Machine-Assisted Verification Using Theorem Proving and Model Checking

To Appear in *Mathematical Programming Methodology*, (ed.) M. Broy.

N. Shankar[1]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025 USA

Abstract. Theorem proving and model checking are complementary approaches to the verification of hardware designs and software algorithms. In theorem proving, the verification task is one of showing that the formal description of the program *implies* the formal statement of a putative program property, while model checking demonstrates that the program is a model that *satisfies* the putative property. Theorem proving is completely general but typically requires significant human guidance, whereas model checking though restricted to a limited range of properties of small (essentially) finite-state systems, is largely automatic. This paper is a tutorial on the combined use of theorem proving and model checking as mechanized in the PVS specification and verification environment.

Keywords. Mechanical verification, Theorem proving, Model checking

1 Introduction

There are both scientific and pragmatic reasons for pursuing the formal verification of hardware and software systems. The scientific reason is to develop the body of knowledge needed to rigorously calculate and predict the behavior of a hardware or software system from an algorithmic description of its design. The pragmatic reason is that it is increasingly the case that extremely complex systems are being designed and deployed in ways that even trivial errors can have serious consequences. A formal method is used to write precise and unambiguous specifications of the requirements, the design assumptions, and the system design, and to demonstrate by rigorous calculation that the requirements are met by the design and the assumptions.

The conventional wisdom in the engineering of complex systems has been that the most troublesome errors are typically introduced early in the design lifecycle. Validation methods such as simulation and testing are not only inefficient and labor-intensive, they contribute very little in the early design phases. An effective formal method must therefore support the early detection and elimination of high-level design errors.

As with any engineering endeavour, mathematics is the basic tool for modelling and analyzing the behavior of hardware and software systems. However, in contrast to other engineering uses of mathematics, the analysis of discrete systems requires the extremely rigorous, large-scale application of logic and mathematics. Rigorous, repeatable, large-scale formal analysis requires effective and highly efficient mechanization.

The mathematics used in the formal development of programs and proofs can range from the trivial to the abstruse. The trivial parts of verification can and should be handled automatically through the use of appropriate decision procedures. At the higher levels, that is in the selection of induction schemes, the choice and usage of lemmas, case analysis, and quantifier instantiations, an effective mechanization must provide low-level automation that can harnessed through high-level user control.

PVS (Prototype Verification System) is a specification and verification environment that provides mechanized support for developing and harnessing mathematics for the purpose of formal verification [28]. PVS carefully integrates an expressive specification language with powerful theorem proving capabilities, and also integrates low-level inference procedures with high-level proof strategies. The inference procedures available in PVS include decision procedures for equality and linear arithmetic, symbolic model checking, and conditional rewriting and simplification. The ideas and implementation underlying PVS are still undergoing development but the system has already been used in some substantial verification projects. These projects include the verification of hardware processors [14, 25], floating-point hardware [26, 31], real-time, distributed, and fault-tolerant algorithms [18, 22, 23, 32], and in the construction of background libraries for use in verification.

This paper is a tutorial on the use of PVS for mechanized verification. Verification, even when restricted to mechanized verification using PVS, is too vast a subject to be the topic of a single paper. We will narrow our focus in this paper to the verification of transition systems, particularly finite-state ones. The main purpose of this paper is to describe how such transition systems can be specified and verified using the combination of theorem proving and model checking available in PVS.

2 Brief Overview of PVS

PVS (Prototype Verification System) is a specification and verification environment that exploits the synergy between an expressive specification language and powerful support for interactive and automated proof construction. The specification language of PVS is based on simply typed higher-order logic with several extensions such as parametric theories, predicate subtypes, dependent types, and abstract datatypes. The PVS typechecker generates proof obligations corresponding to the subtype restrictions. These proof obligations must eventually be discharged by means of the PVS proof checker before a specification can be regarded as well-formed.

A simple example of the use of predicate subtypes occurs in the case of division.[1]

```
nonzero_real: TYPE = {r: real | r /= 0}                                  1
```

In the type definition in $\boxed{1}$, the type `nonzero_real` is declared to be a subtype of the type `real` of real numbers containing all the reals distinct from `0`. The division operation can then be introduced in $\boxed{2}$ with its denominator type constrained to range over the nonzero reals.

```
  /: [real, nonzero_real -> real]                                          2
```

Note that '`/`' is known to be an infix symbol in the PVS grammar. This means that when an expression of the form `1/0` is typechecked, the obviously false proof obligation `0 /= 0` is generated corresponding to the subtype constraint on the denominator of the division operator.

This kind of subtyping also extends to higher-order types, i.e., functions, relations, functions of functions, and so on. For example, the higher-order predicate `injective?` characterizes the class of injective functions from domain type `D` to range type `R`. This predicate definition is embedded in a parametric theory `functions` in $\boxed{3}$ that takes `D` and `R` as type parameters. The predicate `injective?` can easily be used to define the subtype of injective functions of the function type `[D -> R]`.

```
functions [D, R: TYPE]: THEORY                                            3
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

END functions
```

PVS has an interactive proof checker that can be used to construct a proof tree by invoking proof commands. Each proof command either completes a branch of the proof, or generates subgoals that add new branches to the

[1]The examples in this section are adapted from the PVS prelude library which contains a basic collection of PVS theories for logic, functions, relations, sets, and various data types.

current proof branch. The PVS proof calculus is based on Gentzen's sequent calculus. The primitive proof commands are themselves quite powerful. Some of them employ decision procedures for quantifier-free theories of equality and linear arithmetic, conditional rewriting, and model checking. Compound proof commands can be composed from simpler ones by means of *strategies* which are programs that can control the application of proof commands with constructs for recursion, branching, and backtracking.

3 Peterson's Algorithm for Mutual Exclusion

The main goal of this paper is to illustrate the use of the specification, theorem proving, and model checking capabilities of PVS in the verification of transition systems. Peterson's algorithm for 2-process mutual exclusion is a useful example of a transition system since it is simple and familiar. The version of the algorithm presented here is simplified for ease of exposition but does loosely capture the key ideas underlying Peterson's algorithm [30].

