Effective Theorem Proving for Hardware Verification*

D. Cyrluk,[1] S. Rajan,[2] N. Shankar,[3] and M.K. Srivas[3]
{cyrluk, sree, shankar, srivas}@csl.sri.com

[1] Dept. of Computer Science, Stanford University, Stanford CA 94305 and Computer Science Laboratory, SRI International, Menlo Park, CA 94025
[2] Integrated Systems Design Laboratory, Department of Computer Science, University of British Columbia, Vancouver, Canada and Computer Science Laboratory, SRI International, Menlo Park, CA 94025
[3] Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA

Abstract. The attractiveness of using theorem provers for system design verification lies in their generality. The major practical challenge confronting theorem proving technology is in combining this generality with an acceptable degree of automation. We describe an approach for enhancing the effectiveness of theorem provers for hardware verification through the use of efficient automatic procedures for rewriting, arithmetic and equality reasoning, and an off-the-shelf BDD-based propositional simplifier. These automatic procedures can be combined into general-purpose proof strategies that can efficiently automate a number of proofs including those of hardware correctness. The inference procedures and proof strategies have been implemented in the PVS verification system. They are applied to several examples including an N-bit adder, the Saxe pipelined processor, and the benchmark Tamarack microprocessor design. These examples illustrate the basic design philosophy underlying PVS where powerful and efficient low-level inferences are employed within high-level user-defined proof strategies. This approach is contrasted with approaches based on tactics or batch-oriented theorem proving.

1 Introduction

The past decade has seen tremendous progress in the application of formal methods for hardware design and verification. Much of the early work was on applying proof checking and theorem proving tools to the modeling and verification of hardware designs [14, 16]. Though these approaches were quite general, the

* This work was supported in part by the following funding sources: NASA Langley Research Center contract NAS1-18969, ARPA Contract NAG2-891 administered by NASA Ames Research Center, NSF Grant CCR-930044, the Semiconductor Research Corporation contract 92-DJ-295 (to the University of British Columbia), and the Philips Research Laboratories, Eindhoven, The Netherlands.

verification process required a significant human input. More recently, there has been a large body of work devoted to the use of model-checking, language-containment, and reachability analysis to finite-state machine models of hardware [8]. The latter class of systems work automatically but they do not yet scale up efficiently to realistic hardware designs. The challenge then is to combine the generality of theorem proving with an acceptable and efficient level of automation.

Our main thesis is that in order to achieve a balance between generality, automation, and efficiency, a verification system must provide powerful and efficient primitive inference procedures that can be combined by means of user-defined, general-purpose, high-level proof strategies. The efficiency of the inference procedures is important in order to verify complex designs with a greater level of automation. To achieve efficiency, each individual primitive inference procedure must itself perform a powerful and well-defined inference step using well-chosen algorithms and data structures. A number of related deductive operations must be tightly integrated into such a step. The efficiency resulting from a powerful and tightly integrated inference procedure cannot typically be obtained by composing very low-level inference steps by means of tactics. On the other hand, a fully automatic batch-oriented theorem prover has the drawback of being a toolkit with only a single tool. Doing exploratory proof development with such a theorem prover is tedious because of the low bandwidth of interaction. It is difficult to reconcile efficiency with generality in a fully automated theorem prover since a single proof strategy is being applied to all theorems.

The above design philosophy has formed the guiding principle for the implementation of the Prototype Verification System (PVS) [22, 23] developed at SRI. PVS is designed to automate the tedious and obvious low-level inferences while allowing the user to control the proof construction at a meaningful level. Exploratory proofs are usually carried out at a level close to the primitive inference steps, but greater automation can be achieved by defining high-level proof strategies.

In this paper, we present some automatic inference procedures used in the PVS proof checker, show how these inference procedures can be combined into general-purpose proof strategies, and examine the impact of these strategies on the automation of hardware proofs. The primitive inference procedures in PVS include arithmetic and equality decision procedures, an efficient hashing-based conditional rewriter, and a propositional simplifier based on binary decision diagrams (BDDs). The interaction between rewriting, arithmetic, and BDD-based propositional simplification yields a powerful basis for automation. The capabilities of the inference procedures are available to the user of the proof checker as primitive inference steps. These primitive inference steps can either be used as part of an interactive proof attempt or embedded inside a high-level proof strategy. We have developed a basic proof strategy in terms of these inference steps that is particularly effective for automating proofs of microprocessors and inductively defined hardware circuits. The core of this strategy consists of first

carrying out a symbolic execution of the hardware and its specification by expanding and simplifying the relevant definitions; the case structure of the symbolic execution is then brought to the surface, and BDD-based propositional simplification is used to generate subgoals that are typically proved by means of the decision procedures. We present the proof strategy and demonstrate its utility on a number of examples including an N-bit ripple carry adder circuit, Saxe's pipelined microprocessor [24] and the Tamarack processor [18]. The point of these examples is to illustrate efficiency and generality that can be derived from the inference capabilities present in PVS. This work is still at a preliminary stage and we feel that there is plenty of scope for obtaining even greater generality, efficiency, and automation by pursuing the line of development indicated in this paper.

