Computer-Aided Computing*

Invited talk to be presented at MPC '95: Third international conference on the Mathematics of Program Construction, Kloster Irsee, Germany, July 1995

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/shankar.html
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. Formal program design methods are most useful when supported with suitable mechanization. This need for mechanization has long been apparent, but there have been doubts whether verification technology could cope with the problems of scale and complexity. Though there is very little compelling evidence either way at this point, several powerful mechanical verification systems are now available for experimentation. Using SRI's PVS as one representative example, we argue that the technology of mechanical verification is already quite effective. PVS derives its power from an integration of theorem proving with type-checking, decision procedures with interactive proof construction, and more recently, model checking with theorem proving. We discuss these individual aspects of PVS using examples, and motivate some of the challenges that lie ahead.

1 Introduction

Should hardware and software designs be formally verified, and if so, should the verification process be mechanized? The answer to these questions depends crucially on whether it is feasible to formally verify large and complex systems with demonstrable benefits and without an immoderate amount of effort. The basic thesis defended in this paper is that powerful and sophisticated mechanization is needed to facilitate the productive use of formal methods.

Formality and mechanization go hand-in-hand. Formal syntax and proof theory provide the foundation for building mechanized tools. Mechanization supplies the speed, accuracy, and repeatability needed for formal manipulations. Even so, mechanization is widely regarded as more of a hindrance than an aid. We argue that mechanization is not inherently unwieldy, and that through careful engineering, mechanization can be made responsive and flexible.

Several excellent mechanized verification tools have been developed in the recent past. A few of these have even been put to serious industrial use, and

* Supported by by NSF Grant CCR-930044.

the indications are that through these tools, formal methods are slowly gaining acceptance at least for the design of safety-critical systems.

We focus here on one such verification system, PVS, merely out of familiarity [23].[2] We describe the key facets of the design of PVS and illustrate their role in effective verification by means of examples. Our main observation is that general-purpose tools like PVS must be integrated with domain-specific tools, notations, methodologies in order to achieve truly effective mechanization.

We first give a brief introduction to PVS in Section 2. We then discuss three important features of PVS in some detail in Sections 3, 4, and 5. In Section 6, conclusions are drawn from this discussion about the challenges that lie ahead.

2 A Brief Introduction to PVS

PVS (Prototype Verification System) is intended as an environment for constructing clear and precise specifications and for developing readable proofs that have been mechanically verified. While many of the individual ideas in the system pre-date PVS, what is new is the coherent realization of these ideas in a single system. The key elements of the PVS design are captured by the following combinations:

- *An expressive language with powerful deductive capabilities.* The PVS specification language is based on classical, simply typed, higher-order logic with type constructors for functions, records, tuples, predicate subtypes, dependent types, and abstract datatypes. Typechecking is undecidable for PVS. The PVS typechecker checks for simple type correctness and generates proof obligations corresponding to predicate subtypes. PVS also has parametric theories so that it is possible to capture, say, the notion of sorting with respect to arbitrary sizes, types, and ordering relations. By exploiting these features, researchers at NASA Langley Research Center and SRI have developed a very general bit-vector library. Paul Miner at NASA has developed a specification of portions of the IEEE 854 floating-point standard in PVS [22].
- *Powerful decision procedures with user interaction.* PVS proofs are constructed interactively. The primitive inference steps for constructing proofs are quite powerful. They make extensive use of efficient decision procedures for equality and linear arithmetic. They also exploit the tight integration between rewriting, the decision procedures, and the use of type information.

[2] The design and development of PVS was a collaborative effort involving several people, primarily Sam Owre. Other contributors include David Cyrluk, Patrick Lincoln, Steven Phillips, Sree Rajan, John Rushby, Mandayam Srivas, Friedrich von Henke, and Carl Witty. Ricky Butler (NASA), Damir Jamsek (ORA), and Paul Miner (NASA) provided valuable support in testing and validating PVS. The development of PVS was funded by SRI International.

PVS also uses BDD-based propositional simplification so that it can combine the capability of simplifying very large propositional expressions with equality, arithmetic, induction, and rewriting.