The mutual exclusion problem is to ensure that two processes are never simultaneously in their critical section. The processes are labelled p and q. The behavior of each process is modelled by control variables `pcp` and `pcq` ('pc' for *program control*) that determine the control states of p and q, respectively. When `pcp` (respectively, `pcq`) is set to the value `critical`, this indicates that p (respectively, q) is in its critical section. The noncritical section is divided into a `sleeping` phase and a `trying` phase. Each process is allowed to test whether the other process is sleeping but the implementation details for such a test are omitted.[2] Access to the critical section is arbitrated by means of a shared boolean variable called `turn`.

In addition to a stuttering or skip transition, each process has three transitions. In its transition from a `sleeping` state to a `trying` state, process p (respectively, q) sets the `turn` variable to `FALSE` (`TRUE`). From the `trying` state, p (correspondingly, q) enters its `critical` state when either q (p) is `sleeping`, or `turn` is `TRUE` (`FALSE`). When exiting from the `critical` state to a `sleeping` state, process p (q) once again sets the `turn` variable to `FALSE` (`TRUE`). The algorithm as described above is obviously an abstraction of a more refined system that can contain transitions within the `sleeping`, `trying`, and `critical` phases. These transitions are not observable at the level of abstraction at which the algorithm has been described. Each process has a stuttering (skip) transition to allow internal steps that have no effect on the observable state. The stuttering step allows for a refinement map to be established between the abstract and concrete transition systems [1].

We first demonstrate the formalization of the above variant of Peterson's protocol by means of predicates and relations in the higher-order logic of

[2]The status of the other process can be observed by using a shared boolean variable for each process which indicates whether the process is in its `sleeping` state.

PVS. We then show how a proof of the mutual exclusion property can be established by means of the PVS proof checker. We then describe how model checking can be used to establish both safety and liveness properties of this simple finite-state protocol.

3.1 A Formal Description of Peterson's Algorithm

There are a large number of programming notations and program logics that can be used to formally express the algorithm that has been informally described above, but algorithms of this form can just as easily be modelled in a conventional predicate calculus. The description below uses the higher-order logic of PVS.

The first step is to capture the notion of computations over a computation state using the PVS logic. This is formalized in the theory `state` in 4 which takes the type (also named) `state` as a parameter. This type parameter can be instantiated when the `state` theory is imported within another theory. A computation is modelled as an infinite sequence of computation states, i.e., of type `sequences[states]` which is just the function type `[nat -> state]`. An assertion is just a predicate on states, i.e., an expression of type `pred[state]` (which is shorthand for the type `[state -> bool]`).

```
state[state: TYPE] : THEORY                                          4
BEGIN
  IMPORTING  sequences[state]

  statepred: TYPE = PRED[state]   %assertions
  Action: TYPE =  PRED[[state, state]]
  computation : TYPE = sequence[state]

  pp: VAR statepred
  action: VAR Action
  aa, bb, cc: VAR computation

  5
END state
```

A program characterizes a set of computations and is specified as a pair of an initialization assertion and an action, a binary transition relation on states which constrains the allowable transitions of a computation. The higher-order predicate `Init` is defined so that `Init(pp)(aa)` holds of assertion `pp` and computation `aa` exactly if `pp` holds of the initial state `aa(0)`. The predicate `Inv` of an action `action` and computation `aa` is defined to hold exactly when `action` holds of each pair of adjacent states. When `Run(pp, action)(aa)` holds, a computation `aa` is a satisfying run of a given program consisting of initialization predicate `pp` and action `action`. The predicate `Inv` is overloaded to also apply to an assertion pp and holds exactly when pp holds of every state in a given computation.

```
Init(pp)(aa) : bool = pp(aa(0))                                    5

Inv(action)(aa) : bool =
  (FORALL (n : nat) : action(aa(n), aa(n+1)))

Run(pp, action)(aa): bool =
  (Init(pp)(aa) AND Inv(action)(aa))

Inv(pp)(aa) : bool =
 (FORALL (n : nat) : pp(aa(n)))
```

Using the theory state, we can now describe the Peterson algorithm by supplying the appropriate state type, the initialization predicate, and transition relation. This is done in the theory mutex shown in 6, 7, and 8. The type of the global computation state is given by a record type with a field corresponding to each program variable. The two program control variables correspond to the fields pcp and pcq of the enumerated type PC consisting of three values sleeping, trying, and critical.

```
mutex  : THEORY                                                    6
  BEGIN

   PC : TYPE = {sleeping, trying, critical}
   state : TYPE =
             [# pcp : PC,
                turn: bool,
                pcq : PC #]

   IMPORTING  state[state]
   s, s0, s1: VAR state

   7
```

The processes p and q are specified independently. The initialization predicate for p is given by I_P which asserts that p is initially in its sleeping state. The transition relation G_P for p is the disjunction (nondeterministic choice) of four actions: the stuttering action, the sleeping-to-trying action, the entry into the critical state, and the exit from the critical to sleeping state. The expression s0 WITH [pcp := trying, turn := FALSE]) represents the result of updating the record s0 as indicated at the fields pcp and turn.

```
  I_P(s) : bool = (sleeping?(pcp(s)))                                    7

  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 ]))

 8
END mutex
```

The specification of q is symmetrical and is shown in 8

```
  I_Q(s) : bool = (sleeping?(pcq(s)))                                    8

  G_Q(s0, s1): bool =
    (  (s1 = s0)                   %stutter
     OR
       (sleeping?(pcq(s0)) AND     %try
          s1 = s0 WITH [pcq := trying, turn := TRUE])
     OR
       (trying?(pcq(s0))  AND      %enter critical
         (NOT turn(s0) OR sleeping?(pcp(s0))) AND
        s1 = s0 WITH [pcq := critical])
     OR (critical?(pcq(s0)) AND     %exit critical
         s1 = s0 WITH [pcq := sleeping, turn := TRUE]))
```

The entire system is specified in 9 in terms of the initialization predicate `I` which is the conjunction of `I_P` and `I_Q`, and the transition relation `G` which is the disjunction (i.e., interleaving) of actions `G_P` and `G_Q`.

```
 I(s) : bool = (I_P(s) AND I_Q(s))                                       9

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

3.2 Deductive Verification of Peterson's Algorithm

Having specified the algorithm in terms of an initialization predicate and a transition relation, we can now attempt to prove that the algorithm guarantees mutual exclusion. The statement is in the theory `mutex_proof` in 10.

(The theory `connectives` that is imported by the `mutex_proof` theory contains the lifted definitions of the boolean connectives so that, for example, `AND(pp, qq)` is defined as the predicate `(LAMBDA s: pp(s) AND qq(s))`.) The assertion `safe` holds of any state where at most one of p or q is critical. It is then easy to state the theorem that `safe` is an invariant for any computation corresponding to the above algorithm specification.

