Theorem Proving for Verification*

John Rushby

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
rushby@csl.sri.com

Abstract. The challenges in using theorem proving for verification of parallel systems are to achieve adequate automation, and to allow human guidance to be expressed in terms of the system under examination rather than the mechanisms of the prover. This paper provides an overview of techniques that address these challenges.

1 Introduction

Formal verification is accomplished by constructing a mathematical model of the computer program, hardware, or system concerned, and then using calculation to determine whether the model satisfies desired properties. This is exactly analogous to the ways in which mathematical modeling and calculation are used to validate the structural design of a bridge, say, or the aerodynamic properties of a wing. However, the appropriate mathematics for modeling computer systems is formal logic, and calculation is accomplished by formal deduction—which has far higher computational complexity (indeed, many questions are algorithmically undecidable) than solvers for the partial differential equations used to model many physical phenomena. Consequently, formal calculation often requires recourse to human guidance or to simplified models.

Formal deduction by human-guided theorem proving (i.e., interactive proof checking) can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill. On the other hand, the models employed with automated methods such as model checking may be so simplified or "downscaled" that their verification does not necessarily guarantee that the property concerned will hold of the actual system. Mechanization of formal deduction in support of verification must therefore strike a delicate balance between the extent and the form of human guidance required, the heuristic and algorithmic automation provided, and the convenience and fidelity of the modeling supported.

In the following sections I briefly sketch some of the main issues in theorem proving for formal verification, particularly for parallel processes, and outline the choices made in the PVS and SAL verification systems [3, 20].[1]

* This research was supported by DARPA through USAF Rome Laboratory Contract F30602-96-C-0204 and USAF Electronic Systems Center Contract F19628-96-C-0006.

[1] These systems are the work of my colleagues at SRI, most notably Sam Owre and N. Shankar.

2 Choice of Logic and Language

Formal verification requires a formal model of the system concerned (usually, a model of its assumed environment is required also), and descriptions of the properties to be examined. When theorem proving provides the means of calculation, both of these models are constructed as formal specifications in a single specification language. Necessary "background" knowledge of the domain is also specified in the same language and verification is accomplished by showing that the required properties are entailed by the other elements of the specification:

$$background + environment + system \vdash requirement.$$

One of the most fundamental choices in this approach to verification is that of the logic on which to base the specification language. There is a tradeoff, at least in theory, between expressiveness of the logic and the automation that can be provided for it. Thus, most classical theorem proving research focusses on standard first-order logic because it can be semi-decided by resolution. In practice, however, the difficulties in performing formal verification are dominated by the need to deal effectively with interpreted theories such as arithmetic and datatypes, where resolution offers little help. Thus, restriction to standard first-order logic provides few benefits, while imposing an impoverished notational foundation on the specification language.

The discipline of strong typing is at least as beneficial in specification as in programming, and it is also very convenient to be able to supply functions as arguments to other functions (e.g., to parameterize a sorting function with respect to the sorting criterion) and to be able to quantify over functions (e.g., to specify what it means for a sorting criterion to be well-founded). Higher-order logic provides both these benefits, together with an adequate treatment of set theory, and is the foundation for several popular specification languages, including PVS, where the type system is further extended with predicate subtypes and dependent types. Some of the conveniences and benefits of these choices are illustrated in [22].

Although some of the problems in theorem proving for higher-order logic are harder than for first-order (e.g., matching is undecidable), this is of little practical moment: instances of higher-order problems that arise in practice are usually in some easy fragment. The difficulties that dominate practical verification concern the treatment of interpreted theories, selection and instantiation of axioms and lemmas, and case-splitting—and these are essentially the same in all frameworks. The core capabilities required to perform these are outlined in the next section.

3 Basic Mechanization

As noted above, practical verification problems demand the ability to deal effectively with (sub)formulas involving equality and uninterpreted functions, real and integer linear arithmetic, abstract datatypes (such as lists and trees), and combinations of these. The following formulas illustrate some of these (the symbol $\supset$ denotes implication, and formulas are implicitly universally quantified in their free variables).

Equality: $x = y \wedge f(f(f(x))) = f(x) \supset f(f(f(f(f(y))))) = f(x)$.
Linear Arithmetic: $x \leq y \wedge x \leq 1 - y \wedge 2 \times x \geq 1 \supset F(4 \times x) = F(2)$.
Datatypes: $\mathrm{car}(\mathrm{cons}(a, b)) = a$.
Combination: $x \leq \mathrm{car}(a) \supset \mathrm{car}(a) \geq \mathrm{car}(\mathrm{cons}(x, b))$.

