A Tutorial on Using PVS for Hardware Verification*

S. Owre, J. M. Rushby, N. Shankar and M. K. Srivas
{owre, rushby, shankar, srivas}@csl.sri.com

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA

Abstract. PVS stands for "Prototype Verification System." It consists of a specification language integrated with support tools and a theorem prover. PVS tries to provide the mechanization needed to apply formal methods both rigorously and productively.
This tutorial serves to introduce PVS and its use in the context of hardware verification. In the first section, we briefly sketch the purposes for which PVS is intended and the rationale behind its design, mention some of the uses that we and others are making of it. We give an overview of the PVS specification language and proof checker. The PVS language, system, and theorem prover each have their own reference manuals, [1,2,3] which you will need to study in order to make productive use of the system. A pocket reference card, summarizing all the features of the PVS language, system, and prover is also available.
The purpose of this tutorial is not to describe in detail the features of PVS and how to use the system. Rather, its purpose is to introduce some of the more unique and powerful capabilities that are provided by PVS and demonstrate how these features can be used in the context of hardware verification. We present completely worked out proofs of two hardware examples. One of the examples is a pipelined microprocessor that has been used as benchmark for model checkers and the other is a parameterized implementation of an N-bit ripple-carry adder.

1 Introducing PVS

PVS stands for "Prototype Verification System." It consists of a specification language integrated with support tools and a theorem prover. PVS tries to pro-

* This tutorial was presented at the Second International Conference on Theorem Provers in Circuit Design, Theory, Practice, and Experience, Bad Herrenalb, Germany, September 26-28, 1994.

[1] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[2] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[3] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

vide the mechanization needed to apply formal methods both rigorously and productively.

The specification language of PVS is a higher-order logic with a rich type-system, and is quite expressive; we have found that most of the mathematical and computational concepts we wish to describe can be formulated very directly and naturally in PVS. Its theorem prover, or proof checker (we use either term, though the latter is more correct), is both interactive and highly mechanized: the user chooses each step that is to be applied and PVS performs it, displays the result, and then waits for the next command. PVS differs from most other interactive theorem provers in the power of its basic steps: these can invoke decision procedures for arithmetic and equality, a BDD-based propositional simplifier, efficient hashing-based automatic conditional rewriting, induction, and other relatively large units of deduction; it differs from other highly automated theorem provers in being directly controlled by the user. We have been able to perform some significant new hardware verification exercises quite economically using PVS; we have also repeated some verifications first undertaken in other systems and have usually been able to complete them in a fraction of the original time (of course, these are previously solved problems, which makes them much easier for us than for the original developers).

PVS is the most recent in a line of specification languages, theorem provers, and verification systems developed at SRI, dating back over 20 years. That line includes the Jovial Verification System [13], the Hierarchical Development Methodology (HDM) [25,26], STP [30], and EHDM [22,27]. We call PVS a "Prototype Verification System," because it was built partly as a lightweight prototype to explore "next generation" technology for EHDM, our main, heavy-weight, verification system. Another goal for PVS was that it should be freely available, require no costly licenses, and be relatively easy to install, maintain, and use. Development of PVS was funded entirely by SRI International.

The purpose of this tutorial is not to describe in detail the features of PVS and how to use the system. Rather, its purpose is to introduce some of the more unique and powerful capabilities that are provided by PVS and demonstrate how these features can be used in the context of hardware verification. We present completely worked out proofs of two hardware examples. One of the examples is a pipelined microprocessor that has been used as benchmark for testing the capacity of model checkers to handle datapath-oriented circuits. While the size of the datapath is irrelevant in a theorem proving exercise, we wanted to see if the proof would go through just as automatically as in a model checker. The second example is one of the circuits supplied as a TPCD benchmark: a parameterized implementation of an N-bit ripple-carry adder. The second example illustrates proof by induction.

1.1 Design Goals for PVS

The design of PVS was shaped by our experience in doing or contemplating early-lifecycle applications of formal methods. Many of the larger examples we have done concern algorithms and architectures for fault-tolerance (see [23] for an overview). We found that many of the published proofs that we attempted to check were in fact, incorrect, as was one of the important algorithms. We have also found that many of our own specifications are subtly flawed when first written. For these reasons, PVS is designed to help in the detection of errors as well as in the confirmation of "correctness." One way it supports early error detection is by having a very rich type-system and correspondingly rigorous typechecking. A great deal of specification can be embedded in PVS types (for example, the invariant to be maintained by a state-machine can be expressed as a type constraint), and typechecking can generate proof obligations that amount to a very strong consistency check on some aspects of the specification.

Another way PVS helps eliminate certain kinds of errors is by providing very rich mechanisms for conservative extension—that is, definitional forms that are guaranteed to preserve consistency. Axiomatic specifications can be very effective for certain kinds of problem (e.g., for stating assumptions about the environment), but axioms can also introduce inconsistencies—and our experience has been that this does happen rather more often than one would wish. Definitional constructs avoid this problem, but a limited repertoire of such constructs (e.g., requiring everything to be specified as a recursive function) can lead to excessively constructive specifications: specifications that say "how" rather than "what." PVS provides both the freedom of axiomatic specifications, and the safety of a generous collection of definitional and constructive forms, so that users may choose the style of specification most appropriate to their problems.[4]

The third way that PVS supports error detection is by providing an effective theorem prover. The design rationale behind the PVS theorem prover was to provide automatic support for obvious and tedious parts of a proof while giving the user the ability to guide the prover at higher levels of a proof. This goal is accomplished by implementing the primitive inference steps of PVS using automatic rewriting and efficient decision procedures for arithmetic and propositional logic. This approach makes PVS an effective system for hardware verification since most hardware proofs need significant amount of rewriting and case analyses.