The next section gives a brief overview of PVS. In section 3, we describe a general-purpose strategy for hardware proofs in PVS and illustrate this strategy with an N-bit adder circuit. Section 4 describes the use of the PVS inference procedures in the development of verification strategies for microprocessor designs. We present our conclusions in the last section.

2 An Overview of PVS

PVS is an environment for writing specifications and developing proofs [23]. It serves as a prototype for exploring new approaches to mechanized formal methods. The primary goal of PVS is to combine an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving power. PVS has been strongly influenced by the observation that theorem proving capabilities can be employed to enrich the type system of a typed logic, and conversely, that an enriched type system facilitates expressive specifications and effective theorem proving. PVS has also been guided by the experience that much of the time and effort in verification is in debugging the initial specification or proof idea. A high bandwidth of interaction is useful at the exploratory level whereas more automated high-level proof strategies are desirable at an advanced stage of proof development. PVS has been used to verify several complex fault-tolerant algorithms, real-time and distributed protocols, and several other applications [20].

2.1 The Specification 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]]`). The type system of PVS includes *predicate subtypes* that consist of exactly those elements of a

given type satisfying a given predicate. 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 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))]`). Note that the application `a(r)` is used to access the `a` field of record `r`, and the application `PROJ_2(t)` is used to access the second component of a tuple `t`. PVS specifications are packaged as *theories*.

2.2 The Proof Checker

The PVS proof checker is intended to serve as a productive medium for debugging specifications and constructing readable proofs. The human verifier constructs proofs in PVS by repeatedly simplifying a conjecture into subgoals using inference rules, until no further subgoals remain. A proof goal in PVS is represented by a sequent. PVS differs from most proof checkers in providing primitive inference rules that are quite powerful, including decision procedures for ground linear arithmetic. The primitive rules also perform steps such as quantifier instantiation, rewriting, beta-reduction, and boolean simplification. PVS has a simple strategy language for combining inference steps into more complicated proof strategies. In interactive use, when prompted with a subgoal, the user types in a proof command that either invokes a primitive inference rule or a compound proof strategy. For example, the `skolem` command introduces Skolem constants for universal-strength quantifiers while the `inst` command instantiates an existential-strength quantifier with its witness. The `lift-if` command invokes a primitive inference step that moves a block of conditionals nested within one or more sequent formulas to the top level of the formula. The `prop` command invokes a compound propositional simplification strategy (or tactic) that is a less efficient alternative to the use of a BDD-based simplifier described below. Various other commands are discussed below. Proofs and partial proofs can be saved, edited, and rerun. It is possible to extend and modify specifications during a proof; the finished proof has to be rerun to ensure that such changes are benign.

While a number of other theorem provers do use decision procedures, PVS is distinguishing in the aggressiveness with which it uses them. It is also unique in the manner in which the decision procedures and automatic rewriting are engineered to interact with each other in implementing the primitive inference commands of PVS. The user can invoke the functionality provided by the interacting decision procedures in its full power in a single command (`assert`) or in limited forms by means of a number of smaller commands. Some of these primitive commands are described in the following sections.

2.3 The Ground Decision Procedures

The ground decision procedures of PVS are used to simplify quantifier-free Boolean combinations of formulas involving arithmetic and equality, and to propagate type information. PVS makes extremely heavy use of these decision procedures.

Consider a formula of the form `f(x) = f(f(x)) IMPLIES f(f(f(x))) = f(x)`, where the variable `x` is implicitly universally quantified. This is really a *ground* (i.e., variable-free) formula since the universally quantified variable `x` can be replaced by a newly chosen (Skolem) constant, say `c`. We can then negate this formula and express this negation as the conjunction of literals: `f(c) = f(f(c)) AND NOT f(f(f(c))) = f(c)`. We can then prove the original formula by refuting its negation. To refute the negation we can assert the information in each literal into a data structure until a contradiction is found. In this case, we can use a *congruence closure* data structure to rapidly propagate equality information.

Congruence closure [12] plays a central role in several other systems including the Stanford Pascal Verifier [21] and Ehdm [9]. This basic procedure can be extended in several ways. One basic extension is to the case of ground linear inequalities over the real numbers. In the example, `a < 2*b AND b < 3*c AND NOT 3*a < 18*c`, the refutation can be obtained by eliminating the variable `b` in the second inequality in favor of `a`. Another extension is to the case of ground arrays or functions. This is important in hardware examples where memory can be represented as a function from addresses to data. For example, the decision procedure can deduce `(func WITH [(j) := val])(i)` to be equal to `func(i)` under the assumption `i < j`. The PVS decision procedures combine congruence closure over interpreted and uninterpreted functions and relations with refutation procedures for ground linear inequalities over the real numbers and arrays [26]. This procedure is also extended to integer inequalities in an incomplete though effective manner. The ground decision procedures can, for example, refute `i > 1 AND 2*i < 5 AND NOT i = 2`.