The first-order (i.e., quantified) forms of these theories are decidable individually, but their combination is not. Experience shows that the most effective way to deal with this difficulty is to decide the unquantified combination, and to handle quantifier instantiation separately by heuristic or interactive means. Algorithms for deciding unquantified combinations of suitable theories were introduced by Shostak [28] and by Nelson and Oppen [19]; equality with uninterpreted functions is decided simply by the method of congruence closure [26], while linear arithmetic can be decided by several methods, including loop residue [27] and SUP-INF [25]. PVS uses methods based on those of Shostak [8].

Rewriting is another capability that is crucial to effective verification: it can be used to expand definitions and automatically to locate and instantiate appropriate lemmas or axioms. Although some theories can be decided by rewriting whenever any rule applies, most verification problems demand more controlled application. Suitable heuristics are those that rewrite a conditional rule only when its condition can first be discharged (or, equivalently, when it can be decided which case of a top-level if-then-else applies in an unconditional rule). Thus, the following rewrite rule would be applied in contexts containing $factorial(n)$ where n is known to be zero or is known to be nonzero, but not otherwise:

$$factorial(n : nat) \stackrel{\mathrm{def}}{=} \text{if } n = 0 \text{ then } 1 \text{ else } n \times factorial(n - 1).$$

Such heuristics require use of decision procedures during rewriting, together with recursive application of the rewriter, so that the integration between the rewriter and decision procedures becomes a critical element in the overall performance of the theorem prover. Forward chaining is another mechanism for invoking suitable lemmas and axioms automatically and, like rewriting, it is most effective when integrated with decision procedures.

Propositional calculus must also be decided efficiently, so procedures based on BDDs (ordered binary decision diagrams) or on the Davis-Putnam or Stålmark methods are attractive ([12] provides references to all of these and describes an interesting new method). However, it is seldom that the best approach is to expand everything out and then call a propositional satisfiability procedure on the (usually huge) result: rather, intelligent case analysis (that avoids the unnecessary work of dealing repeatedly with similar subgoals) is what is required, so that propositional calculus is another capability that must be integrated with rewriting and decision procedures. That integration then forms the core mechanization that supports the remaining parts of the theorem prover.

Those remaining parts consist of interactive and tactical guidance having the goal of reducing the overall problem to decidable subproblems as quickly as possible. Given the core mechanization described above, the tasks left interactive human guidance are selecting the rules and definitions to make available to the rewriter, explicitly invoking lemmas and axioms not having the form of rewrite (or forward chaining) rules, suggesting instantiations for quantified variables, and guiding case analysis where necessary.

Systematic patterns of interactions can be captured as macros or, more powerfully, as programs (having the ability to adjust their control flow according to the formulas of the theorem being proved and the outcome of other proof steps): such "tactic" or "strategy" languages provide a convenient way for developers and users to extend the capabilities of the core mechanization. First introduced in the LCF system [10] (whose "metalanguage" became the programming language ML), such languages and extension capabilities are found in many verification systems, including PVS.

4 The Challenge of Human Guidance

The mechanisms described above are present to some degree in all theorem provers that have proved effective for verification problems.[2] These provers can be quite productive for skilled users, but still are insufficiently automatic for general use. The main barrier to greater automation is quantifier instantiation: heuristic instantiation too often guesses incorrectly and sends the proof down a wrong path.[3] It seems that some combination of proof search (shades of resolution!) and exploitation of decision procedures can significantly improve quantifier elimination (PVS includes experimental capabilities of this sort), but that some human guidance will always be required for verification problems of any substance. The design challenge then becomes one of choosing the most effective and acceptable form to present this guidance.

Theorem provers currently require guidance to presented in their terms: an interactive theorem prover such as PVS requires the user to suggest case-splits, lemmas, and so on during the course of a proof; a noninteractive prover such as ACL2 receives human guidance through the selection and ordering of the lemmas it is invited to prove. Neither of these forms of guidance seems acceptable to nonspecialists: they wish to provide guidance in terms of the *system design* (its properties and structure), not in terms of its proof. The example of model checking is instructive here: models must often be significantly downscaled to be acceptable to a model checker, yet this form of "guidance" does seem acceptable to nonspecialists—presumably because it directly concerns, and is conducted in terms of, the design. I return to this topic in Section 7, but need first to discuss the additional issues raised by parallel processes.