Our experience has been that the act of trying to prove properties about specifications is the most effective way to truly understand their content and to identify errors. This can come about incidentally, while attempting to prove a "real" theorem, such as that an algorithm achieves its purpose, or it can be done deliberately through the process of "challenging" specifications as part of a validation process. A challenge has the form "if this specification is right, then the following ought to follow"—it is a test case posed as a putative theorem; we "execute" the specification by proving theorems about it.[5]

[4] Unlike EHDM, PVS does not provide special facilities for demonstrating the consistency of axiomatic specifications. We do expect to provide these in a later release, but using a different approach than EHDM.

[5] Directly executable specification languages (e.g., [2,17]) support validation of spec-

1.2 Uses of PVS

PVS has so far been applied to several small demonstration examples, and a growing number of significant verifications. The smaller examples include the specification and verification of ordered binary tree insertion [28], the Boyer-Moore majority algorithm, an abstract pipelined processor, Fischer's real-time mutual exclusion protocol, and the Oral Messages protocol for Byzantine agreement. Examples of this scale can typically be completed within a day. More substantial examples include the correspondence between the programmer and RTL level of a simple hardware processor [11], the correctness of a real-time railroad crossing controller [29], a variant of the Schröder-Bernstein theorem, and the correctness of a distributed agreement protocol for a hybrid fault model consisting of Byzantine, symmetric, and crash faults [19]. These harder examples can take from several days to a week.

Currently, PVS is being applied to the requirements specification of selected aspects of the control software for NASA's space shuttle project and to verify a commercial pipelined microprocessor, AAMP5, being built for avionics applications at Rockwell International.

2 The PVS Language

The PVS specification language builds on a classical typed higher-order logic. The base types consist of booleans, real numbers, rationals, integers, natural numbers, lists, and so forth. The primitive type constructors include those for forming function (e.g., `[nat -> nat]`), record (e.g., `[# a : nat, b : list[nat]#]`), and tuple types (e.g., `[int, list[nat]]`). PVS departs from simply typed logics by allowing *predicate subtypes*. A predicate subtype consists of exactly those elements of a given type satisfying a given predicate so that, for example, the subtype of positive numbers is given by the type {`n : nat | n > 0`}. Predicate subtypes are used to explicitly constrain the domains and ranges of operations in a specification and to define partial functions, e.g., division, as total functions on a specified subtype. In general, typechecking with predicate subtypes is undecidable.[6] PVS contains a further useful enrichment to the type system in the form of *dependent* function, record, and tuple constructions where the type of one component of a compound value depends on

ifications by running conventional test cases. We think there can be merit in this approach, but that it should not compromise the effectiveness of the specification language as a tool for deductive analysis; we are considering supporting an executable subset within PVS.

[6] PVS does have an algorithmic typechecker that checks for type correctness relative to the simple types. It generates proof obligations corresponding to predicate subtypes. The typical proof obligations can be automatically discharged by the PVS decision procedures. The provability of such proof obligations is the only source of undecidability in the PVS type system so that none of the benefits of decidable typechecking are lost.

the value of another component. PVS terms include constants, variables, abstractions (e.g., `(LAMBDA (i : nat): i * i)`), applications (e.g., `mod(i, 5)`), record constructions (e.g., `(# a := 2, b := cons(1, null) #)`), tuple constructions (e.g., `(-5, cons(1, null))`), function updates (e.g., `f WITH [(2) := 7]`), and record updates (e.g., `r WITH [a := 5, b := cons(3, b(r))]`). PVS specifications are packaged as *theories* that can be parametric in types and constants. Type parametricity (or *polymorphism*) is used to capture those concepts or results that can be stated uniformly for all types. PVS also has a facility for automatically generating abstract datatype theories (containing recursion and induction schemes) for a class of abstract datatypes [28].

3 The PVS Proof Checker

The central design assumptions in PVS are that

- The purpose of an automated proof checker is not merely to prove theorems but also to provide useful feedback from failed and partial proofs by serving as a rigorous skeptic.
- Automation serves to minimize the tedious aspects of formal reasoning while maintaining a high level of accuracy in the book-keeping and formal manipulations.
- Automation should also be used to capture repetitive patterns of argumentation.
- The end product of a proof attempt should be a proof that, with only a small amount of work, can be made humanly readable so that it can be subjected to the *social process* of mathematical scrutiny.

In following these design assumptions, the PVS proof checker is more automated than a low-level proof checker such as AUTOMATH [12], LCF [15], Nuprl [7], Coq [8], and HOL [16], but provides more user control over the structure of the proof than highly automated systems such as Nqthm [3,4] and Otter [21]. We feel that the low-level systems over-emphasize the formal correctness of proofs at the expense of their cogency, and the highly automated systems emphasize theorems at the expense of their proofs.

What is unusual about PVS is the extent to which aspects of the language, the typechecker, and proof checker are intertwined. The typechecker invokes the proof checker in order to discharge proof obligations that arise from typechecking expressions involving predicate subtypes or dependent types. The proof checker also makes heavy use of the typechecker to ensure that all expressions involved in a proof are well-typed. This use of the typechecker can also generate proof obligations that are either discharged automatically or are presented as additional subgoals. Several aspects of the language, particularly the type system, are built into the proof checker. These include the automatic use of type constraints by the decision procedures, the simplifications given by the abstract datatype axioms, and forms of beta-reduction and extensionality.

Another less unusual aspect of PVS is the extent to which the automatic inference and decision procedures involving equalities and linear arithmetic inequalities are employed.[7] The most direct consequence of this is that the trivial, obvious, or tedious parts of the proof are often discharged so that the user can focus on the intellectually demanding parts of the proof, and the resulting proof is also easier to read. PVS also provides an efficient conditional rewriter that interacts very closely with its decision procedures to simplify conditions during rewriting. More details about the rewriting and the decision procedures used in PVS are described in [10]. The capabilities of the inference and decision procedures, which play a central role in almost all proofs in PVS are made available to the user by means of the following primitive inference steps.

