PVS: An Experience Report

S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA
{owre, rushby, shankar, dave_sc}@csl.sri.com
URL: http://pvs.csl.sri.com

Abstract. PVS is a comprehensive interactive tool for specification and verification combining an expressive specification language with an integrated suite of tools for theorem proving and model checking. PVS has many academic and industrial users and has been applied to a wide range of verification tasks. In this note, we summarize some of its applications.

1 Introduction to PVS

PVS (Prototype Verification System) is an environment for constructing clear and precise specifications and for efficient mechanized verification. It is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. The PVS specification language is a typed higher-order logic with a richly expressive type system with predicate subtypes and dependent types. Typechecking in this language requires the services of a theorem prover to discharge proof obligations corresponding to subtyping constraints.

The development of PVS began in 1990, and it was first made publicly available in 1993. Subsequent releases have increased its robustness and speed, and added a host of new capabilities. The essential features of PVS have already been described in prior publications [30, 32, 40], and comprehensive details can be found in the system manuals that are available from the PVS web site at http://pvs.csl.sri.com. In this note, we indicate the capabilities of the system through a survey of some of the applications for which it has been used. Due to space constraints, this is only a small sampling of the applications that have been performed using PVS, and even those that are mentioned are often given without full citations (we generally cite only the most accessible and the most recent works). We apologize to all PVS users whose work is omitted or mentioned without citation, and refer all readers to the online PVS Bibliography for a comprehensive list of citations to work concerning PVS [38].

We divide PVS activities and applications into a few broad subject areas: library development, requirements analysis, hardware verification, fault-tolerant algorithms, distributed algorithms, semantic embeddings/backend support, real-time and hybrid systems, security and safety, and compiler correctness.

* The development of PVS was funded by SRI International through Internal R&D funds. Various applications and customizations have been funded by NSF Grants CCR-930044 and CCR-9509931, and by contracts F49620-95-C0044 from AFOSR, NASA-20334 from NASA, and N00014-92-C-2177 from ONR.
2 PVS Library Development

A major cost in undertaking formal specification and verification is that of developing formalizations for all the “background knowledge” that is required. PVS libraries help reduce this cost by providing formalizations for many common mathematical domains. Good libraries are challenging to develop: not only must they provide foundational definitions and axiomatizations that are correct, together with a body of derived constructions and lemmata that are rich enough to support development of clean, succinct, and readable specifications, but they must express these in a way that allows the PVS theorem prover to make effective use of them.

The “prelude” library built into PVS provides many useful definitions and theorems covering basic mathematical concepts such as sets, bags, functions, relations, and orderings, together with properties of real and integer arithmetic outside the domain of the PVS decision procedures (principally those involving nonlinear arithmetic).

External PVS libraries provide finite sets, floor and div/mod, bitvectors, coalgebras, real analysis, graphs, quaternions, $\mu$-calculus, and linear and branching time temporal logics. Development of libraries is very much a community effort in which sharing, modification, and extension has allowed the PVS libraries to grow into effective, robust and reusable assets. For example, the library for undirected graphs was developed by NASA Langley to support a proof of Menger’s theorem [7]. This was extended to directed graphs by the University of Utah to support analysis of PCI bus transactions [28], and subsequently re-adopted and generalized by NASA.

3 Requirements

There is extensive evidence that requirements capture is the most error-prone stage in the software engineering lifecycle, and that detection and removal of those errors at later stages is very costly. Requirements provide a fruitful application area for formal methods because relatively “lightweight” techniques have proved effective in detecting numerous and serious errors. PVS supports these activities by providing direct support for consistency and completeness checking of tabular specifications [31], and through the process of “formal challenges” [39] where expected properties are stated of a specification and examined by theorem proving or model checking.

PVS has been used by multiple NASA centers to analyze requirements for the Cassini Spacecraft [13] and for the Space Shuttle [9], and by the SafeFM project (University of London) in the analysis of requirements for avionics control systems [12].

4 Hardware Verification

Applications of PVS to hardware verification fall into two broad classes. One class is concerned with verification of the complete microarchitecture against
the instruction set architecture seen by machine code programmers. While the presence of pipelining and other optimizations introduces complexities, the basic approach to this class of verifications depends on efficient symbolic simulation and equality reasoning, which in PVS are achieved by its tight integration of cooperating decision procedures with rewriting, combined with BDD-based simplification. PVS has been used for the full or partial verification of microcoded avionics and Java processors developed by Rockwell Collins [18], as well as for a number of smaller DLX-like processors with complex pipelines.

The other class of hardware applications concerns the complex circuits, algorithms, and protocols that are the building blocks of modern processors; these applications are sufficiently difficult that success depends on finding an effective methodology. Examples include verification of SRT dividers and other arithmetic circuits at NASA [27] and SRI, out-of-order execution at the University of Utah and SRI [23] and the Weizmann Institute [36], and cache-coherence at Stanford University [33]. Some applications are best handled using a combination of tools; PVS was used in this way by Fujitsu for the validation of the high-level design of an ATM switch [37].

