A Brief Overview of the PVS User Interface

Sam Owre[1]

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

An overview of the PVS system is presented from a user interface perspective. We present the interfaces from the PVS Lisp core to Emacs, Tcl/Tk, the Prover, markup languages, and some of the various back-end and front-end systems that have been integrated with PVS.

1 Introduction

PVS is an open source verification system that has been in use since it was first released in 1993. The PVS interface historically was simply Emacs, with the Lisp image comprising most of PVS as a subprocess. This is still the standard way to use PVS, but over the years it has been substantially augmented with browsing tools, enhanced prover interfaces, ground evaluation, graphical displays, and LaTeX, HTML, and XML output. In addition, it has been used as both a back-end and front-end with many systems, and has a ground evaluator that even allows PVS to be used as a scripting language. Figure 1 shows the basic architecture of PVS from a user perspective. The rest of this paper is an overview of some of the aspects of the PVS system, focusing on the user interface.

2 Emacs

The basic User Interface for PVS is Emacs (or XEmacs), an extensible and very flexible editor. The PVS Lisp image runs as a subprocess of Emacs, with PVS Emacs commands translated to forms for the underlying Lisp. For example, a proof is started by placing the cursor on the lemma to be proved, and issuing the `M-x prove` command. This sends the current line and theory information to Lisp, which then locates the internal (typechecked) form of the lemma and starts the proof.

[1] This work was partially supported by NSF CCR-ITR-0325808 and CNS-0823086.

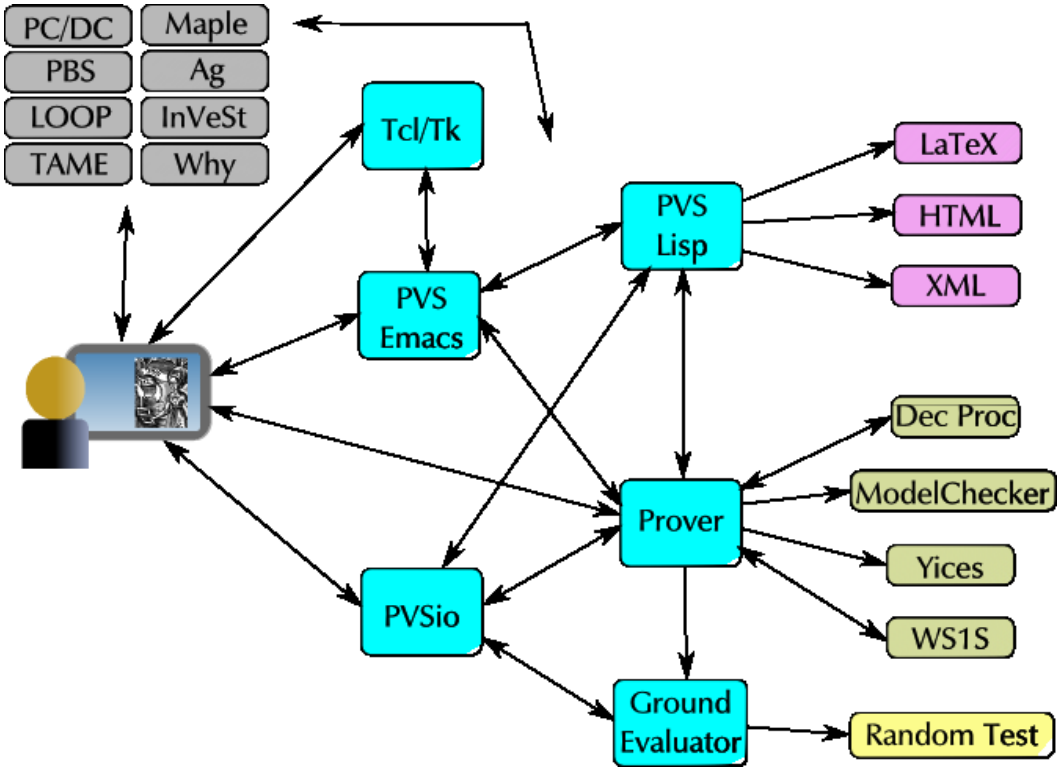

Fig. 1. PVS User Interface Overview

Specifications are edited in a special `pvs-mode` in Emacs, which in addition to the usual keyword highlighting, provides numerous functions, all of which are available from the PVS menu.