5 Verifying Parallel Processes

Theorem provers operate on specification languages that are basically logics: models of computation are not built-in but must be constructed on the resources of the specification logic concerned. The construction can be chosen to suit the application: for some purposes, sequential programs are best modeled functionally, while for others a Hoare-style axiomatic treatment may be preferred. Parallel programs can also be modeled in

[2] These differ primarily in their choice of logic—ACL2 [16] uses an unquantified and untyped first order logic—and in the extent to which they employ decision procedures—those of the HOL [11] tradition generally eschew them.

[3] Unquantified logics such as ACL2 avoid the problem of instantiation, but instead must use induction (arguably a harder problem) in similar situations.

a variety of ways—for example, as state machines or process algebras. To make things concrete, I will focus on their representation as state machines, and will concentrate on safety properties.

In this representation, a parallel program is modeled as a nondeterministic automaton over a possibly infinite set of states. Given set of states S, initiality predicate I on S, and transition relation T on S, a predicate P on S is *inductive* for $\mathcal{S} = (S, I, T)$ if

$$I(s) \supset P(s) \tag{1}$$

and

$$P(s) \wedge T(s,t) \supset P(t). \tag{2}$$

The *reachable states* are those characterized by the smallest (ordered by implication) inductive predicate R on $\mathcal{S}$. A predicate G is an *invariant* or *safety property* if it is larger than R (i.e., includes all reachable states). Since the focus here is on safety (as opposed to liveness) properties, we need not be concerned with the acceptance criterion on the automaton $\mathcal{S}$.

Using theorem proving, the usual way to establish that a predicate G is invariant to show that it is inductive—i.e., we attempt to prove the verification conditions (1) and (2) with G substituted for P. Unfortunately, most safety properties are not inductive, and must be strengthened (i.e., replaced by a smaller property) to make them so. Typically, this is done by conjoining additional predicates in an incremental fashion, so that G is replaced by

$$G^i_\wedge = G \wedge G_1 \wedge \cdots \wedge G_i \tag{3}$$

until an inductive $G^m_\wedge$ is found. This process can be made systematic, but is always tedious. In one well-known example, 57 such strengthenings were required to verify a bounded retransmission communications protocol [14]; each G_{i+1} was discovered by inspecting a failed proof for inductiveness of $G^i_\wedge$, and the process consumed several weeks.

Some improvements can be made in this process: static analysis [4] and automated calculations of (approximations to) fixpoints of weakest preconditions or strongest postconditions [7] can discover many useful invariants that can be used to seed the process as $G_1, \ldots, G_i$. Nonetheless, the transformation of a desired safety property into a provably inductive invariant remains the most difficult and costly element in verification of parallel programs using theorem proving. One of the chief attractions to model checking is that it calculates reachability directly and thereby eliminates the need to strengthen proposed invariants to make them inductive. The drawbacks to model checking are that the system must have a finite (and fairly small—typically less than about 2^{300}) number of states, and must be represented in a very concrete form (e.g., no uninterpreted functions).

6 Abstraction

A very attractive idea is to combine theorem proving and model checking in a way that preserves their advantages while each compensates for the deficiencies of the other. Abstraction provides a bridge between the two methods: we construct a simpler model of the system that can be model checked, and use theorem proving to establish that the simpler model is a property-preserving abstraction of the original system [18]. If S_c, I_c, T_c, P_c and S_a, I_a, T_a, P_a represent the concrete (i.e., original) stateset, initiality predicate, transition relation, and desired invariant, and their abstract counterparts, respectively, and $\phi : S_c \to S_a$ is the abstraction function, then the proof obligations that must be discharged are the following.

Initiality simulation: $I_c(s) \supset I_a(\phi(s))$,
Transition simulation: $T_c(s,t) \supset T_a(\phi(s), \phi(t))$,
Safety Preservation: $P_a(\phi(s)) \supset P_c(s)$, and
Abstract Invariant: $I_a(s) \supset \mathrm{AG}(T_a, P_a)(s)$, where AG is the "globally always" modality of CTL.

The first three can be discharged by theorem proving, and the last by model checking. Here, model checking the reduced model does verify the original, because the other proof obligations ensure that the reduced model is a property-preserving abstraction of the original. The necessary integration of theorem proving and model checking can be cleanly handled by using the mechanisms of a model checker to provide a decision procedure for the finite fragment of a μ-calculus (quantified Boolean logic extended with fixpoint operators or $\Box$ and $\Diamond$ modalities) [21]. Such μ-calculi can be specified within higher-order logic and are sufficiently expressive to define CTL model checking.

