Specification, Proof Checking, and Model Checking for Protocols and Distributed Systems with PVS [1]

Tutorial Notes for FORTE/PSTV '97
Osaka, Japan, November 1997

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

[1]This work was supported by the National Science Foundation under contract CCR-9509931.

Abstract

This tutorial uses a simple example to demonstrate some of the techniques and issues in the formal specification and analysis of protocols. A simple cache-coherence protocol is formally specified as a transition relation in PVS and a desired property of the protocol is proposed. The specification is then "downscaled" to create a finite-state instance with just two caches, and the property is examined using model checking. An attempt is then made to establish the property for the full protocol using theorem proving, but it is discovered that the property is not inductive. A strengthened invariant is then proposed, checked by model checking in the downscaled instance, and then established by theorem proving. Finally, an alternative approach is illustrated in which theorem proving is used to justify a property-preserving abstraction to finite state, which is then analyzed by model checking. The specifications and proofs appearing in this tutorial can be downloaded from `http://www.csl.sri.com/forte97.html` and then examined using PVS.

The tutorial is not intended as an introduction to PVS, but as an illustration of some topics in its application to protocols and as an introduction to the use of theorem proving and model checking techniques in combination.

Contents

Chapter 1

Introduction

The purpose of this tutorial is to illustrate some techniques and issues in the formal analysis of protocols (i.e., distributed, concurrent algorithms) using theorem proving, model checking, and the combination of these two methods. The illustrations use PVS, which is a formal verification system (i.e., a specification language integrated with a mechanized theorem prover and a model checker), but this tutorial is not an introduction to PVS—that topic is addressed by several other tutorials available from the PVS web page at `http://www.csl.sri.com/pvs.html` (follow the link to `tutorials&examples`). The salient attributes of PVS are that it provides a specification language based on higher-order logic with a very rich type system, and an interactive theorem prover with significant automation including decision procedures for several common theories (such as equality and linear arithmetic) [21]. PVS has hundreds of users worldwide; you can discover some of their applications at `http://www.csl.sri.com/pvs-bib.html`. The PVS system is freely available from `http://www.csl.sri.com/pvs.html` as a binary image for Sun, AIX, and Linux workstations.

There are, of course, many systems specifically intended for the formal description and analysis of protocols; these systems generally provide a specification language tailored for that application, together with a simulator, a model checker, or both; examples include the many tools for LOTOS [13], and the Spin system [12]. A *simulator* allows representative, or interesting, runs of a protocol to be examined and is useful for gaining an understanding of its operation in normal and abnormal circumstances, and may also reveal bugs. However, because of its distributed and concurrent nature, a protocol generally can exhibit vast numbers of different runs, and simulation can examine only a fraction of these. Techniques related to *model checking*[1], on the other hand, provide *complete* exploration of all the runs exhibited

[1]Strictly, model checking refers to methods in which a protocol is specified as a finite state transition system, a property is specified as a formula in some temporal logic, and the check is whether the transition system is a Kripke model for the formula. The check is performed either

by a *finite-state* description of the protocol. Because few protocols are naturally finite state (i.e., they generally have unbounded data structures such as buffers, or an unbounded number of participants [e.g., processes, processors, users]), they must be modified to satisfy the finite-state requirement—in fact, the requirement is not just for a *finite* state space, but for a relatively *small* state space, so the modification is often severe. The modification generally takes the form of "downscaling"—arbitrarily limiting the participants to some small number (typically two or three), and tightly bounding the size of data types and data structures—but sometimes it is done by abstraction (i.e., creating a simpler protocol that is—hoped to be—equivalent to the real one for the property concerned). Because model checking is applied to a modified description of the protocol, it is possible that it will miss some bugs, and may also raise some false alarms (the latter can be eliminated by testing putative bugs found by model checking against a simulation of the full protocol). Nonetheless, model checking has become extremely popular in the last few years because experience indicates that checking *all* the runs of a reduced version of the protocol is more effective at finding bugs than simulating *some* of the runs of the full thing.

Methods based on mechanized theorem proving can complement those based on simulation and model checking by allowing the complete behavior of the full protocol to be analyzed. Theorem proving with even the best mechanized theorem provers is a far less automated process than simulation or model checking, however. Furthermore, mechanized theorem provers are usually provided as components of *verification systems* intended for generic application. To apply theorem proving to protocols, therefore, it is first necessary to develop appropriate ways for representing protocols within the specification language supported by a verification system, and to find systematic ways for applying its theorem prover to this particular class of problems. Some verification systems, such as PVS, provide a model checker as well as a theorem prover and do so in an integrated way that allows the two methods to be used in combination [26]. The benefits from the approach to protocol analysis that can be supported by such a generic verification system include the following.

- The approach is not restricted to one class of system, nor one type of analysis, in the way that specialized methods are; it is often useful to combine several techniques (for example, theorem proving and model checking).
- It provides a way to prototype development of new specialized methods.
- By making the representational issues explicit, it can help students understand the techniques underlying the more specialized methods.

by brute-force enumeration of the state space [3,25], or symbolically [20]. Related methods include those based on language inclusion [17], those based on refinement in process algebras [27], and those that perform straightforward state exploration [5]; because they use similar underlying technology, these are all loosely referred to as model checking methods.

This tutorial aims to make concrete some of the ideas and issues raised above. It attempts to do so by working through a single simple example using model checking, theorem proving, and their combination. The approach taken is almost identical to that described, for a much more difficult example, by Havelund and Shankar [11], and their paper is a useful source for additional information. Another paper by Shankar is a good source for a more technical exposition of many of the topics raised here [33].

Chapter 2

A Simple Example: Cache Coherence

The example is a simplification of the single-bus cache coherence protocol from Futurebus+ (without split transactions) [15].[1] A memory and several processors, each with a local cache, are attached to a bus. The caches maintain "coherence" (explained below) by *snooping* (i.e., observing) all bus transactions and updating their local states appropriately. Figure 2.1 portrays a system with n processors (and their caches) labeled P_0 to P_{n-1}.

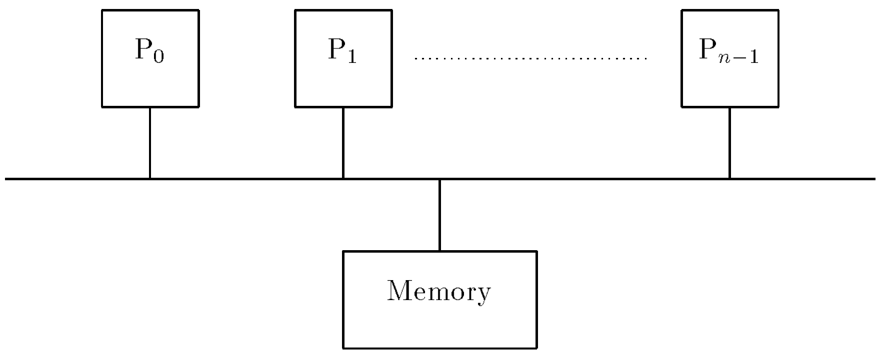

Figure 2.1: Single Bus System

Memory is organized as *lines* (i.e., groups of contiguous addresses treated as a unit for coherence purposes). Each cache can store copies of the values of several lines; each copy can be either `shared` (i.e., read-only) or `exclusive` (i.e., writable).

[1]The specification and its verification by combined theorem proving and model checking are due to Sergey Berezin and N. Shankar.

When a processor needs to read a value from a memory location, it checks whether its line is present in its cache. If it is, the processor obtains the value from the cache; if not (in which case the line is said to be `invalid` for that cache), it issues a `read-shared` transaction on the bus. Any cache that has an `exclusive` copy of that line can respond by sending its value over the bus (and changing the status of its own copy to `shared` or `invalid`), otherwise the memory supplies the value. In either case, the requesting cache will *snarf* (i.e., load) the line as a `shared` copy when its value comes over the bus (other caches may choose, opportunistically, to do the same). When a processor needs to write a value to a memory location, it checks whether it has an `exclusive` copy of the relevant line in its cache. If not, it first obtains a `shared` copy of the line as described above (unless it already has one), then issues a `read-modified` transaction and changes the status of its copy to `exclusive`. Other caches that have a copy of the line change the status of their copies to `invalid` when they see a `read-modified` transaction. A processor that has an `exclusive` copy of a line can relinquish it by issuing a `write-back` transaction. This causes the cache to write its value back to memory and to mark the line as `invalid`[2].

