A Brief Overview of PVS

Sam Owre and Natarajan Shankar*

Computer Science Laboratory,
SRI International

Abstract. PVS is now 15 years old, and has been extensively used in research, industry, and teaching. The system is very expressive, with unique features such as predicate subtypes, recursive and corecursive datatypes, inductive and coinductive definitions, judgements, conversions, tables, and theory interpretations. The prover supports a combination of decision procedures, automatic simplification, rewriting, ground evaluation, random test case generation, induction, model checking, predicate abstraction, MONA, BDDs, and user-defined proof strategies. In this paper we give a very brief overview of the features of PVS, some illustrative examples, and a summary of the libraries and PVS applications.

1 Introduction

PVS is a verification system [17], combining language expressiveness with automated tools. The language is based on higher-order logic, and is strongly typed. The language includes types and terms such as: numbers, records, tuples, functions, quantifiers, and recursive definitions. Full predicate subtype is supported, which makes typechecking undecidable. For example, division is defined such that the second argument is nonzero, where nonzero is defined:

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

Note that this means PVS is total; partiality is only supported via subtyping. Dependent types for records, tuples, and function types is also supported. Here is a record type (introduced with `[# #]`) representing a finite sequence, where the `seq` is an array with domain depending on the `length`.

```
finseq: TYPE = [# length: nat, seq: [below[length] -> T] #]
```

Beyond this, the PVS language has structural subtypes (i.e., a record that adds new fields to a given record), dependent types for record, tuple, and functions, recursive and corecursive datatypes, inductive and coinductive definitions, theory interpretations, and theories as parameters, conversions, and judgements that provide control over the generation of proof obligations. Specifications are

* This material is based on work performed at SRI and supported by the National Science Foundation under Grants No. CCR-ITR-0326540 and CCR-ITR-0325808.

given as collections of parameterized theories, which consist of declarations and formulas, and are organized by means of importings.

The PVS prover is interactive, but with a large amount of automation built in. It is closely integrated with the typechecker, and features a combination of decision procedures, BDDs, automatic simplification, rewriting, and induction. There are also rules for ground evaluation, random test case generation [11], model checking, predicate abstraction, and MONA. The prover may be extended with user-defined proof strategies.

PVS has been used as a platform for integration. It has a rich API, making it relatively easy to add new proof rules and integrate with other systems. Examples of this include the model checker, Duration Calculus [19], MONA [12], Maple [1], Ag [14], and Yices. The system is normally used through a customized Emacs interface, though it is possible to run it standalone (PVSio does this). PVS is open source, and is available at `http://pvs.csl.sri.com`. PVS is a part of SRI's Formal Methods Program [6].

In the following sections we will describe a basic example, give a brief description of some more advanced examples, describe some of the available libraries, and finally describe some of the applications.

2 PVS Examples

Ordered Insertion Ordered binary trees are a fundamental data structure used to represent more abstract data structures such as sets, multisets, and associative arrays. The basic idea is that the values in the nodes are totally ordered, and the values of the nodes on the left are less than the current node, which in turn is less than those on the right. The first step is to define a binary tree. In PVS, this can be specified using a datatype.

```
binary_tree[T: TYPE]: DATATYPE BEGIN
  leaf: leaf?
  node(val: T, left, right: binary_tree): node?
 END binary_tree
```

A `binary_tree` is constructed from the `leaf` and `node` *constructors*; the `node` constructor takes three arguments, and has *accessors* `val`, `left`, and `right`, and *recognizers* `leaf?` and `node?`. This is all parameterized by the type `T`. To reference a `binary_tree` where the type `T` is `int`, simply type `binary_tree[int]`.

When this is typechecked by PVS, theories are created that include many axioms such as extensionality and induction, and various mapping and reduction combinators [17].

The theory of ordered binary trees is parameterized with a type and a total ordering, an `ordered?` predicate is defined, and an `insert` operation specified. The main theorem states that if a tree is ordered, then the tree obtained after inserting an element is also ordered. This is naturally an inductive argument. In the actual specification available at `http://pvs.csl.sri.com/examples/`

`datatypes/datatypes.dmp` a helper lemma is used to make the induction easier. The proof then is quite simple; both lemmas use `induct-and-rewrite!`, including the helper lemma as a rewrite rule in the proof of the main lemma.

Inductive and Coinductive Definitions. PVS provides a mechanism for defining inductive and coinductive definitions. A simple example of an inductive definition is the transitive closure of an arbitrary relation `R`.

```
TC(R)(x, y): INDUCTIVE bool =
   R(x, y) OR (EXISTS z: R(x, z) AND TC(R)(z, y))
```

This is simply a *least fixedpoint* with respect to a given domain of elements and a set of rules, which is well-defined if the rules are *monotonic*, by the well known Knaster-Tarski theorem. Under these conditions the greatest fixedpoint also exists and corresponds to *coinductive* definitions. Inductive and coinductive definitions have induction principles, and both must satisfy additional constraints to guarantee that they are well defined.

Corecursive Datatypes. The ordered binary tree example is based on an inductively defined *datatype*, which is defined inductively and describes an *algebra*, as described by Jacobs and Rutten [7]. It is also possible to define *codatatypes*, corresponding to *coalgebras*. The simplest codatatype is the stream:

```
stream[T: TYPE]: CODATATYPE BEGIN
  cons(first:T, rest:stream): cons? END stream
```

This describes an infinite stream. Instead of induction, properties of coalgebras are generally proved using bisimulation, which is automatically generated for codatatypes, as induction is for datatypes.

```
colist[T: TYPE]: CODATATYPE BEGIN
  null: null?
  cons(first: T, rest: colist): cons?
 END colist
```

The colist example abovce looks like the `list` datatype, but includes both finite and infinite lists. This is often useful in specifications; for example, to describe a machine that may run forever, or may halt after some steps. Without coalgebras, the usual approaches are to model halting as stuttering, or to model it as a union type—both of which have various drawbacks.

