PVS: A Prototype Verification System

Reprint from: Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Saratoga, NY, 1992, pages 748–752; Volume 607 of Springer Verlag *Lecture Notes in Artificial Intelligence*.

S. Owre, J. M. Rushby, and N. Shankar
SRI International Computer Science Laboratory
Menlo Park, CA 94025 USA

1 Introduction

PVS is a prototype system for writing specifications and constructing proofs. Its development has been shaped by our experiences studying or using several other systems[1] and performing a number of rather substantial formal verifications (e.g., [5,6,8]). PVS is fully implemented and freely available. It has been used to construct proofs of nontrivial difficulty with relatively modest amounts of human effort. Here, we describe some of the motivation behind PVS and provide some details of the system.

Automated reasoning systems typically fall in one of two classes: those that provide powerful automation for an impoverished logic, and others that feature expressive logics but only limited automation. PVS attempts to tread the middle ground between these two classes by providing mechanical assistance to support clear and abstract specifications, and readable yet sound proofs for difficult theorems. Our goal is to provide mechanically-checked specifications and proofs that contribute to the social process by which purported theorems come to be discarded or accepted, and designs for critical systems get certified.

PVS combines an expressive logic with a powerful but highly interactive proof checker that supports top-down proof exploration and construction. In addition to its proof checker, the PVS system includes a parser, prettyprinter, and typechecker. We describe the PVS logic and proof checker in the following sections.

[1] Lack of space prevents us from discussing or explicitly referencing the many systems and notations that have influenced us in one way or another. These include Affirm, Automath, EHDM, EKL, EVES, FDM, Gypsy, HOL, IMPLY, IMPS, LCF, LP, Muse, Nqthm, Nuprl, OBJ, Ontic, PC-Nqthm, RAISE, RRL, STP, TPS, Tecton, VDM, Veritas, and Z among others. Most of our ideas can be traced to one or other of these earlier efforts.

2 The PVS Specification Logic

The philosophy behind the PVS logic has been to exploit mechanization in order to augment the expressiveness of the logic. PVS features a strongly typed higher-order logic with a rich type system. Higher-order logic was chosen since we and others have found it conducive to the construction of compact and perspicuous specifications. Strong typing is needed to keep higher-order logic consistent, but typechecking is also a simple and effective way to discover very many errors in specifications.

PVS specifications are structured into parameterized theories that can have constraints attached to the parameters. Constraints can also be attached to the types in a PVS specification. These choices make it possible to be very explicit about allowed instantiations of theories, and about the domains and ranges of functions, thereby contributing to the clarity of the specification. The price paid is that typechecking in PVS is not algorithmically decidable: it can require theorem proving to establish that expressions satisfy the constraints attached to parameters and types. However, the inference mechanisms of PVS perform most of the necessary theorem proving automatically, and thereby allow an enriched specification language at little cost.

For instance, the division operation is typed so that it is constrained to nonzero divisors. Constraints are attached to types in PVS using *predicate subtypes*, so that the signature for division can be given as:

```
nonzero : TYPE = {x : rational | x /= 0}
/ :  [rational, nonzero -> rational]
```

where `rational` is the (built-in) type of rational numbers, and `nonzero` is defined here to be the subtype of the rational numbers different from zero. When the PVS typechecker is invoked on the formula:

```
x /= y IMPLIES (y - x)/(x - y) < 0
```

it recognizes that the divisor expression `(x - y)` must be shown to be nonzero and generates a proof obligation (known as a *type correctness condition*) of the form:

```
(FORALL (y, x: rational): x /= y IMPLIES (x - y) /= 0)
```

Notice that the logical context in which `(x - y)` occurred appears as part of the hypothesis to the proof obligation. The PVS decision procedures are powerful enough to automatically discharge the large majority of proof obligations, such as the one above; more difficult ones must be proved under user-guidance, and can be postponed until convenient.