Higher-level inference steps can be defined by means of strategies (akin to LCF tactics [16]) written in a simple strategy language. Typical strategies include heuristic instantiation of quantifiers, propositional and arithmetic simplification, and induction and rewriting. The PVS proof checker strives to strike a careful balance between an automatic theorem prover and a low-level proof checker. Through the use of BDD-based simplification, simple PVS proof strategies can be defined for efficiently and automatically verifying simple processor designs and N-bit arithmetic circuits [9].

– *Model checking with theorem proving.* Many forms of finite-state verification, such as linear temporal logic model checking, language containment, and bisimulation checking, can be expressed in the mu-calculus [3, 13]. The higher-order logic of PVS is used to define the least and greatest fixpoint operators of the mu-calculus. When the state type is finite, the mu-calculus expressions are translated into the propositional mu-calculus and a propositional mu-calculus model checker can be used as a decision procedure. The finite types are those built from booleans and scalars using records, tuples, or arrays over subranges. Fairness cannot be expressed in CTL, but it can be defined using the mu-calculus.

  With this integration, PVS includes a useful model checker that is complemented with the theorem proving capabilities of PVS. In particular, PVS can be used to define property-preserving abstractions from an unbounded-state system to a finite-state one [5, 11]. The induction and rewriting capabilities of PVS can also be used in conjunction with model checking. We have in fact verified two illustrative examples with a combination of theorem proving and model checking: a processor-memory combination and an N-process mutual exclusion algorithm [27].

A variety of examples have been verified using PVS. It has been used to verify fault-tolerant agreement protocols under Byzantine and hybrid fault models, and real-time protocols such as Fischer's mutual exclusion algorithm and a generalized railroad crossing [10, 18]. The most substantial use of PVS has been in the verification of the microcode for selected instructions of a commercial-scale microprocessor called AAMP5 designed by Rockwell-Collins and containing about 500,000 transistors [21].

We now discuss each of the above facets of PVS in greater detail.

3 Combining Theorem Proving and Typechecking

The PVS specification language is based on classical, simply typed higher-order logic, but the type system has been augmented with subtypes and dependent types.

Subtypes and Proof Obligations. Early in the design of PVS, it was decided that a simply typed language in which all types were distinct would be too restrictive and would disallow types such as the even numbers and the prime numbers which could be seen as subtypes of the integers. Once this kind of subtyping was permitted, it was quite clear that the language should permit subtype definitions with respect to arbitrary predicates. This kind of *naive comprehension* is acceptable in a typed language since the type classification rules out any Russellian inconsistencies. With arbitrary *predicate subtypes* in a higher-order logic, not only can we define the various useful subsets of the numbers but also such subtypes as the injective and surjective functions between two types. In PVS, the injective function space `injection` can be defined as a predicate subtype using the higher-order predicate `injective?` as shown below. The notation `(injective?)` is an abbreviation for {`f` | `injective?(f)`} which is the subtype of functions from `D` to `R` for which the predicate `injective?` holds.

```
functions [D, R: TYPE]: THEORY
 BEGIN
  f, g: VAR [D -> R]
  x, x1, x2: VAR D

  injective?(f): bool = (FORALL x1, x2: (f(x1) = f(x2) => (x1 = x2)))

  injection: TYPE = (injective?)


 END functions
```

Once we define such a subtype, we can define the type `even` of even numbers and declare a function double as an injective function from the type of natural numbers `nat` to the subtype `even`. Note that bound variables can also have their types *locally* declared in the binding as in `(j : nat)` below.

```
even: TYPE = {i : nat | EXISTS (j : nat): i = 2 * j}

double : injection[nat, even] = (LAMBDA (i : nat): 2 * i)
```

When the declaration of `double` is typechecked, the typechecker generates two proof obligations or type correctness conditions (TCCS):

```
double_TCC1: OBLIGATION (FORALL (i: nat): EXISTS (j: nat): 2 * i = 2 * j)

double_TCC2: OBLIGATION injective?[nat, even]((LAMBDA (i: nat): 2 * i))
```

The first TCC checks that the result computed by `double` is an even number. The second TCC checks that the definition of `double` is injective. Both TCCs are proved quickly and automatically using the default TCC strategy employed by the PVS proof checker. Proofs of more complicated TCCs can be constructed interactively. The PVS decision procedures are very handy for discharging typical TCC proof obligations. These decision procedures automate the theories of equality and linear arithmetic with uninterpreted function symbols and are discussed further in Section 4.