2.4 The Simplifier

The congruence closure data structure is used to maintain and update contextual information. Any relevant subtype constraints on terms are also recorded in these data structures. The *beta-reduction* of lambda-redexes, and datatype tuple, record and update access are among the automatic simplifications supported by PVS. We do not describe the arithmetic simplifications except to say that they evaluate expressions where the arithmetic operations (`+`, `-`, `*`, and `/`) are applied to numerical values and reduce any arithmetic expressions to a sum-of-products form. The Boolean simplifications are similarly straightforward and they simplify expressions involving the constants `TRUE` and `FALSE` and the operators `NOT`, `OR`, `AND`, `IMPLIES`, and `IFF`. In the simplification of conditional

expressions, the *test* part of the conditional is used in the simplification of the *then* and *else* parts. These simplifications are shown below where '$l \Longrightarrow r$' is shown as '$l \Longrightarrow r$':

1. (IF A THEN s ELSE t ENDIF) $\Longrightarrow$ s, if A $\Longrightarrow$ TRUE
2. (IF A THEN s ELSE t ENDIF) $\Longrightarrow$ t, if A $\Longrightarrow$ FALSE
3. (IF A THEN s ELSE s ENDIF) $\Longrightarrow$ s
4. (IF A THEN s ELSE t ENDIF) $\Longrightarrow$ (IF A' THEN s' ELSE t' ENDIF), if A $\Longrightarrow$ A', s $\Longrightarrow$ s' assuming A', and t $\Longrightarrow$ t' assuming $\neg$A'

When the **record** command is invoked on a PVS sequent of the form $A_1, \ldots, A_m \vdash B_1, \ldots, B_n$, the simplified form of each atomic A_i (or $\neg B_i$) is recorded in the congruence closure data structures. This information is then used to simplify the remaining formulas in the sequent. The **simplify** command simplifies the formulas using the ground decision procedures and simplifier without recording any new information into the data structures.

2.5 The PVS Rewriter

A (conditional) rewrite rule is a formula of either the form $A \supset p(b_1, \ldots, b_n)$ or $A \supset l = r$. The former case can be reduced to the latter form as $A \supset p(b_1, \ldots, b_n) =$ **TRUE**. In the latter case, the PVS rewriter then simplifies an instance $\sigma(l)$ of l to $\sigma(r)$ provided the hypothesis instance $\sigma(A)$ simplifies (using simplification with decision procedures and rewriting) to **TRUE** as must any type correctness conditions (TCCs) generated by the substitution σ. The free variables in A and r must be a subset of those in l. The hypothesis can be empty and definitions can also be used as rewrite rules.

There is also a restriction of rewriting where, if the right-hand side $\sigma(r)$ of a rewrite is an **IF-THEN-ELSE** expression, then the rewrite is not applied unless the test part of the conditional simplifies to **TRUE** or **FALSE**. This restricted form of rewriting serves to prevent looping when recursive definitions are used as rewrite rules and to control the size of the resulting expression. The above heuristic restriction on rewriting relies on the effectiveness of the simplifications given by the decision procedures. This heuristic is quite important in the context of processor proofs where the next state of the processor should be computed as long as there is an explicit clock tick available.

For efficiency, PVS maintains a hash-table where corresponding to a term a, the result of the most recent rewriting of a is kept along with the logical *context* at the time of the rewrite. A context consists of the congruence closure data structures and the current set of rewrite rules stored internally at the time of rewrite. This way, if the term a is encountered within the same logical context, the result of the rewrite is taken from this hash-table and the rewriting steps are not repeated. The information that an expression could not be rewritten in a context is also cached. This information is perhaps the more heavily used than the information about successful rewriting.

The context is modified by the proof tree structure and the IF-THEN-ELSE structure of an expression. In case the term a is encountered in a strictly larger logical context (facts have been added to the congruence closure data structures or the set of rewrite rules has been expanded) then the result of the rewrite is taken from the hash-table and further rewritten using the current larger logical context.

The rewriter described above is used automatically in simplification. It can be invoked by the **do-rewrite** command. The **assert** command combines the functionality of **record**, **simplify**, and **do-rewrite**.

2.6 The Power of Interaction

The following example illustrates the power that a close interaction between rewriting and the decision procedures can provide for the user in PVS. Such a close interaction is not as easily accomplished if the decision procedures and rewriting were implemented as separate tactics or strategies.

```
t: nat
s: VAR state
MAR_t0: AXIOM t /= 0  => dest(IR(s)) = MAR(s)

MDR5(s): data =
    IF t <= 2
       THEN IF p(t)
            THEN MDR(s)
            ELSE rf(s) WITH [(MAR(s)) := MDR(s)](dest(IR(s)))
            ENDIF
       ELSE somedata
    ENDIF

property: THEOREM t < 3 & p(0) => MDR(s) = MDR5(s)
```

In the PVS specification shown above, the goal is to prove **property** from the axioms **MAR_t0** and the definition of **MDR5**, where the constants **p**, **MDR**, **MAR**, etc., are declared elsewhere. The constant **rf** is a function that maps addresses to data. Assuming that the definition of **MDR5** and **MAR_t0** have been entered as rewrite rules (via the command **auto-rewrite**), **property** can be proved (after skolemizing the variable **s** and flattening the implication) by simply using the command **assert** on the resulting sequent twice. The first invocation of **assert** is able to rewrite **MDR5(s)** to the **IF-THEN-ELSE** expression beginning at **p(t)** since, when **t** is a **nat**, **t <= 2** can be deduced from **t < 3** by the decision procedure.