The drawback to this approach is that discharging the transition simulation proof obligation usually requires auxiliary invariants: in the bounded retransmission example cited earlier, 45 invariants were required to discharge this obligation. An exciting way to reduce this burden is described in the next section.

7 Automated Abstraction

Instead of constructing a downscaled system model by hand and then proving that it is, indeed, a property-preserving abstraction, an alternative approach is to *calculate* a reduced system description that has this property by construction. Given an abstraction function relating the original and abstracted state spaces, a verification condition can be generated for each pair of abstract states that specifies the conditions under which no transition is required between those states in the abstracted system description (the condition is that there is no transition between any pair of original states that map to those abstract states). If this verification condition can be proved (using automatic proof procedures), then the transition can be omitted from the abstracted system description; if not, then it is conservative to include the transition. For the bounded retransmission protocol, this approach is able to compute automatically an abstracted system description suitable for model checking [5].

A particularly useful form of abstraction is *predicate abstraction*, in which predicates on the concrete state are abstracted to Boolean values [23]. The abstraction function is then defined implicitly in terms of the list of predicates to be abstracted, and the abstracted system can be constructed by a very efficient method [24]. This approach is available as a standard command in PVS and also is able to verify the bounded retransmission protocol automatically in a few seconds.

One of the great attractions of approaches based on automated abstraction is that they make very effective use of fully automated theorem proving: instead of requiring the theorem prover to tackle one big theorem (the desired invariant), they instead invoke it on large numbers of smaller problems (the verification conditions that construct the abstracted system). Not only are these problems smaller and simpler (making the automation more likely to succeed), but it is not catastrophic if the automation sometimes fails to prove true verification conditions: the only penalty is that the abstraction will be more conservative than necessary—but it may still be good enough.

When an automatically constructed abstraction is *not* good enough (i.e., when model checking fails to verify the abstracted property because the abstracted system is insufficiently precise), it is sometimes possible to use the counterexample trace to help refine the abstraction. In other cases, auxiliary invariants may be required—and the combination of automated abstraction and model checking can sometimes be used in a recursive fashion to help develop these. The idea is to construct an even simpler abstraction than the one desired, and then use a model checker to calculate the reachable states of that abstraction (most model checkers can do this). The reachable states characterize the strongest invariant of that abstraction; their concretization (the inverse of abstraction) is an invariant—and plausibly a strong one—of the original system. An invariant discovered in this way can often be strengthened further by a few iterations of a strongest postcondition computation. This approach suggests iterated application of these techniques (one abstraction is used to calculate an invariant that helps calculate another abstraction, and so on), which in turn suggests a blackboard architecture in which a common intermediate representation is used to accumulate useful properties and abstractions of a design in a form that many different tools can both use and contribute to. A common intermediate representation also provides a way to support many source notations without inordinate cost. Verification systems that explore this approach include SAL [3], which is under development at SRI, and InVeSt [6], which is a collaboration between Verimag and SRI (InVeSt is built on PVS and is being integrated with SAL). Other experiments using related methods have been reported recently [1, 9].

The great benefit of these approaches is that human guidance to the process of iterated abstraction and model checking is conducted in terms of properties related to the design: the user suggests predicates to be abstracted, examines counterexample traces or derived invariants, and suggests new predicates. Theorem proving is a central element in the mechanization, but is fully automated and conducted behind the scenes.

8 Summary

Verification using theorem proving has two great advantages over algorithmic methods such as model checking: it can deal with unbounded or infinite systems, and it can support highly expressive, yet abstract, specifications of system models and properties. Its disadvantages are that it generally requires extensive human guidance, and that guidance is expressed in terms of the operation of the theorem prover, rather than in terms of the problem. Powerful automation, including extensive use of decision procedures and integration with model checking through automated abstraction, promises to overcome these disadvantages.

Opportunities for further research include new decision procedures and new ways to exploit decision procedures (e.g., abstracting to WS1S [2]), returning counterexamples from failed proof attempts, improved methods for generating and strengthening invariants, and new methods for integrating the techniques of theorem proving with those of model checking and reachability analysis.

Acknowledgment

My colleague N. Shankar is the source of most of the ideas described here, and also provided helpful comments that improved their presentation.

References

Papers on formal methods and automated verification by SRI authors can generally be found at `http://www.csl.sri.com/fm-papers.html`.

[1] Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [13], pages 146–159.

[2] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, pages 188–203, Berlin, Germany, March 2000.

[3] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, NASA Langley Research Center, Hampton, VA, June 2000. To appear.