```
mutex_proof: THEORY                                                    10
 BEGIN
   IMPORTING mutex, connectives[state]
   s, s0, s1: VAR state

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

   safety_proved: CONJECTURE
      (FORALL (aa: computation):
        Run(I, G)(aa)
        IMPLIES Inv(safe)(aa))

   19
END mutex_proof
```

We can now try the proof of the conjecture[3] `safety_proved`. We show a partial transcript of the proof starting with 11. The proof commands are supplied in response to the `Rule?` prompt following the display of the current subgoal of the proof. The `skosimp` command introduces an arbitrary constant `aa!1` for the universally quantified parameter `aa`. It also "flattens" the implication across the sequent turnstile. The interpretation of a PVS sequent is as an implication between the conjunction of (*antecedent*) formulas above the sequent turnstile '`|-------`' and the disjunction of (*consequent*) formulas below the turnstile. The antecedent formulas are numbered `-1`, `-2`, etc., and the consequent formulas are numbered `1`, `2`, and so on. Since each subgoal sequent in a proof contains some new formulas and some old formulas that are inherited from the parent goal, the numbers corresponding to new formulas are in braces, e.g., {1}, whereas the numbers for the old formulas are in brackets, e.g., *[-1]*.

[3]The keyword `CONJECTURE` has no special semantic significance in PVS and we could just as well have used `LEMMA`, `FACT`, `THEOREM`, `PROPOSITION`, etc.

```
safety_proved :                                                        11

  |-------
{1}   (FORALL (aa: computation):
            Run(I, G)(aa) IMPLIES Inv(safe)(aa))

Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
safety_proved :

{-1}   Run(I, G)(aa!1)
  |-------
{1}   Inv(safe)(aa!1)
```

We need to establish the invariant `safe` by induction on the transitions used to construct the computation `aa!1` but the induction variable is hidden in the definition of `Inv`. Expanding `Inv` in 12 yields a universally quantified formula where we can apply induction over the parameter `n`.

```
Rule? (expand "Inv")                                                   12
Expanding the definition of Inv,
this simplifies to:
safety_proved :

[-1]   Run(I, G)(aa!1)
  |-------
{1}   (FORALL (n: nat): safe(aa!1(n)))
```

Invoking induction on `n` causes PVS to instantiate the natural number induction scheme (since the type of `n` is `nat`) with a suitable induction predicate, and carry out further simplifications that yield two subgoals corresponding to the base and induction steps. In the base case, proving that the invariant holds of the initial state of the computation is straightforward and can be carried out by the `grind` command which expands out some of the definitions in the subgoal and then applies the PVS decision procedures.

```
Rule? (induct "n")                                                    13
Inducting on n,
this yields  2 subgoals:
safety_proved.1 :

[-1]   Run(I, G)(aa!1)
  |-------
{1}   safe(aa!1(0))

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.1.
```

This brings up the induction step in $\boxed{14}$ which requires a proof that the invariant `safe` is preserved in any transition from `aa!1(j)` to `aa!1(j + 1)`. For reasons of efficiency and succinctness of proof, we would like to prove this by case-splitting on the possible transitions allowed by the transition relation `G`. We achieve this by expanding out the definitions in the only antecedent formula using the proof command `simplify-with-rewrites`, replacing the universally quantified variable `j` in the consequent by the constant `j!1` using `skosimp*`, then instantiating the pre-state and post-state variables in the transition `G` by `aa!1(j!1)` and `aa!1(j!1 + 1)` using `inst?`, respectively. Propositional simplification using `prop` can then be used to generate the 10 subgoals corresponding to the 10 possible transitions. Since the intermediate formulas in the above sequence of proof steps are large and unreadable, they have been suppressed by using the PVS proof command `apply` to package these four steps into a single atomic proof step.

```
safety_proved.2 :                                                     14

[-1]   Run(I, G)(aa!1)
  |-------
{1}   (FORALL (j: nat): safe(aa!1(j)) IMPLIES safe(aa!1(j + 1)))

Rule? (apply (then (simplify-with-rewrites - :defs T)
           (skosimp*)
                   (inst?)
                   (prop)))

Applying
   (THEN (SIMPLIFY-WITH-REWRITES - :DEFS T)
         (SKOSIMP*) (INST?) (PROP)),
this yields  10 subgoals:
```

The first subgoal in $\boxed{15}$ corresponds to a stuttering transition which trivially preserves the invariant. It is proved here by `grind` but it can also be proved by other simpler proof commands.

```
safety_proved.2.1 :                                                    15

{-1}   (aa!1(1 + (j!1 + 1 - 1)) = aa!1(j!1 + 1 - 1))
{-2}   safe(aa!1(j!1))
{-3}   (sleeping?(pcp(aa!1(0))))
{-4}   (sleeping?(pcq(aa!1(0))))
  |-------
{1}   safe(aa!1(j!1 + 1))

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.2.1.
```

The second subgoal in $\boxed{16}$ corresponds to the transition of p from a `sleeping` to `trying` state. This easily preserves the invariant at most one of p or q is `critical` since p is not critical in state `aa!1(j!1 + 1)`. This subgoal is also proved by `grind` which essentially expands the definition of `safe` and performs the required equality and enumerated datatype reasoning.

```
safety_proved.2.2 :                                                    16

{-1}   sleeping?(pcp(aa!1(j!1 + 1 - 1)))
{-2}   aa!1(1 + (j!1 + 1 - 1))
          = aa!1(j!1 + 1 - 1) WITH [pcp := trying, turn := FALSE]
{-3}   safe(aa!1(j!1))
{-4}   (sleeping?(pcp(aa!1(0))))
{-5}   (sleeping?(pcq(aa!1(0))))
  |-------
{1}   safe(aa!1(j!1 + 1))

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of safety_proved.2.2.
```

The third and more troublesome subgoal shown in $\boxed{17}$ corresponds to the p transition from a `trying` to `critical` state. Here, we need to rule out the possibility that the invariant can be violated because q is already in a `critical` state.