1. `Bddsimp` performs efficient BDD-based propositional simplification on the current goal.
2. `Do-rewrite` performs automatic conditional rewriting on expressions in the current goal using rewrite rules stored in the underlying database used by the inference procedures. PVS provides several commands for the user to make rewrite rules out of definitions, lemmas and axioms and enter them in the database. The rewriter invokes the decision procedures to simplify conditions of conditional rewrite rules.
3. `Assert` invokes the arithmetic and equality decision procedures on the current goal. Besides trying to prove the subgoal using the decision procedures, it performs the following tasks
   - it stores the subgoal information in the underlying database, allowing automatic use to be made of it later.
   - it simplifies the subgoal using the decision procedures using rewriting as well as other simplification techniques.

In order to learn how to use the PVS proof checker, one must first understand the sequent representation used by PVS to represent proof goals, the commands used to move around and undo parts of the proof tree, and the commands used to get help. One must then understand the syntax and effects of proof commands used to build proofs. Many of these commands are extremely powerful even in their simplest usage. Several of these commands can be more carefully directed by supplying them with one or more optional arguments. The advanced user will also need to understand how to define proof strategies that capture repetitive patterns of proof commands, and commands used for displaying, editing, and replaying proofs. There are about 20 basic commands and a similar number of commonly used high-level strategies.

[7] The Ontic system [20] is a proof checker where decision procedures are ubiquitously used. Nqthm [3,4], Eves [24], and IMPS [14] also rely heavily on the use of decision procedures.

4 Rest of the Tutorial

In the following sections we introduce some of the details of PVS system by working the complete proof of correctness of two examples. This will introduce some of the most useful commands and provide a glimpse into the philosophy behind PVS. PVS uses EMACS as its interface by extending EMACS with PVS functions, but all the underlying capabilities of EMACS are available. Thus the user can read mail and news, edit nonPVS files, or execute commands in a shell buffer in the usual way. All PVS commands are entered as extended EMACS commands. The proof checker runs as a subprocess inside EMACS.

5 A Pipelined Microprocessor

In this section we develop a complete proof of a correctness property of the controller logic of a simple pipelined processor design described at a register-transfer level. The design and the property verified are both based on the processor example given in [5]. The example has been used as a benchmark for evaluating how well finite state-enumeration based tools, such as model checkers, can handle datapath-oriented circuits with a large number of states by varying the size of the datapath. From the perspective of a theorem prover, the size of the datapath is irrelevant because the specification and proof are independent of the datapath size. As a theorem proving exercise, the challenge is to see if the proof can be done just as automatically as a model checker. As we will see in the following, in PVS the proof can be obtained by repeatedly invoking one of its primitive commands `assert`.

5.1 Informal Description

Figure 1 shows a block diagram of the pipeline design. The processor executes instructions of the form `(opcode src1 src2 dstn)`, i.e., "destination register `dstn` in the register file `REGFILE` becomes some `ALU` function determined by `opcode` of the contents of source registers `src1` and `src2`. Every instruction is executed in three stages (cycles) by the processor:

1. *Read*: Obtain the proper contents of the register file at `src1` and `src2` and clock them into `opreg1` and `opreg2`, respectively.
2. *Compute*: Perform the ALU operation corresponding to the opcode (remembered in `opcoded`) of the instruction and clock the result into `wbreg`.
3. *Write*: Update the register file at the destination register (remembered in `dstndd`) of the instruction with the value in `wbreg`.

The processor uses a three-stage pipeline to simultaneously execute distinct stages of three successive instructions. That is, the read stage of the current instruction is executed along with the compute stage of the previous instruction

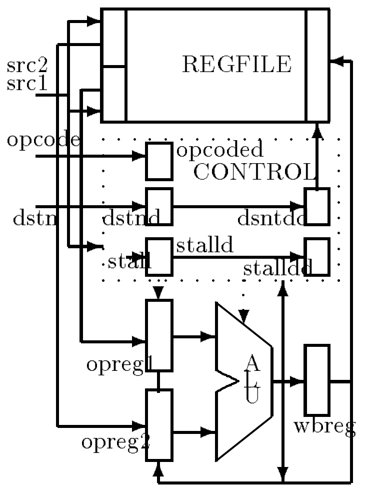

Fig. 1. A Pipelined Microprocessor

and the write stage of the previous-to-previous instruction. Since the `REGFILE` is not updated with the results of the previous and previous-to-previous instructions while a read is being performed for the current instruction, the controller "bypasses" `REGFILE`, if necessary, to get the correct values for the read. The processor can abort, i.e., treat as `NOP`, the instruction in the read stage by asserting the `stall` signal true. An instruction is aborted by inhibiting its write stage by remembering the `stall` signal until the write stage via the registers `stalld` and `stalldd`. We verify that an instruction entering the pipeline at any time gets completed correctly, i.e., will write the correct result into the register file, three cycles later, provided the instruction is not aborted.

5.2 Formal Specification

PVS specifications consist of a number of files, each of which contains one or more theories. A theory is a collection of declarations: types, constants (including functions), axioms that express properties about the constants, and theorems and lemmas to be proved. Theories may import other theories; Every entity used in a theory must be either declared in an imported theory or be part of the prelude (the standard collection of theories built-in to PVS).

The microprocessor specification is organized into three theories, selected parts of which are shown in Figures 2 and 3. (The complete specification can be found in [31].) The theory `pipe` (Figure 2) contains a specification of the design and a statement of the correctness property to be proved. The theories `signal` and `time` (Figure 3) imported by `pipe` declares the types `signal` and `time` used in `pipe`.