The interface is built on a modified version of ILISP [12], allowing the same interface to be used both for developing the PVS system and for creating PVS specifications. In fact, as it is just an extended version of Emacs, PVS may be used to undertake any task normally done using Emacs. PVS Lisp makes requests of Emacs by means of specially formatted strings, that are recognized by the output filter associated with the PVS Lisp subprocess. For example, by this means PVS Lisp can create a buffer and have it displayed in Emacs.

3 Tcl/Tk Interface

The Tcl/Tk interface provides some graphical interface, in particular, it allows proof trees and theory hierarchies to be viewed and manipulated. This is especially useful for large proofs or specifications. The displays are mouse-sensitive; clicking on a theory name in the theory hierarchy will display the corresponding theory specification in an Emacs buffer, and clicking on a sequent symbol in the proof tree window pops up a Tck/Tk window showing the full sequent at that point in the proof tree.

Tcl/Tk is invoked as a subprocess of PVS, and strings are passed from the PVS Lisp process to Emacs, which passes them on to Tcl/Tk. This works in both directions. Unfortunately, the interface is slow, inflexible, and buggy. In particular, there is a bug that seems to be due to a difficult to track race condition that happens when rerunning a large proof as the Tcl/Tk window tries to keep up. We plan on moving to Gtk in the future, which

can be invoked directly from PVS Lisp as foreign functions. In addition to fixing the race condition, this should make it easier to create more graphical interfaces to PVS, as process of going from Lisp to Emacs to Tcl/Tk and back again is unwieldy.

4 Prover interaction

The PVS prover is interactive; starting form a goal sequent, the user constructs a proof tree using available prover commands. The prover provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. These proof commands can be combined to form proof strategies. To make proofs easier to debug, the PVS proof checker permits proof steps to be undone, and checkpointed, and allows the specification to be modified during the course of a proof. After modification, the prover offers to rerun the proof to see that it is still valid. It marks all formulas whose proofs depend on the modified declaration as unchecked. To support proof maintenance, PVS allows proofs (and partial proofs) to be edited and rerun, and allows for multiple proofs to be associated with a formula. Currently, the proofs generated by PVS can be made presentable but they still fall short of being humanly readable.

New strategies and rules may be defined as described in [18], using the `defstep` and `addrule` functions, which may be added to an automatically loaded PVS strategies file. Typically only `defstep` is used to define new user strategies in terms of existing ones. In this way, strategies are built up from primitive rules, and only they need to be trusted. However, some extensions require the addition of new rules, which must be done carefully as soundness may be compromised.

Note that although the strategy language allows arbitrary calls to Lisp, the proofs may be rerun in a mode in which all strategies have been expanded to their primitive rules, in which the Lisp calls are no longer made. In this way the soundness of PVS relies only on the primitive rules and the core execution engine.

5 Generating Latex, HTML, and XML

PVS specifications are in ASCII, which is fine for developing specifications and proofs, but it is often desirable to present them differently. Toward this end, PVS includes facilities for generating LaTeX, HTML, and XML output. The LaTeX output can be generated for specifications or proofs, and the user has control over the mapping from PVS identifiers and operators to LaTeX. The HTML and XML are similar, but only available for specifications. The HTML interface does provide links that lead from a symbol to its corresponding declaration.

The XML output provides much more than the LaTeX and HTML output, as it is a complete representation of the internal typechecked form of PVS entities. This makes it easy to map from PVS to other systems, which is very difficult to do directly from PVS specifications and proofs. Not only is the PVS grammar difficult to parse, but the overloading and automatic conversions allowed by PVS makes it impossible to know how to interpret a concrete expression without typechecking it. The XML form solves this, as it directly represents the parse tree, and provides full resolutions for each identifier. The XML representation includes enough information that the original concrete syntax may be generated, and we have generated an XSLT script that does this.