```
safety_proved.2.3 :                                               17

{-1}   turn(aa!1(j!1 + 1 - 1))
{-2}   trying?(pcp(aa!1(j!1 + 1 - 1)))
{-3}   aa!1(1 + (j!1 + 1 - 1))
         = aa!1(j!1 + 1 - 1) WITH [pcp := critical]
{-4}   safe(aa!1(j!1))
{-5}   (sleeping?(pcp(aa!1(0))))
{-6}   (sleeping?(pcq(aa!1(0))))
  |-------
{1}   safe(aa!1(j!1 + 1))
```

Indeed, the application of the `grind` command in 18 to this subgoal fails to prove it.

```
Rule? (grind)                                                     18
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
safety_proved.2.3 :

{-1}   turn(aa!1(j!1))
{-2}   trying?(pcp(aa!1(j!1)))
{-3}   aa!1(1 + j!1) = aa!1(j!1) WITH [pcp := critical]
[-4]   safe(aa!1(j!1))
{-5}   critical?(pcq(aa!1(j!1)))
[-6]   (sleeping?(pcp(aa!1(0))))
[-7]   (sleeping?(pcq(aa!1(0))))
  |-------

Rule?
```

Since `grind` fails to prove this subgoal, we need to investigate whether it is in fact provable. The application of `grind` does provide a counterexample by generating a scenario where q is `critical` in state `aa!1(j!1)` and `turn(aa!1(j!1)` is `TRUE`. This cannot happen in the actual protocol since the `turn` field is checked to be `FALSE` when q becomes `critical` and p never assigns `TRUE` to `turn`. The proof fails because the invariant is not strong enough to rule out the scenario where q is `critical` but the `turn` field happens to be `TRUE`.

The above failure is instructive since it allows us to construct a strengthened invariant. The new invariant `strong_safe` in 19 is a conjunction asserting that when p or q is critical, the corresponding test in the guard for entry to the `critical` state holds. This invariant can actually be generated by simple heuristics (for related heuristics, see [4,5,24]) The test in the guard for the entry to the `critical` state for p (q) is a local invariant for p (q) that is free from interference from q (p).

```
strong_safe(s) : bool =                                                    19
   ((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: computation):
        Run(I, G)(aa)
      IMPLIES Inv(strong_safe)(aa))

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

The invariance of `strong_safety` stated in `strong_safety_proved` is proved by following the same outline as the previous proof.[4] The theorem `strong_safe_implies_safe` is easily proved by a single `grind` command. This implication is quite obvious because, by `strong_safe`, if p and q are simultaneously `critical`, then `turn` would have to be simultaneously both `TRUE` and `FALSE`.

To summarize the main points:

- A shared state, interleaving model of concurrent programs such as those used by Manna and Pnueli's temporal logic [24], UNITY [10], and TLA [21] can be formalized by means of predicates, relations, and sequences in a higher-order logic such as HOL [16, 17] or PVS [27].
- With sufficiently powerful mechanization based on term rewriting and decision procedures, it is possible to construct mechanically verified proofs of program properties that are faithful to their informal counterparts.
- Failed invariance proofs can be used to diagnose the source of the failure.

3.3 Model Checking Mutual Exclusion

Peterson's mutual exclusion algorithm, as described above, is reused in this section to illustrate verification methodologies based on model checking. The state type `state` in the theory `mutex` in $\boxed{6}$ in Section 3.1 is a record type with one boolean field `turn` and two fields, `pcp` and `pcq`, ranging over a 3-valued enumerated type `PC`. This yields a state space with only 18 possible states. By the invariant `strong_safe`, we know that 4 of these 18 possible states are unreachable from an initial state. The proof in Section 3.2 involves a

[4] It is possible to prove `strong_safety_proved` with fewer manual interactions but this typically yields a less efficient verification and a less readable proof.

ten-way case analysis which seems rather excessive for such a small reachable state space. It is clearly more expedient to carry out such verifications by means of an exhaustive exploration of the reachable state space as is done with model checking [11, 12]. Model checking technology has developed to a point where it is possible to deal with quite large state spaces. It has also been extended to handle real-time constraints where the state space is actually uncountable [2,3]. The advantages of model checking over deductive verification are that model checking is largely automatic and can provide useful counterexample traces from failed verification attempts. We show in this section that the model checking verification paradigm is also formalizable within the framework of a higher-order logic such as PVS.

In the deductive approach to verification as used in the proof of `strong_safety_proved`, we are essentially proving that given a program M and a property A, that $\vdash M \supset A$ holds. In model checking, the program M is described by an automaton or a *Kripke model* and the property A as a temporal formula, and the verification establishes $M \models A$.

The basic ideas underlying model checking are quite simple. The state space consisting of 18 states in the Peterson algorithm along with the transition relation `G` which defines the set of directed transition edges between states, can be seen as defining a Kripke model on which the truth of formulas in a temporal logic can be evaluated. The most convenient temporal logic from the viewpoint of model checking is the branching time temporal logic CTL. The temporal operators of CTL can be characterized as minimal or maximal fixed points of monotone predicate transformers over the state space. The CTL operators are definable in terms of minimal and maximal fixed point operators can be defined in higher-order logic. We first examine the formalization of a fixed point calculus, the mu-calculus, in PVS before presenting the definitions of the CTL operators.

3.3.1 The Boolean Mu-Calculus

The boolean mu-calculus (BMC) [9, 29] consists of boolean formulas and n-ary boolean predicates.[5]

Boolean formulas are built from the truth values `TRUE` and `FALSE` and the atomic propositions, e.g., a, b, c, by means of negation $\neg A$, disjunction $A \vee B$, boolean quantification $\exists x.\ A[x]$ or $\forall x.\ A[x]$, or by the application of an n-ary predicate r to n boolean formulas $A_1, \ldots, A_n$ as in $r(A_1, \ldots, A_n)$.

The n-ary predicates are defined either by *lambda-abstraction* so that $\lambda x_1, \ldots, x_n : A[x_1, \ldots, x_n]$ or as *least fixed points* $\mu Z.\ P[Z]$ ("the least or strongest Z such that $Z = P[Z]$") where Z is a variable ranging over n-ary boolean predicates and P is a *monotone predicate transformer*. The n-ary

[5]The boolean mu-calculus is slightly different from the modal mu-calculus [15,20] where transitions are represented by labels rather than boolean functions.

boolean predicates can be ordered so that $r \leq s$ ("r is stronger than s") when $(\forall x_1, \ldots, x_n.\ r(x_1, \ldots, x_n) \supset s(x_1, \ldots, x_n))$. A monotone predicate transformer P maps n-ary boolean functions to n-ary boolean functions so that for any n-ary predicates r and r', if $r \leq r'$, then $P[r] \leq P[r']$.[6] The greatest fixed point $\nu Z.\ P[Z]$ of a monotone predicate transformer P can be defined dually as $\neg \mu Z.\ \neg P[\neg Z]$.