To be useful, a cache coherence protocol such as this should possess certain properties. One is that the caches should not alter the functional behavior of the system: that is, the system should look "the same" to each processor, whether or not the caches are present (the caches are then said to be *coherent*). Of course, a given processor performing a fixed sequence of memory accesses can still see many different behaviors in a concurrent distributed system such as this, depending on the activity of other processors and the order in which concurrent transactions serialize on the bus. Thus we can only require that any trace of memory values read and written by a processor in the presence of the caches should also be *possible* in their absence; this is the *sequentially consistent* memory model—the behavior perceived by the processors should be consistent with some sequential interleaving of their reads and writes [18].[3] Other properties we might be interested in include *latency*: how long a processor might have to wait to get a value. In this tutorial, however, we focus on a subproblem of sequential consistency.

2.1 Formalization and Representation in PVS

In applying formal methods to protocol analysis, we first construct a mathematical model of some aspects of the protocol, and then use suitable techniques (e.g.,

[2]This is more restrictive than necessary: the protocol could allow the line to be retained as `shared` and other processors could opportunistically snarf a `shared` copy of the value as it is written to memory.

[3]Note, this does not mean that a processor that writes a value to a location and then reads the same location will get that value: another processor may have written a different value in between.

theorem proving or model checking) to examine its properties. As we construct the mathematical model, we must choose how detailed or abstract to make it, and whether some aspects should be omitted or simplified. As a general rule, we should model only those aspects that are relevant to the properties of interest, and in only as much detail as is necessary for this purpose. These choices require skill and judgment: include too much detail and the formal analysis may become excessively complex, or even infeasible; include too little, and the model will become inaccurate and the analysis may fail to detect bugs (or report spurious ones).

For the cache-coherence example, the ultimate property of interest is sequential consistency. To analyze this, we would need to model the reading and writing of memory, and would need to formulate a precise specification for sequential consistency [7]. This requires quite a lot of work, so we might decide instead to focus on "internal" properties of the protocol itself: for example, at most one cache should have an `exclusive` copy of any given line. This is certainly a necessary condition for sequential consistency. By restricting our attention to such internal properties, we can ignore the reading and writing of memory and can focus our mathematical model on the way the protocol manages the caches.

Another way in which we can simplify the modeling and analysis is to restrict the size of the caches. Examination of the informal description of the protocol shows that each cache line is treated independently; thus, it really seems to make no difference whether we model m cache lines, or seventy-seven, or just one. By considering only a single cache line, we will simplify the formal model and also reduce the size of its state space—which is significant for model checking.[4]

We can summarize the simplifying decisions we have made as follows.

- We will model just the cache-coherence protocol itself, and will not model the reading and writing of external memory.
- We will restrict the model to just a single cache line.
- We will analyze this model against the requirement that at most one cache has an `exclusive` copy of the line.

The next step is to construct a suitable formal model and to represent it in the specification language of PVS. There are two well-established methods for formalizing distributed computations: one method focusses on the *events* exchanged among the components (these methods are generally *process algebras*), the other focuses on the *states* of the components and their transitions. PVS has been used with both methods (see, e.g., [6, 16]), but this example will focus on the state-based method.

[4] Actually, "at most one `exclusive` copy" is insufficient to ensure sequential consistency, but the counterexample requires more than one cache line, so this kind of simplification needs to be undertaken with care and caution.

The basic idea is to describe the system as a state machine (not necessarily finite-state): that is, we describe the possible states of the system, and how the system can change state under the influence of internal or external events. First, we need to specify types for processors, transactions, and the state of a cache.

```
cache_array[N : posnat]: THEORY
BEGIN
  processor: TYPE = below(N)

  line_status : TYPE = {invalid, shared, exclusive}

  transactions : TYPE = {idle, read_shared, read_modified, write_back}
```

This specification begins with the declaration of a PVS theory parameterized by a positive integer `N`, which is the number of processors (and their associated caches). The first declaration in the theory then specifies that processors are numbered `0,...,N-1` (this is the PVS type `below(N)`). The other two declarations then enumerate the status that may be recorded for the line by each cache, and the different kinds of transaction that can occur on the bus.

The protocol must keep note of the `line_status` of the cache line in each of the caches. This can be represented in PVS by an array (or function—the two are equivalent in the PVS specification language) from `processor` to `line_status`. The *signature* of such an array is written as follows in PVS.

```
[processor -> line_status]
```

The protocol must also note the state of the bus, which has two attributes: the current transaction, and the identity of the processor that issued it (which we will call the `bus_master`). All these items can be packaged together in a PVS record type, which we name `protocol_state`.

```
  protocol_state : TYPE =
     [# cache : [processor -> line_status],
        transaction : transactions,
        bus_master : processor #]
```

If `ps` is a variable of type `protocol_state`, then the current transaction on the bus is denoted `transaction(ps)`[5] and the status of the cache of processor `p` is given by `cache(ps)(p)`. The test whether this cache has the `exclusive` status is then written `cache(ps)(p) = exclusive` or, more commonly, `exclusive?(cache(ps)(p))`.

The protocol will be formally specified as a state machine whose state is given by the type `protocol_state`. Notice that this state is not necessarily finite, because

[5] Note that PVS uses function-like notation for record field access, rather than the more usual "dot" notation `ps.transaction`.

there is no bound on the number of processors. To specify the state machine, we need to describe its *transition relation*: that is, given any particular state, we must specify its possible successor states. Notice that there may be more than just a single successor state (i.e., the machine is nondeterministic, and must be modeled by a transition *relation*, and not a next-state *function*) because each processor cache has a certain degree of autonomy. For example, if there are two caches with `exclusive` status, then we do not care which of them responds to a `read_shared` transaction; of course, we hope to prove that there can be at most one cache with the `exclusive` status, but that is something we need to demonstrate, not assume.

A relation is a set of pairs, and in a higher-order logic such as PVS, sets are identified with predicates—that is, functions with range type `bool`. Thus, the transition relation for the cache protocol is specified in PVS as a function `p_next` from pairs of states to `bool`.

```
ps, ps0, ps1: VAR protocol_state
p_next(ps0, ps1): bool
```

The `p_next` function will specify the allowed relationship between predecessor state `ps0`, and successor state `ps1`. We specify the function by decomposing it into four cases: one for each of the three types of bus transaction, and one "idle" case where no transaction is pending. Each case is specified as the conjunction of a predicate that "recognizes" the case (e.g., `idle?(t)`) and a relation the "executes" it (e.g., `do_idle(ps0, ps1)`).

```
p_next(ps0, ps1): bool =
  LET p = bus_master(ps0), t=transaction(ps0) IN
   (   (idle?(t) AND do_idle(ps0, ps1))
    OR (read_shared?(t) AND do_read_shared(ps0, ps1)(p))
    OR (read_modified?(t) AND do_read_modified(ps0, ps1)(p))
    OR (write_back?(t) AND do_write_back(ps0, ps1)(p)))
```

The idle transaction is simply specified: it does nothing.

```
do_idle(ps0, ps1): bool = (ps1 = ps0)
```

The transition when processor `p` initiates a `read_shared` transaction is specified by the predicate `do_read_shared(ps0, ps1)(p)` as follows.

```
                                                              1
p, q, r: VAR processor

do_read_shared(ps0, ps1)(p): bool  =
  invalid?(cache(ps0)(p)) AND shared?(cache(ps1)(p))
  AND ((EXISTS q: exclusive?(cache(ps0)(q))
          AND NOT exclusive?(cache(ps1)(q))
          AND (FORALL r: r/=q IMPLIES
                 exclusive?(cache(ps1)(r)) =
                 exclusive?(cache(ps0)(r))))
       OR (FORALL q: NOT exclusive?(cache(ps0)(q))
                 AND NOT exclusive?(cache(ps1)(q))))
```

We will examine the components of this specification in order, reformatting them slightly to assist comprehension. The first conjunct of the specification

```
invalid?(cache(ps0)(p)) AND shared?(cache(ps1)(p))
```

indicates that the transaction can only be issued by a processor whose cache has `invalid` status (i.e., it does not hold a copy of the cache line) in predecessor state `ps0`, and results in that processor obtaining the `shared` status in successor state `ps1`. There are two ways that the value of the cache line can be provided: either some processor with `exclusive` access provides it, or the memory does.