5 Fault-Tolerant Algorithms

Mechanisms for fault tolerance are a significant component of many safety-critical systems; they can account for half the software in a typical flight-control system, and are sufficiently complicated that they can become its primary source of failure! Verifications of practical fault-tolerant designs are quite difficult and are often achieved incrementally, as more real-world complexities are layered on to a basic algorithm. The parameterized theories and strict dependency checking of PVS help in these incremental constructions.

For example, formal analysis of Byzantine fault tolerant clock synchronization has been elaborated over nearly a decade, with contributions from SRI and NASA Langley (using a predecessor to PVS) and the University of Ulm, culminating in verification of the algorithm used in a commercial system for safety-critical automobile control [35]. Comparable developments at SRI, NASA, Allied Signal, and Ulm have verified practical algorithms for consensus, diagnosis, and group membership, together with overall architectures for state machine replication and time-triggered execution of synchronous algorithms.

6 Distributed Algorithms

The fault tolerance applications described above employ synchronous algorithms. Other distributed algorithms are often asynchronous and are generally modeled as transition relations. Safety properties are traditionally verified by invariance arguments, and generation of suitably strong invariants is the major methodological challenge. More recent approaches employ abstraction to a finite-state (or other tractable) system that can be model checked. PVS has a model checker integrated with its theorem prover, so that it is able to perform all the stages of
such approaches. Examples include communications protocols [19] and garbage
collection algorithms, parallel simulation algorithms [44] and parallelizing tech-
niques [8], and operating system buffer-cache management [34].

Current research focuses on methods for automating the generation of ab-
stractions and invariants [1,5,41].

7 Semantic Embeddings and Backend Support

For some applications it is convenient to use a customized logic for both spec-
ification and reasoning. Such logics can be encoded in the higher-order logic of
PVS using either shallow or deep semantic embeddings. Examples include the
Duration Calculus [42], DisCo [26], the B method [29], and coalgebraic treat-
ments of Java classes [25]. An advantage of these embeddings over dedicated
verification support is that the full expressiveness and power of PVS is available
for all the auxiliary concepts and data types that are required.

An API for semantic embeddings of other logics is currently under develop-
ment; this will allow specifications and proofs to be presented directly in the
notation of the embedded logic.

An alternative to semantic embedding is to use PVS to discharge proof obli-
gations generated by the support tool for another language. This route has been

8 Real-Time and Hybrid Systems

Formal treatments of real-time systems often employ special temporal or Hoare
logics. Some of these have been supported by semantic embedding in PVS, as
described above; others include timed automata [4], the language Trio [2], and
the compositional method of Hoeman [22]. Applications include several standard
test-pieces, such as the Fisher's mutual exclusion algorithm, the Generalized
Railroad Crossing, and the Steam Boiler, as well as some realistic protocols.

A real-time kernel for supporting Ada95 applications on a uniprocessor em-
bedded system has also been developed in PVS at the University of York [14].

9 Security and Safety

Strong protection of data belonging to different processes is required for both
security and safety in several applications. A formulation of this property in
terms of "noninterference" forms one of the PVS tutorial examples. More elab-
orate and realistic treatments based on the same idea have been developed for
security at Secure Computing Corporation [21], and for safe "partitioning" in
avionics at NASA [45] and Rockwell Collins [49].

Ongoing work at SRI is developing an efficient approach for the verification
of cryptographic protocols, while the special security problems arising in active
networks have been formalized at the University of Cincinnati [11].
10 Compiler Correctness

In most system developments, correctness of the translation from source code to object code is not a source of major concern. Testing is performed on object code, which is fortuitously effective in finding errors introduced during compilation, assembly and linking. For critical developments, however, further assurance may be required.

PVS has been used to perform the verification of a compiler for a small safety critical language [43], and to reason about object code in terms of flow graphs [47]. The Verifix project (http://i46s11.info.uni-karlsruhe.de/verifix/) at the Universities of Karlsruhe, Kiel, and Ulm has verified several compilation and optimization algorithms (including some expressed as abstract state machines, ASMs, where errors were found) and has also developed a collection of PVS theories for reasoning about operational and denotational semantics in this context. Another application related to programming language implementation is the security of Java style dynamic linking [10].

11 Summary

The applications sketched above give an idea of the range of projects for which PVS has been used and also provide a resource for those undertaking similar work. Additional descriptions can be found in the PVS Bibliography, which provides over 250 citations [38].

The development of PVS has been strongly influenced by practical applications and by feedback from users, and we expect this to continue. Enhancements currently in progress include direct and very fast execution for a substantial subset of the PVS language (this supports computational reflection [46], as well as improved validation of specifications [16]), and faster and more automated theorem proving. Those planned for the near future include support for refinement and a more open system architecture.

References

18. David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Hu and Vardi [24], pages 39–44.


The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.