Let $\perp$ represent the n-ary relation $\lambda p_1, \ldots, p_n.$ FALSE. Also let $k = 2^n$ and i range from 0 to k, then The least fixed point can be expressed directly in QBF as the disjunction $P[\perp] \vee \ldots P^i[\perp] \vee \ldots P^k[\perp]$. Typically, the computation of the fixed point will converge in a lot fewer than k iterations.

It turns out to be convenient to use reduced ordered binary decision diagrams (ROBDDs) [6–8] to represent the boolean functions defined by the boolean mu-calculus. A binary decision diagram (BDD) is a binary tree with propositional variables labelling the nodes, `TRUE`/`FALSE` arrows out of each node, and `TRUE`/`FALSE` leaves. An ordered BDD (OBDD) is BDD in which the ordering of propositional variables along each branch is consistent with some fixed total ordering on these variables. A reduced OBDD is an OBDD with a directed acyclic graph structure so that there are no duplicate occurrences of equivalent subtrees, and any redundant nodes, namely those for which both the `TRUE` and `FALSE` outgoing edges point to the same subgraph, have been eliminated. ROBDDs are a canonical representation for boolean functions. The size of an ROBDD is quite sensitive to the chosen ordering of propositional variables. The ROBDD for a circuit that checks that exactly 0 or 2 of a, b, c are high, is shown in Figure 3.3.1.

3.4 The Relational Mu-Calculus

Fixed point operators extend first-order logic by allowing the definition of operations such as transitive closure, but these operators are actually *definable* in a higher-order logic. In the higher-order logic of PVS, subsets of a given type are modelled by predicates (i.e., the characteristic predicate of the set) over the type. The PVS formalization of the relational mu-calculus is shown in $\boxed{20}$. The PVS theory `mucalculus` takes a parameter `T` corresponding to the state type of the computation system or Kripke model. Predicates on the state type, i.e., assertions, are of the type `pred[T]`, namely, `[bool -> T]`. The stronger-than ordering on predicates of the state type is given by the relation `p1 <= p2`. A predicate transformer is a mapping from predicates to predicates. A monotone predicate transformer is one that preserves this ordering relation as checked by `ismono`. A fixed point of a predicate transformer is characterized by the higher-order relation `isfix`.

[6] It is typical to restrict $P[Z]$ to be *syntactically monotone* so that Z only occurs within an even number of negations.

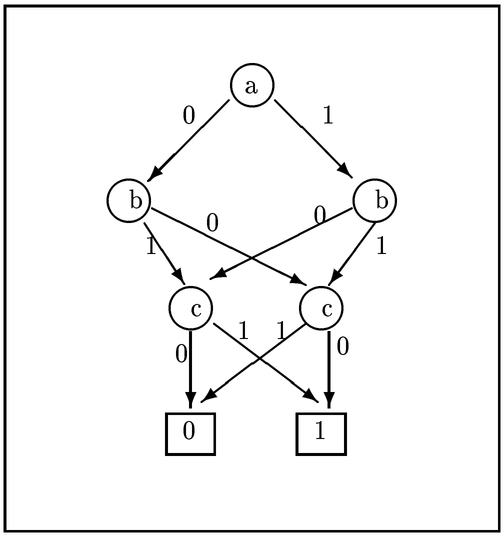

Figure 1: ROBDD for even parity on a, b, c

```
mucalculus[T:TYPE]: THEORY                                        20
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 = FORALL s: p1(s) IMPLIES p2(s)

  ismono(pp):bool =
    FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2)

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

  21
END mucalculus
```

The greatest lower bound of a set of sets (predicates) is the intersection over all these sets as defined by `glb`. The `member` operation is defined in PVS prelude library so that `member(x, p)` is just `p(x)`. The pre-fixed point of a predicate transformer `pp` is the set {`p | pp(p) <= p` }. The least fixed point is defined in $\boxed{21}$ as the greatest lower bound of the pre-fixed points and this is used to define the μ or `mu` least fixed point operator.

```
glb (setofpred):pred[T] =                                            21
  LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES  p(s))

lfp (pp):pred[T] = glb({p | pp(p) <= p})

mu(pp): pred[T] = lfp(pp)

22
```

Correspondingly, the greatest fixed point is defined in $\boxed{22}$ as the least upper bound of the post-fixed points of a monotone predicate transformer which is then used to define the ν operator.

```
lub (setofpred):pred[T] =                                            22
  LAMBDA s: EXISTS p: member(p,setofpred) AND p(s)

gfp (pp):pred[T] = lub({p | p <= (pp(p))})

nu (pp):pred[T] = gfp(pp)
```

The fixed point operators can be explained intuitively in terms of infinite disjunctions as follows. Let $\top$ be $\lambda z_1 \ldots, z_n$: **true** and $\bot$ be $\lambda z_1, \ldots, z_n$: **false**, then if P is $\cup$-continuous (i.e., $P[\cup_{i\in\omega} Z_i] = \cup_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $j < k$), and Q is $\cap$-continuous (i.e., $P[\cap_{i\in\omega} Z_i] = \cap_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $k < j$), then

$$
\begin{aligned}
(\mu Z.P[Z])(z_1, \ldots, z_n) &= \bigvee_i P^i[\bot](z_1, \ldots, z_n) \\
(\nu Z.Q[Z])(z_1, \ldots, z_n) &= \bigwedge_i Q^i[\top](z_1, \ldots, z_n)
\end{aligned}
$$

When the state type is finite, all monotone predicate transformers are $\cap$-continuous and $\cup$-continuous.

Using PVS, we can prove that the `lfp` operator does actually return the least fixed point by the familiar Tarski-Knaster argument. Let l represent $\bigcap\{X|P[X] \leq X\}$:

1. l is a fixed point of P:
   (a) $P[l] \leq l$: For each l' in $\{X|P[X] \leq X\}$, we have $P[l'] \leq l'$ (since $l' \in \{X|P[X] \leq X\}$) and $l \leq l'$ (since l is a lower bound of $\{X|P[X] \leq X\}$), this yields (by monotonicity of P), $P[l] \leq P[l'] \leq l'$.