Dependent Types. The next example illustrates the use of dependent types where the dependencies are expressed using predicate subtypes. The theory `newstacks` introduces an implementation of stacks using a record consisting of a `size` field indicating the number of elements in the stack, and an `elements` field which is an array of `size` elements. The `newstacks` theory is parametric in the element type. The type `below(size)` is just $\{\mathtt{i} | \mathtt{i} < \mathtt{size}\}$. The empty stack `empty` has `size` zero and an empty array of elements. The `push` operation yields a nonempty stack. The `pop` operation is only defined for nonempty stacks.

```
newstacks[T: TYPE]: THEORY
  BEGIN

  i, j: VAR nat
  stack: TYPE = [# size: nat, elements: ARRAY[below(size) -> T] #]
  u, s: VAR stack
  x, y: VAR T
  e: T
  nonemptystack?(s): bool = (size(s) > 0)

  empty: stack = (# size := 0, elements := (LAMBDA (j | j < 0): e) #)

  push(x, s): (nonemptystack?) =
        (# size := size(s) + 1,
           elements := elements(s) WITH [(size(s)) := x] #)

  ns: VAR (nonemptystack?)
  pop(ns): stack  =
        (# size := size(ns) - 1,
           elements := (LAMBDA (j | j < size(ns) - 1):
                          elements(ns)(j)) #)

  top(ns): T = (elements(ns)(size(ns) - 1))

  END newstacks
```

Typechecking the above theory yields seven TCCS, all of which are proved automatically by the TCC proof strategy. Note that the pop operation updates the stack by decrementing the `size` field by one, and simultaneously domain-restricting the `elements` field to track the dependency. It is interesting to see what happens when this domain-restriction is omitted by redefining `pop` as shown below.

```
pop(ns): stack  =
      ns WITH [ (size) := size(ns) - 1]
```

In this case, the typechecker successfully typechecks the theory but we are left with one TCC that the proof checker does not prove.

```
pop_TCC2: OBLIGATION
     (FORALL (ns: (nonemptystack?), x1:
                below(size(ns)), y1: below(size(ns) - 1)):
        y1 < size(ns) AND x1 < size(ns) - 1);
```

It is easy to see that this is an unprovable TCC since it requires showing that if `x1` is a natural number less than `size(ns)`, then it is also less than `size(ns)`$-$`1`. Thus there can be two kinds of type errors: one, where the typechecker finds a blatant syntactic violation at the level of arities or simple types, and the second, a more subtle, semantic class of errors where the typechecker generates TCCs that are unprovable.

The PVS specification language has a number of other features that exploit the interaction between theorem proving and typechecking. These include *assumptions* on theory parameters that must be discharged when a theory is imported with actual parameters, user-specified *type conversions* that are automatically inserted when the corresponding type mismatch is detected, *typing judgements* which can be used to assert a more refined type for an existing operation. PVS also has a facility for automatically generating abstract datatype theories with induction and recursion schemes.

Type information is used heavily within a PVS proof. The predicate subtype constraints are automatically asserted to the contextual database of assertions maintained by the decision procedures. Most of the abstract datatype simplifications are also carried out automatically. The `typepred` command of the PVS proof checker makes the type constraints of a given expression explicit in a proof subgoal. The `extensionality` command introduces extensionality axiom schemes corresponding to abstract datatypes or function, record, or tuple types into a proof subgoal. Quantifier instantiations are typechecked and can generate TCC subgoals during a proof attempt. We do not describe these features in further detail here.

The main point of the above discussion is that PVS takes an expansive view of types and automated typechecking to encompass both syntactic and semantic restrictions. The crucial design decision in PVS is the strict separation between automatic typechecking for the simply typed fragment and proof obligation generation for subtyping and other extensions. The practical experience with PVS has been that the type system does rapidly detect a lot of common specification errors.

