PVS: Combining Specification, Proof Checking, and Model Checking*

Appeared in the Proceedings of FMCAD'96, LNCS 1166

N. Shankar

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/pvs.html
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

PVS is an automated tool for specification and verification that is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. PVS strives for a balance between automation and interaction by using decision procedures to simplify the tedious and obvious steps in a proof leaving the user to interactively supply the high-level steps in a verification. Decision procedures for BDD-based propositional simplification and model checking, equality, and linear arithmetic are integrated into PVS. The utility of the language and deductive features of PVS have been demonstrated in a number of examples, including the specification and partial verification of the Rockwell-Collins AAMP5 processor design [9], and the verification of an SRT divider [13]. This tutorial concentrates on the integration of individual verification techniques available in PVS.

1 The PVS Language

The PVS specification language is based on classical, simply typed higher-order logic augmented with subtypes and dependent types. The use of higher-order logic in PVS allows variables to not only range over datatypes such as numbers, lists, and trees, but also over predicates, functions, functions of functions, etc. The strong type system is needed to disallow self-application of functions and predicates which can lead to inconsistencies such as those related to Russell's paradox. Strong typing is also useful as a way of quickly detecting common specification errors by means of a typechecker. Subtyping in PVS can be used to define refined forms of types such as those for even numbers, prime numbers, injective functions, and order-preserving maps. Typechecking is undecidable for the PVS type system. The PVS typechecker automatically checks for simple type correctness and generates proof obligations corresponding to predicate subtypes.

* The development of PVS was funded by SRI International through IR&D funds. Various applications and customizations have been funded by NSF Grant CCR-930044, NASA, ARPA contract A721, and NRL contract N00015-92-C-2177. The PVS system is the result of developed as a collaborative effort involving Sam Owre, John Rushby, Mandayam Srivas, David Cyrluk, Pat Lincoln, Sree Rajan, Judy Crow, among many others.

These proof obligations can be discharged through the use of the PVS proof checker. PVS also has parametric theories so that it is possible to capture, say, the notion of sorting with respect to arbitrary sizes, types, and ordering relations. By exploiting subtyping, dependent typing, and parametric theories, researchers at NASA Langley Research Center and SRI have developed background libraries for basic concepts such as bit vectors, finite sets, and cardinalities. Paul Miner at NASA has developed a specification of portions of the IEEE 854 floating-point standard in PVS [10].

The PVS specification language has a number of features that exploit the interaction between theorem proving and typechecking. Conversely, type information is used heavily within a PVS proof so that predicate subtype constraints are automatically asserted to the decision procedures, and quantifier instantiations are typechecked and can generate TCC subgoals during a proof attempt. The practical experience with PVS has been that the type system does rapidly detect a lot of common specification errors. These language features have been used to obtain readable hardware descriptions in which certain simple semantic errors (e.g., out-of-bounds array references) are guaranteed to be absent.

2 The PVS Proof Checker

PVS has an interactive proof checker where the user constructs a proof by directing the system to apply specific inference steps. PVS has a powerful base of primitive inference rules so that the use of decision procedures and rewriting are built into the core of the system. PVS has a strategy language for defining strategies in a manner similar to the metalanguage tactics and tacticals of LCF [5] and related systems such as HOL [6] and Nuprl [2].

Decision Procedures. PVS employs decision procedures including the congruence closure algorithm for equality reasoning along with various decision procedures for various theories such as linear arithmetic, arrays, and tuples, in the presence of uninterpreted function symbols [14]. PVS does not merely make use of decision procedures to prove theorems but also to record type constraints and to simplify subterms in a formula using any assumptions that *govern* the occurrence of the subterm. These governing assumptions can either be the test parts of surrounding conditional (IF-THEN-ELSE) expressions or type constraints on governing bound variables. Such simplifications typically ensure that formulas do not become too large in the course of a proof. Also important, is the fact that automatic rewriting is tightly coupled with the use of decision procedures, since many of the conditions and type correctness conditions that must be discharged in applying a rewrite rule succumb rather easily to the decision procedures.

Symbolic Model Checking. This approach to verification has been described in McMillan [8] and Burch, Clarke, McMillan, Dill, and Hwang [1]. The integration of symbolic model checking as a decision procedure in PVS is described below in Section 4.

Strategies. The PVS proof checker provides powerful primitive inference steps that make heavy use of decision procedures, but proof construction solely in terms of even these inference steps can be quite tedious. PVS therefore provides a language for defining high-level inference strategies. This language includes recursion, a `let` binding construct, a backtracking `try` strategy construction, and a conditional `if` strategy construction.