(b) $l \leq P[l]$: By 1a and monotonicity of P, $P[l]$ is itself a pre-fixed point, and hence $l \leq P[l]$.

2. If $P[Z] = Z$ then $l \leq Z$: This is obvious since if Z is a fixed point, it is a pre-fixed point.

The above description of the relational mu-calculus is general in the sense that there are no restrictions on the size of state type `T`. We say that the state type is finite when it is constructed from booleans and scalars by tuple, record, or array type constructors. When the state type `T` is finite, the least and greatest fixed points can be explicitly computed. In PVS, this is done by carrying out a binary encoding the variables of state type as an n-tuple of booleans so that a relational mu-calculus formula on a finite type is reduced to a BMC formula. The resulting BMC formula can then be processed by a package due to Janssen [19] which employs binary decision diagrams (BDDs) to check the validity of mu-calculus formulas.

3.5 Using the Mu-Calculus to Define CTL

Formulas in a branching-time temporal logic such as CTL characterize sets of states in a Kripke model, whereas formulas in a linear-time temporal logic (LTL) characterizes sets of computation traces in a Kripke model. CTL and LTL are incomparable in their expressiveness. CTL formulas consist of:

- atomic propositions: a, b, ...
- propositional combinations: $A \wedge B$, $A \vee B$, $\neg A$, $A \supset B$.
- universal path quantified formulas: $\mathbf{AG}A$ ("along every computation path, always A"), $\mathbf{AF}A$ ("along every path, eventually A"), A **EU** B (" .
- existential path quantified formulas: $\mathbf{EG}A$ ("along some path always A"), $\mathbf{EF}A$ ("along some path eventually A")

We give just three simple instances of the definitions of CTL operators using the relational mu-calculus as formalized in PVS. As a prelude to this, we first *lift* the boolean connectives to the domain of predicates in a pointwise manner. The `connectives` theory overloads the prefix/infix operators `AND`, `OR`, `NOT` to operate on predicates rather than booleans. As an example, the expression `f AND g` for predicates `f` and `g` returns a predicate of the form `(LAMBDA u: f(u) AND g(u))`.

```
connectives [ T: TYPE ] : THEORY                                   23
  BEGIN

  f, g: VAR pred[T]
  u,v,w: VAR T

  AND(f,g)(u):bool =  f(u) AND g(u);
  OR(f,g)(u):bool =  f(u) OR g(u)
  NOT(f)(u):bool =  NOT f(u);
  IMPLIES(f,g)(u):bool = f(u) IMPLIES g(u);
  IFF(f,g)(u):bool = f(u) IFF g(u)

  TRUE(u) : bool = TRUE;
  FALSE(u) : bool = FALSE;

  END connectives
```

The CTL operators as defined below are indexed by the next-state relation N of the computation. The temporal operator EX is defined so that EX(N,p) holds of a state if p holds of some next state. The temporal formula EG(N,p) holds of a state if there is an infinite computation path leading out where p always holds. The temporal formula EU(N,p,q) holds of a state if there is computation path where q eventually holds and upto which point p holds.

```
ctlops[state : TYPE]: THEORY                                       24
 BEGIN
  IMPORTING mucalculus[state], connectives[state]
  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 =  (EXISTS v: (f(v) AND N(u, v)))

  EG(N,f):pred[state] = nu (LAMBDA Q: (f AND EX(N,Q)))

  EU(N,f,g):pred[state] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))
  ⋮
END ctlops
```

It is not hard to establish that the above definitions capture the intended meaning of the CTL operators. In the case of EX, the correspondence is obvious. The definition of EG is an infinite conjunction of the form

```
f AND EX(N, f) AND EX(N, (EX(N, f))) ....
```

This is easily seen to be equivalent to the existence of an infinite computation path on which f always holds.

Similarly, the definition of `EU(N, f, g)` expands out to the infinite disjunction

```
g OR (f AND EX(N, g)) OR (f AND EX(N, (f AND EX(N, g)))) ....
```

Since this is true exactly when at least one of the disjuncts holds, and each disjunct is equivalent to the existence of a finite path of `f` states terminating in a `g` state.

Invariance properties of transition systems can be captured in CTL as `AG` formulas. First, `EF(N, f)` is defined to hold on a state exactly when there is a computation path leading from that state to a state where `f` holds. Then `AG(N, f)` is defined to hold of any state from which no `NOT f` state is reachable.

25

```
EF(N,f):pred[state] = EU(N, TRUE, f)

AG(N,f):pred[state] = NOT EF(N, NOT f)
```

We can then, in 26, state the main invariant for Peterson's algorithm to assert that any state reachable by a computation path from a state `s` where the initialization predicate `I` holds, satisfies the assertion `safe` defined in 10.

26

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

This property can be automatically proved by means of symbolic model checking. The `model-check` proof command expands out the initialization predicate `I`, the transition relation `G`, and also rewrites the CTL operators into their mu-calculus definitions. The type `state` is then checked to be finite and the formula is translated into the boolean mu-calculus and processed by a BDD-based validity checker.

27

```
safety :

  |-------
[1]   (FORALL (s: state): I(s) IMPLIES AG(G, safe)(s))

Rule? (model-check)
   ⋮
By rewriting and mu-simplifying,

Q.E.D.
```

The fixed point in the above case is computed in just two iterations by the BDD package and the BMC validity checking only takes milliseconds. With more complicated transition systems, the above definition of `AG` might be less efficient since it computes the set of states that can reach a state violating the invariant, and there could be many such states. In such cases, there is an alternative definition of `AG` that computes the set of states reachable from an initial state. The set of states reachable by a computation path of `N` transitions from a state in `P` can be defined by the predicate `Reachable(N, P)`.

```
Reachable(N, P)(s): bool =
  mu(LAMBDA Q:
      P OR
      (LAMBDA s2: (EXISTS s1: Q(s1) AND N(s1, s2))))(s)
```
28

The predicate `AGI(N, P, f)` then checks if the set of states reachable by a computation path from a state in `P` all satisfy the predicate `f`.

```
AGI(N, P, f): bool
 = (FORALL u: Reachable(N, P)(u) IMPLIES f(u))
```
29

The safety property of Peterson's algorithm can then be expressed using `AGI`.

```
safety: LEMMA AGI(G, I, safe)
```
30

The mu-calculus model checker is similarly efficient on this "forward" version of `AG`. The latter definition turns out to be significantly more efficient on certain examples such as a finite-state version of the Philips bounded retransmission protocol.