The first of these possibilities is specified as follows.

```
EXISTS q: exclusive?(cache(ps0)(q)) AND NOT exclusive?(cache(ps1)(q))
       AND (FORALL r: r/=q IMPLIES
        exclusive?(cache(ps1)(r) = exclusive?(cache(ps0)(r))))
```

This says that a possible transition is one where some processor `q` having `exclusive` access gives up that status (we do not specify whether it goes to `shared` or `invalid` status), and no other processor either gives up or gains `exclusive` access.

The second possibility is specified as follows.

```
FORALL q: NOT exclusive?(cache(ps0)(q)) AND NOT exclusive?(cache(ps1)(q))
```

This says that a possible transition is one in which no processor has exclusive access either before or after (implicitly, the memory is understood to provide the value in this case, but we are not modeling that aspect).

The transition when processor `p` initiates a `read_modified` transaction is specified by `do_read_modified(ps0, ps1)(p)` as follows.

```
do_read_modified(ps0, ps1)(p): bool =
  shared?(cache(ps0)(p))
  AND (FORALL q: cache(ps1)(q) =
          IF p = q THEN exclusive
          ELSIF shared?(cache(ps0)(q)) THEN invalid
          ELSE cache(ps0)(q) ENDIF)
```

This says that the initiating processor p must start in the shared status and that in the successor state p will have exclusive status and all other processors with the shared status will change to the invalid status.

The transition when processor p initiates a write_back transaction is specified by do_write_back(ps0, ps1)(p) as follows.

```
do_write_back(ps0, ps1)(p) : bool =
   exclusive?(cache(ps0)(p))
   AND cache(ps1) = cache(ps0) WITH [(p) := invalid]
```

This says that p must have exclusive status in the predecessor state, and that the successor state is identical, except that p changes to the invalid status.

In order to complete the specification, we need to suitably constrain the initial states. We will require that all processors start off with the invalid status.

```
p_init(ps): bool = FORALL p: invalid?(cache(ps)(p))
```

We can now specify the safety property that we require of this protocol: in any state ps, at most one processor should have the exclusive status.

```
p_safe(ps): bool =
  FORALL p, q: exclusive?(cache(ps)(p)) AND exclusive?(cache(ps)(q))
    IMPLIES p = q
```

This property will be true of all reachable states if it is true of all initial states and is preserved by the transition p_next.

```
                                                                        2
invariant: THEOREM (p_init(ps) IMPLIES p_safe(ps))
  AND (p_safe(ps0) AND p_next(ps0, ps1) IMPLIES p_safe(ps1))
```

The full specification is shown in Appendix A.1.1.

2.2 Model Checking a Finite Instance

Now it will be relatively difficult to establish the property $\boxed{2}$ by theorem proving, so first we may wish to test its plausibility by model checking a finite instance—say the case where there are only two processors. Model checking is usually a much more automatic process than theorem proving: its disadvantage is that it can usually be applied only to finite-state systems, and to a fairly limited class of properties (typically those that can be specified in a temporal logic).

In order to render our specification suitable for model checking, we must reduce its state space, and must express the property to be examined in the logic supported by the model checker. We can reduce the state space by fixing N, the number of processors, to be some small finite number: specifically, 2. The model checker of PVS supports the branching time temporal logic called *Computation Tree Logic* (CTL). We do not define CTL here, and do not explain the general technology of CTL model checking, nor the manner of its implementation in PVS; see [32] for coverage of these topics. In CTL as encoded in PVS, an invariant property is specified in the form `AG(trans, prop)(s)`, where `trans` is the transition relation of the system concerned, `prop` is the property of interest, and `s` is the starting state. The `AG` operator of CTL then says the `prop` must be true of all states that can be reached from `s` by the transitions of `trans`. Thus, the invariant above can be recast into CTL as follows.

```
                                                                          3
  invariant_ctl: THEOREM  p_init(ps) IMPLIES AG(p_next, p_safe)(ps)
```

Ideally, with these simple modifications to our specification, we should be able to submit the formula `invariant_ctl` to the PVS model checker and discover that it is, indeed, true. Unfortunately, the PVS model checker does not, at present, handle numeric subranges (as in the `processor` type), so it is necessary to manually modify the specification to expand out all quantification over this type. The expanded specification is shown in Appendix A.2.1; the need for this ugly modification will be removed in a forthcoming version of PVS. Using the model checker (with the PVS proof commands `(expand "p_next") (model-check)`), PVS verifies the property `invariant_ctl` in a couple of seconds.

Specifications sometimes model check for the wrong reason (e.g., the property may be an implication with a false antecedent), so it is often worthwhile to introduce a deliberate error into a specification and then check that the model checking fails. In this example, we can strike out the third line in the specification for `do_read_modified` in $\boxed{1}$ (i.e., the line `AND NOT exclusive?(cache(ps1)(q))`), thereby allowing a processor to retain the `exclusive` status after it has provided the line to a processor requesting `shared` status. The PVS model checker reports correctly that the modified specification fails to satisfy the property `invariant_ctl`. Unlike most standalone model checkers, the PVS model checker does not currently provide a counterexample trace when a property fails.

2.3 Proving the Invariant

Satisfied that model checking a finite instance has shown the specification and property to be at least plausible, we can now move on to verify the general case using theorem proving. When we invoke the PVS theorem prover on the formula `invariant`, we are presented with the following initial goal.

```
invariant:
  |-------
{1} (FORALL (ps: protocol_state,
          ps0: protocol_state, ps1: protocol_state):
     (p_init(ps) IMPLIES p_safe(ps))
       AND (p_safe(ps0) AND p_next(ps0, ps1) IMPLIES p_safe(ps1)))

Rule?
```

The `Rule?` prompt is inviting us to enter PVS proof commands, which are given in Lisp syntax. The proof commands `(skosimp)` and `(ground)` eliminate the quantification (replacing the variable `ps`, for example, by the *Skolem constant* `ps!1`) and perform propositional simplification. We are then presented with the first of two cases.

```
                                                                    4
This yields 2 subgoals:
invariant.1:

{-1}    p_init(ps!1)
  |-------
{1}    p_safe(ps!1)

Rule?
```

PVS presents proof goals as *sequents*, in which a collection of *antecedent* formulas (here just one) appears above the sequent line (`|-------`), and a collection (here again, just one) of *conclusion* formulas appears below; the sequent is true if the conjunction of formulas above the line implies the disjunction of formulas below (if there are no formulas below the line then we need a contradiction among those above). PVS proof commands transform the current sequent to one or more simpler (we hope) sequents whose truth implies the original one.

The sequent in $\boxed{4}$ is inviting us to prove that the property `p_safe` is true of any protocol state `ps!1` that satisfies the `p_init` property. The proof is a trivial one that follows by expanding definitions and applying obvious deductions. It is dispatched by the powerful PVS proof command `(grind)`. We are then presented with the second subgoal.

```
This completes the proof of invariant.1.

invariant.2:

{-1}    p_safe(ps0!1)
{-2}    p_next(ps0!1, ps1!1)
  |-------
{1}    p_safe(ps1!1)

Rule?
```

Here we are being invited to prove that `p_safe` is preserved by `p_next`. Our first step is to expand the definition of `p_next` and split the proof into cases corresponding to each of the four kinds of transaction. This is accomplished by the proof commands `(expand "p_next")` and `(ground)`.

```
Applying propositional simplification and decision procedures,
this yields 4 subgoals:
invariant.2.1:

{-1}    idle?(transaction(ps0!1))
{-2}    do_idle(ps0!1, ps1!1)
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)

Rule?
```

This first subgoal concerns the `idle` transaction and is easily dispatched with `(grind)`. Skipping the second and third subgoals temporarily, the fourth subgoal concerns the case of a `write_back` transaction.

```
invariant.2.4:

{-1}    write_back?(transaction(ps0!1))
{-2}    do_write_back(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)

rule?
```

This is dispatched by the command `(grind: if-match all)`, which instructs `grind` to consider more instantiations for quantified variables than it would normally.