[4] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[5] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [15], pages 319–331.

[6] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. InVeSt: A tool for the verification of invariants. In Hu and Vardi [15], pages 505–510.

[7] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[8] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, Volume 1104 of Springer-Verlag *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996.

[9] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [13], pages 160–171.

[10] 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.

[11] 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.

[12] Jan Friso Groote and Joost P. Warners. The propositional formula checker HeerHugo. *Journal of Automated Reasoning*, 24(1–2):101–125, February 2000.

[13] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[14] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, Volume 1051 of Springer-Verlag *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996.

[15] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, Volume 1427 of Springer-Verlag *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998.

[16] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000.

[17] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[18] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[19] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[20] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[21] 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*, Volume 939 of Springer-Verlag *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995.

[22] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[23] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997.

[24] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [13], pages 443–454.

[25] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[26] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.
[27] Robert E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
[28] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

A Example: Bakery

This appendix uses Lamport's well-known "Bakery" Algorithm [17] for distributed mutual exclusion to illustrate some of the topics outlined in the body of the paper. The specifications and verifications presented here use PVS, but I do not provide an introduction to PVS: I hope the material is fairly self-explanatory, but those who would like to understand more can obtain the PVS system, its manuals, and its tutorials from `http://pvs.csl.sri.com`. The PVS files for the specifications and proofs used here are available from `http://www.csl.sri.com/~rushby/movep2k.html`.

The Bakery Algorithm is intended to ensure mutual exclusion among a collection of concurrent processes. The algorithm resembles the queuing discipline in a bakery or deli, where a machine dispenses tickets printed with numbers that increase monotonically; each customer desiring service takes a ticket and the unserved customer with the lowest number is served. The algorithm employs this basic idea without needing the centralized ticket dispenser: a process desiring to enter its critical section sets its counter (which takes the role of the ticket) to be one greater than that of any other process. The process whose counter is least may enter its critical section; on leaving the critical section, it sets its counter back to zero. The property we would like to verify for this algorithm is mutual exclusion: at most one process is in its critical section at any time.

In constructing a formal model for this system, we must make some choices about the level of detail that should be considered. The algorithm uses shared memory and is sensitive to the exact "memory model" used: for example, what is the value obtained by a process that reads the counter of another process simultaneously with that process updating it? It requires judgment to select a suitable level of detail in constructing the formal model. If we wish to examine the mutual exclusion property of a particular implementation of the algorithm, we will need to consider the memory model employed—which could be quite challenging. Instead (or at first), we may prefer to focus on the behavior of the algorithm in an ideal environment with sequentially consistent memory.

For example purposes, I will use an interleaving model of concurrency (where there are no simultaneous accesses) and limit attention to the case of just two processes. And although each process will perform activities other than merely execute the Bakery Algorithm, I will abstract these details away and assume that each process is always in one of three "phases":

idle: performing work outside its critical section,
trying: to enter its critical section, or
critical: performing work inside its critical section.

Each process has a system state comprising two values:

- The value of its counter, recorded in the variables `t1` or `t2` (for Process 1 and 2, respectively) which is a natural number, and
- The phase it is in, recorded in its "program counter" `pc1` or `pc2` respectively, which can take values `idle`, `trying`, `critical`.

The algorithm can then be specified by the following transitions in a Unity-like notation.

```
Type
    phase: enum {idle, trying, critical};
    counter: nat;
Var
    pc1, pc2: phase;
    t1, t2: counter;
Initially
    t1 := 0;        t2 := 0;
    pc1 := idle;    pc2 := idle;
Rules
    pc1 = idle ==>  t1 := t2+1; pc1 := trying;
    pc2 = idle ==>  t2 := t1+1; pc2 := trying;

    pc1 = trying & (t1 < t2 | t2 = 0) ==> pc1 := critical;
    pc2 = trying & (t1 > t2 | t1 = 0) ==> pc2 := critical;

    pc1 = critical ==> t1 := 0;   pc1 := idle;
    pc2 = critical ==> t2 := 0;   pc2 := idle;

    else ==> do nothing;
Invariant
    NOT (pc1=critical & pc2=critical);
```

There is a peculiarity of this specification that a process that *can* make a move *must* do so (e.g., an `idle` process must go to its trying phase). I have introduced this peculiarity in order to provoke outcomes that help me illustrate certain points during the analysis.