6 PVS as a Back-end

It is often desirable to have the PVS typechecking and theorem proving available at the back-end of a system. This can easily be done by invoking PVS in *raw* mode, which runs it without the Emacs interface. In this mode it waits for Lisp input, and returns the results, exactly in the way it does with Emacs. There are several functions (e.g., `typecheck-formula-decl` and `prove-formula-decl`) that provide support for proving individual formulas, without generating a full theory. Several existing systems have used PVS as a back-end typechecker and/or theorem prover. Skakkebæk [21] made a deep embedding of the Duration Calculus in his PC/DC system.

César Muñoz implemented a shallow embedding of the B-method [1] into a front-end for PVS called PBS [14]. The B-method is a state-oriented formal method for software development that provides a uniform language, the abstract machine notation, to specify, design, and implement systems. The method is founded on set theory with a first-order predicate calculus, which is embedded into the higher-order logic of PVS.

The LOOP project [22, 6] has developed a tool for specifying and verifying properties of Java programs, using PVS as a back-end. It represents Java objects as coalgebras, and has been used to prove properties of some Java libraries, as well as proving properties of smartcards, as part of the Verificard project.

TAME (Timed Automata Modeling Environment) [4, 3] is a system for specifying several classes of automata, providing templates, a set of auxiliary theories, and specialized prover strategies for specifying and proving properties of automata models.

An interface between the Maple computer algebra system and the PVS theorem prover was implemented [2]. The interface allows Maple users access to the robust and strongly typechecked proof environment of PVS. The environment was extended by a library of proof strategies for use in real analysis. This provides both strong typechecking and theorem proving capabilities to Maple users.

Carlos Pómbo [19], used PVS to provide the semantics of $\mathbf{A_g}$ specifications, defining the semantics of First Order Dynamic Logic and Fork Algebras, along with rules and strategies that allow a user to reason in $\mathbf{A_g}$. Here conversions were defined, such as a meaning function, and arguments such as the current world of the Kripke structure, that by default are included in the prover interaction, but add clutter to the proof. In this case the function for pretty-printing applications was modified in order to suppress the meaning function and the world argument.

There are many other systems that use PVS as a back-end, including Pamela [7], InVeSt [5], the Java Interactive Verification Environment (JIVE) [13], TRIO [10], SOCOS [11], and Why [9]. This is just a partial list.

7 PVS as a Front-end

PVS has also been used as a front-end to several systems. Generally this involves creating a proof rule that interacts with the specified system. This interaction can be through a shell, or directly via foreign function calls. The usual method is to define supporting theories in PVS, define a translation from these theories to the target system, and to define a rule that performs the translation and invokes the system. If the system is intended to return more than simply true or false, a translator must also be provided to convert the results into a

valid PVS sequent. Note that, in general, the soundness of the resulting proof depends on the soundness of the underlying system.

For decision procedures, a special interface was created making it easy to implement new decision procedures. This was used to integrate ICS in earlier versions of PVS.

The built-in PVS model checker [20] is an example of this, in which the model checker only returns true, finishing the proof of this sequent, or unknown with an explanation, leaving the sequent untouched. The model checker relies on the mu-calculus, and theories to support this were provided in the PVS prelude.

The Mona WS1S system was integrated into PVS [17] in the same way. Yices was recently integrated as well, as an end-game prover. This greatly speeds up many kinds of proofs.

PVS may also be used for programming, by using the ground evaluator to translate specifications to Lisp or the Clean functional programming language (see the description at http://clean.cs.ru.nl/). This opens up many possibilities. Using semantic attachments, one can evaluate, test, and animate specifications [8]. The PVS random tester [16] builds on the ground evaluator, and allows specifications to be randomly tested, which is often useful for detecting bugs in specifications before attempting difficult proofs.

César Muñoz developed PVSio [15] an extension of the ground evaluator that makes it simple to define new attachments, use the ground evaluator during proof, and even create PVS scripts that may be used from the command line as with any other scripting language.