A second invocation of **assert** attempts to rewrite each branch of the resulting **IF-THEN-ELSE** expression. The **p(t)** case is trivially true. In the **NOT p(t)** case, the decision procedures deduce that **t = 0** is false from **p(0)**. This triggers the rewrite rule **MAR_t0** so that the goal becomes **MDR(s) = rf(s)**

`WITH [(MAR(s)) := MDR(s)](MAR(s))`, which the equality procedures simplify to true.

2.7 BDD Simplifier

Binary decision diagrams (BDDs) are widely used in the design, synthesis, and verification of digital logic. They provide an efficient representation for the simplification of propositional formulas. A BDD-based propositional simplifier was recently added to PVS as a primitive inference step. Prior to the introduction of this simplifier, PVS used a propositional simplification tactic called `prop` that was found to be unsatisfactory since it could generate subgoals that were just permutations of other subgoals.

The basic idea behind the use of BDD-based simplification in PVS is to transform the goal sequent into a Boolean expression where the atomic formulas have been replaced by propositional variables. This formula is given as input to an off-the-shelf BDD package by means of an external function call. Note that PVS is implemented in Common Lisp whereas the BDD package is a C program. The BDD package simplifies the given formula into an equivalent formula in conjunctive normal form that is easily translated into a collection of subgoal sequents by replacing the propositional variables back to the corresponding atomic formulas. It is also possible to provide the BDD simplifier with some contextual information using the *restriction* operator, also known as *cofactoring*, provided by the BDD package. The restriction operation is used to simplify one BDD representation assuming another containing the contextual assumptions.

The BDD simplifier we use is an efficient implementation from EUT [17]. The simplifier uses Reduced Ordered BDD (ROBDD), a canonical representation of boolean expressions, with an associated set of algorithms [5]. The BDD simplifier is invoked in a PVS proof with the command `bddsimp`.

3 The Nature of Hardware Proofs, and Our Thesis

We have described some of the built-in deductive capabilities of PVS. A PVS proof is constructed by interactively (or automatically) invoking these inference steps to simplify the given goal into simpler subgoals until all the subgoals are trivially true. At the highest level, the user directs the verification process by elaborating and modifying the specification, providing relevant lemmas, and backtracking on the fruitless paths in a proof attempt. At the next level of an interactive PVS proof, particularly a hardware proof, the user provides the following crucial inputs:

Quantifier elimination: Since the decision procedures work on ground formulas, the user must eliminate the relevant universal-strength quantifiers by introducing Skolem constants or suggesting induction schemes. Existential-strength quantifiers are eliminated 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.

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

The use of decision procedures for arithmetic and equality yields a significant advantage in that the outcome of a proof attempt is not as logically sensitive to the decisions made in performing the second and third tasks as is the case in provers without decision procedures. However, decisions made during the second and the third tasks critically impact the efficiency of the proof. For example, the extent to which the defined function symbols are unfolded determines the number of cases to be considered during case analysis. Performing case analysis on selected boolean expressions before rewriting can make rewriting more productive and reduce the size of the resulting expressions, whereas a naive case analysis can lead to a needless combinatorial blowup in proof size.

In most of our experiments with hardware proofs, we found that we needed to intervene manually during rewriting and case analysis tasks only to control the complexity of the proof. Our experience suggested that the second and the third tasks could be completely automated for most hardware proofs given an efficient rewriting and propositional simplification engine used in conjunction with the arithmetic decision procedures. In the next section, we illustrate the above thesis on an N-bit ripple-carry adder example.

3.1 An N-bit Adder

The theory `adder` shown below describes the implementation and the correctness statement of the adder. The theory is parameterized with respect to the length of the bit-vectors. It imports the theory `full_adder` which contains a specification of a full adder circuit with output carry bit `fa_cout` and the sum bit `fa_sum`, and the theory `bv` which specifies the bit-vector type (`bvec[N]`) and related bit-vector functions. An N-bit bit-vector is represented as an array, i.e., a function from the type `below[N]` of natural numbers less than `N` 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 carry bit that ripples through the full adders is specified recursively by means of the function `nth_cin`[4]. 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 the least

[4] Recursive function definitions in PVS must have an associated `MEASURE` function to ensure termination. The typechecker automatically generates type correctness proof obligations to show that the measure of the argument to every recursive invocation the function is less than the measure of the original argument.

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.

```
adder[N: posnat] : THEORY

BEGIN

  IMPORTING bv[N], full_adder

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

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

  bv_sum(cin, bv1, bv2)(n): bvec =
    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))

  full_adder_correct:
    LEMMA
      bool2bit(a) + bool2bit(b) + bool2bit(c)
        =   2 * bool2bit(fa_cout(c, a, b))
          + bool2bit(fa_sum(a, b, c))

  adder_correct: LEMMA (FORALL n:
      bvec2nat(n, bv1) + bvec2nat(n, bv2) + bool2bit(cin)
        =   exp2(n + 1) * bool2bit(bv_cout(n, cin, bv1, bv2))
          + bvec2nat(n, bv_sum(cin, bv1, bv2))

END adder
```