Observe that the algorithm has infinite state: the counters can increase without bound. If we wish to analyze the algorithm using finite state methods, we must fix the maximum counter value and then take some special action if an attempt is made to increase a counter beyond that value—but then we are modeling a modified algorithm. Although we cannot verify the original algorithm by model checking the downscaled version, we may be able to find bugs by this means. It is generally worthwhile to debug a design to the extent possible using inexpensive techniques such as typechecking, animation, simulation, and model checking on downscaled models, before undertaking more expensive analyses such as proof. Once we have established some confidence that the mutual exclusion property is true for the downscaled algorithm using model checking and other inexpensive methods, we can attempt to verify it for unbounded counters by theorem proving using PVS.

PVS is a *logic*, it does not have a notion of state, nor of programs, built in, so the specification must embed the transition relation semantics directly—as in the following specification.

```
bakery: THEORY
BEGIN

phase : TYPE = {idle, trying, critical}
state: TYPE = [# pc1, pc2: phase, t1,t2: nat #]
s, pre, post: VAR state

P1_transition(pre, post): bool =
 IF pre`pc1 = idle
   THEN post = pre WITH [(t1) := pre`t2 + 1, (pc1) := trying]
 ELSIF pre`pc1 = trying AND (pre`t2 = 0 OR pre`t1 < pre`t2)
   THEN post = pre WITH [(pc1) := critical]
 ELSIF pre`pc1 = critical
   THEN post = pre WITH [(t1) := 0, (pc1) := idle]
 ELSE post = pre
 ENDIF

P2_transition(pre, post): bool =
 IF pre`pc2 = idle
   THEN post = pre WITH [(t2) := pre`t1 + 1, (pc2) := trying]
 ELSIF pre`pc2 = trying AND (pre`t1 = 0 OR pre`t1 > pre`t2)
   THEN post = pre WITH [(pc2) := critical]
 ELSIF pre`pc2 = critical
   THEN post = pre WITH [(t2) := 0, (pc2) := idle]
 ELSE post = pre
 ENDIF

transitions(pre, post): bool =
  P1_transition(pre, post) OR P2_transition(pre, post)

init(s): bool = s`pc1 = idle AND s`pc2 = idle
  AND s`t1 = 0 AND s`t2 = 0

safe(s): bool = NOT(s`pc1 = critical AND s`pc2 = critical)
```

In this PVS specification, the notation `[# ... #]` is a record type constructor and the backquote indicates access to the fields of a record. The `WITH` construct is used to "update" record or array values.

We desire to show that the predicate `safe` is an invariant. In the next section we will attempt to do this by proving that it is inductive.

A.1 Inductive Invariance Proof

We begin by specifying what it means for predicate `INV` to be an inductive invariant of a transition relation `R` with initiality predicate `INIT`, and then establish as our `first_try` the goal of proving that `safe` is an inductive invariant of the algorithm.

```
INV, INIT: var pred[state]

R: VAR pred[[state,state]]

inductive_invariant(INV, INIT, R): bool =
 FORALL s: INIT(s) => INV(s)
  AND FORALL pre,post: INV(pre) AND R(pre,post) => INV(post)

first_try: LEMMA inductive_invariant(safe, init, transitions)
```

If we start the PVS theorem prover on this goal, we will be presented with the following *sequent*.

```
first_try :
  |-------
{1}   inductive_invariant(safe, init, transitions)
Rule?
```

The proof commands `(EXPAND "inductive_invariant")` and `(GROUND)` open up the definition of `inductive_invariant` and split it into cases. We are then presented with the first of the two cases.

```
This yields  2 subgoals:
first_try.1 :
  |-------
{1}   FORALL s: init(s) => safe(s)
```

This is discharged by the proof command `(GRIND)`, which expands definitions and performs obvious deductions, and we are then presented with the remaining subgoal.

```
This completes the proof of first_try.1.
first_try.2 :
  |-------
{1}   FORALL pre, post:
          safe(pre) AND transitions(pre, post) => safe(post)
```

The commands `(SKOSIMP)`, `(EXPAND "transitions")`, and `(GROUND)` eliminate the quantification and split `transitions` into separate cases for Processes 1 and 2.

```
first_try.2.1 :
{-1}  P1_transition(pre!1, post!1)
[-2]  safe(pre!1)
  |-------
[1]   safe(post!1)
```

A PVS sequent such as this is true if the conjunction of formulas above the "turnstile" line `|-------` implies the disjunction of formulas below. The numbers serve simply to label the formulas; those enclosed in curly braces highlight formulas that are changed from the previous sequent.