BDDs are used as a compact symbolic representation for sets of states, the transition predicate, and the iterates of the predicate transformer used in computing the fixed points. The size of examples that can be handled by symbolic model checking is typically bounded by the size of the BDDs generated during the fixed point computation. As noted earlier, BDD size is sensitive to the variable ordering. Unlike other model checkers, in PVS there is no way for a user to control the variable ordering. Janssen's BDD package, like many other BDD packages, uses a heuristic for dynamically reordering the variables during BDD construction.

CTL as defined is in the alternation-free fragment of the mu-calculus since even though the temporal modalities of CTL are nested, there is no nesting of predicate variables bound by μ or ν in the expanded mu-calculus representation of a CTL formula.

3.6 Liveness

The safety property of mutual exclusion could be trivially satisfied by a program that denies entry into the critical section to both processes. As a desirable correctness property, one would like to demonstrate that if a process is in its `trying` state then it eventually makes the transition to its `critical` state. This *progress* or *liveness* property is not automatically guaranteed by the mutual exclusion algorithm as we modelled above since a valid computation can stutter indefinitely. In order to guarantee progress, we need to rule out computations with indefinite stuttering on states where a non-stuttering transition is possible. This can be done by means of fairness conditions. There is a large variety of notions of fairness. The typical ones are *strong fairness* (if a transition is infinitely often enabled, then it is infinitely often taken), and *weak fairness* (if a transition is indefinitely enabled, then it is eventually taken). These notions of fairness are not expressible in CTL, but CTL can be extended with a weaker notion of fairness that is often adequate.

A *fairness condition* `Ff` is just an assertion, i.e., a predicate on the `state` type of the computation. A fair computation path with respect to a fairness condition `Ff` is one where `Ff` occurs infinitely often. CTL can be extended so that the path quantifiers `A` and `E` are restricted to ranging over fair paths. For example, a fair version of the `EG` operator labelled `fairEG` can be defined using an alternation of a ν and μ operator.

31

```
fairEG(N, f)(Ff): pred[state] =
  nu(LAMBDA P: EU(N, f, f AND Ff AND EX(N, P)))
```

The `fairEG` operator defines a predicate that holds of states `P` such that there exists a path from any `P` state where `f` holds until both the `f` and `Ff` hold and the next state is a `P` state. If `Fflist` is a list of fairness conditions, then the `fairEG` operator can be extended to a list of fairness conditions by means of an operator `FairEG` where `FairEG(N, f)(Ff)` defines the set of set of states that lead off an infinite computation path along which `f` always holds and each `Ff` in `Fflist` holds infinitely often. We omit the definition of the CTL operators constrained with multiple fairness conditions.

The theorem `liveness_p` in $\boxed{32}$ uses `AG` and `FairAF` to assert that in any state reachable from an initial state, if p is `trying`, then is always possible to reach along a fair path, a state where p is eventually `critical`. The restriction of the `AF` to fair paths with the accompanying fairness conditions ignores paths where either p or q is stuck in its `trying` state, or where q never exits the `critical` state.

```
 trying_p?(s): bool = trying?(pcp(s))                                   32
 trying_q?(s): bool = trying?(pcq(s))
 critical_p?(s): bool = critical?(pcp(s))
 critical_q?(s): bool = critical?(pcq(s))

enabled_critical_p?(s) : bool =
   (trying?(pcp(s)) AND
         (turn(s) OR sleeping?(pcq(s))))

 ltl_liveness_1: CONJECTURE
   valid(G, Holds(I) IMPLIES (F(G(Holds(enabled_critical_p?)))
                              OR F(G(Holds(critical_q?)))
                              OR G(Holds(trying_p?) IMPLIES
                                   F(Holds(critical_p?)))))

enabled_critical_q?(s) : bool =
   (trying?(pcq(s)) AND
         (NOT turn(s) OR sleeping?(pcp(s))))

ctl_liveness: LEMMA
   I(s) IMPLIES
    AG(G, trying_p? IMPLIES
              FairAF(G, critical_p?)(
                  (: NOT enabled_critical_q?,
                     NOT enabled_critical_p?,
                     NOT critical_q? :)))(s)
```

Thus, the extension of CTL model checking with even a limited notion of fairness turns out to be useful in verifying liveness properties. CTL model checking is also useful in checking out a variety of small but useful properties. For example, the lemma `liveness_p` since it uses only universal path quantification could be trivially satisfied by an implementation that was deadlocked in its initial state. The lemma `sanity` provides a sanity check that there is a computation path along which p is eventually `critical`.

```
 sanity: LEMMA I(s) IMPLIES EF(G, critical_p?)(s)                       33
```

In summary, model checking is an effective automatic technique for verifying temporal properties of finite-state systems. Model checking can be unified with theorem proving. This is done in PVS by first defining a relational mu-calculus using higher-order logic. The temporal operators of the branching-time temporal logic CTL can then be defined using the relational mu-calculus. When the computation state is of a finite type, relational mu-calculus formulas can be reduced to equivalent formulas in the boolean mu-calculus which can then be checked for validity using ROBDDs.

4 Model Checking Linear-Time Temporal Logic

We already noted the differences in the expressiveness of CTL and LTL. Unlike the branching-time temporal logic CTL, where formulas characterize sets of computation states, a linear-time temporal logic (LTL) formula characterize sets of computation traces. LTL formulas consist of

- atomic propositions: a, b, ...
- propositional combinations: $A \wedge B$, $A \vee B$, $\neg A$, $A \supset B$
- Next formulas : $\mathbf{X}A$ ("next A" or $\circ A$)
- Until formulas : $A\ \mathbf{U}\ B$

We can then define the more familar modalities:

- Eventuality formulas: $\mathbf{F}A$ ("eventually A" or $\Diamond A$) as `TRUE` $\mathbf{U}$ A.
- Stability formulas: $\mathbf{G}A$ ("henceforth A" or $\Box A$) as $\neg\mathbf{F}\neg A$.

Though the model checking problem for LTL formulas is PSPACE-complete and the best known algorithms are exponential in the size of the given formula, LTL is especially useful in capturing useful notions of fairness and in stating and verifying progress properties that require fairness constraints.