4 Combining Decision Procedures with Interactive Proof

Decision Procedures. The use of decision procedures is essential for effective proof construction, as we have already seen above, but it is by no means sufficient. Decision procedures can automatically carry out simplifications that may otherwise require tedious amounts of manipulation. They can make use of highly efficient data structures as for example with congruence closure graphs for equality, and binary decision diagrams for boolean simplification. However, most decision procedures operate as black boxes and do not give much insight into why they have succeeded or failed. PVS therefore employs decision procedures for the low-level manipulations used to support user-directed inferences at the higher levels of the proof.

Integrating decision procedures with interactive proof is a delicate exercise. The brute-force nature of decision procedures has to be balanced against the requirement that in an interactive proof, the individual proof steps must be easy to comprehend. In PVS, decision procedures are used very aggressively, but they operate mostly in the background by recording and propagating equality, inequality, and type information. The simplifications wrought by the decision procedures are not always immediately comprehensible but they are typically ones that would be quite tedious to carry out manually.

The basic PVS decision procedures were originally developed by Shostak in the context of the STP theorem prover [29, 30]. These decision procedures use the congruence closure algorithm for equality reasoning to combine decision procedures for various theories such as linear arithmetic, arrays, and tuples, in the presence of function symbols that are uninterpreted by any of the function symbols. As an example, the decision procedures can prove the assertion `arith` by linear arithmetic, the assertion `eq1` by congruence closure, and `eq2` using a combination of the two.

```
x,y,v: VAR real

arith: THEOREM
  x < 2*y AND y < 3*v IMPLIES 3*x < 18*v

f: [real -> real]

eq1: THEOREM x = f(x) IMPLIES f(f(f(x))) = x

g : [real, real -> real]

eq2: THEOREM x = f(y) IMPLIES g(f(y + 2 - 2), x + 2) = g(x, f(y) + 2)
```

Note that the functions symbols `f` and `g` are uninterpreted.

PVS does not merely make use of decision procedures to prove theorems but also to record type constraints and to simplify subterms in a formula using any assumptions that *govern* the occurrence of the subterm. These governing assumptions can either be the test parts of surrounding conditional (IF-THEN-ELSE) expressions or type constraints on governing bound variables. Such simplifications typically ensure that formulas do not become too large in the course of a proof. Also important, is the fact that automatic rewriting is closely coupled with the use of decision procedures, since many of the conditions and type correctness conditions that must be discharged in applying a rewrite rule succumb rather easily to the decision procedures.

Strategies. The PVS proof checker provides powerful primitive inference steps that make heavy use of decision procedures, but proof construction solely in terms of even these inference steps can be quite tedious. PVS therefore provides a language for defining high-level inference strategies (which are similar to *tactics* in LCF [16,24]). This language includes recursion, a `let` binding construct, a backtracking `try` strategy construction, and a conditional `if` strategy construction. For example, one can use recursion and backtracking to repeatedly apply the disjunctive simplification step `flatten`, and the conjunctive splitting step `split`, to define a simple propositional simplification strategy as shown below.

```
(defstep prop ()
  (try (flatten) (prop) (try (split)(prop) (skip))))
```

The above strategy applies the `flatten` step, and if it succeeds and generates a subgoal, then it recursively applies `prop` to that subgoal. If `flatten` has no effect because there are no available disjunctive simplifications in the goal, then `prop` tries to apply the `split` step in a similar manner. The strategy terminates with the do-nothing step `skip` when there are no disjunctive or conjunctive simplifications available in the goal.

Typical strategies include:

- `inst?` for the heuristic instantiation of quantifiers
- `grind` for repeatedly applying rewriting, skolemization, heuristic instantiation, and if-lifting.
- `induct-and-simplify` for applying induction followed by simplifications similar to those given by `grind`.