The commands `(EXPAND "P1_transition")` and `(GROUND)` split the proof into four cases according to the kind of step made by the process. The first one is discharged by `(GRIND)`, but the second is not and we are presented with the following sequent.

```
first_try.2.1.2 :
[-1]  pre!1`t2 = 0
{-2}  trying?(pre!1`pc1)
[-3]  post!1 = pre!1 WITH [(pc1) := critical]
[-4]  safe(pre!1)
{-5}  critical?(pre!1`pc2)
  |-------
```

When there are no formulas below the line, a sequent can only be true if there is a contradiction among those above the line.

Here, Process 1 enters its critical section (Formula `-3`) because Process 2's counter is zero (Formula `-1`)—but Process 2 is already in its critical section (Formula `-5`). This situation is impossible because Process 2 must have incremented its counter (making it nonzero) when it entered its `trying` phase. But contemplation, or experimentation with the prover, shows that this fact is not provable from the information provided. The same holds for the other unprovable subgoals. The problem is not that `safe` is untrue, but that it is not *inductive*: it does not provide a strong enough antecedent to support the proof of its own invariance.

The solution is to prove a stronger property than the one we are really interested in: we strengthen the desired invariant by conjoining some additional clauses.

```
strong_safe(s): bool = safe(s)
   AND ((s`pc1 = trying OR s`pc1 = critical) => s`t1 /= 0)
   AND ((s`pc2 = trying OR s`pc2 = critical) => s`t2 /= 0)

second_try: LEMMA
       inductive_invariant(strong_safe, init, transitions)
```

Rerunning the proof developed so far, we find that the strengthened invariant deals with the case we just examined, and the symmetric case for Processor 2, but its proof still produces two unproved subgoals. The first of these is the following.

```
second_try.2 :
{-1}  trying?(pre!1`pc1)
{-2}  pre!1`t1 < pre!1`t2
{-3}  post!1 = pre!1 WITH [(pc1) := critical]
{-4}  critical?(pre!1`pc2)
  |-------
{1}   (pre!1`t1 = 0)
```

The situation here is that Process 1 is entering its critical section (Formula `-3`) even though Process 2 is already in its critical section (Formula `-4`). But this cannot happen, because Formula `1` indicates that Process 1 has a nonzero counter (formulas below the line are the negations of their equivalents above the line), so that Process 2 must have had the smaller counter when it entered—thereby contradicting Formula `-2`. Again we need to strengthen the invariant to carry along this fact.

```
inductive_safe(s):bool = strong_safe(s)
 AND ((s`pc1 = critical AND s`pc2 = trying) => s`t1 < s`t2)
 AND ((s`pc2 = critical AND s`pc1 = trying) => s`t1 > s`t2)

third_try: LEMMA
      inductive_invariant(inductive_safe, init, transitions)
```

Finally, we have an invariant that is inductive—and is proved with `(GRIND)`.

A.2 Proof by Manual Abstraction and Model Checking

It does not matter to the operation of algorithm what the actual values of the counters are: all that matters is whether or not each of them is zero, and whether one is less than the other. We can use Booleans to represent these relations and can construct an abstracted specification that operates on this abstracted representation. We begin by introducing the abstracted (finite) state type (whose field names indicate their intended interpretation), the abstraction function `abst` from concrete to abstract states, and the abstracted safety property.

```
abstract_state: TYPE =
  [# pc1, pc2: phase, t1_is_0, t2_is_0, t1_lt_t2: bool #]

as, a_pre, a_post: VAR abstract_state

abst(s): abstract_state =
  (# pc1 := s`pc1, pc2 := s`pc2,
     t1_is_0 := s`t1 = 0, t2_is_0 := s`t2 = 0,
     t1_lt_t2 := s`t1 < s`t2 #)

a_safe(as): bool =
   NOT (as`pc1 = critical AND as`pc2 = critical)
```

Then we specify the abstracted algorithm in terms of its initiality predicate

```
a_init(as): bool =
   as`pc1 = idle AND as`pc2 = idle
     AND as`t1_is_0 AND as`t2_is_0
```