PVS also has a mechanism for automatically generating theories for abstract datatypes that generalizes the *shell* principle of the Boyer-Moore prover [1]. The PVS type system includes numbers, enumerations and uninterpreted types, together with tuple, record, and function types, and also dependent forms of these type constructions. Numerous other features of the language are omitted from this brief description.

Just as the use of powerful inference procedures during typechecking allows the specification language to be enriched, so, conversely, do several of the features of the specification language contribute to the effectiveness of the proof checker: constructions such as abstract datatype definitions, predicate subtypes, and dependent types supply constraints that can be used effectively by the inference mechanisms. Thus we find a synergistic interaction between the language and inference capabilities.

3 The PVS Proof Checker

Our experience with mechanical verification of complex designs and algorithms has led us to conclude that, just as with software, there is a lifecycle to a mechanically-checked proof. In the initial *exploratory* phase of proof development, we are mainly interested in debugging the specification and putative theorems, and in testing and revising the key, high-level ideas in the proof. An important requirement in this phase is early and useful feedback when a purported theorem is, in fact, false. Once the basic intuitions have been acquired and the formalization is stable, the proof checking enters a *development* phase where we take care of the details and construct the proof in larger leaps. Efficiency of proof development is a key requirement here. In the third, *presentation* phase, the proof is honed and polished for presentation in order to be scrutinized by the social process. Readability and intellectual perspicuity of the output is the goal here. The final phase is *generalization* where we carefully analyze the finished proof, weaken and generalize the assumptions, extract the key insights and proof techniques, and make it easier to carry out subsequent verifications of a similar nature. *Maintenance*, is a special application of generalization, where we adapt a verification to slightly changed assumptions or requirements. Robustness of the proof procedure is a useful attribute here.

The goal of the PVS proof checker is to support the efficient development of readable proofs in all stages of the proof development lifecycle. The PVS proof checker implements a small set of powerful primitive inference rules, a mechanism for composing these into proof strategies, a facility for rerunning proofs, and another to check that all secondary proof obligations (such as type correctness conditions) have been discharged. The first two of these are described below.

Representation. A sequent representation is used for proof goals since it nicely encapsulates all the information that is relevant to a branch of the proof for presentation to a user as well as the machine.

Goal-directed Proof Search. A proof is constructed by starting from the conclusion sequent and progressively applying the inference steps to generate subgoals until the subgoals are trivially provable. This makes it easy to present the proof as it is being developed.

Primitive Inferences. The inference steps in PVS were chosen to be powerful in comparison with the simple rules given in textbook introductions to logic. Powerful primitive inferences make the composed inference steps correspondingly more powerful, and allow the proof to be represented in a manner that is robust and can be rerun efficiently. A small and carefully chosen set of primitive inferences makes the system easier to learn and use. Each inference step is flexible, so it can be used in a variety of related ways, and takes optional parameters that adjust its behavior. For example, the beta reduction rule eliminates all redexes (and for flexibility many things are regarded as redexes) from a set of sequent formulas specified by a parameter (the default is all formulas).

The primitive inferences in PVS include the propositional and quantifier steps, beta-reduction, equality replacement, and the use of decision procedures and lemmas. The propositional rules of inference include a propositional axiom rule, a disjunctive simplification rule, a conjunctive splitting rule, the Cut rule for introducing case splits, a rule for lifting IF-conditionals to the top level of a formula (to enable the corresponding case splits), and a rule for deleting formulas from a goal sequent (the weakening rule). The quantifier rules consist of a rule for replacing universally quantified variables with Skolem constants, and a rule for instantiating existentially quantified variables with terms. The equality rules include a rule for beta-reducing redexes, and one for replacing one side of an equality premise by another.

In addition to the rules above, there are rules for introducing an instance of a lemma as a premise formula in a goal sequent, for introducing an extensionality axiom for a given type as a premise formula, for introducing the type constraints of a given expression as premise formulas, for invoking the decision procedures on a goal sequent, and for enabling and disabling the automatic use of rewrite rules.