An Example: The Binomial Theorem. We now examine a somewhat nontrivial example, namely, the proof of the binomial theorem in PVS. First, the number of ways of selecting k distinct elements from a set of n distinct elements, namely $\binom{n}{k}$, was defined as the function `chooses`. The lemma `chooses_step` isolates a simple case of the definition.

```
i, j, k, l, m, n: VAR nat

chooses(n, k):
    RECURSIVE
      nat = (IF k = 0 THEN 1
             ELSIF n < k THEN 0
             ELSE chooses(n - 1, k) + chooses(n - 1, k - 1)
             ENDIF)
     MEASURE (n + k)

chooses_step: LEMMA
      k <= n
        IMPLIES chooses(n + 1, k + 1) =
                chooses(n, k) + chooses(n, k + 1)
```

The exponentiation and summation operations are defined recursively as shown below. The definition of exponentiation holds no surprises, but the summation operation **sigma** is defined on domain that is a dependent tuple type [**n** : **nat**, **rat_array**(**n**)], which is not unlike the type used above to define **stack**. This refined typing of **sigma** helps to streamline the inductive proof of the binomial theorem.

```
x, y, z: VAR rational

exp(x, k):
    RECURSIVE rational =
    (IF k > 0 THEN exp(x, k - 1) * x ELSE 1 ENDIF)
     MEASURE k

rat_array(n): TYPE = [below[n] -> rational]

sigma(n, (f: rat_array(n))):
    RECURSIVE
      rational =
      (IF n = 0
       THEN 0
       ELSE f(n - 1)
            + sigma(n - 1, (LAMBDA (i: below[n - 1]): f(i)))
       ENDIF)
     MEASURE (LAMBDA n, (f: rat_array(n)): n)
```

Three lemmas about the summation operation **sigma** are needed in the main proof of the binomial theorem. All three lemmas are proved using a simple variant of the **induct-and-simplify** strategy. The first one requires two additional steps, but the other two lemmas are proved automatically by the strategy.

```
sigma_split0: LEMMA
      (FORALL n, (f: rat_array(n + 1)):
         sigma(n + 1, f) =
         f(0) + sigma(n, (LAMBDA (i: below[n]): f(i + 1))))

sigma_times: LEMMA
      (FORALL n, (f: rat_array(n)):
         sigma(n, f) * x =
         sigma(n, (LAMBDA (i: below[n]): f(i) * x)))

sigma_plus: LEMMA
      (FORALL n, (f, g: rat_array(n)):
         sigma(n, (LAMBDA (i: below[n]): f(i) + g(i)))
           = sigma(n, f) + sigma(n, g))
```

The binomial theorem was then stated in the form shown below.

```
bin_thm: THEOREM
      exp(x + y, n)
        =
        sigma(n + 1,
              (LAMBDA (k: below[n + 1]):
                 chooses(n, k) * exp(x, k) * exp(y, n - k)))
```

The proof of `bin_thm` is by induction and requires about eighteen interactions in order to apply the lemmas and expand the definitions in an orderly manner. This number of interactions could be reduced through further experimentation. The main point is, however, that with the use of refined typing, decision procedures, and strategies, the number of interactions roughly corresponds to the number of suggestive hints in a textbook proof. The level of detail is also roughly similar to that of a textbook proof. The entire verification occupied less than a day.

5 Integrating Model Checking and Theorem Proving

In the theorem proving approach to program verification, one verifies a property P of a program M by proving $M \supset P$. The model checking approach verifies the same program by showing that the state machine for M is a satisfying model of P, namely $M \models P$. For control-intensive approaches over small finite states, model checking is very effective since a more traditional Hoare logic style proof involves discovering a sufficiently strong invariant. These two approaches have traditionally been seen as incompatible ways of viewing the verification problem. In recent work [27], we were able to unify the two views and incorporate a model checker as decision procedure for a well-defined fragment of PVS.

This integration uses the mu-calculus as a medium for communicating between PVS and a model checker for the propositional mu-calculus. The advantage of this integration can be illustrated by means of an inductive proof of an N-process mutual exclusion protocol where the induction step used the correctness of the 2-process version of the protocol as verified by the model checker. This example is described very briefly below.