The proof of **adder_correct** proceeds by induction on the variable **n** using an induction scheme for the type **below[N]**. This results in a base case that is easily proved by **assert** and an induction case that is displayed below as a sequent containing only one formula.

```
adder_correct2 :

  |-------
{1}   (FORALL (r: below[N]):
         r < N - 1
           AND (FORALL (bv1, bv2: bvec[N]), (cin: bool):
                  bvec2nat(r, bv1) + bvec2nat(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(r + 1, bv1)
                        + bvec2nat(r + 1, bv2)
                        + bool2bit(cin)
                     = exp2(r + 1 + 1)
                        * bool2bit(bv_cout(r + 1, cin, bv1, bv2))
                        +
                        bvec2nat(r + 1,
                                    bv_sum(cin, bv1, bv2))))
```

The general strategy to prove the above goal, as in any inductive proof, is to first introduce skolem constants for the universal-strength variables in the goal and flatten the sequent into a form where the inductive hypothesis is in the antecedent. After that, one has to simplify the conclusion to a point where an instance of the induction hypothesis can be used to discharge the conclusion. The simplification of the conclusion can either be done under control or by brute-force automation. We contrast the two approaches for the adder example below. In both approaches, the first step (**skosimp***) performs the repeated skolemization and flattening of the sequent required at the start of the proof.

Guided Proof	Automatic Proof
(skosimp*)	(skosimp*)
(expand "exp2" 1)	(auto-rewrite-explicit)
(expand "bvec2nat" 1)	(do-rewrite)
(expand "bv_sum" 1 1)	(inst?)
(expand "bv_cout")	(repeat (lift-if))
(expand "nth_cin" 1)	(simplify)
(lemma "full_adder_correct")	(then* (bddsimp)(assert))
(inst?)	
(inst?)	
(assert)	

In the guided proof, shown on the left, we carefully control the rewriting process by selecting a subset of the defined function symbols in the sequent

to unfold in order to keep the size of the proof tree under control. The PVS command **expand** is used to expand function definitions in a controlled manner. The optional second and the third argument to **expand** respectively specify the formula and the occurrence of the symbol to be expanded.

At this point, a careful case analysis on the `bool2bit` values of the three most significant boolean bits under consideration would lead to eight subgoals. We construct a shorter proof by using the lemma `full_adder_correct` about the full adder to eliminate the bit-level case analysis required. The `inst?` command attempts to find a suitable set of instantiations for existential-strength quantifiers in the sequent formulas of a subgoal[5] which in this case include the lemma as well as the induction hypothesis. In this case, it manages to instantiate both the inductive hypothesis and the lemma to the desired substitutions. The lemma `full_adder_correct` cannot be successfully applied as a rewriting rule in this proof because the instances of `bool2bit` do not appear contiguously in the expression to be simplified. The last step in the proof, `assert`, invokes the arithmetic and equality decision procedures of PVS to complete the proof.

A More Automatic Proof

We now describe a more automatic proof of the same theorem, shown on the right side of the above table, that takes a brute-force approach by employing automated rewriting and BDD-based propositional simplification. This strategy is part of a general strategy for proofs involving induction and rewriting that has been used on several other examples. Using this strategy we were able to verify the adder in 130 seconds. Every defined function symbol used directly or indirectly in the sequent is set up as a rewrite rule by invoking `auto-rewrite-explicit`. This set of function symbols includes not only those appearing in the `adder` theory but also those in the theories `full_adder` and `bv` imported by `adder`. The automatic rewriter of PVS is then invoked by means of the `do-rewrite` command to rewrite all the expressions in the sequent using the rewrite rules introduced above. The rewriting process simplifies the conclusion into an equation on two nested conditional (`IF-THEN-ELSE`) expressions. Not surprisingly, the size of the expressions resulting from the rewriting is much larger here than in the intelligent proof.

To automate the case analysis, we repeatedly (using `repeat`[6]) lift all the `IF-THEN-ELSE` conditionals to the topmost level (using `lift-if`). The lifting

[5] It can be used either in a mode in which all possible instances of the lemma are produced or only a single instance is produced.

[6] `Repeat` and `then*` are among the *tacticals* provided by PVS for constructing proof strategies from primitive inference steps and other predefined proof strategies. `Repeat` applies a given proof step repeatedly until its application has no change on the current goal; `then*` applies the first goal from the given list of proof steps to the current goal, and the rest of the steps in the list to each of the subgoals, if any, resulting from the first application.

process transforms the conclusion into a propositional expression in the form of nested **IF-THEN-ELSE** expressions whose leaf nodes are equalities on unconditional expressions. The propositional expression is simplified using **bddsimp**. Rewriting and decision procedures (using **assert**) are applied to any subgoals generated by **bddsimp**.

The latter proof is automatic in the sense that it applies certain coarse-grain inference steps under a simple control strategy without requiring any specific information from the user to guide the proof. Three elements are crucial to making the automatic proof successful. Firstly, we need efficient rewriting that can rewrite large expressions while exploiting contextual information. Second, we need an efficient propositional simplifier to perform the automatic case analysis on very large formulas. The above automatic proof blows up if **bddsimp** is replaced with the tactic-based simplifier (**prop**) of PVS. The third element is the availability of powerful arithmetic decision procedures. This makes the exact syntactic form of the expressions in the simplified sequent less relevant than whether the sequent has enough (semantic) information to complete the proof.