Returning to the third subgoal, we are presented with the case corresponding to a `read_shared` transaction.

```
invariant.2.2:

{-1}    read_shared?(transaction(ps0!1))
{-2}    do_read_shared(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)

Rule?
```

This requires a more interactive proof because the automated strategies of PVS guess incorrect instantiations for variables. We start the proof off with `(grind :if-match nil)`, which is a variant of `grind` in which the prover is instructed to not attempt instantiation of variables. The proof is then completed by half a dozen explicit instantiations; the full proof script is shown in Appendix A.1.2. If you would like to see an interactive proof described in detail, please refer to that of the formula `safety_preserved` on page 21.

Finally, we are presented with the case corresponding to a `read_modified` transaction.

```
invariant.2.3:

{-1}    read_modified?(transaction(ps0!1))
{-2}    do_read_modified(ps0!1, ps1!1)(bus_master(ps0!1))
[-3]    p_safe(ps0!1)
  |-------
[1]    p_safe(ps1!1)

Rule?
```

As with the previous case, we start off with `(grind :if-match nil)` and are presented with the following proof goal.

```
invariant.2.2:

{-1}    q!1 < N
{-2}    p!1 < N
[-3]    read_modified?(transaction(ps0!1))
{-4}    shared?(cache(ps0!1)(bus_master(ps0!1)))
{-5}    FORALL (q: processor):
        cache(ps1!1)(q)
            = IF bus_master(ps0!1) = q THEN exclusive
          ELSIF shared?(cache(ps0!1)(q)) THEN invalid
          ELSE cache(ps0!1)(q)
          ENDIF
{-6}    FORALL (p: processor):
        exclusive?(cache(ps0!1)(p))
            IMPLIES FORALL (q: processor):
            exclusive?(cache(ps0!1)(q)) IMPLIES p = q
{-7}    exclusive?(cache(ps1!1)(p!1))
{-8}    exclusive?(cache(ps1!1)(q!1))
  |-------
{1}    p!1 = q!1

Rule?
```

Inspection, or some fruitless experimentation with the prover, should convince us that this goal is not provable from the information provided: when a `read_modified` transaction is performed, the `bus_master` changes from `shared` to `exclusive` status, and all other processors with `shared` status drop back to `invalid`—but what if there is a processor already in the `exclusive` status? The specification says nothing about this circumstance, so that processor could retain its status, thereby violating the `p_safe` property.

Further thought should convince us that this circumstance is actually impossible: processors can only obtain the `exclusive` status from the `shared` status, and the first processor to gain `exclusive` status causes all other processors with `shared` status to revert to `invalid`. The problem is not that `p_safe` is untrue, but that it is not *inductive*: that is, it does not provide a strong enough antecedent to support the proof of its own invariance. The solution to this difficulty is to prove a somewhat stronger property than the one we are really interested in. In this case, a suitable property is the following.

```
  strong_p_safe(ps): bool =
    FORALL p: exclusive?(cache(ps)(p))
       IMPLIES FORALL q: q /= p IMPLIES invalid?(cache(ps)(q))

  strong_invariant: THEOREM (p_init(ps) IMPLIES strong_p_safe(ps)) AND
     (strong_p_safe(ps0) AND p_next(ps0, ps1) IMPLIES strong_p_safe(ps1))
```

This says not merely that there cannot be two processors with the `exclusive` status, but that if any processor has the `exclusive` status, than all others must have the `invalid` status.

Before we invest effort in attempting to prove this stronger property, it will be worthwhile to model check it for the two-processor case.

```
strong_invariant_ctl: THEOREM
  p_init(ps) IMPLIES AG(p_next, strong_p_safe)(ps)
```

The PVS model checker establishes the truth of this formula in a few seconds, so it is worthwhile to attempt to prove the general case.

The proof for `strong_invariant` in the general case is very similar to that for the original `invariant`, except that this time the subgoal corresponding to the `read_modified` transaction is easily discharged by an interactive proof similar to that for the `read_shared` subgoal; the full proof script is shown in Appendix A.1.2.

2.4 Model Checking and Property-Preserving Abstraction

Model checking a downscaled instance of a protocol is very effective at finding bugs: the model checker often provides a counterexample that can be examined against the full version of the protocol (either by hand or with the aid of a simulator) in order to check that the bug is present in the full version and not just an artifact of the simplification to finite state. When model checking does not report errors, however, we cannot assume that the full protocol is correct—there may be bugs that fail to show up in the simplified, finite-state version of the protocol. For example, it is easy to inject a bug into our cache-coherence protocol that will show up only if three or more processors are considered, and that will therefore not be discovered in model checking the two-processor case.

Theorem proving can handle the general case of a protocol, but the proofs are often difficult and long. More significant as a source of difficulty than formal proof itself, however, is the need to strengthen the desired invariant to make it inductive. For our cache-coherence protocol, we needed only a single strengthening, but for other (quite simple) protocols it is not uncommon to iterate through dozens of strengthenings in order to reach an inductive property. Typically, each iteration involves an unsuccessful proof attempt that yields insight and a new term to be conjoined to the putative invariant. Havelund and Shankar [11] report the example of a simple communications protocol (known as the Bounded Retransmission Protocol, or BRP) that required 57 strengthenings of the invariant.

Some of the most attractive and promising ideas in protocol verification seek to combine the complementary strengths of model checking and theorem proving. One class of approaches uses model checking to help generate the invariants needed for theorem proving. Using such an approach, Saïdi and Graf [30] were able to generate the invariants needed for the BRP example automatically. Another method for combining theorem proving and model checking is to use theorem proving to justify a property-preserving finite-state abstraction of the protocol, which can then be analyzed by model checking [4,19]. Unlike the downscaled versions of a protocol that are constructed when model checking is used for debugging, a property-preserving abstraction is faithful to the full protocol in the sense that model checking a certain property for the abstracted protocol is sufficient to establish a related property for the full protocol. We illustrate this approach for our cache protocol.

We would like to establish the `invariant` property, or its CTL equivalent `invariant_ctl` ($\boxed{2}$ and $\boxed{3}$, respectively) for the general N-processor case using a combination of theorem proving and model checking. Rather than just cut the full specification down to the 2-processor case for model checking, we instead seek a finite-state abstraction that preserves the essential structure of the N-processor case. In particular, we would like to develop an abstracted state type `abstract_state` with corresponding transition relation `a_next` and initiality predicate `a_init`, together with an abstracted safety property `a_safe` and an abstraction function `abst` from `protocol_state` to `abstract_state`, such that the following four properties hold.

```
                                                                    5
init_simulation: THEOREM
    p_init(ps) IMPLIES a_init(abst(ps))

next_simulation: THEOREM
    p_next(ps0, ps1) IMPLIES a_next(abst(ps0), abst(ps1))

safety_preserved: THEOREM a_safe(abst(ps)) IMPLIES p_safe(ps)

as: VAR abstract_state

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

That these four conditions are sufficient to establish `invariant_ctl` is a consequence of general results established in [4].

The goal is to arrange matters so that the fourth of these conditions can be established by model checking, and the other three by fairly simple theorem proving. The key to obtaining a finite-state model that is faithful to the full protocol is to notice that it is not important to the safety property we are interested in exactly how many processors have the `exclusive` status, but only whether there are zero, one, or many of them, and whether or not there are also some processors with the `shared` status. Thus, we introduce the following enumerated type.

```
abstract_status : TYPE = {all_invalid,
                          some_shared,
                          one_exclusive,
                          many_exclusive,
                          exclusive_shared}
```

Each value of type `abstract_status` abstracts the status of N concrete caches. The intended correspondence between the abstract and concrete status is given in the following table.

Abstract Status	Interpretation for N Caches
`all_invalid`	all `invalid`
`some_shared`	some `shared`, no `exclusive`
`one_exclusive`	one `exclusive`, no `shared`
`many_exclusive`	many `exclusive`, no `shared`
`exclusive_shared`	some `exclusive`, some `shared`

This correspondence is formalized in the following function which gives the `abstract_status` corresponding to the array of `line_status` values for the N-processor system.[6]

```
abst_cache(c: [processor -> line_status]) : abstract_status =
  IF (EXISTS p: exclusive?(c(p)))
  THEN IF (EXISTS q: shared?(c(q)))
       THEN exclusive_shared
       ELSIF (EXISTS p: (FORALL q: exclusive?(c(q)) IMPLIES p = q))
       THEN one_exclusive
       ELSE many_exclusive ENDIF
  ELSIF (EXISTS p: shared?(c(p)))
  THEN some_shared
  ELSE all_invalid ENDIF
```

The state type of the abstract specification, and the abstraction function between the concrete and abstract states are then specified as follows.