The Mu-Calculus. The general mu-calculus over a given *state* type essentially provides operators for defining least and greatest fixpoints of monotone predicate transformers. The propositional mu-calculus is a restriction of the general mu-calculus where the state type is an n-tuple of booleans. The propositional mu-calculus thus extends quantified boolean formulas (i.e., propositional logic with boolean quantification) to include the application of n-ary relational terms to n argument formulas [20]. The relational terms can constructed by means of lambda-abstraction, or by taking the least fixpoint $\mu Q.F[Q]$ where Q is an n-ary predicate variable and F is a monotone predicate transformer. The greatest fixpoint operation can be written as $\nu Q.F[Q]$ and defined as $\neg\mu Q.\neg F[\neg Q]$. The mu-calculus can be easily defined in PVS as shown in the theory `mu_calculus` below.

```
mucalculus[T:TYPE]: THEORY
  BEGIN

  s: VAR T
  p,p1,p2: VAR pred[T]
  predtt: TYPE = [pred[T]->pred[T]]
  pp: VAR predtt
  setofpred: VAR pred[pred[T]]

  IMPORTING orders[pred[T]]
  <=(p1,p2):bool = FORALL s: p1(s) IMPLIES p2(s)

  ismono(pp):bool =
    FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2)

  glb (setofpred):pred[T] =
    LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES  p(s))

  % least fixpoint
  lfp (pp):pred[T] = glb(p | pp(p) <= p)

  mu(pp): pred[T] = lfp(pp)

END mucalculus
```

We omit the detailed explanation of the above specification except to note that `pred[T]` is an abbreviation for the type `[T -> bool]` which consists of the predicates over `T`. When a predicate transformer `pp` is monotonic (given by `ismono`), the least fixpoint operator `lfp` yields a fixpoint. The usual Tarski-Knaster argument for this can be easily verified in PVS.

More usefully, the temporal operators of the branching time logic CTL can be defined using the mu-calculus. A sampling of these definitions is shown below. Given a next-state relation `N`, `EX(N, f)` holds of a state if there exists a next state where `f` holds. The assertion `EG(N, f)` holds at a state if there is an infinite path along which `f` always holds. The assertion `AG(N, f)` holds at a state if `f` holds along all outgoing paths.

```
ctlops[state : TYPE]: THEORY
  BEGIN
  IMPORTING mucalculus[state], connectives[state]

  u,v,w: VAR state
  f,g,Q,P,p1,p2: VAR pred[state]
  Z: VAR pred[[state, state]]

  N: VAR [state, state->bool]

  EX(N,f)(u):bool =  (EXISTS v: (f(v) AND N(u, v)))

  EG(N,f):pred[state] = nu (LAMBDA Q: (f AND EX(N,Q)))

  EU(N,f,g):pred[state] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))

  EF(N,f):pred[state] = EU(N,(LAMBDA u: TRUE),f)

  AG(N,f):pred[state] = NOT EF(N, NOT f)

END ctlops
```

Fairness is not definable in CTL but fair variants of the CTL operators can be defined in the mu-calculus.

When the state type is finite, i.e., constructed inductively from the booleans and scalar types using records, tuples, or arrays over subranges, the mu-calculus over such finite types (and the corresponding CTL) can be translated into the propositional mu-calculus and model checking can be used as a decision procedure for this fragment. We do not discuss the details of this encoding here (see [27]).

An efficient model checking algorithm for the propositional mu-calculus was presented by Emerson and Lei [13], and the symbolic variant employing BDDs was presented by Burch, *et al* [4]. The above embedding of the mu-calculus and CTL owes much to these seminal papers.

Compositionality. We now digress from our main point to describe a technique for compositional verification of concurrent systems [28]. We say that the parallel composition $P_1 || P_2$ of two specifications P_1 and P_2 is *compositional* if every local property, that is, a property of one of the P_i, is also a global property of the composition $P_1 || P_2$. A system component P is specified as a triple $\langle I, G, R \rangle$ consisting of an initialization predicate I on the state type, and two *actions* or binary predicates on the state type, G and R. The action G represents the *guarantee* action performed by P, and the action R is a constraint on actions performed by the environment. A trace satisfying a specification P is sequence of states where the initial state satisfies I and the adjacent states are related by G or R actions, indicating an interleaving between system and environment actions.

The composition of two specifications P_1 given by $\langle I_1, G_1, R_1 \rangle$ and P_2 given by $\langle I_2, G_2, R_2 \rangle$ is the system

$$\langle I_1 \wedge I_2, \qquad (G_1 \wedge R_2) \vee (G_2 \wedge R_1), \qquad R_1 \wedge R_2 \rangle.$$

Here, conjunction (disjunction) of predicates on state is the pointwise conjunction (disjunction) of the predicate applied to individual states. The main point is that there is no attempt here to discharge any rely-guarantee proof obligations since we are dealing with specifications (where I, G and R are only partially specified) and not programs. The main outcome of the above form of composition is that any trace of $P_1 || P_2$ is also a trace of P_1 and of P_2. If a property is simply a subset of traces, then any property of P_1 or P_2 is also a property of $P_1 || P_2$.