Theory Interpretations. PVS has support for interpreting one theory in terms of another. This allows uninterpreted types and constants to be interpreted, and the axioms of the uninterpreted theory are translated into proof obligations, thus guaranteeing soundness. Theory interpretations are primarily used for refinement and to prove consistency for an axiomatically defined theory. PVS theory interpretations are described in the theory interpretations report [17].

Model Checking and Predicate Abstraction. PVS includes an integrated model checker that is based on the μ-calculus.[1]. To use the model checker, a finite transition system must be defined with an initial predicate and a transition relation. The PVS prelude provides CTL temporal operators, as well as several definitions of fairness. Examples making use of the model checker may be found in the report [13], which also describes the PVS table construct in some detail.

Model checking requires finite (and usually small) domains, but real systems are generally not finite. Thus in order to apply model checking to a system one must first map it to a finite abstraction. One powerful technique for doing this semi-automatically is *predicate abstraction* [16], which has been integrated into the PVS theorem prover as the `abstract` rule.

Ground Evaluation and PVSio. PVS is primarily a specification language, but it is possible to recognize an executable subset, and hence to actually run PVS. This is efficiently done with the ground evaluator, described in [18, 5].

César Muñoz has extended the ground evaluator with *PVSio* [10], which includes a predefined library of imperative programming language features such as side effects, unbounded loops, input/output operations, floating point arithmetic, exception handling, pretty printing, and parsing. The PVSio library is implemented via semantic attachments. PVSio is now a part of the PVS distribution.

3 PVS Libraries and Applications

PVS has an extensive set of libraries available. To begin with, there is the prelude—a preloaded set of theories defining many concepts, ranging from booleans through relations, functions, sets, numbers, lists, CTL, bit-vectors, and equivalence classes (see the prelude report [17] for details). The PVS distribution includes extensions of the basic finite sets and bit-vector theories given in the prelude.

NASA Langley has been working with PVS for many years, and has developed extensive libraries, available at `http://shemesh.larc.nasa.gov/fm/fm-pvs.html`. This includes libraries for algebra, analysis, calculus, complex numbers, graphs/digraphs, number theory, orders, series, trigonometric functions, and vectors. They have also contributed the Manip and Field packages, which make it easier to do numeric reasoning.

PVS has been used for applications in teaching, research, and industry. Formal specification and verification are inherently difficult, so the focus tends to be on applications with high cost of failure, or critical to life or national security. Thus most applications are to requirements [2, 8], hardware [9], safety-critical applications [3], and security [15]

[1] See Chapter 7 of *Model Checking* [4] for an introduction to the μ-calculus.

4 Conclusions and Future Work

PVS contains several features that we have omitted from this brief introduction. PVS is still under active maintenance and development. We have many features we hope to add in the future, including polymorphism, reflection, proof generation, faster rewriting and simplification, a declarative proof mode, counterexamples, proof search and other target languages for the ground evaluator.

[1] A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001*, volume 2152 of *LNCS*, pages 27–42, Edinburgh, Scotland, September 2001. Springer-Verlag.

[2] Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):139–181, 2000.

[3] V. Carreño and C. Muñoz. Aircraft trajectory modeling and alerting algorithm verification. In M. Aagaard and J. Harrison, editors, *TPHOLs 2000*, volume 1869 of *LNCS*, pages 90–105, Portland, OR, August 2000. Springer-Verlag.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2001. Available from `http://www.csl.sri.com/users/rushby/abstracts/attachments`.

[6] Formal Methods Program. Formal methods roadmap: PVS, ICS, and SAL. Technical Report SRI-CSL-03-05, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2003. Available at `http://fm.csl.sri.com/doc/roadmap03`.

[7] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[8] T. Kim, D. Stringer-Calvert, and S. Cha. Formal verification of functional properties of an SCR-style software requirements specification using PVS. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, volume 2280 of *LNCS*, pages 205–220, Grenoble, France, April 2002. Springer-Verlag.

[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] César Muñoz. *Rapid Prototyping in PVS*. National Institute of Aerospace, Hampton, VA, 2003. Available from `http://research.nianet.org/~munoz/PVSio/`.

[11] Sam Owre. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM)*, Seattle, WA, August 2006. Available at `http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf`.

[12] Sam Owre and Harald Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *CAV '2000*, volume 1855 of *LNCS*, pages 548–551, Chicago, IL, July 2000. Springer-Verlag.

[13] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, at `http://www.csl.sri.com/csl-95-12.html`. Also published as NASA Contractor Report 201729.

[14] Carlos López Pombo, Sam Owre, and Natarajan Shankar. A semantic embedding of the Ag dynamic logic in PVS. Technical Report SRI-CSL-02-04, Computer Science Laboratory, SRI International, Menlo Park, CA, October 2004.

[15] John Rushby. A separation kernel formal security policy in PVS. Technical note, Computer Science Laboratory, SRI International, Menlo Park, CA, March 2004.

[16] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *CAV '97*, volume 1254 of *LNCS*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[17] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide, PVS Language Reference, PVS Prover Guide, PVS Prelude Library, Abstract Datatypes in PVS, and Theory Interpretations in PVS*. Computer Science Laboratory, SRI International, Menlo Park, CA, 1999. Available at `http://pvs.csl.sri.com/documentation.shtml`.

[18] Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In A. Pettorossi, editor, *11th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 01)*, LNCS, pages 1–24. Springer-Verlag, 2002. Available at `ftp://ftp.csl.sri.com/pub/users/shankar/lopstr01.pdf`.

[19] Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.