```
abstract_state : TYPE = [# cache: abstract_status,
                           transaction : transactions #]

abst(ps) : abstract_state =
   (# cache := abst_cache(cache(ps)), transaction := transaction(ps) #)
```

[6]This abstraction is actually set up to preserve the `strong_p_safe` property; it also preserves the weaker `p_safe` property, but a slightly different abstraction could be used if that was the only goal.

The transition relation of the abstract system is specified by cases in a similar manner to the concrete one. Note that the names `do_idle` etc. are *overloaded* here with new definitions; PVS can disambiguate which function definition (concrete or abstract) is intended from the types of its arguments.

```
as0, as1: VAR abstract_state

a_next(as0, as1): bool =
  LET t = transaction(as0) IN
    ((idle?(t) AND do_idle(as0, as1))
       OR (read_shared?(t) AND do_read_shared(as0, as1))
       OR (read_modified?(t) AND do_read_modified(as0, as1))
       OR (write_back?(t) AND do_write_back(as0, as1)))
```

The case for the idle transaction is specified in the obvious way.

```
do_idle(as0, as1): bool = (as0 = as1)
```

For the `read_shared` transaction, the system must clearly end up in the `some_shared` abstract status, and may start off from any of the `all_invalid`, `some_shared`, or `one_exclusive` abstract statuses. This suggests the following specification.

```
do_read_shared(as0, as1): bool = LET x=cache(as0) IN
  (all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
   AND some_shared?(cache(as1))
```

However, we also need to account for the case where a `read_shared` transaction is issued in an inappropriate initial state (i.e., one with the many `many_exclusive` or `exclusive_shared` status); this shouldn't happen but, as noted earlier, that is something we should prove, not assume. We do not care about the subsequent state in this case, but must allow some transition. This suggests we disjoin the negation of the desired antecedent state to the transition relation, as follows.

```
do_read_shared(as0, as1): bool = LET x=cache(as0) IN
  ((all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
    AND some_shared?(cache(as1)))
  OR NOT (all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))
```

Now this expression has the abstract form `(a AND b) OR NOT a`, which is equivalent to `a IMPLIES b`. Also, the expression `(all_invalid?(x) OR some_shared?(x) OR one_exclusive(x))` is surely the safety property that we wish to establish. Hence, our final specification can be written as follows.

```
a_safe(as): bool =                                                      6
  LET x = cache(as) IN
   all_invalid?(x) OR some_shared?(x) OR one_exclusive?(x)

do_read_shared(as0, as1): bool =
  (a_safe(as0) IMPLIES some_shared?(cache(as1)))
```

For the `read_modified` transaction, the system must start off in an abstract status that corresponds to some processors having `shared` access (i.e., `some_shared?(x) OR exclusive_many?(x)`) and ends up in the status `one_exclusive` or `exclusive_many` depending on whether the starting state was `some_shared` or `exclusive_many`, respectively.

```
do_read_modified(as0, as1): bool =
  LET x = cache(as0), y = cache(as1) IN
      (some_shared?(x) OR exclusive_shared?(x))
         AND IF some_shared?(x) THEN one_exclusive?(y)
             ELSE many_exclusive?(y) ENDIF
```

For the `write_back` transaction, the system must start off in a status that includes some `exclusive` access. If it was `one_exclusive`, then the subsequent status should be `all_invalid`; if it was one `many_exclusive` or `exclusive_shared`, then we have a problem and do not care about the subsequent status, so we leave it unspecified using the `IMPLIES` construction previously seen in $\boxed{6}$.

```
do_write_back(as0, as1): bool =
  LET x = cache(as0), y = cache(as1) IN
    (one_exclusive?(x) OR many_exclusive?(x) OR exclusive_shared?(x))
      AND (one_exclusive?(x) IMPLIES all_invalid?(y))
```

With this abstracted specification (shown in full in Appendix A.3.1), the safety property `abs_invariant_ctl` of $\boxed{5}$ (page 18) is easily verified by model checking.

The other three conditions of $\boxed{5}$, which are part of the theory `refinement` shown in full in Appendix A.3.3, require theorem proving. The first condition, `init_simulation`, is proved by `(grind)`. The third condition, `safety_preserved`, requires a simple interactive proof. We begin with `(grind :if-match nil)`, which produces two subgoals.

```
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
safety_preserved.1 :

{-1}    p!2 < N
{-2}    p!1 < N
{-3}    q!1 < N
{-4}    (FORALL (q: processor[N]): exclusive?(cache(ps!1)(q))
                 IMPLIES p!2 = q)
{-5}    exclusive?[N](cache(ps!1)(p!1))
{-6}    exclusive?[N](cache(ps!1)(q!1))
  |-------
{1}    (EXISTS (q: processor[N]): shared?(cache(ps!1)(q)))
{2}    p!1 = q!1

Rule?
```

Formula `-4` tells us that if any processor `q` has `exclusive` status, then `q` equals `p!2`. But formulas `-5` and `-6` tell us that processors `p!1` and `p!2` both have `exclusive` status. The commands `(inst-cp -4 "p!1")` and `(inst -4 "q!1")` respectively crate a copy of formula `-4` and instantiate its variable `q` with `p!1` and with `q!1` to produce the following goal.

```
This simplifies to:
safety_preserved.1 :

[-1]    p!2 < N
[-2]    p!1 < N
[-3]    q!1 < N
{-4}    exclusive?(cache(ps!1)(q!1)) IMPLIES p!2 = q!1
[-5]    exclusive?(cache(ps!1)(p!1)) IMPLIES p!2 = p!1
[-6]    exclusive?[N](cache(ps!1)(p!1))
[-7]    exclusive?[N](cache(ps!1)(q!1))
  |-------
[1]    (EXISTS (q: processor[N]): shared?(cache(ps!1)(q)))
[2]    p!1 = q!1

Rule?
```

It is now a trivial consequence of propositional and equality reasoning that the formulas `-4` to `-7` imply that `p!1 = q!1`, and hence discharge formula `2` in the conclusion. The PVS decision procedures, which are invoked by the `(assert)` command, perform this reasoning and complete the proof of this subgoal. We are then presented with the other subgoal.

```
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of safety_preserved.1.

safety_preserved.2 :

{-1}    p!1 < N
{-2}    q!1 < N
{-3}    exclusive?[N](cache(ps!1)(p!1))
{-4}    exclusive?[N](cache(ps!1)(q!1))
  |-------
{1}    (EXISTS (p: processor[N]): exclusive?(cache(ps!1)(p)))
{2}    p!1 = q!1