For example, we can then state and prove the LTL variant of the property `ctl_liveness` in $\boxed{32}$ as `ltl_liveness` in $\boxed{34}$. The formula asserts that for traces defined using the transition relation `G`, if the initial state satisfies `I`, then whenever p is `trying`, it is eventually `critical`, unless either p or q's entry into the critical section is enabled indefinitely but not taken, or q is stays indefinitely in its critical section. The `Holds(r)` construct essentially represents an assertion `r` as an atomic LTL proposition. The formula `valid(G, A)` asserts that every G computation path satisfies the LTL formula `A`.

```
ltl_liveness: LEMMA                                                    34
   valid(G, Holds(I) IMPLIES (F(G(Holds(enabled_critical_p?)))
                             OR F(G(Holds(critical_q?)))
                             OR F(G(Holds(enabled_critical_q?)))
                             OR G(Holds(trying_p?) IMPLIES
                                  F(Holds(critical_p?)))))
```

The proof of `ltl_liveness` is by means of a strategy that converts an LTL validity assertion into an model checking problem in CTL augmented with fairness which is then discharged by means of the BDD-based mu-calculus validity checker. The details of this transformation are omitted. The key idea in model checking an LTL formula A is to extend each computation state s with labels corresponding to those subformulas of A that are satisfied by some computation path leading out of s. It is sufficient to label the states with formulas of the form $\mathbf{X}B$ since this is enough for computing the satisfiability of the other subformulas. Since subformulas of the form $B\ \mathbf{U}\ C$ are treated as being equivalent to $C \vee \mathbf{X}B\ \mathbf{U}\ C$, this raises the possibility that the satisfiability of $B\ \mathbf{U}\ C$ is given by a computation path along which $\mathbf{X}B\ \mathbf{U}\ C$ holds indefinitely but C never holds. Fairness conditions can be used to rule out such false computation paths by insisting that $C \vee \neg \mathbf{X}B\ \mathbf{U}\ C$ should hold infinitely often along a satisfying computation path.

5 Combining Theorem Proving and Model Checking

The previous sections have shown that:

- The same logical representation of the verification problem can be used for deductive verification or for model checking.
- Model checking can be viewed as a decision procedure for a finite-state fixed point fragment of the PVS logic.

The combination of the deductive and algorithmic (model checking) approaches has certain advantages:

1. Many significant properties of transition systems often depend only on the finite-state control part of the system where model checking can be quite effective.
2. Theorem proving can be used to *abstract* out the relevant finite-state control part of a larger, possibly infinite-state system in a *property-preserving* manner.
3. Model checking can be used to verify properties of system components that can then be composed using theorem proving.
4. Model checking can be combined with induction in the verification of parametric systems consisting of N similar processes.

The combination of theorem proving and model checking in PVS has been used in exactly this manner to verify an N-process mutual exclusion protocol that is an iterated version of Peterson's 2-process protocol. Here theorem proving is used to set up the induction and discharge the base case. The induction hypothesis is used to derive an abstraction from an $n + 1$ process algorithm to a 2-process algorithm that is sufficient for verifying the mutual exclusion property. The 2-process abstraction is easily verified by model checking.

The combination of induction and model checking can also be used to verify progress properties. Here the combination of theorem proving and model checking is particularly useful since deductive proofs of liveness are usually quite tedious.

6 Conclusions

In this brief tutorial on the mechanized verification of transition systems, we have applied the PVS specification language, proof checker, and model checker to the verification of a simple instance of Peterson's mutual exclusion algorithm. The main conclusions to be drawn from this exercise are:

- The key concepts and logics used in the formalization of simple transition systems can be captured in a higher-order logic such as that of HOL [16] or PVS [28].
- Model checking and automated theorem proving are complementary but compatible verification techniques.
- Symbolic model checking using BDDs can be viewed as a decision procedure for a well-defined fragment of the PVS logic.
- The same representation of transition systems can be used as a basis for verification using either theorem proving or model checking or both in combination.
- Effective mechanized verification can be achieved with the combination of low-level decision procedures such as model checking, BDDs, and linear arithmetic, and high-level user interaction.

There are a number of open research questions in the mechanized verification of transition systems that merit further investigation. These include:

- The automated generation of invariants and invariant strengthenings.
- The discovery of property-preserving abstractions of transition systems.
- Experimentation with decision procedures for various data domains such as monadic second-order logic and recursive datatypes.
- Domain-specific techniques for application areas such as security, fault tolerance, real-time and hybrid systems, control systems, and communication protocols.

Mechanized formal verification is rapidly gaining acceptance particularly within the hardware design community as a useful technique for detecting design errors. Model checking has been especially popular in this regard because it is mostly automatic and yields intelligible counterexamples. Formal specification has also been gaining in popularity as a methodology for capturing requirements at varying levels of abstraction. Mechanized theorem proving usually requires some degree of manual guidance and is therefore not yet widely popular but deductive proof is clearly necessary to tie together high-level specification with low-level verification techniques. Such a unification can only be achieved by integrating various verification paradigms and techniques within a single coherent semantic framework.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.

[3] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993.

[4] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer Verlag. To appear.

[5] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. In Ugo Montanari and Francesca Rossi, editors, *Principles and Practice of Constraint Programming – CP '95*, volume 976 of *Lecture Notes in Computer Science*, Cassis, France, 1995. Springer Verlag.

[6] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[10] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[11] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. de Bakker, W.-P.de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175, Noordewijkerhout, The Netherlands, 1994. Springer Verlag.

[12] E. M. Clarke and O. Grumberg. Research on automatic verication of finite state concurrent systems. In *Annual Review of Computer Science*, pages 269–290. Annual Reviews, Inc., 1987.

[13] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[14] David Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In Srivas and Camilleri [33], pages 172–186.

[15] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In *Proceedings, Symposium on Logic in Computer Science*, pages 267–278, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[16] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[17] Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 387–439, New York, NY, 1989. Springer-Verlag.

[18] Jozef Hooman. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, Lübeck, Germany, September 1994. Springer-Verlag.

[19] G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[20] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[21] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[22] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [13], pages 292–304.

[23] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[24] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems, Volume 2: Safety*. Springer Verlag, New York, NY, 1995.

[25] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[26] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Srivas and Camilleri [33], pages 64–78.

[27] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[28] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[29] David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[30] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[31] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[32] Natarajan Shankar. Verification of real-time systems using PVS. In Courcoubetis [13], pages 280–291.

[33] Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.