Decision Procedures. Formal proofs of even trivial facts can be quite difficult to construct, and the typical user is seldom curious about the trivial details. Decision procedures help to automatically discharge such trivial subgoals. PVS uses decision procedures for ground equalities and linear inequalities (based on the work of Shostak [9]) in order to simplify IF-expressions, datatype expressions, function definitions, and conditions of conditional rewrite rules. As a result, the user usually only sees the relevant case of a large definition and often never has to deal with the conditions of a conditional rewrite rule. Through this use of decision procedures, there are fewer cases to a proof and the sequents themselves are kept to manageable size.

Proof Strategies. It is useful to be able to compose frequently used patterns of proofs into single steps. We call these *strategies* in PVS. Typical PVS strategies are propositional simplification, which applies all the proposition proof rules and returns with the remaining subgoals. A more powerful version of this strategy employs the decision procedures as well. Rewriting with a definition or lemma is also described by a strategy. We have implemented a simple language with two basic constructs for describing strategies. The `(IF condition strategy1 strategy2)` construct evaluates the `condition` against the current proof goal and applies `strategy1` or `strategy2` accordingly. The `(TRY strategy1 strategy2 strategy3)` applies `strategy1` to the current goal and if that succeeds, then `strategy2` is applied to the resulting subgoals, otherwise `strategy3` is applied to the current proof goal. These simple constructs can be combined to achieve the effects of *tactics* and *tacticals* in LCF-style systems [3]. For example, the propositional simplification strategy `(prop*)` has the form

```
(prop*) = (TRY (propax) nil (TRY (dsimp) (prop*) (TRY (split) (prop*) nil)))
```

where `nil` indicates that there are no further steps and the current goal should be postponed, `(propax)` checks if the current goal is propositional axiom, and `(dsimp)` and `(split)` represent the primitive inference steps for disjunctive simplification and conjunctive splitting, respectively.

4 Experience and Prospects

Only a few modest-sized example verifications have so far been carried out using PVS; these include the correctness of the Boyer-Moore majority algorithm [2], the proof that insertion into an ordered binary tree preserves order, the *Oral Messages* Algorithm for Byzantine Agreement [4], Cantor's theorem, the Schröder-Bernstein theorem, and the equivalence of a pipelined and an unpipelined microprocessor design [7]. All of these examples took on the order of a day or less of human effort to verify on the first attempt using PVS. The time taken to rerun finished proofs is on the order of minutes.

Future Work. Although we are reasonably satisfied that PVS is an effective tool for developing readable specifications and formal proofs with considerable human efficiency, we are still significantly short of our goal of employing mechanization to produce proofs that humans find truly perspicuous. We would also like to extract robust and reusable proof outlines from individual proofs. We plan to enhance the expressive power of the specification language by introducing structural subtypes, inductive definitions, and refinement mappings between theories. We also plan to define powerful higher-level proof strategies to further mechanize proof construction, and enhance the user-interface to the system.

Acknowledgements. Friedrich von Henke (SRI, currently at U. of Ulm, Germany), David Cyrluk (Stanford), Judy Crow (SRI), Steven Phillips (Stanford), Carl Witty (currently at MIT), contributed to the design, implementation, and testing of PVS. We also thank Mark Moriconi, Director of the SRI Computer Science Laboratory, for his support. Development of PVS was funded entirely by SRI International.

References

[1] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[2] Robert S. Boyer and J Strother Moore. MJRTY—a fast majority vote algorithm. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, volume 1 of *Automated Reasoning Series*, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[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] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[5] John Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In Vytopil [10], pages 237–257.

[6] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. In *SIGSOFT '91: Software for Critical Systems*, pages 1–15, New Orleans, LA, December 1991. Expanded version to appear in *IEEE Transactions on Software Engineering*, January 1993.

[7] James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. Technical Report 78, DEC Systems Research Center, Palo Alto, CA, September 1991.

[8] Natarajan Shankar. Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization. In Vytopil [10], pages 217–236.

[9] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[10] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Nijmegen, The Netherlands, January 1992. Springer Verlag, volume 571 of *Lecture Notes in Computer Science*.