The theory `pipe` is parameterized with respect to the types of the register address, data, and the opcode field of the instructions. A theory parameter in

```
pipe[addr: TYPE, data: TYPE, opcodes: TYPE]: THEORY
  BEGIN
   IMPORTING signal, time

   ASSUMING
    addr_nonempty: ASSUMPTION (EXISTS (a: addr): TRUE)
    data_nonempty: ASSUMPTION (EXISTS (d: data): TRUE)
    opcodes_nonempty: ASSUMPTION (EXISTS (o: opcodes): TRUE)
   ENDASSUMING

   t: VAR time

   %% Signal declarations
      opcode: signal[opcodes]
      src1, src2, dstn: signal[addr]
      stall: signal[bool]
      aluout: signal[data]
      regfile: signal[[addr -> data]]
       ...

    %% Specification of constraints on the signals
      dstnd_ax: AXIOM dstnd(t+1) = dstn(t)
      dstndd_ax: AXIOM dstndd(t+1)= dstnd(t)
      .....

      regfile_ax: AXIOM regfile(t+1) =
                         IF stalldd(t) THEN regfile(t)
                         ELSE regfile(t)
                              WITH [(dstndd(t)) := wbreg(t)]
                         ENDIF
      opreg1_ax: AXIOM opreg1(t+1) =
                    IF src1(t) = dstnd(t) & NOT stalld(t)
                       THEN aluout(t)
                    ELSIF src1(t) = dstndd(t) & NOT stalldd(t)
                          THEN wbreg(t)
                    ELSE regfile(t)(src1(t)) ENDIF
       opreg2_ax: AXIOM ...

       aluop: [opcodes, data, data -> data]
       ALU_ax: AXIOM aluout(t) = aluop(opcoded(t), opreg1(t),
                                 opreg2(t))
      correctness: THEOREM (FORALL t:
        NOT(stall(t)) IMPLIES regfile(t+3)(dstn(t)) =
                     aluop(opcode(t), regfile(t+2)(src1(t)),
                                 regfile(t+2)(src2(t))) )
  END pipe
```

Fig. 2. Microprocessor Specification

PVS can be either a type parameter or a parameter belonging to a particular type, such as `nat`. Since `pipe` does not impose any restriction on its parameters, other than the requirement that they be nonempty, which is stated in the `ASSUMING` part of the theory, one can instantiate them with any type. Every entity declared in a parameterized theory is implicitly parameterized with respect to the parameters of the theory. For example, the type `signal` declared in the parameterized theory `signal` is a parametric type denoting a function that maps `time` (a synonym for `nat`) to the type parameter `T`. (The type `signal` is used to model the wires in our design.) By importing the theory `signal` uninstantiated

```
signal[val: TYPE]: THEORY
  BEGIN
   signal: TYPE = [time -> val]
  END signal

time: THEORY
  BEGIN
   time: TYPE nat
  END signal
```

Fig. 3. Signal Specification

in `pipe`, we have the freedom to create any desired instances of the type `signal`.

In this tutorial, we use a *functional* style of specification to model register-transfer-level digital hardware in logic. In this style, the inputs to the design and the outputs of every component in the design are modeled as signals. Every signal that is an output of a component is specified as a function of the signals appearing at the inputs to the component.

This style should be contrasted with a *predicative* style, which is commonly used in most HOL applications. In the predicative style every hardware component is specified as a predicate relating the input and output signals of the component and a design is specified as a conjunction of the component predicates, with all the internal signals used to connect the components hidden by existential quantification. A proof of correctness for a predicative style specification usually involves executing a few additional steps at the start of the proof to essentially transform the predictative specification into an equivalent functional style. After that, the proof proceeds similar to that of a proof in a functional specification. The additional proof steps required for a predicative specification essentially unwind the component predicates using their definitions and then appropriately instantiate the existentially quantified variables. An automatic way of performing this translation is discussed in [31], which illustrates more examples of hardware design verification using PVS.

Getting back to our example, the microprocessor specification in `pipe` consists of two parts. The first part declares all the signals used in the design—the inputs to the design and the internal wires that denote the outputs of components. The composite state of `REGFILE`, which is represented as a function from `addr` to `data`, is modeled by the signal `regfile`. The signals are declared as uninterpreted constants of appropriate types. The second part consists of a set of AXIOMs that specify the the values of the signals over time. (To conserve space, we have only shown the specification of a subset of the signals in the design.) For example, the signal value at the output of the register `dstnd` at time `t+1` is defined to be that of its input a cycle earlier. The output of the ALU, which is a combinational component, is defined in terms of the inputs at the same time instant.

In PVS, one can use a descriptive style of definition, as illustrated in this example, by selectively introducing properties of the constants declared in a theory as AXIOMs. Or, one can use the definitional forms provided by the language to define the constants. An advantage of using the definitions is that a specification is guaranteed to be consistent, although it might be overspecified. An advantage of the descriptive style is that it gives better control over the degree to which one wants to define an entity. For example, one could have specified `dstnd` prescriptively by using the conventional function definition mechanism of PVS. PVS's function definition mechanism would have forced us to specify the value of the signal at time `t = 0` to ensure that the function is total. In the descriptive style used, we have left the value of the signal at `0` unspecified.

In the present example, the specifications of the signals `opreg1` and `opreg2` are the most interesting of all. They have to check for any register collisions that might exist between the instruction in the read stage and the instructions in the later stages and bypass reading from the register file in case of collisions. The `regfile` signal specification is recursive since the register file state remains the same as its previous state except, possibly, at a single register location. The `WITH` expression is an abbreviation for the result of updating a function at a given point in the domain value with a new value. Note that the function `aluop` that denotes the operation ALU performs for a given `opcode` is left completely unspecified since it is irrelevant to the controller logic.

The theorem `correctness` to be proved states a correctness property about the execution of the instruction that enters the pipeline at `t`, provided the instruction is not aborted, i.e., `stall(t)` is not true. The equation in the conclusion of the implication compares the actual value (left hand side) in the destination register three cycles later, when the result of the instruction would be in place, with the expected value. The expected value is the result of applying the `aluop` corresponding to the opcode of the instruction to the values at the source field registers in the register file at `t+2`. We use the state of the register file at `t+2` rather than `t` to allow for the results of the two previous instructions in the pipeline to be completed.

