Protocol Verification in Nuprl [⋆]

Amy P. Felty[1], Douglas J. Howe[1], and Frank A. Stomp[2]

[1] Bell Labs, Murray Hill, NJ 07974, USA. {felty,howe}@bell-labs.com
[2] Dept. of Comp. Sci., UC Davis, Davis, CA 95616, USA. stomp@cs.ucdavis.edu

Abstract. This paper presents work directed toward making the Nuprl interactive theorem prover a more effective tool for protocol verification while retaining existing advantages of the system, and describes application of the prover to verifying the SCI cache coherence protocol. The verification is based, in part, on formal mathematics imported from another theorem-proving system, exploiting a connection we implemented between Nuprl and HOL. We have designed and implemented a type annotation scheme for Nuprl's logic that allows type information to be effectively applied by the system's automated reasoning facilities. This is significant because Nuprl's powerful constructive type theory buys much of its expressive power and flexibility at the cost of giving up the more manageable kinds of type system found in other logics.

1 Introduction

Nuprl [2] is an interactive theorem-proving system in the lineage of LCF. One of its main distinguishing characteristics is its highly expressive formal logic, a constructive type theory whose classical variant has expressive power equivalent to conventional set theory (ZFC) [12, 6].

Nuprl has been extensively applied, and its expressive power has been shown to be a substantial advantage in a variety of domains, but little work has been specifically directed toward effectiveness for the kind of large-scale practical applications where the bulk of the formal mathematics is highly complicated, but shallow and representationally simple.

This paper describes our work in this direction, and features an application of Nuprl to prove safety properties of the SCI cache coherence protocol [8].

We chose SCI as an example partly because its complexity is representative of the scale of algorithms which can be currently handled by mechanized tools. Model checking systems that have been applied to the protocol suffer from state explosion at a small number of processors, though even so some bugs have been found [11]. A second reason for choosing it is that a proof method and supported invariants have already been worked out [3].

Our work has been to improve Nuprl for these kinds of applications without compromising existing advantages of the system by, *e.g.*, adding restrictions to the logic. There are three parts to this work.

[⋆] In *Proceedings of the Tenth International Conference on Computer-Aided Verification, June 1998.*

Imported mathematics. Verification using an interactive theorem-prover requires a great deal of basic formal mathematics about elementary data structures and models. Building it is time-consuming, and is largely duplication of effort since these basic facts tend to be similar across systems. To avoid doing this ourselves, we *import* some basic mathematics from HOL [5], a system that has, over the years, accumulated a large corpus of mathematics of the kind useful for software/hardware verification. The paper [7] gives the basic design of the connection between HOL and Nuprl, and [4] gives an extension to it and an application to a moderately difficult problem in metamathematics. Our work, though just a first step, establishes that sharing mathematics can be useful in software/hardware verification.

Type Annotation. Nuprl buys its expressive power at the cost of some traditional aspects of type systems. In particular, the type theory's flexibility is in large part due to the fact that terms are *untyped* in the sense that one cannot determine from the syntax of an expression what, if any, type it is a member of. In this way, Nuprl is similar to set theory, with types being analogous to sets. This is a problem for automation for two reasons. First, it is often important for terms to come with their types; for example, in term rewriting, type information can enable a useful form of conditional rewriting. Second, typing properties require proof, so, for example, every time a lemma is instantiated, the instantiating objects must be proved to have the right types. We have designed and implemented an annotation scheme where terms are decorated with types in such a way that types can (almost always) be efficiently maintained during inference, but no new syntactic restrictions are placed on the logic. We have obtained roughly a factor of 10 speedup in term rewriting (the main workhorse in Nuprl proofs). Unfortunately, the implementation wasn't completed until part-way through the SCI effort, so a good deal of work was done without its benefit.

Tactic support. We represent the protocol and its specification using a familiar kind of embedding of a Unity-like language. We used Nuprl's *tactic* mechanism to implement a suite of automated reasoners specialized to this model.