The automatic proof used above can be packaged in a PVS proof strategy and used on other hardware examples. The core of our strategy for automating hardware proofs begins with the user suggesting the initial induction variable and the induction scheme when this is not obvious from the type of induction variable. An automatic strategy takes over from that point and completes the proof in the following manner. First, the PVS rewriter is set up to automatically rewrite every defined function symbol directly or indirectly used in the the theorem to be proved until rewriting is no longer productive. In the next step, all the boolean conditions appearing as the boolean part of conditional expressions in the formulas are lifted to the topmost level. The resulting nested boolean expression is propositionally simplified into a finite number of subgoals. The last step consists of applying arithmetic and equality decision procedures on each of the subgoals resulting from the propositional simplification. We have applied this strategy to an *n-bit* ALU [7] that executes 12 microoperations. The completely automatic verification took 90 seconds on a SPARC 10. The same strategy is also effective on several non-hardware examples [25].

4 Microprocessor Verification

The automatic inference procedures used in the PVS proof checker have also allowed us to highly automate the task of microprocessor verification. PVS is a relatively new system that has been evolving over the course of our processor verification effort. As more automatic inference procedures have been added to PVS, our effectiveness at automating microprocessor verification has significantly increased. Here we illustrate the usefulness and importance of automatic inference procedures in PVS from the point of view of processor verification. These examples are quite different from the N-bit adder described in Section 3

but the basic idea underlying the proof strategy given there is easily adapted for our present purpose.

We take the approach of describing the specification and implementation of microprocessors in terms of state transition systems. The state of the microprocessor consists of the state of the memory, register file, and internal registers of the processor (these would generally include the program counter, memory address register, and pipeline registers if the processor is pipelined, etc.).

The microprocessor verification problem is to show that the traces induced by the implementation transition system are a *subset* of the traces induced by the specification transition system, where *subset* has to be carefully defined by use of an abstraction mapping. The details of this approach are beyond the scope of this paper (see [2,11,24,27,29][7]).

In this approach, the proof of correctness makes use of an *abstraction* function that maps an implementation state into a *corresponding* specification state. Correctness can then be reduced to showing that for any execution trace of the implementation machine there exists a *corresponding* execution trace of the specification machine.

The implementation machine may run at a different rate than the specification machine [11,27]. For example, in the case of the Saxe pipeline example [24], the specification machine takes one state transition to execute each instruction, but the implementation machine might take five cycles to execute branch instructions, but only one cycle for non-branch instructions. In the following we assume that the specification machine always takes one cycle to execute an instruction. We also assume that the number of cycles that the implementation machine takes to execute an instruction can be given as a function of the current state and current input. (This restriction can be slightly relaxed to deal with interrupts which might arrive a bounded number of cycles into the future.)

4.1 A Proof Strategy for Microprocessor Correctness

We denote the function that determines the number of cycles that the implementation machine takes to complete an instruction as `num_cycles`. We assume that this information is provided by the hardware designer or verifier.

The first step in verifying the correctness of the microprocessor is to split the proof into cases based on the definition of `num_cycles`. Thus for each case we have a precise number through which we have to cycle the implementation machine.

In the microprocessor verifications we have looked at, the state variables of the specification state are simply a subset of the state variables of the implementation state. The abstraction mapping maps to each specification register the corresponding implementation register, but not necessarily from the exactly

[7] The precise details followed in these papers are somewhat different.

corresponding state. For example, the abstraction mapping for the Saxe pipeline is such that the specification program counter is mapped from the corresponding implementation program counter and the specification register file is mapped from the implementation register file, but three cycles into the future. See [11,24] for details. If the abstraction mapping is given this way, then once the proof is split according to the definition of **num_cycles**, the resulting statement of correctness is usually an instance of a decidable fragment of the theory Ground Temporal Logic (*GTL2*) [10].

The problem is to come up with an effective procedure for deciding this theory. One obvious strategy is to completely rewrite the next-state functions and abstraction mapping until a large **IF-THEN-ELSE** is generated, then perform a case analysis on the resulting expression and check that each resulting case is valid. This naive strategy has proven to be ineffective for both the Saxe pipeline and Tamarack microprocessors, let alone anything more complex. However, the automatic inference procedures of PVS have allowed us to develop a less naive strategy that is still highly automatic and does succeed in proving the correctness of both the Saxe pipeline and Tamarack microprocessors:

```
(then* (skosimp*)
       (auto-rewrite-all-theories)
       (typepred-impl-state)
       (record)
       (cycle-split)
       (record)
       (rewrite-lift-if-simplify-and-assert)
       (auto-rewrite-all-theories!)
       (rewrite-lift-if-simplify-and-assert))
```

where the **rewrite-lift-if-simplify-and-assert** strategy is just:

```
(then* (assert) (repeat (lift-if)) (bddsimp) (assert)).
```

The above strategy consists of first skolemizing, then instructing PVS to use the axioms and definitions of the processor as rewrite rules (**auto-rewrite-all-theories**), then invoking the type predicate of the implementation state-type ((**typepred-impl-state**)). This is necessary in case there is a pipeline invariant associated with the machine state [11, 27]. The proof goal is then split (**cycle-split**) according to the **num_cycles** function. The current case is recorded in the ground decision procedures and **assert** is called which invokes automatic rewriting. The rewriting here will halt once it is incapable of simplifying a right hand side that is an **IF-THEN-ELSE**. Any resulting **IF-THEN-ELSE**s are then lifted and **bddsimp** is called to generate the resulting cases. The **assert** command is used to finish up each of these cases. Sometimes this is enough to complete the proof. If it is not then the (**assert, lift-if, bddsimp, assert**) cycle is repeated, but this time with PVS directed to completely rewrite, even through unsimplifiable **IF-THEN-ELSE**s.