5.3 Proof of Correctness

The next step is to typecheck the file, which parses and checks for semantic errors, such as undeclared names and ambiguous types. Typechecking may build new files or internal structures such as *type correctness conditions* (*TCCs*). The TCCs represent *proof obligations* that must be discharged before the `pipe` theory can be considered typechecked. The typechecker does not generate any TCCs in the present example. If, for example, one of the assumptions, say for `addr`, in the `ASSUMING` part of the theory was missing, the typechecker would generate the following TCC to show that the `addr` type is nonempty. The declaration of the signal `src1` forces generation of this TCC because a function is nonexistent if its range is empty.

```
% Existence TCC generated (line 17) for src1: signal[addr]
% May need to add an assuming clause to prove this.
  % unproved
src1_TCC1: OBLIGATION (EXISTS (x1: signal[addr]): TRUE);
```

The PVS proof checker runs as a subprocess of Emacs. Once invoked on a theorem to be proved, it accepts commands directly from the user. The basic objective of developing a proof in PVS as in other subgoal-directed proof checkers (e.g., HOL), is to generate a *proof tree* in which all of the leaves are trivially true. The nodes of the proof tree are sequents, and while in the prover you will always be looking at an unproved leaf of the tree. The *current* branch of a proof is the branch leading back to the root from the current sequent. When a given branch is complete (i.e., ends in a true leaf), the prover automatically moves on to the next unproved branch, or, if there are no more unproven branches, notifies you that the proof is complete.

The primitive inference steps in PVS are a lot more powerful than in HOL. So, it is not necessary to build complex tactics to handle tedious lower level proofs in PVS. A user knowledgeable in the ways of PVS can typically get proofs to go through mostly automatically by making a few critical decisions at the start of the proof. However, PVS does provide the user with the equivalent of HOL's tacticals, called *strategies*, and other features to control the desired level of automation in a proof.

The proof of the microprocessor property shown below follows a certain general pattern that works successfully for most hardware proofs. This general proof pattern, variants of which have been used in other verification exercises [1, 18], consists of the following sequence of general proof tasks.

Quantifier elimination: Since the decision procedures work on ground formulas, the user must eliminate the relevant universal quantifiers by skolemization or selecting variables on which to induct and existential quantifiers by suitable instantiation.

Unfolding definitions: The user may have to simplify selected expressions and defined function symbols in the goal by rewriting using definitions, axioms or lemmas. The user may also have to decide the level to which the function symbols have to rewritten.

Case analysis: The user may have to split the proof based on selected boolean expressions in the current goal and simplify the resulting goals further.

Each of the above tasks can be accomplished automatically using a short sequence of primitive PVS proof commands. The complete proof of the theorem is shown below. Selected parts of the proof session is reproduced below as we describe the proof.

```
1: ( then* (skosimp)
2:         (auto-rewrite-theory ‘‘pipe’’ :always? t)
3:         (repeat (do-rewrite))
4:         (apply (then* (repeat (lift-if))
5:                       (bddsimp)
6:                       (assert))))
```

In the proof, the names of strategies are shown in *italics* and the primitive inference steps in `type-writer font`. (We have numbered the lines in the proof for reference.) `Then*` applies the first command in the list that follows to the current goal; the rest of the commands in the list are then applied to each of the subgoals generated by the first command application. The `apply` command used in line 5 makes the application of a compound proof step implemented by a strategy behave as an atomic step.

The first goal in the proof session is shown below. It consists of a single formula (labeled {1}) under a dashed line. This is a *sequent*; formulas above the dashed lines are called *antecedents* and those below are called *succedents*. The interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the succedents.

```
correctness :

  |-------
{1}   (FORALL t: NOT (stall(t))
                 IMPLIES regfile(t + 3)(dstn(t)) =
                    aluop(opcode(t), regfile(t + 2)(src1(t)),
                          regfile(t + 2)(src2(t))))
```

The quantifier elimination task of the proof is accomplished by the command `skosimp`, which skolemizes all the universally quantified variables in a formula and flattens the sequent resulting in the following goal. Note that `stall(t!1)` has been moved to the succedent in the sequent because PVS displays every atomic formula in its positive form.

```
Rule? (skosimp)
Skolemizing and flattening, this simplifies to:
correctness :

  |-------
{1}   (stall(t!1))
{2}   regfile(t!1 + 3)(dstn(t!1))
        =
        aluop(opcode(t!1), regfile(t!1 + 2)(src1(t!1)),
              regfile(t!1 + 2)(src2(t!1)))
```

The next task—unfolding definitions—is performed by the commands in lines 2 through 3. PVS provides a number of ways of unfolding definitions ranging from unfolding one step at a time to automatic rewriting that performs unfolding in a brute-force fashion. Brute-force rewriting usually results in larger expressions than controlled unfolding and, hence, potentially larger number of cases to consider. If a system provides automatic and efficient rewriting and case analysis facilities, then one can dare to use the automatic approach, as we do here. In PVS automatic rewriting is performed by first entering the definitions and AXIOMs that one wants to be used for unfolding as rewrite rules. Once entered, the commands, such as `do-rewrite` and `assert`, that perform rewriting as part of their repertoire repeatedly apply the rewrite rules until none of the rules is applicable. To control the size of the expression resulting from rewriting and the potential for looping, the rewriter uses the following restriction for stopping a rewrite: If the right-hand-side of a rewrite is a conditional expression, then the rule is applied only if the condition simplifies to true or false.

Here our aim is to unfold every signal in the sequent so that every signal expression contains only the start time `t!1`. So, we make a rewrite rule out of every AXIOM in the theory `pipe` by means of the command `auto-rewrite-theory` on line 2. We also force an over-ride of the default restriction for stopping rewriting by setting the tag[8] `always?` to true in the `auto-rewrite-theory` command and embed `do-rewrite` inside a `repeat` loop to force maximum rewriting. In the present example, the rewriting is guaranteed to terminate because every feedback loop is cut by a sequential component.