Rule?
```

This is asking us to show that there exists a processor with `exclusive` status. But above the line we are told that `p!1` and `q!1` both have this status. PVS can therefore be told to guess a suitable instantiation for the variable `p` (using the command `(inst?)`), and this completes the proof.

The remaining formula in $\boxed{5}$ is the one called `next_simulation`. This requires quite a long interactive proof; the script is shown in Appendix A.3.4.

Chapter 3

Discussion and Exercises

We have seen three methods of formal analysis applied to a simple protocol.

Model Checking a Finite-State Instance: This was the simplest and most automatic form of analysis. It is often effective at finding bugs, but the drastic "downscaling" used to create the finite-state instance (in our case, the restriction to just two processors) means that it cannot verify the full protocol.

Research in model checking is addressing the "state explosion problem" (the tendency for the number of states that must be explored to grow exponentially with the number of components considered) using symbolic methods, reductions based on special properties of the system or property considered (e.g., symmetry [14], partial order [8]), and is also exploring compositional methods [10].

Using Theorem Proving to Verify an Invariant: This was conceptually straightforward, but required strengthening the invariant to obtain one that is inductive. In our example, only a single strengthening was required, but dozens may be required in more difficult cases.

Much recent work focusses on systematic and automated ways to generate and strengthen invariants; see, for example [2]. Other work focusses on methods that are particularly effective for certain classes of applications: see, for example [22–24]

The proofs required a fair degree of interactive guidance, and therefore a certain familiarity with PVS. Examination of the proofs shows that interaction is chiefly required to select the correct instantiation for variables. Current research and development for PVS should result in improved automation in this regard.

Model-Checking and Theorem Proving in Combination: This required insight to construct a finite-state model that was not merely a downscaled version of the full protocol, but a property-preserving abstraction of it. Theorem proving was used to demonstrate that it was an abstraction, and model checking was used to verify the desired safety property of the abstraction. In this example, one of the proofs needed to justify the abstraction was longer than those needed to verify invariance in the previous approach. This seems not to be the case in more realistic examples. As noted above, improved selection of variable instantiations would significantly increase the automation of this proof.

Another approach establishes properties of N-process systems using the combination of induction and model checking, where the latter is used to discharge the inductive and/or base cases established by the former [31, 33]. Other recent work investigates a different way of using model checking and theorem proving in combination: using model checking to help develop invariants for use by theorem proving [28–30].

We hope the examples in this tutorial illustrate the point that model checking and theorem proving (and simulation, though that was not demonstrated) have complementary strengths and are best used in combination. In *loose* combination, model checking of downscaled instances can be used to check for bugs before undertaking the investment of theorem proving the full protocol; model checking can also be used to test strengthened formulas as they are proposed during the interactive search for an inductive invariant. In *tight* combination, theorem proving can be used to justify abstractions that can be model checked, and model checking can be used to help find invariants for theorem proving. Although model checking of downscaled instances is the dominant method for industrial applications today, we expect that combined methods will find increasing application as their tool support matures.

3.1 Suggested Exercises

The specifications and proofs described in this tutorial can be downloaded from the PVS Web Site (see the Appendix for instructions). You are encouraged to experiment with these in PVS; first, try stepping through the proofs supplied (using the command `M-x x-step-proof` to start the proof and its graphical representation, then typing the two characters `<tab> 1` to execute each proof step in turn). Once you are familiar with the existing proofs, try constructing your own for some of the simpler formulas, or cases.

When you have gained confidence in your ability to reproduce or improve on the given examples, consider modifying the specifications and making any necessary changes to the proofs as suggested below.

- The initial condition specifies that all processors should start with the `invalid` status. Investigate weaker conditions (e.g., `NOT exclusive`, or `p_safe`). Test them first by model checking in the downscaled instance, then try the corresponding invariant and abstraction proofs.

- Repeat the verification and model checking for the approach based on property-preserving abstraction, but with the `strong_p_safe` property.

- The `write_back` transaction causes the `bus_master` to change from `exclusive` to `invalid` status. Investigate allowing this processor to retain `exclusive` status, or to change to `shared` status. What if other processors snarf the value (i.e., change from `invalid` to `shared` status) during `write_back` under these conditions?

- Introduce a bug into the specification. Investigate how the bug comes to light in the different forms of analysis. Introduce a bug that is only manifested when three or more processors are considered.

- Only the memory or a processor with `exclusive` status can supply the line to a `read_shared` transaction. Examine the case where a processor with `shared` status can also respond. Does this not potentially allow multiple processors to respond, causing collisions on the bus? Does the formal model expose this? If not, how can it be extended to do so?

- (This is an advanced exercise:) Show that the protocol provides sequential consistency [7].

Appendix A

PVS Specification and Proof Listings

You can obtain the PVS "dump file" for all the specifications and proofs listed here from http://www.csl.sri.com/forte97.html. If you copy that file into a local one called cache.dmp and give PVS the command M-x undump-pvs-files and respond cache.dmp when prompted, the specifications and proofs will be recreated in your machine. The specification files are all imported by a theory called top, whose only purpose is to tie them all together for convenience.

```
top: THEORY

BEGIN

  N: posint
  IMPORTING cache_array[N], cache2, abs_cache, refinement[N]

END top
```

You can step through any proof by moving the cursor to the specification and formula concerned and giving the command M-x step-proof (alternatively, the command M-x x-step-proof additionally displays a picture of the developing proof tree); repeatedly typing the two keystrokes <tab> 1 will then step through the proof one command at a time.

A.1 N-Processor Cache Specification and Proof

This is the specification developed in Section 2.1 and the invariant proof of Section 2.3.

A.1.1 N-Processor Cache Specification

The specification here differs slightly from that in Section 2.1 in that the definition of the `transactions` type is broken out into a separate theory so that it can also be imported into the downscaled and abstract specifications.

```
trans: THEORY
BEGIN
   transactions : TYPE = {idle, read_shared, read_modified, write_back}
END TRANS

cache_array[N : posnat]  : THEORY

BEGIN

  IMPORTING trans

  processor : TYPE = below(N)

  line_status : TYPE = invalid, shared, exclusive

  protocol_state : TYPE =
     [# cache : [processor -> line_status],
        transaction : transactions,
        bus_master : processor #]

  p, q, r : VAR processor

  ps, ps0, ps1: VAR protocol_state

  do_idle(ps0, ps1): bool = (ps1 = ps0)

% want shared (read) access

  do_read_shared(ps0, ps1)(p): bool  =
    invalid?(cache(ps0)(p)) AND shared?(cache(ps1)(p))
    AND ((EXISTS q: exclusive?(cache(ps0)(q))
            AND NOT exclusive?(cache(ps1)(q))
            AND (FORALL r: r/=q IMPLIES
                   exclusive?(cache(ps1)(r)) =
                   exclusive?(cache(ps0)(r))))
         OR (FORALL q: NOT exclusive?(cache(ps0)(q))
                   AND NOT exclusive?(cache(ps1)(q))))

% want to make my copy exclusive (writable)

  do_read_modified(ps0, ps1)(p): bool =
```

```
    shared?(cache(ps0)(p))
    AND (FORALL q: cache(ps1)(q) =
            IF p = q THEN exclusive
            ELSIF shared?(cache(ps0)(q)) THEN invalid
            ELSE cache(ps0)(q) ENDIF)

% want to write an exclusive copy back to memory

  do_write_back(ps0, ps1)(p) : bool =
     exclusive?(cache(ps0)(p))
     AND cache(ps1) = cache(ps0) WITH [(p) := invalid]

  p_next(ps0, ps1): bool =
    LET p = bus_master(ps0), t=transaction(ps0) IN
     (   (idle?(t) and do_idle(ps0, ps1))
      OR (read_shared?(t) and do_read_shared(ps0, ps1)(p))
      OR (read_modified?(t) and do_read_modified(ps0, ps1)(p))
      OR (write_back?(t) and do_write_back(ps0, ps1)(p)))

  p_init(ps): bool = FORALL p: invalid?(cache(ps)(p))

  p_safe(ps): bool =
    FORALL p, q: exclusive?(cache(ps)(p)) AND exclusive?(cache(ps)(q))
      IMPLIES p = q

  invariant: THEOREM (p_init(ps) IMPLIES p_safe(ps))
    AND (p_safe(ps0) AND p_next(ps0, ps1) IMPLIES p_safe(ps1))

  strong_p_safe(ps): bool =
    FORALL p: exclusive?(cache(ps)(p))
       IMPLIES FORALL q: q /= p IMPLIES invalid?(cache(ps)(q))

  strong_invariant: THEOREM (p_init(ps) IMPLIES strong_p_safe(ps)) AND
     (strong_p_safe(ps0) AND p_next(ps0, ps1) IMPLIES strong_p_safe(ps1))

END cache_array
```

A.1.2 Proof Scripts of Invariant Properties for the N-Processor Cache Specification

The proof script for `invariant` is marked "unfinished" because the proof fails to attempt the case for a `read_modified` transaction (the `(postpone)` command skips this case). This case cannot be proved because the formula is not inductive.

