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

To appear in CAV'96

S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas

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

PVS (Prototype Verification System) is an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified. It is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. For example, the type system of PVS requires the use of theorem proving to establish type correctness, and conversely, type information is used extensively during a proof. Similarly, decision procedures are heavily used in order to simplify the tedious and obvious steps in a proof leaving the user to interactively supply the high-level steps in a verification. Model checking is one such decision procedure that is used to discharge temporal properties of specific finite-state systems.

A variety of examples from functional programming, fault tolerance, and real time computing have been verified using PVS [8]. The most substantial use of PVS has been in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5 designed by Rockwell-Collins and containing about 500,000 transistors [6]. Most recently, PVS has been applied to the verification of the design of an SRT divider [10]. The key elements of the PVS design are described below in greater detail below.

1 Combining Theorem Proving and Typechecking

The PVS specification language is based on classical, simply typed higher-order logic, but the type system has been augmented with subtypes and dependent types. Though 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. 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 a very general bit-vector library. Paul Miner at NASA

* 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.

has developed a specification of portions of the IEEE 854 floating-point standard in PVS [7].

In PVS, the injective function space `injection` can be defined as a higher-order predicate subtype using the higher-order predicate `injective?` as shown below. The notation `(injective?)` is an abbreviation for {`f` | `injective?(f)`} which is the subtype of functions from `D` to `R` for which the predicate `injective?` holds.

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

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

  injection: TYPE = (injective?)

 END functions
```

We can also define the subtype `even` of even numbers and declare a function `double` as an injective function from the type of natural numbers `nat` to the subtype `even`.

```
 even: TYPE = {i : nat | EXISTS (j : nat): i = 2 * j}

 double : injection[nat, even] = (LAMBDA (i : nat): 2 * i)
```

When the declaration of `double` is typechecked, the typechecker generates two proof obligations or type correctness conditions (TCCS). The first TCC checks that the result computed by `double` is an even number. The second TCC checks that the definition of `double` is injective. Both TCCs are proved quickly and automatically using the default TCC strategy employed by the PVS proof checker. Proofs of more complicated TCCs can be constructed interactively.

The PVS specification language has a number of other 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.

2 Combining Decision Procedures with Interactive Proof

Decision Procedures. PVS employs decision procedures include 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 [11]. 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 closely 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.

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 (which are similar to *tactics* in LCF [3]). 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.

3 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 [9], we were able to unify the two views and incorporate a model checker as decision procedure for a well-defined fragment of PVS.

This integration uses the mu-calculus as a medium for communicating between PVS and a model checker for the propositional mu-calculus. We have used this integration to verify a complicated communication protocol by means of abstraction and model checking [4], and also to prove the correctness of an N-process mutual exclusion protocol in such a way that the induction step used the correctness of the 2-process version of the protocol as verified by the model checker.

The general mu-calculus over a given *state* type essentially provides operators for defining least and greatest fixpoints of monotone predicate transformers. In Park's mu-calculus, the state type is restricted to n-tuple of booleans and extends quantified boolean formulas (i.e., propositional logic with boolean quantification) to include the application of n-ary boolean predicates to n argument formulas. The relational terms can constructed by means of lambda-abstraction, or by taking the least fixpoint $\mu Q.F[Q]$ where Q is an n-ary predicate variable and F is a monotone predicate transformer. The greatest fixpoint operation can be written as $\nu Q.F[Q]$ and defined as $\neg\mu Q.\neg F[\neg Q]$. The mu-calculus can be easily defined in PVS. The temporal operators of the branching time temporal logic CTL can be defined using the mu-calculus [1]. An efficient model checking algorithm for the propositional mu-calculus was presented by Emerson and Lei [2], and the symbolic variant employing BDDs was presented by Burch, *et al* [1]. PVS employs a BDD-based mu-calculus validity checker due to Janssen [5].

When the state type is finite, i.e., constructed inductively from the booleans and scalar types using records, tuples, or arrays over subranges, the mu-calculus over such finite types (and the corresponding CTL) can be translated into the Boolean mu-calculus and model checking can be used as a decision procedure for this fragment. We do not discuss the details of this encoding here (see [9]).

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. E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.
3. 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.
4. 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.
5. G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
6. 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.
7. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA Langley Research Center, 1995.
8. 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.
9. 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.
10. H. Ruess, M. K. Srivas, and N. Shankar. Modular verification of SRT division. In Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification, CAV*

'96, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.
11. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.