At the end of automatic rewriting, the succedent we are trying to prove is in the form of an equation on two deeply nested conditional expressions as shown below in an abbreviated fashion. The various cases in conditional expression shown above arise as a result of the different possible conflicts between instructions in the pipeline. The equation we are trying to prove contains two distinct, but equivalent conditional expressions, as in `IF a THEN b ELSE c ENDIF = IF NOT a THEN c ELSE b ENDIF`, that can only be proved to be equal by performing a case-split on one or more of the conditions. While `assert` simplifies the leaves of a conditional expression assuming every condition along the path to the leaves holds, it does not split propositions. One way to perform the case-splitting task automatically is to "lift" all the `IF-THEN-ELSE`s to the top so that the equation is transformed into a propositional formula with unconditional equalities as atomic predicates. After performing such a lifting, one can try to reduce the resulting proposition to true using the propositional simplification command `bddsimp`. If `bddsimp` does not simplify the proposition to true, then it is most likely the case that equations at one or more of the leaves of the proposition need to be further simplified, by `assert`, for instance, using the conditions along the path. If the propositional formula does not reduce to true or false, `bddsimp`

[8] Tags are one of the ways in which PVS permits the user to modify the functionality of proof commands.

produces a set of subgoals to be proved. In the present case, each of these goals can be discharged by `assert`. The compound proof step appearing on lines 4 through 6 of the proof accomplishes the case-splitting task.

```
correctness :

  |-------
[1]   (stall(t!1))
{2}   aluop(opcode(t!1),
            IF src1(t!1) = dstnd(t!1) & NOT stalld(t!1)
              THEN aluop(opcoded(t!1), opreg1(t!1), opreg2(t!1))
            ELSIF src1(t!1) = dstndd(t!1) & NOT stalldd(t!1)
            THEN wbreg(t!1)
            ELSE regfile(t!1)(src1(t!1)) ENDIF,
            ....
            ENDIF)
        = aluop(opcode(t!1),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE regfile(t!1) WITH [(dstndd(t!1)) := wbreg(t!1)]
                ENDIF
              ELSE ...
              ENDIF(src1(t!1)),
              IF stalld(t!1) THEN IF stalldd(t!1) THEN regfile(t!1)
                ELSE ... ENDIF
              ELSE ...
              ENDIF(src2(t!1)))
```

We have found that the sequence of steps shown above works successfully for proving safety properties of finite state machines that relate states of the machine that are finite distance apart. If the strategy does not succeed then the most likely cause is that either the property is not true or that a certain property about some of the functions in the specification unknown to the prover need to be proved as a lemma. In either case, the unproven goals remaining at the end of the proof should give information about the probable cause.

6 An N-bit Ripple-Carry Adder

The second example we consider is the verification of a parametrized N-bit ripple-carry adder circuit. The theory `adder`, shown in Figure 4, specifies a ripple-carry adder circuit and a statement of correctness for the circuit.

The theory is parameterized with respect to the length of the bit-vectors. It imports the theories (not shown here) `full_adder`, which contains a specification of a full adder circuit (`fa_cout` and `fa_sum`), and `bv`, which specifies the bit-vector type (`bvec[N]`) and functions. An N-bit bit-vector is represented as an

```
adder[N: posnat] : THEORY
BEGIN
  IMPORTING bv[N], full_adder

  n: VAR below[N]
  bv, bv1, bv2: VAR bvec
  cin: VAR bool

  nth_cin(n, cin, bv1, bv2): RECURSIVE bool =
      IF n = 0 THEN cin
      ELSE fa_cout(nth_cin(n - 1, cin, bv1, bv2), bv1(n - 1), bv2(n - 1))
      ENDIF
    MEASURE n

  bv_sum(cin, bv1, bv2): bvec =
    (LAMBDA n: fa_sum(bv1(n), bv2(n), nth_cin(n, cin, bv1, bv2)))

  bv_cout(n, cin, bv1, bv2): bool =
    fa_cout(nth_cin(n, cin, bv1, bv2), bv1(n), bv2(n))

  adder_correct_n: LEMMA
      bvec2nat_rec(n, bv1) + bvec2nat_rec(n, bv2) + bool2bit(cin)
        = exp2(n + 1) * bool2bit(bv_cout(n, cin, bv1, bv2))
          + bvec2nat_rec(n, bv_sum(cin, bv1, bv2))

  adder_correct: THEOREM
      bvec2nat(bv1) + bvec2nat(bv2) + bool2bit(cin)
        = exp2(N) * bool2bit(bv_cout(N - 1, cin, bv1, bv2))
          + bvec2nat(bv_sum(cin, bv1, bv2))
END adder
```

Fig. 4. Adder Specification

array, i.e., a function, from the the type `below[N]`, a subtype of `nat` ranging from `0` through `N-1`, to `bool`; the index `0` denotes the least significant bit. Note that the parameter `N` is constrained to be a posnat since we do not permit bit vectors of length `0`. The `adder` theory contains several declarations including a set of variable declarations in the beginning. PVS allows logical variables to be declared globally within a theory so that the variables can be used later in function definitions and quantified formulas.

The carry bit that ripples through the full adder is specified recursively by means of the function `nth_cin`. Associated with this definition is a *measure* function, following the `MEASURE` keyword, which will be explained below. The function `bv_cout` and `bv_sum` define the carry output and the bit-vector sum of the adder, respectively. The theorem `adder_correct` expresses the conventional correctness statement of an adder circuit using `bvec2nat`, which returns the natural number equivalent of an N-bit bit-vector. Note that variables that are left free in a formula are assumed to be universally quantified. We state and prove a more general lemma `adder_correct_rec` of which `adder_correct` is an instance. For a given `n < N`, `bvec2nat_rec` returns the natural number equivalent of the least significant n-bits of a given bit-vector and `bool2bit` converts the boolean constants `TRUE` and `FALSE` into the natural numbers `1` and `0`, respectively.