Two Process Mutual Exclusion. Peterson's mutual exclusion algorithm [26] is a easy exercise for a model checker, and is somewhat less easily verified by means of theorem proving. We merely sketch the details of the PVS verification of this algorithm. Compositionality is not necessary for specifying this algorithm, but it is used here to properly abstract the N-process version where compositionality is need.

Given processes `P` and `Q`, the global system state consists of five boolean variables which can be represented as a record with five fields in PVS. The state component `tryp` (`tryq`) indicates whether `P` (`Q`) is trying to get into or is already in its critical section. The state component `CSP` (`CSQ`) indicates whether `P` (`Q`) is in its critical section. The state component `turn` is used to arbitrate between `P` and `Q`. The initialization predicate for `P` asserts that `tryp` (`tryq`) and `CSP` (`CSQ`) are both `FALSE`. In its G or guarantee action, the process `P` can either stutter, enter its trying phase by setting `tryp` to `TRUE` and `turn` to `TRUE`, or enter its critical section by first checking whether `TURN` or `trypq` is `FALSE`, and then setting `CSP` to `TRUE`. The R or rely part of P requires `tryp` and `CSP` to be left unchanged. The specification of the `Q` process is similar except that `turn` is set to `FALSE` on entering the trying phase and checked to be `TRUE` before entry into the critical section. The main safety property of the protocol can be stated as an **AG** property in the CTL fragment shown above. In the statement below, `abs_pet` is the composition of `P` and `Q`. The I componenent is given by `init(abs_pet)`.

```
correct: LEMMA
      init(abs_pet)(s) IMPLIES
       AG((LAMBDA s0, s1: G(abs_pet)(s0, s1) OR R(abs_pet)(s0, s1)),
          (LAMBDA s0: NOT (CSP(s0) AND CSQ(s0))))(s)
```

The proof of `correct` using model checking takes only about fifteen seconds of machine time, most of which is spent outside the model checker.

N-process mutual exclusion. The algorithm we use here is a simple recursive variant of the two-process Peterson algorithm. The `N` processes are numbered from `0` to $\mathtt{N}-\mathtt{1}$. A process `i` can be at a critical section level between `i` and `N` given by the variable `cs(i)`. The object of the algorithm is to ensure that at most one process is at critical section level `N`. Each process `i` has a `turn` component `turn(i)` for coordinating with its predecessor. Each process also as boolean component `try(i)` to indicate whether it is trying to enter the critical section or is already in its critical section. Each process `i` starts a critical section level `i` and with `try(i)` set to `FALSE`. It executes a two-process Peterson protocol with the one preceding process `j`, if any, such that `cs(j)` is `i`. The "winner" reaches critical section level $\mathtt{i}+\mathtt{1}$ and gets to compete with the $\mathtt{i}+\mathtt{1}$st process. The rely conditions ensure that the `turn` variables are modified in a constrained manner, since otherwise the algorithm might violate the mutual exclusion property.

The protocol for `n` processes is given by `Pet(n)`. The correctness assertion is stated as `safety` below.

```
AG_C(Cprog, Cprop) : bool =
 (FORALL s: init(Cprog)(s)
   IMPLIES AG((LAMBDA s0, s1: G(Cprog)(s0, s1) OR
                              R(Cprog)(s0, s1)),
              Cprop)(s))

safety: LEMMA AG_C(Pet(n),
             (LAMBDA s: (FORALL (j, k: below[n]):
                          cs(s)(j) >= n AND  cs(s)(k) >= n
                      IMPLIES j = k)))
```

The informal proof of the protocol is by induction. If there is exactly one process, the proof is trivial. This base case of the proof could also have been verified by means of abstraction and model checking. At the induction step, we know that there is at most one process below `i` that is at critical section level `i`. We can use this induction hypothesis to abstract the first `i` processes to the process `P` in the two-process algorithm, and to abstract process `i` to the process `Q`. If we have ensured that the abstraction preserves the correctness property, then the correctness of the `N`-process algorithm follows. Such abstractions have been studied by Clarke, Grumberg, and Long [5], and by Dams, Grumberg, and Gerth [11].