The intuition behind this strategy is that the first form of rewriting takes care of the simple parts of the proof that require only rewriting and limited amount of case analysis. The second, unrestricted, rewriting takes care of the resulting cases that need to expand to large `IF-THEN-ELSE`s and require lots of case analysis. In the Saxe pipeline this type of reasoning is needed to verify the correctness of the register bypass logic. Note that this strategy, while not identical to the basic hardware strategy described earlier, has the same core strategy, namely the `(do-rewrite, lift-if, bddsimp, assert)` cycle.

We have also applied the same strategy to the Tamarack microprocessor first verified by Joyce [18]. This microprocessor is microcoded but not pipelined. Only the first restricted form of rewriting is necessary to finish the Tamarack's proof of correctness. This is because the case splitting generated by the `num_cycles` function is sufficient to generate all the relevant cases and to direct the rewriter through a single path through the microcode. In the Saxe pipeline more case analysis is needed to deal with the register bypass logic.

Note that prior to adding hashing and bddsimp to PVS we had verified the correctness of the Saxe pipeline, but only with manual assistance. The verification originally done by Saxe et al [24] also required user assistance.

5 Experimental Results

The following table summarizes the performance of PVS's automatic strategy with and without the improvements to PVS's automatic inference procedures. The timings were made on a SPARC 10.

Processor	Hashing and BDDs	No Hashing	Neither
Adder	127 sec.	160 sec.	unfin.
ALU	87 sec.	92 sec.	unfin.
Saxe Pipeline	605 sec.	1400 sec.	unfin.
Tamarack	545 sec.	unfin.	unfin.

Note that hashing was much more important in the microprocessor examples. These examples typically use more rewriting. In the ongoing verification of a simplified version of the MIPS R3000 we find that we get exponential savings due to hashing.

6 Related Work

The HOL system [13] is prototypical of the proof checkers that are based on very simple primitive inference rules combined using tactics. The more powerful primitive inference mechanisms of PVS can, in principle, be developed as tactics in HOL. For example, Boulton [3] has implemented a decision procedure for Presburger arithmetic as a tactic in HOL. However, this procedure does not

handle equality over uninterpreted function symbols and, unlike in PVS, is not tightly integrated with the simplification and rewriting procedures. It would be interesting to see if the same degree of integration can be accomplished as effectively in a tactic-based approach and whether individual tactics can match the performance of inference procedures that use specialized algorithms and data structures. The HOL system favors tactics over special-purpose inference procedures since the latter might introduce unsoundness. This is an important consideration: the inference procedures of PVS do need to be scrutinized and tested with great care and rigor, but once this is done, they do not need to be justified down to basic inference steps with each application.

The "super-duper" tactic developed in [1] for hardware proofs is similar to the core strategy described in this paper. The similarity lies in the fact that both combine rewriting, case-splitting and simplifications in a loop for automating hardware proofs. The main differences are in (1) our use of decision procedures for congruence closure, arithmetic, and BDDs, and (2) our conditional rewriter interacts very closely with the decisions procedures and uses several optimizations. This interaction allows rewriting to be more effective, i.e., successful in simplifying more often, and efficient. We have found that the efficiency and effectiveness of rewriting are very crucial in the core strategy being applicable for large examples. The tactic in [1] is also designed to process predicative style of hardware specifications, whereas ours is suited for functional style.

Kumar, Schneider, and Kropf have developed a system MEPHISTO and a sequent calculus prover FAUST [19] which jointly can automatically verify a class of bit-level hardware circuits specified in a relational style popularized by Michael Gordon. Their system cannot automate proofs of complex circuits, such as microprocessors, that use data types since they do not have rewriting and arithmetic capability. This system does incorporate first-order BDD-based techniques that can handle some data types and parameterized hardware. Although our automatic strategy presented in this paper is designed for proving hardware specified in a functional style, we were able to automatically prove all but two of their eight circuits by modifying our strategy slightly to use the heuristic instantiation capability supported by PVS. We had to provide manual instantiations for the other two examples.

The Boyer-Moore theorem prover, Nqthm, is the best known of the batch-oriented theorem proving systems used in hardware verification [4]. Many of its deductive components are quite similar to those in PVS. The system uses a fast propositional simplifier, and also includes a rewriter and a linear arithmetic package. The latest release of the system has been heavily optimized for efficiency. As an experiment, we used Nqthm without any libraries to prove the N-bit adder. The Nqthm formalization of this theorem was slightly different from that of PVS. We found that the theorem could not be proved automatically. It took several hours of effort to fine-tune the definitions and to determine the lemmas needed to help the theorem prover with its proof. Though a significant human effort was required to complete the proof, Nqthm was eventually able to prove the main theorem in about 14 seconds of CPU time (on a Sparc 10/41).