```
Proof scripts for theory cache_array:
```

```
cache_array.invariant: unfinished

(""
 (SKOSIMP)
 (GROUND)
 (("1" (GRIND))
  ("2"
   (EXPAND "p_next")
   (GROUND)
   (("1" (GRIND))
    ("2"
     (GRIND :IF-MATCH NIL)
     (("1"
       (INST -8 "q!2")
       (ASSERT)
       (INST -9 "q!1")
       (ASSERT)
       (INST -9 "q!2")
       (ASSERT))
      ("2" (INST -6 "p!1") (ASSERT))))
    ("3" (POSTPONE)) ("4" (GRIND :IF-MATCH ALL))))))
```

The strengthened invariant `strong_invariant` is inductive, and its proof is complete,

```
cache_array.strong_invariant: proved - complete

(""
 (SKOSIMP)
 (GROUND)
 (("1" (GRIND))
  ("2"
   (EXPAND "p_next")
   (GROUND)
   (("1" (GRIND))
    ("2"
     (GRIND :IF-MATCH NIL)
     (("1"
       (INST -8 "p!1")
       (ASSERT)
       (INST -9 "p!1")
       (ASSERT)
       (INST -9 "q!1")
       (ASSERT))
      ("2" (INST -6 "p!1") (ASSERT))))
    ("3"
     (GRIND :IF-MATCH NIL)
```

```
    (INST -5 "p!1")
    (LIFT-IF)
    (GROUND)
    (("1"
      (REVEAL -1)
      (INST -1 "q!1")
      (LIFT-IF)
      (GROUND)
      (INST -8 "q!1")
      (ASSERT)
      (INST -8 "p!1")
      (ASSERT))
     ("2" (INST -6 "p!1") (ASSERT) (INST -6 "bus_master(ps0!1)") (ASSERT))))
   ("4" (GRIND :IF-MATCH ALL))))))
```

A.2 Downscaled Specification and Model Check

This is the specification and proof described in Section 2.2, for the N-processor case instantiated for $N = 2$.

A.2.1 Downscaled Specification

All that should have been necessary to create this specification is to instantiate the theory `cache_array` with the value 2. The PVS "judgement" `1 HAS_TYPE below(2)` is added to the specification to help the typechecker localize the checks that the constants `0` and `1` really are of type `below(2)`.

The PVS model checker performs symbolic model checking for specifications whose types are "hereditarily finite"—that is, built recursively from booleans, finite enumerations, and functions and records of these. Unfortunately, the PVS model checker cannot presently deal with subranges of the integers—as in the type `below(2)` used for `processor` (except in the domain of a function—as in the `cache` field of `protocol_state`), so it is necessary to create an enumeration type `procs = {a, b}` for use in the `bus_master` field of `protocol_state` and to convert from `procs` to `processor` in the definition of `p_next`.

It is also necessary to replace quantification over the `processor` type by explicit conjunctions and disjunctions. For example:

```
FORALL q: foo(q)
```

becomes

```
foo(0) AND foo(1)
```

while

```
  EXISTS q: foo(q)
```

becomes

```
  foo(0) OR foo(1).
```

These adjustments result in the following very ugly specification.

```
cache2: THEORY
BEGIN

  IMPORTING trans

  processor: TYPE = below(2)

  JUDGEMENT 1 HAS_TYPE below(2)

  line_status: TYPE = invalid, shared, exclusive

  procs: TYPE = a, b

  protocol_state:  TYPE =
        [# cache: [processor -> line_status],
           transaction: transactions,
           bus_master: procs #]

  p, q, r: VAR processor

  ps, ps0, ps1: VAR protocol_state

  p_init(ps): bool =
    invalid?(cache(ps)(0)) AND invalid?(cache(ps)(1))

  do_idle(ps0, ps1): bool = (ps1 = ps0)

  do_read_shared(ps0, ps1)(p): bool =
    invalid?(cache(ps0)(p))
        AND shared?(cache(ps1)(p))
          AND
        ((exclusive?(cache(ps0)(0))
              AND NOT exclusive?(cache(ps1)(0))
                AND exclusive?(cache(ps1)(1))
                  = exclusive?(cache(ps0)(1))
              OR exclusive?(cache(ps0)(1))
                AND NOT exclusive?(cache(ps1)(1))
                  AND exclusive?(cache(ps1)(0))
                    = exclusive?(cache(ps0)(0)))
             OR
```

```
         (NOT exclusive?(cache(ps0)(0))
              AND NOT exclusive?(cache(ps1)(0))
                AND NOT exclusive?(cache(ps0)(1))
                  AND NOT exclusive?(cache(ps1)(1))))

do_read_modified(ps0, ps1)(p): bool =
  shared?(cache(ps0)(p))
      AND
    (LET q = 0
       IN cache(ps1)(q)
             = IF p = q THEN exclusive
           ELSIF shared?(cache(ps0)(q)) THEN invalid
           ELSE cache(ps0)(q)
           ENDIF)
        AND
      (LET q = 1
         IN cache(ps1)(q)
               = IF p = q THEN exclusive
             ELSIF shared?(cache(ps0)(q)) THEN invalid
             ELSE cache(ps0)(q)
             ENDIF)

do_write_back(ps0, ps1)(p): bool =
  exclusive?(cache(ps0)(p))
      AND cache(ps1) = cache(ps0) WITH [(p) := invalid]

p_next(ps0, ps1): bool =
  LET p = (IF a?(bus_master(ps0)) THEN 1 ELSE 0 ENDIF),
      t = transaction(ps0)
    IN (ps1 = ps0
            OR (read_shared?(t) AND do_read_shared(ps0, ps1)(p))
              OR
            (read_modified?(t) AND do_read_modified(ps0, ps1)(p))
                OR (write_back?(t) AND do_write_back(ps0, ps1)(p)))

p_safe(ps): bool =
  NOT (exclusive?(cache(ps)(0)) AND exclusive?(cache(ps)(1)))

p_safe_strong(ps): bool =
  (exclusive?(cache(ps)(0)) IMPLIES invalid?(cache(ps)(1)))
      AND (exclusive?(cache(ps)(1)) IMPLIES invalid?(cache(ps)(0)))

IMPORTING MU@ctlops[protocol_state]

invariant_ctl: THEOREM p_init(ps) IMPLIES AG(p_next, p_safe)(ps)
```

```
  strong_invariant_ctl: THEOREM
        p_init(ps) IMPLIES AG(p_next, p_safe_strong)(ps)

END cache2
```

Notice that in order to support model checking, **invariant** and **strong_invariant** have been replaced by their CTL equivalents, and the appropriate instantiation of the CTL definitions are imported into the specification from the PVS MU (μ-calculus) library.

A.2.2 Proof Scripts for Downscaled Specification

PVS generates two proof obligations called TCCs (Type Correctness Condition) from this specification in order to ensure that the constants 0 and 1 appearing in various contexts are members of the type `below(2)`. These TCCs are the following.

```
% Subtype TCC generated for  1
  % proved - complete
one_TCC1: OBLIGATION 1 < 2;

% Subtype TCC generated (line 24) for  0
  % proved - complete
p_init_TCC1: OBLIGATION 0 < 2;
```

Both TCCS are discharged automatically by the command `M-x tcp`, which invokes the default proof strategy `(subtype-tcc)`. The two invariant properties are proved by expanding the definition of the `p_next` transition relation, and then invoking the model checker.

```
Proof scripts for theory cache2:


cache2.one_TCC1: proved - complete

("" (SUBTYPE-TCC))


cache2.p_init_TCC1: proved - complete

("" (SUBTYPE-TCC))


cache2.invariant_ctl: proved - complete

("" (EXPAND "p_next") (MODEL-CHECK))
```

```
cache2.strong_invariant_ctl: proved - complete

("" (EXPAND "p_next") (MODEL-CHECK))
```

A.3 Abstracted Specification, Refinement Proof, and Model Check

The abstracted specification described in Section 2.4 is actually constructed in two parts: one giving the abstract specification itself, and the other providing the constructions necessary to establish the "refinement" link between the specifications of the abstract and the N-processor protocols.

A.3.1 Abstract Specification

The abstract specification also provides definitions for the CTL forms of the abstracted `invariant` and `strong_invariant`.