6.1 Typechecking

The typechecker generates several TCCs (shown in Figure 5 below) for `adder`. These TCCs represent *proof obligations* that must be discharged before the `adder` theory can be considered typechecked. The proofs of the TCCs may be postponed until it is convenient to prove them, though it is a good idea to view them to see if they are provable.

```
% Subtype TCC generated (line 13) for n - 1
  % unproved
nth_cin_TCC1: OBLIGATION (FORALL n: NOT n = 0 IMPLIES n - 1 >= 0 AND n - 1 < N)

% Subtype TCC generated (line 31) for N - 1
  % unproved
adder_correct_TCC1: OBLIGATION N - 1 >= 0
```

Fig. 5. TCCs for Theory `adder`

The first TCC is due to the fact that the first argument to `nth_cin` is of type `below[N]`, but the type of the argument (`n-1`) in the recursive call to `nth_cin` is integer, since `below[N]` is not closed under subtraction. Note that the TCC includes the condition `NOT n = 0`, which holds in the branch of the `IF-THEN-ELSE` in which the expression `n - 1` occurs. A TCC identical to the this one is generated for each of the two other occurrences of the expression `n-1` because `bv1` and `bv2` also expect arguments of type `below[N]`. These TCCs are not retained because they are subsumed by the first one.

The second TCC is generated by the expression `N-1` in the definition of the theorem `adder_correct` because the first argument to `bv_cout` is expected to be the subtype `below[N]`.

There is yet another TCC that is internally generated by PVS but is not even included in the TCCs file because it can be discharged trivially by the typechecker, which calls the prover to perform simple normalizations of expressions. This TCC is generated to ensure that the recursive definition of `nth_cin` terminates. PVS does not directly support partial functions, although its powerful subtyping mechanism allows PVS to express many operations that are traditionally regarded as partial. The measure function is used to show that recursive definitions are total by requiring the measure to decrease with each recursive call. For the definition of `nth_cin`, this entails showing `n-1 < n`, which the typechecker trivially deduces.

In the present case, all the remaining TCCs are simple, and in fact can be discharged automatically by using the `typecheck-prove` command, which attempts to prove all TCCs that have been generated using a predefined proof strategy called `tcc`.

6.2 Proof of Adder_correct_n

The proof of the lemma uses the same core strategy as in the microprocessor proof except for the quantifier elimination step. Since the specification is recursive in the length of the bit-vector, we need to perform induction on the variable `n`. The user invokes an inductive proof in PVS by means of the command `induct` with the variable to induct on (`n`) and the induction scheme to be used (`below_induction[N]`) as arguments. The induction used in this case is defined in the PVS prelude and is parameterized, as is the type `below[N]`, with respect to the upper limit of the subrange.

This command generates two subgoals: the subgoal corresponding to the base case, which is the first goal presented to prove, is shown in Figure 6.

```
adder_correct.1 :

  |-------
1   (N > 0
        IMPLIES
        (FORALL
         (bv1: bvec[N], bv2: bvec[N], cin: bool):
           bvec2nat_rec(0, bv1) + bvec2nat_rec(0, bv2)
             + bool2bit(cin)
             = exp2(0 + 1) * bool2bit(bv_cout(0, cin, bv1, bv2))
                 + bvec2nat_rec(0, bv_sum(cin, bv1, bv2))))
```

Fig. 6. Base Step

The goal corresponding to the inductive case is shown below.

```
The remaining siblings are:
adder_correct_n.2 :

  |-------
{1}   (FORALL (r: below[N]):
        r < N - 1
          AND (FORALL (bv1, bv2: bvec[N]), (cin: bool):
                 bvec2nat_rec(r, bv1) + bvec2nat_rec(r, bv2)
                   + bool2bit(cin)
                   = exp2(r + 1) * bool2bit(bv_cout(r, cin, bv1, bv2))
                       + bvec2nat_rec(r, bv_sum(cin, bv1, bv2)))
          IMPLIES (FORALL (bv1, bv2: bvec[N]), (cin: bool):
                     bvec2nat_rec(r + 1, bv1)
                       + bvec2nat_rec(r + 1, bv2)
                       + bool2bit(cin)
                       = exp2(r + 1 + 1)
                           * bool2bit(bv_cout(r + 1, cin, bv1, bv2))
                           +
                           bvec2nat_rec(r + 1,
                                        bv_sum(cin, bv1, bv2))))
```

Fig. 7. Inductive Step

The base and the inductive steps can be proved automatically using essentially the same strategy used in the microprocessor proof. A complete proof of `adder_correct_n` is shown in Figure 7.

```
 1: ( spread (induct ``n'' 1 ``below_induction[N]'')
 2:   ( ( then*  (skosimp*)
 3:               (auto-rewrite-defs :always? t)
 4:               (do-rewrite)
 5:               ( repeat (lift-if))
 6:               ( apply ( then* (bddsimp)(assert))))
 7:     ( then* (skosimp*)
 8:              (inst?)
 9:              (auto-rewrite-defs :always? t)
10:              (do-rewrite)
11:              ( repeat (lift-if))
12:              ( apply ( then* (bddsimp)(assert))))))
```

The strategy *spread* used on line 1 applies the first proof step (**induct**) and then applies the i^{th} element of the list of commands that follow to the i^{th} subgoal resulting from the application of the first prof step. Thus, the proof steps listed on lines 2 through 6 prove the base case of induction, the steps on lines 7 through 12 prove the inductive case, and the proof step on line 13 takes care of the third TCC subgoal.

Let us consider the base case first. The **induct** command has already instantiated the variable `n` to `0`. The remaining variables are skolemized away by **skosimp***. To unfold the definitions in the resulting goal, we use the command **auto-rewrite-defs**, which makes rewrite rules out of the definition of every function either directly or indirectly used in the given formula. The rest of the proof proceeds exactly as for the microprocessor.

The proof of the inductive step follows exactly the same pattern except that we need to instantiate the induction hypothesis and use it in the process of unfolding and case-analysis. PVS provides a command **inst?** that tries to find instantiations for existential-strength variables in a formula by searching for possible matches between terms involving these variables with ground terms inside formulas in the rest of the sequent. This command finds the desired instantiations in the present case. The rest of the proof proceeds as in the basis case.