Typical strategies include those for heuristic instantiation of quantifiers, repeated skolemization, simplification, rewriting, and quantifer instantiation, and induction followed by simplification and rewriting. There are about a hundred strategies currently in PVS but only about thirty of these are commonly used. The others are used as intermediate steps in defining more powerful strategies. The use of powerful primitive inference steps makes it possible to define a small number of robust and flexible strategies that usually suffice for productive proof construction. Such strategies help in the automated or nearly automated verification of a variety of hardware examples [3,4,9,13].

3 Example Illustrating Rewriting Combined with Decision Procedures

Figure 1 shows a simple 3-stage pipelined processor [1]. Each instruction consists of an opcode `opcode`, two source register addresses `src1` and `src2`, and a destination register address `dstn`. The first stage loads the source register contents into the operand registers `opreg1` and `opreg2`. The second stage applies the ALU operation given by the `opcode` to the contents of the operand registers and stores the result in the write-back register `wbreg`. The final stage writes the contents of `wbreg` into the `dstn` address in the register file. The machine can nondeterministically introduce stall signals in order to implement delayed branching. The hardware must ensure that stalled instructions are not executed and that the outputs of ALU and `wbreg` are properly loaded into the operand registers in case an operand of a new instruction is the result of one of the instructions already in the pipeline.

The state of the machine is modelled as a collection of signals, one per wire, so that at time tick `t` for `t` $\geq$ 0, the value of signal `wbreg`, for instance, is `wbreg(t)`. The state transitions of the system can be modelled as equations that describe the value of a signal at time `t + 1` in terms of the values of various signals at time `t`. The main correctness property is stated as asserting that for an unstalled instruction at time `t`, the destination register of the register file at time `t+3` registers the result of applying the ALU operation given by the opcode to the contents of the operand registers.

```
correctness: THEOREM (FORALL t:
 NOT(stall(t)) IMPLIES
     regfile(t+3)(dstn(t)) =
       aluop(opcode(t), regfile(t+2)(src1(t)),
             regfile(t+2)(src2(t))) )
```

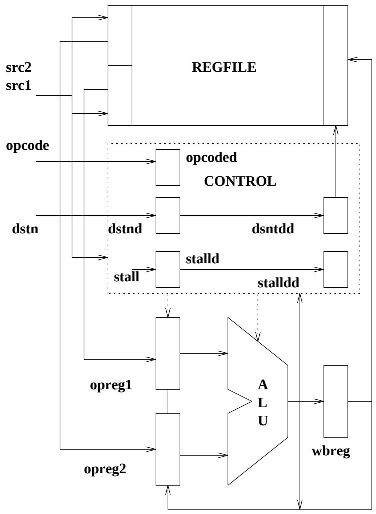

Fig. 1. A 3-Stage Pipelined Processor

This correctness property is proved in about five steps by first symbolically executing the machine to obtain an equality between two large conditional expressions representing the possible values of the left-hand side and the right-hand side of the correctness statement. By case-splitting on the conditions in the conditional expression, and applying BDD-based boolean simplification, we get seventeen subgoals which are easily discharged by the PVS decision procedures. This example can be verified for arbitrary datapath widths in PVS in under a minute of CPU time whereas a verification of this example for a fixed datapath width using model checking takes several hours [1]. The pipeline example described above is unrealistically small but it illustrates the need for integrating rewriting, arithmetic and equality simplification, and BDD-based propositional simplification in order to provide effective deductive support for hardware verification.

4 Integrating Model Checking and Theorem Proving

In the theorem proving approach to program verification, one verifies a property P of a program M by proving $M \supset P$. The model checking approach verifies the same program by showing that the state machine for M is a satisfying model of P, namely $M \models P$. For control-intensive approaches over small finite states, model checking is very effective since a more traditional Hoare logic style proof involves discovering a sufficiently strong invariant. These two approaches have traditionally been seen as incompatible ways of viewing the verification problem.

In recent work [12], we were able to unify the two views and incorporate a model checker as decision procedure for a well-defined fragment of PVS.