and its transition relation.

```
a_P1_transition(a_pre, a_post): bool =
 IF a_pre`pc1 = idle
   THEN a_post = a_pre WITH [(t1_is_0) := false,
                             (t1_lt_t2) := false,
                             (pc1) := trying]
 ELSIF a_pre`pc1 = trying
         AND (a_pre`t2_is_0 OR a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc1) := critical]
 ELSIF a_pre`pc1 = critical
   THEN a_post = a_pre WITH [(t1_is_0) := true,
                             (t1_lt_t2) := NOT a_pre`t2_is_0,
                             (pc1) := idle]
 ELSE a_post = a_pre
ENDIF

a_P2_transition(a_pre, a_post): bool =
 IF a_pre`pc2 = idle
   THEN a_post = a_pre WITH [(t2_is_0) := false,
                             (t1_lt_t2) := true,
                             (pc2) := trying]
 ELSIF a_pre`pc2 = trying
         AND (a_pre`t1_is_0 OR NOT a_pre`t1_lt_t2)
   THEN a_post = a_pre WITH [(pc2) := critical]
 ELSIF a_pre`pc2 = critical
   THEN a_post = a_pre WITH [(t2_is_0) := true,
                             (t1_lt_t2) := false,
                             (pc2) := idle]
 ELSE a_post = a_pre
ENDIF

a_trans(a_pre, a_post): bool = a_P1_transition(a_pre, a_post)
                               OR a_P2_transition(a_pre, a_post)
```

Next, we state (without justification) the four verification conditions from Section 6 that are sufficient to establish that `safe` is an invariant using the abstraction function `abst`.

```
init_simulation: THEOREM init(s) IMPLIES a_init(abst(s))

trans_simulation: THEOREM
   transitions(pre,post) IMPLIES a_trans(abst(pre),abst(post))

safety_preserved: THEOREM a_safe(abst(s)) IMPLIES safe(s)

abs_invariant_ctl: THEOREM
   a_init(as) IMPLIES AG(a_trans, a_safe)(as)
```

The conditions `init_simulation` and `safety_preserved` are proved by `(GRIND)`, and `abs_invariant_ctl` is proved by `(MODEL-CHECK)` (which calls the PVS model checker).

However, trans_simulation results in two unproved cases; the first is shown below.

```
trans_simulation.2.6.1 :
[-1]  post!1 = pre!1
{-2}  idle?(pre!1`pc1)
  |-------
{1}   critical?(pre!1`pc2)
[2]   pre!1`t1 = 0
[3]   pre!1`t1 > pre!1`t2
{4}   idle?(pre!1`pc2)
{5}   pre!1`t1 < pre!1`t2
```

The problem here is that when the two counters are equal but nonzero, the concrete algorithm drops through to the ELSE case and requires the pre and post states to be the same—whereas in the abstracted algorithm, this situation can satisfy the condition for Process 2 to enter its critical section. This is because NOT t1_lt_t2 abstracts t1 >= t2 rather than t1 > t2. However, this situation can never arise, because each counter is always incremented to be strictly greater than the other. We can establish this fact as an invariant that is proved with (GRIND).

```
not_eq(s): bool = s`t1 = s`t2 => s`t1 = 0

extra: LEMMA inductive_invariant(not_eq, init, transitions)
```

A modified version of the verification condition for simulation (stated here without justification) allows us to use a known invariant to establish it.

```
strong_trans_simulation: THEOREM
   inductive_invariant(not_eq, init, transitions)
    AND not_eq(pre) AND not_eq(post)
    AND transitions(pre, post)
      IMPLIES a_trans(abst(pre), abst(post))
```

This modified verification condition is discharged by the following proof.

```
 (SKOSIMP)
 (EXPAND "transitions")
 (GROUND)
 (("1" (EXPAND "P1_transition")
       (APPLY (THEN (GROUND) (GRIND))))
  ("2" (EXPAND "P2_transition")
       (APPLY (THEN (GROUND) (GRIND)))))
```

A.3 Proof by Automated Abstraction and Model Checking

We have seen that proof by inductive invariance requires several strengthenings of the desired invariant to obtain one that is inductive, and that justification of a manually constructed abstraction also requires an auxiliary invariant. However, the PVS mechanism for automatic construction of abstractions succeeds without further assistance. We first state the desired safety property as a CTL formula over the original (infinite state) system.

```
auto_abstract: THEOREM
   init(s) IMPLIES AG(transitions, safe)(s)
```

Then we instruct PVS to abstract the resulting sequent on the predicates `t1=0`, `t2=0`, and `t1 < t2` and to model check the resulting finite state system. This is done by the following command.

```
 (ABSTRACT-AND-MC
  ( "lambda(s) :s`t1=0"
    "lambda(s) :s`t2=0"
    "lambda(s) :s`t1 < s`t2 "))
```

PVS constructs the abstracted system (automatically checking 62 verification conditions while doing so) and model checks the result—all in less than 10 seconds on a 500 MHz Pentium.