangle$.

A run of a P is a sequence of states: the initial state satisfies I, and adjacent pairs of states satisfy G or R.

Programs are composed by taking the interleaving conjunction:

$P_1 \| P_2$ is $\langle\ I_1 \wedge I_2,$
$\quad (P_1 \wedge R_2) \vee (P_2 \wedge R_1),$
$\quad R_1 \wedge R_2 \rangle.$

Any run of P_i is a run of $P_1 \| P_2$, so any (linear-time) *property* of P_i is a property of $P_1 \| P_2$.

McMillan (p.138): In other languages (UNITY, for example ...) it is possible for any variable of a process to be changed by another process. This would have unfortunate consequences for induction over processes and other compositional methods.

Useful CTL/mu-calculus Theorems in PVS

- $\mathbf{AG}_N$**true**
- Sufficiency of invariant-restricted next-state relation:

$$\mathbf{AG}_N f \wedge \mathbf{AG}_{[N/f]} g \supset \mathbf{AG}_N g$$

- Given
  a concrete program $\langle I_C, C \rangle$,
  an abstract program $\langle I_A, A \rangle$,
  a concrete invariant f,
  a concrete property c
  an abstract property a,
  a *good* abstraction mapping h maps the concrete state to the abstract state:
  1. $I_C(u) \supset I_A(h(u))$
  2. $[C/f](u, v) \supset A(h(u), h(v))$, and
  3. $f(u) \wedge a(h(u)) \supset c(u)$.

  Then, $I_C \supset \mathbf{AG}_C\ c$ follows from $I_A \supset \mathbf{AG}_A\ a$.

Other Relevant Work

- Verification of Safety property for Philips Bounded Retransmission Protocol using abstraction and model checking [Havelund&Shankar, FME'96].
  - Needed a 'forwards' definition of $\mathbf{AG}$ for efficiency.
- Automatic generation of invariants [Bensalem&Lakhnech&Saidi, CAV'96].
- Extension to LTL model checking [Schuerman&Rajan&Shankar].
- Model checking and simulation checking for SCR-style tabular specification [Rushby&Owre& Shankar].

Related Work

Mu-calculus/CTL model checking: [Emerson&Lei], [Burch&Clarke&Dill&McMillan].

Coupling of theorem proving and model checking:

- HOL/Voss [Joyce&Seger]
- COSPAN/TLP [Kurshan&Lamport]
- SVE/SEDUCT [Dingel&Filkorn]
- Isabelle [Müller&Nipkow]
- STeP [Manna, *et al*; Sipma, *et al*]
- HOL/CTL [Schneider&Kropf]

Abstraction and model checking: [Clarke&Grumberg&Long], [Graf], [Gerth&Grumberg&Dams], [Bensalem, *et al*]

Model checking and induction: [Kurshan&McMillan] and [Wolper&Lovinfosse].

Research Issues

- Explicit-state *versus* symbolic model checking.
- Symmetry and partial-order state-space reduction.
- Symbolic state representations other than BDDs.
- Integrating hybrid (continuous environment with discrete control) model checking and theorem proving.
- Counterexample generation.

Conclusions

- Model checking and theorem proving can fruitfully coexist.
- The mu-calculus is a good medium for tying model checking with theorem proving.
- Model checking is a decision procedure for an identifiable fragment of PVS.
- Model checking combined with abstraction, induction, and compositionality can simplify the verification of unbounded-state systems.
- But the efficiency of the mu-calculus model checker is not comparable with SMV, SPIN, Murphi, COSPAN, etc., and a counterexample capability is sorely needed.