The above example is of course purely illustrative since the algorithm is a somewhat naive and contrived one. It shows how theorem proving and model checking can be combined to simplify an argument by using compositionality, induction, and abstraction.

6 Observations and Conclusion

Despite all our claims above, mechanical verification is not easy but the main difficulty is in constructing a correct proof. Mechanization is the only reasonable technique for keeping the verification process honest. Furthermore, mechanization delivers extremely valuable feedback by catching even subtle errors that might have otherwise gone unnoticed. There are several examples of published proofs of important results where the errors have gone unnoticed until they were revealed by mechanical verification. Mechanical verification is not a substitute for the social process [12] but is merely a tool that aids the social process. There is nothing inherently inhuman about mechanized verification. A mechanical verifier plays the role of an implacable skeptic that highlights weaknesses in an argument such as misapplied lemmas, missing cases, and inappropriate inductions or instantiations.

We have argued that computer-aided computing can be realized by carefully designing language features for mechanized syntactic and semantic checks, and by integrating powerful and efficient decision procedures and model checking within an interactive proof checking framework.

There still are certain inflexibilities with mechanical verification since the technology is still in its infancy. For example, it is not yet easy to introduce new notation or switch formalisms. Computing is rife with such formalisms. It is typically not easy to introduce derived inference rules as is frequently done in informal mathematics. Informal mathematics also often mixes mathematical and metamathematical argumentation, and this too is not easily mechanized.

The case for mechanized verification is very strong. We now know that it is feasible to verify large systems. Bevier, Hunt, Moore, and Young [1] present a proof of a system consisting of a microprocessor, an assembler for this microprocessor, a compiler for a Pascal-like language with the assembler as its target language, and an operating system kernel. This verification used the Boyer-Moore theorem prover [2]. There are many other impressive verification examples carried out on a variety of systems including Coq [7], EVES [8], IMPS [14], HOL [17], Isabelle [25], LP [15], Nuprl [6], and RRL [19], to name a few.

The main challenge is to make mechanical verification less arduous and more perspicuous. Specific challenges are in

- the design of expressive specification languages that interact synergistically with mechanization
- the integration of decision procedures with interactive proof development
- support for multiple formalisms
- the design of customized domain-specific verification methodologies.

References

1. William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
2. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, NY, 1988.
3. J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
5. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
6. R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
7. 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.
8. Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. EVES: An overview. In S. Prehn and W. J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag. Volume 1: Conference Contributions.
9. 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.
10. David Cyrluk, Patrick Lincoln, Paliath Narendran, Sam Owre, Sreeranga Ragan, John Rushby, Natarajan Shankar, Jens Ulrik Skakkebæk, Mandayam Srivas, and Friedrich von Henke. Seven papers on mechanized formal verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1995.
11. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994.
12. Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.
13. E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.

14. William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2):213–248, October 1993.
15. Stephen J. Garland and John V. Guttag. LP: The Larch prover. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction (CADE)*, volume 310 of *Lecture Notes in Computer Science*, pages 748–749, Argonne, IL, May 1988. Springer-Verlag.
16. 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.
17. 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.
18. Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Real Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994. IEEE Computer Society.
19. D. Kapur and H. Zhang. *RRL: A User's Manual*. General Electric Corporate Research and Development, Schenectady, NY, March 1986. Unpublished Manuscript.
20. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, December 1983.
21. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society. To appear.
22. Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA Langley Research Center, 1995.
23. 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. Three volumes: Language, System, and Prover Reference Manuals.
24. L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, Cambridge, England, 1987.
25. Lawrence C. Paulson. *Isabelle: A generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
26. G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
27. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, Lecture Notes in Computer Science, Liege, Belgium, June 1995. Springer-Verlag. To appear.
28. N. Shankar. A lazy approach to compositional verification. Technical Report SRI-CSL-93-8, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
29. 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)*, volume 138 of *Lecture Notes in Computer Science*, New York, NY, 1982. Springer-Verlag.
30. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.