8 Proof discovery and maintenance

PVS has limited capabilities for browsing formulas and proofs, and copying proofs from one formula to another. Proof trees may be displayed, and proofs may be single-stepped and check-pointed. Declarations may be modified and added during a proof.

Proof discovery and maintenance is a wide open area of research. Much more is needed, for example, it should be possible to match the current sequent to formulas in the prelude or existing libraries and list the ones likely to be useful. This is quite difficult for several reasons: the libraries might not be referenced, and may only be available remotely, the matching formulas may be useless because some precondition is false, or because an inequality is in the wrong direction. Formulas might not be considered because theories were developed with different names, though they are actually relating to the same entities - for example groups could be defined in one theory using $\star$ and using $+$ in another, thus rendering syntactic matches useless.

9 Conclusion

PVS has a rich user interface, which we have outlined here. It continues to grow, and new paradigms are being explored. In our view, PVS may be treated as a tool bus, allowing exploration of interfaces between often disparate tools. The system is open source, and we encourage any and all additions to the system. More information, and instructions for obtaining and installing PVS are available at http://pvs.csl.sri.com.

References

[1] J.-R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The B-method. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, Volume 552 of Springer-Verlag *Lecture Notes in Computer Science*, pages 398–405, Noordwijkerhout, The Netherlands, October 1991. Volume 2: Tutorials.

[2] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, Volume 2152 of Springer-Verlag *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, September 2001.

[3] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Proceedings of FMSP '00: The Third Workshop on Formal Methods in Software Practice*, pages 25–36, Association for Computing Machinery, Portland, OR, August 2000.

[4] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, July 1998. Informal proceedings available at http://www.win.tue.nl/cs/ipa/uitp/proceedings.html.

[5] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, pages 505–510, Vancouver, Canada, June 1998.

[6] C.-B. Breunesse, N. Cataño, M. Huisman, and B. P. F. Jacobs. Formal methods for smart cards: An experience report. *Science of Computer Programming*, 55(1–3):53–80, March 2005.

[7] Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, Volume 1349 of Springer-Verlag *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997.

[8] Judy Crow, Sam Owre, John Rushby, N. Shankar, and Dave 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.

[9] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification (tool paper). In *Computer Aided Verification*. pages 173–177, 2007.

[10] Carlo A. Furia, Matteo Rossi, Dino M, and Angelo Morzenti. Automated compositional proofs for real-time systems. *Theoretical Computer Science*, pages 326–340, 2006.

[11] Ralph johan Back, Johannes Eriksson, and Magnus Myreen. Verifying invariant based programs in the SOCOS environment. In *In Teaching Formal Methods: Practice and Experience (BCS Electronic Workshops in Computing). BCS-FACS*, 2006.

[12] Todd Kaufmann, Chris McConnell, Ivan Vazquez, Marco Antoniotti, Rick Campbell, and Paolo Amoroso. *ILISP User Manual*, 2002. Available with at http://sourceforge.net/projects/ilisp/.

[13] J. Meyer, P. Müller, and A. Poetzsch-Heffter. *The JIVE System Implementation Description*, 2000. Available at http://softech.informatik.uni-kl.de/old/en/publications/jive.html.

[14] César Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-1, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1999.

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

[16] 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.

[17] Sam Owre and Harald Rueß. Integrating WS1S with PVS. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, Volume 1855 of Springer-Verlag *Lecture Notes in Computer Science*, pages 548–551, Chicago, IL, July 2000.

[18] Sam Owre and N. Shankar. Writing PVS proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pages 1–15, Hampton, VA, September 2003. Available at http://research.nianet.org/fm-at-nia/STRATA2003/.

[19] 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. Available at http://pvs.csl.sri.com/papers/AgExample/.

[20] N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, Volume 1166 of Springer-Verlag *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996.

[21] 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.

[22] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, Volume 2031 of Springer-Verlag *Lecture Notes in Computer Science*, pages 299–312, Genova, Italy, April 2001.