Since the inductive proof pattern shown above is applicable to any iteratively generated hardware designs we have packaged it into a general proof strategy called **name-induct-and-bddrewrite**. The strategy is parameterized with respect to induction scheme to be used and the set of rewrite rules to be used for unfolding. We have used the strategy to prove an N-bit ALU [6] that executes 12 microoperations by cascading N 1-bit ALU slices.

7 Summary

This tutorial gives an overview of some of the unique and important capabilities of PVS. PVS is built to combine a very expressive specification language with effective theorem proving to produce a system to apply formal methods productively. PVS does pay a performance penalty because of the need for the prover to invoke the typechecker more often than in other provers that support a less expressive type system. We are working on reducing the amount of type information that the prover needs to generate and maintain during a proof and also on further optimizing some of our inference procedures. These optimizations should be available with future releases of PVS. In the following, we summarize some of the language and system features that were not covered in the tutorial.

PVS provides a fairly extensive set of commands for determining the status of specification elements such as theories and formulas. For example, the user can inquire whether a theory has been typechecked or a proof has been completed and if the proof is current. It has commands that perform *proof chain analysis* to see the proof status of all the lemmas that a theorem is dependent on.

When a formal specification and verification is complete, it is usually desirable to present it to others in as readable a form as possible. PVS provides commands for generating Latex versions of the specifications and proofs that can be included in typeset documents. The output produced can be controlled by user-supplied tables so that mathematical notation, including infix and mis-fix symbols and sub and superscripts can be created easily.

An important language feature that we haven't illustrated here is the abstract data type feature. This feature is similar to the definitional principle supported by the Boyer-Moore theorem prover, but is generalized to abstract data types with arbitrary constructors. The system provides facility for automatically generating abstract data type theories (containing recursion and induction schemes) from a syntactic definition of the operations of the data type.

7.1 Getting and Using PVS

At the moment, PVS is readily available only for Sun SPARC workstations, although versions of the system do exist for the IBM Risc 6000 (under AIX) and DECSystems (under Ultrix). PVS is implemented in Common Lisp (with CLOS), and has been ported to Lucid and Allegro. All versions of PVS require GNU EMACS, which must be obtained separately.

PVS requires about 30 megabytes of disk space. In addition, any system on which it is to be run should have a minimum of 100 megabytes of swap space and 32 megabytes of real memory (more is better).

To obtain the PVS system, send a request to `pvs-request@csl.sri.com`, and we will provide further instructions for obtaining a tape or for getting the system by FTP. All installations of PVS must be licensed by SRI. A nominal distribution fee is charged for tapes; there is no charge for obtaining PVS by FTP.

References

1. Mark D. Aagard, Miriam E. Leeser, and Phillip J. Windley. Toward a super duper hardware tactic. In *Proceedings of the HOL User's Group Workshop*, pages 401–414, 1993.
2. Heather Alexander and Val Jones. *Software Design and Prototyping using* me too. Prentice Hall International, Hemel Hempstead, UK, 1990.
3. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.
4. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.
5. J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 2^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.
6. F. J. Cantu. Verifying an *n-bit* arithmetic logic unit. Blue book note 935, University of Edinburgh, June 1994.
7. R. L. Constable, *et al. Implementing Mathematics with the Nuprl*. Prentice-Hall, New Jersey, 1986.
8. T. Coquand and G. P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proceedings of EUROCAL 85, Linz (Austria)*, Berlin, 1985. Springer-Verlag.
9. Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.
10. D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, pages 287–305, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.
11. David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
12. N. G. de Bruijn. A survey of the project Automath. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.
13. B. Elspas, M. Green, M. Moriconi, and R. Shostak. A JOVIAL verifier. Technical report, Computer Science Laboratory, SRI International, January 1979.
14. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. Technical Report M90-19, Mitre Corporation, 1991.
15. 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.
16. M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, Dordrecht, The Netherlands, 1988.
17. Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods, and VDM*. International Computer Science Series. Addison-Wesley, Wokingham, England, 1988.

18. R. Kumar, K. Schneider, and T. Kropf. Structuring and automating hardware proofs in a higher-order therem proving environment. *Formal Methods in System Design*, 2(2):165–223, 1993.
19. Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Courcoubetis [9], pages 292–304.
20. D. A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press, 1989.
21. W. McCune. OTTER 2.0 users guide. Technical Report ANL-90/9, Argonne National Laboratory, 1990.
22. P. Michael Melliar-Smith and John Rushby. The Enhanced HDM system for specification and verification. In *Proc. VerkShop III*, pages 41–43, Watsonville, CA, February 1985. Published as ACM Software Engineering Notes, Vol. 10, No. 4, Aug. 85.
23. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Some lessons learned. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, pages 482–500, Odense, Denmark, April 1993. Volume 670 of *Lecture Notes in Computer Science*, Springer-Verlag.
24. W. Pase and M. Saaltink. Formal verification in m-EVES. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 268–302, New York, NY, 1989. Springer-Verlag.
25. L. Robinson, K. N. Levitt, and B. A. Silverberg. *The HDM Handbook.* Computer Science Laboratory, SRI International, Menlo Park, CA, June 1979. Three Volumes.
26. Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. *Communications of the ACM*, 20(4):271–283, April 1976.
27. John Rushby, Friedrich von Henke, and Sam Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1991.
28. N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
29. Natarajan Shankar. Verification of real-time systems using PVS. In Courcoubetis [9], pages 280–291.
30. R. E. Shostak, R. Schwartz, and P. M. Melliar-Smith. STP: A mechanized logic for specification and verification. In D. Loveland, editor, *6th International Conference on Automated Deduction (CADE)*, New York, NY, 1982. Volume 138 of *Lecture Notes in Computer Science*, Springer-Verlag.
31. M.K. Srivas, et. al. Hardware verification using pvs: A tutorial. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1994. A Forthcoming Technical Report.