The same example was proved in PVS without any lemmas and very little human input in about 130 seconds.

Burch and Dill [6] report on an automatic stand-alone strategy for microprocessor verification. Although they have not attempted the two examples reported here they report impressive timings for the automatic verification of a small version of the DLX processor [15]. They also describe a method for automating the generation of the abstraction mapping.

7 Conclusions

Automated theorem proving technology clearly has a great deal to contribute to hardware verification since hardware proofs tend to fall into certain systematic patterns. Our contention is that if theorem provers are to be effective in hardware verification, we must employ powerful and efficient deductive components within high-level strategies that capture the patterns of hardware proofs. More specifically, we have argued that:

- Hardware proofs tend to fall into certain patterns so that it is possible to obtain greater automation.
- Effective theorem proving is best achieved by mechanizing the tedious and routine deductive steps so that the human effort can be concentrated on the difficult parts of the proof.
- We can combine automation with efficiency by employing powerful and well-integrated mechanized procedures as can be obtained through the use of decision procedures and BDD-based propositional simplification.
- Batch-oriented theorem provers like Nqthm do contain tightly integrated and highly mechanized inference procedures, but they require a significant amount of tedious human effort in the exploratory phase of proof development.
- PVS strikes a balance between the tactic-based approach and those based on batch-oriented theorem proving. In PVS, efficient mechanization is used to automate the tedious and obvious deductive steps. Proofs can be constructed interactively under human control. Further mechanization can be obtained by defining high-level strategies in terms of tactics.

We have shown how hardware proofs can be automated in PVS through the use of a powerful mechanization of various useful inference steps and the definition of simple proof strategies that invoke these inference steps. We have illustrated the use of PVS an N-bit adder, a pipelined processor, and a simple unpipelined processor. The basic approach shown here is being applied to the mechanization of the correctness proofs of industrial-strength processors including the MIPS R3000 architecture and a commercial avionics processor AAMP5.

AAMP5 is a microcoded pipelined processor built at the Collins Avionics Division of Rockwell International for Avionics applications. It is a complex

CISC processor containing more than half a million transistors and is designed to execute a stack-oriented machine. One of the main purposes in undertaking this project [28], which is sponsored by NASA Langley Research Center and Rockwell International, was to see how well techniques developed and tested on small examples would scale to a commercial processor of significant complexity. We have successfully used the core strategy described in the paper to verify a number of instructions (identified by Rockwell engineers) of AAMP5. The verification revealed several errors some unknown to Rockwell and some planted by Rockwell engineers as a challenge to us.

Acknowledgements. John Rushby provided a great deal of support and encouragement for this work and supplied detailed comments on drafts of this paper. Sam Owre answered a number of questions regarding PVS and also proofread the paper. The N-bit ripple-carry adder example comes from a PVS library for bit-vectors being developed by Rick Butler and Paul Miner of NASA.

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. Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE, Computer Society Press, July 1988.
3. R. J. Boulton. The HOL arith library. Technical report, University of Cambridge Computer Laboratory, 1992.
4. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, NY, 1988.
5. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. of the 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.
6. J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification '94*, pages 68–80. Volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.
7. F. J. Cantu. Verifying an *n-bit* arithmetic logic unit. Blue book note 935, University of Edinburgh, June 1994.
8. E. M. Clarke and O. Grümberg. Research on automatic verification of finite-state concurrent systems. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, *Annual Review of Computer Science, Volume 2*, pages 269–290. Annual Reviews, Inc., Palo Alto, CA, 1987.
9. *User Guide for the* EHDM *Specification Language and Verification System, Version 6.1.* Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes.
10. D. Cyrluk and P. Narendran. Ground temporal logic—a logic for hardware verification. In David Dill, editor, *Computer-Aided Verification '94*, pages 247–259. Volume 818 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.

11. David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, December 1993.
12. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, October 1980.
13. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.
14. Mike Gordon. Proving a computer correct. Technical Report TR 42, University of Cambridge, Computer Laboratory, 1983.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.
16. Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, December 1989.
17. G. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
18. J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher order logic. Technical Report 100, Computer Lab., University of Cambridge, 1986.
19. 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.
20. Patrick Lincoln, Sam Owre, John Rushby, N. Shankar, and Friedrich von Henke. Eight papers on formal verification. Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.
21. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. CSD Report STAN-CS-79-731, Stanford University, Stanford, CA, March 1979.
22. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Saratoga, NY, June 1992. Volume 607 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.
23. S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes.
24. James B. Saxe, Stephen J. Garland, John V. Guttag, and James J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.
25. N. Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
26. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
27. Mandayam Srivas and Mark Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.
28. Mandayam Srivas and Steve Miller. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. Technical report. A Forthcoming NASA Contractor Report.
29. P. Windley and M. Coe. A correctness model for pipelined microprocessors. In *Proceedings of Theorem Provers in Circuit Design*, 1994.