```
abs_cache: THEORY
  BEGIN

  IMPORTING trans

  abstract_status:
      TYPE =
        all_invalid, some_shared,
         one_exclusive, many_exclusive, exclusive_shared

  abstract_state:
      TYPE = [# cache: abstract_status, transaction: transactions #]

  IMPORTING MU@ctlops[abstract_state]

  as, as0, as1: VAR abstract_state

  a_safe(as): bool =
    LET x = cache(as) IN
      all_invalid?(x) OR some_shared?(x) OR one_exclusive?(x)

  a_init(as): bool = all_invalid?(cache(as))

  do_idle(as0, as1): bool = (as0 = as1)

  do_read_shared(as0, as1): bool =
    (a_safe(as0) IMPLIES some_shared?(cache(as1)))
```

```
  do_read_modified(as0, as1): bool =
    LET x = cache(as0), y = cache(as1) IN
      (some_shared?(x) OR exclusive_shared?(x))
      AND IF some_shared?(x)
            THEN one_exclusive?(y)
            ELSE many_exclusive?(y)
          ENDIF

  do_write_back(as0, as1): bool =
    LET x = cache(as0), y = cache(as1) IN
      (one_exclusive?(x) OR many_exclusive?(x) OR exclusive_shared?(x))
          AND
        (one_exclusive?(cache(as0)) IMPLIES all_invalid?(cache(as1)))

  a_next(as0, as1): bool =
    LET t = transaction(as0) IN
     ((idle?(t) AND do_idle(as0, as1))
       OR (read_shared?(t) AND do_read_shared(as0, as1))
          OR (read_modified?(t) AND do_read_modified(as0, as1))
             OR (write_back?(t) AND do_write_back(as0, as1)))

  abs_invariant_ctl: THEOREM a_init(as) IMPLIES AG(a_next, a_safe)(as)

  END abs_cache
```

A.3.2 Model Check for Abstract Specification

```
("" (EXPAND "a_next") (MODEL-CHECK))
```

A.3.3 Refinement Specification

This is the specification that provides the abstraction function between `protocol_state` and `abstract_state`. It also states the three conditions necessary to establish that the concrete specification is a refinement of the abstract specification.

```
refinement[N: posnat]: THEORY
BEGIN

  IMPORTING abs_cache, cache_array[N]

   p, q : VAR processor
   ps, ps0, ps1: VAR protocol_state

   abst_cache(c: [processor -> line_status]) : abstract_status =
```

```
     IF (EXISTS p: exclusive?(c(p)))
     THEN IF (EXISTS q: shared?(c(q)))
          THEN exclusive_shared
          ELSIF (EXISTS p:
                  (FORALL q: exclusive?(c(q)) IMPLIES p = q))
          THEN one_exclusive
          ELSE many_exclusive ENDIF
     ELSIF (EXISTS p: shared?(c(p)))
     THEN some_shared
     ELSE all_invalid ENDIF

    abst(ps) : abstract_state =
     (# cache := abst_cache(cache(ps)),
        transaction := transaction(ps) #)

   init_simulation: THEOREM
      p_init(ps) IMPLIES a_init(abst(ps))

   next_simulation: THEOREM
      p_next(ps0, ps1) IMPLIES a_next(abst(ps0), abst(ps1))

   safety_preserved: THEOREM a_safe(abst(ps)) => p_safe(ps)

   strong_safety_preserved: THEOREM a_safe(abst(ps)) => strong_p_safe(ps)

END refinement
```

A.3.4 Proof Scripts for Refinement Specification

The first, third, and fourth proofs are straightforward. The second is quite long and moderately difficult: it takes about two minutes to run the proof on a Pentium Pro.

```
Proof scripts for theory refinement:


refinement.init_simulation: proved - complete

("" (GRIND))


refinement.next_simulation: proved - complete

(""
 (SKOSIMP)
 (EXPAND "p_next")
 (PROP)
 (("1" (EXPAND "a_next") (FLATTEN) (HIDE 2 3 4) (GRIND :IF-MATCH NIL))
```

```
("2"
 (EXPAND "do_read_shared")
 (PROP)
 (("1"
   (EXPAND "a_next")
   (FLATTEN)
   (HIDE 1 3 4)
   (GRIND :IF-MATCH NIL)
   (("1" (INST-CP -9 "p!2") (REDUCE :IF-MATCH ALL)) ("2" (REDUCE))
    ("3" (REDUCE)) ("4" (REDUCE))))
  ("2"
   (EXPAND "a_next")
   (FLATTEN)
   (HIDE 1 3 4)
   (GRIND :IF-MATCH NIL)
   (("1" (INST -3 "p!2") (ASSERT)) ("2" (INST? 2))
    ("3" (INST -2 "p!1") (ASSERT)) ("4" (INST? 2))))))
("3"
 (EXPAND "a_next")
 (FLATTEN)
 (HIDE 1 2 4)
 (EXPAND "do_read_modified")
 (PROP)
 (("1" (AUTO-REWRITE -3) (HIDE -3) (GRIND))
  ("2" (AUTO-REWRITE -3) (HIDE -3) (GRIND))
  ("3"
   (GRIND :IF-MATCH NIL)
   (("1" (AUTO-REWRITE -7) (INST - "q!1") (GRIND :IF-MATCH NIL))
    ("2"
     (INST 2 "bus_master(ps0!1)")
     (SKOSIMP*)
     (INST + "q!1")
     (INST - "q!1")
     (REDUCE :IF-MATCH NIL))
    ("3"
     (INST 2 "bus_master(ps0!1)")
     (INST - "bus_master(ps0!1)")
     (REDUCE :IF-MATCH NIL))))
  ("4"
   (GRIND :IF-MATCH NIL)
   (("1" (INST - "q!1") (REDUCE))
    ("2"
     (INST? -4 :IF-MATCH ALL)
     (INST-CP - "p!1")
     (INST - "bus_master(ps0!1)")
     (GRIND))
```

```
      ("3" (INST + "bus_master(ps0!1)") (REDUCE)) ("4" (INST?)) ("5" (INST?))
      ("6" (INST?))))))
  ("4"
   (EXPAND "do_write_back")
   (EXPAND "a_next")
   (FLATTEN)
   (HIDE 1 2 3)
   (GRIND :IF-MATCH NIL)
   (("1" (INST? - :IF-MATCH ALL) (GROUND)) ("2" (INST?)) ("3" (INST?))))))


refinement.safety_preserved: proved - complete

(""
 (GRIND :IF-MATCH NIL)
 (("1" (INST-CP -4 "p!1") (INST -4 "q!1") (ASSERT)) ("2" (INST?))))


refinement.strong_safety_preserved: proved - complete

(""
 (GRIND :IF-MATCH NIL)
 (("1" (INST 1 "q!1") (INST-CP -4 "p!1") (INST -4 "q!1") (ASSERT))
  ("2" (INST?))))
```

References

Papers by SRI authors are available from `http://www.csl.sri.com/fm.html`.

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1], pages 323–335.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[4] 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.

[5] David L. Dill. The Murϕ verification system. In Alur and Henzinger [1], pages 390–393.

[6] Bruno Dutertre and Steve Schneider. Embedding CSP in PVS. an application to authentication protocols. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 121–136, Murray Hill, NJ, August 1997. Springer-Verlag.

[7] Rob Gerth. Verifying sequentially consistent memory: Problem definition. Eindhoven Institute of Technology, April 1993. Available from `http://www.research.digital.com/SRC/tla/Intro.ps.Z`.

[8] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[9] Orna Grumberg, editor. *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[10] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

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

[12] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[13] *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior.* International Organization for Standardization—Information Processing Systems—Open Systems Interconnection, Geneva, Switzerland, September 1988. ISO Standard 8807.

[14] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in Systems Design*, 9(1/2):41–75, August 1996.

[15] *International Standard: Information Technology—Microprocessor Systems — Futurebus+ — Logical Protocol Specification.* ISO/IEC 10857 and ANSI/IEEE Std. 896.1, 1994.

[16] Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In *Formal Methods Europe FME '97*, volume 1313 of *Lecture Notes in Computer Science*, pages 589–604, Graz, Austria, September 1997. Springer-Verlag.

[17] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach.* Princeton University Press, Princeton, NJ, 1994.

[18] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–692, September 1979.

[19] 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.

[20] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[21] 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.

[22] Seungjoon Park. Practical verification of cache coherence algorithms with an automatic theorem-prover. In *2nd International Workshop on Verification of Infinite State Systems: Infinity '97*, Bologna, Italy, July 1997. To appear.

[23] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed actions. In Alur and Henzinger [1], pages 300–310.

[24] Seungjoon Park and David L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 1996.

[25] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Turin, Italy, April 1982. Springer-Verlag.

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

[27] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1994.

[28] Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996. Springer-Verlag.

[29] Hassen Saïdi. The Invariant Checker: Automated deductive verification of reactive systems. In Grumberg [9], pages 436–439.

[30] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Grumberg [9], pages 72–83.

[31] N. Shankar. Computer-aided computing. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1995.

[32] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy, editor, *Mathematical Methods in Program Development*. Springer-Verlag, 1997. To appear. Draft available at `http://www.csl.sri.com/~shankar/shankar-papers.html`.

[33] Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996. Springer-Verlag.