Our integration uses Park's relational mu-calculus [1, 11] as a medium for communicating between PVS and a model checker for the propositional mu-calculus. The idea here is that a Kripke model is captured by defining a state type, typically given by a record type construction, and a next-state relation on this state type which defines the state transitions or the edges in the Kripke model. Temporal properties such as those expressible in the branching-time temporal logic CTL can be characterized by fixpoints computed over such a Kripke model using the least and greatest fixpoint operators `mu` and `nu` over monotone predicate transformers. Such fixpoint operators can easily be defined using the higher-order logic of PVS. A CTL formula characterizes a predicate over states in this Kripke model. CTL formula are built from the propositional atoms using the propositional connectives and the modalities `EX`, `EG`, and `EU`. With respect to the Kripke model, the CTL formula `EX(N, f)` holds on those states that have a single edge leading to a state where the formula `f` holds. The CTL formula `EG(N, f)` holds on those states that have an infinite outgoing path of edges along which the formula `f` always holds. The CTL formula `EU(N, f, g)` holds on those states that have a path of zero or more edges leading to a state where `g` holds so that `f` holds along each intermediate state on the path. The other CTL modalities can be defined in terms of the above modalities. In particular, `AG(N, f)` holds of those states that do not have a path to any state where the formula `f` is falsified.

```
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))))
```

Model checking CTL properties of a given transition system is then achieved by translating the CTL operators into the mu-calculus definitions, then binary coding the finite state type so that the result can be processed by a BDD-based boolean mu-calculus checker.

We can also prove theorems about the CTL operators using their fixpoint characterizations. In particular, we can prove theorems concerning the use of property-preserving abstractions to reduce the state space of a system to manageable levels. Such an abstraction has been used to verify a complicated communication protocol by means of a combination of theorem proving and model checking [7].

CTL cannot express fairness but we can once again define fair versions of the CTL operations in the mu-calculus. For example, the operator `fairEG(N, f)(Ff)` is the predicate that holds of a state `u` such that there is a *fair path*, i.e., one where the predicate `Ff` holds infinitely often, such that the predicate `f` holds on every state on the path. This is expressed in the mu-calculus by the definition shown below.

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

Fair variants of the other connectives can be defined using `fairEG`. Multiple fairness conditions, i.e., using a list of predicates in place of the single fairness predicate `Ff`.

One direct use of fairCTL model checking is that it can be employed for model checking linear-time temporal logic LTL. In joint work with Schuermann and Rajan, we have incorporated LTL model checking into PVS. LTL formulas characterize sets of computation traces, unlike CTL formulas which characterize sets of states in the Kripke model. Various notions of fairness are easily captured in LTL. Many popular verification formalisms are based on LTL. Other forms of finite-state verification such as bisimulation and language inclusion can also be verified in terms of the mu-calculus. Transition systems can be described both synchronously in the style of SMV [8] or asynchronously as in the mutual exclusion example described below.

5 Example: Peterson's Mutual Exclusion Algorithm

Peterson's 2-process mutual exclusion algorithm ensures that two processes P and Q are never simultaneously in their critical section. In order to model this algorithm in PVS, we first identify the state space as a record datatype consisting of P's program counter `pcp`, a boolean `turn` variable, and Q's program counter `pcq`. Each process can either be in a `sleeping`, `trying` or `critical` state according to the value of the program counter field of the state. This is formalized in PVS in the theory `mutex` as shown.

```
PC : TYPE = {sleeping, trying, critical}

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

The behavior of each process is modelled as a pair $\langle I, N\rangle$ where I is an initialization predicate on `state` characterizing the valid initial states of the process, and N is a next-state relation between states. For process P, the initialization predicate is given by `I_P` and its next-state relation is given by `G_P`. Initially, process P is in its `sleeping` state. In each transition, P either goes from `sleeping` to `trying` by setting `turn` to `FALSE`, or from `trying` to `critical` by checking whether `turn` is `TRUE` or Q is `sleeping`, or exits the `critical` state by setting `turn` to `FALSE`. Process Q has a symmetrical set of transitions. The composed system is described by conjoining the initialization predicates of P and Q, and by taking the disjunction of the two transition relations in order to obtain the combined transition relation representing the interleaving of P and Q transitions.

```
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))
```

The main safety property of a two-process Peterson algorithm is very easily verified by means of model checking. The correctness of an N-process version of this algorithm can be reduced to that of the two-process case by means of induction and abstraction. This combination of induction, abstraction, and model checking can be applied to a number of other parametric N-process algorithms.

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

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

6 Conclusions

We claim that no single technique such as rewriting, BDDs, or model checking is effective for all aspects of hardware verification. Many examples need the careful integration of these techniques. We have shown some simple examples to illustrate the integration available in PVS. This combination of techniques has been applied to some larger examples such as an SRT divider and Rockwell-Collins AAMP series of processors. The automation available in PVS on these examples can be further improved through the use of more decision procedures (e.g., bit vectors) and better verification methodologies (e.g., abstraction, induction).

References

1. 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.
2. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
3. D. Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD '96)*, 1996. This volume.
4. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.
5. M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
6. 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.
7. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
8. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.
9. 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.
10. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA Langley Research Center, 1995.
11. David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 19.
12. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
13. 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*, number 1102 in Lecture Notes in Computer Science, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.
14. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.