One might ask why not just use HOL (for example)? The answer is that we are aiming to make Nuprl an effective tool for a wide range of formal problems related to protocol verification. For example, we want to be able to reason about abstraction and refinement methods (see [1] for an example), an area where expressive power can be a great advantage. Of course, there are verification tasks, such as checking that the atomic state transitions of a system preserve a property, where expressive power may be less important and where the speed and effectiveness of basic inference mechanisms, such as term rewriting, is crucial. One goal of our work is to enhance the second kind of reasoning without imposing restrictions that affect the first kind.

Our proof is completely constructive (by choice). While we don't see much application for this fact in this particular case, it is noteworthy that constructivity has not gotten in the way. It may be possible to engineer constructive proofs of protocols from which one can synthesize, for example, programs that track simulations of the protocol and produce interesting data about the current state.

In the rest of the paper we describe the SCI correctness proof and the improvements we made to Nuprl. The proof is not yet finished, though it is nearing completion. A description of what remains to be done is included later in the paper. Details of the completed formalization will be available on the web at www.cs.bell-labs.com/~felty/sci/.

2 SCI Cache Coherence and its Formalization in Nuprl

This section gives an overview of the SCI cache coherence protocol and its formalization in Nuprl. Before proceeding to the overview, we give a brief description of Nuprl. Formal mathematics in Nuprl is organized in a single library, which is broken into files simulating a theory structure. Library objects can be definitions, display forms, theorems, comments or objects containing ML code. Definitions define new operators, possibly with binding structure, in terms of existing Nuprl terms and previously defined operators. Display forms provide notations for defined and primitive operators. These notations need not be parsable since Nuprl uses structure editors. Theorems have tree structured proofs, possibly incomplete. Each node has a sequent, and represents an inference step. The step is justified either by a primitive rule, or by a *tactic*. Nuprl's notion of tactic is derived from that of LCF, as is HOL's.

Nuprl's type theory has a rich set of type constructors. The following are some example types: $\Pi n \in N \,.\, B^n \to B^n$,

$$\{\, x \in N \; \mathit{list} \,|\, x \neq \mathit{nil} \,\}, \quad \Sigma\, n \in N \,.\, B^n, \quad (x,y) : Z \times N^{+}//(x_1 y_2 = y_1 x_2).$$

The first of these can be thought of as the type of functions mapping an n and an n-ary bit-vector to an n-ary bit-vector. The second is the type of nonempty list of natural numbers, the third is the collection of pairs (n, b) such that b is an n-ary bit-vector, and the last is a quotient type representing the rational numbers represented as pairs of integers with the usual equivalence relation.

2.1 SCI Cache Coherence

The SCI protocol is an IEEE standard for specifying communication between multiprocessors in a shared memory model [8]. Due to the space limitations we present a very high-level description of our model of the cache coherence part of that protocol. A detailed description of our model can be found in [3].

Processors which try to access the store form a doubly linked list. This list can be thought of as prioritizing processors so that read and write conflicts do not arise. The protocol is distributed; there is no global cache or global data structure for the linked list. Instead each processor p has a set of local variables which keeps track of, for instance, its view of the cache (cv_p), knowledge of whether or not its view is valid (cs_p), and its current successor ($succ_p$) and predecessor ($pred_p$) on the linked list, if any. All communication is via point-to-point message passing. Since a very large number of processors could be on the network, a huge amount of concurrency is present, complicating the understanding of the protocol. (The IEEE standard specifies an upper bound of 64,000 processors. The proof we are formalizing proves the correctness for an arbitrary finite number of processors.)

The protocol is specified as a set of guarded actions. For example, the following is an action executed by the memory controller m.

```
$buf[m]?read\_cache\_freshQ(p)\longrightarrow$
  if $status_m = Gone$ then $buf[p]!read\_cache\_freshR(m, head_m, cv_m, gone)$
  else $buf[p]!read\_cache\_freshR(m, head_m, cv_m, ok)$ fi;
  $head_m := p$; if $status_m = Home$ then $status_m := Fresh$ fi
```

Here, the guard indicates that this action can be executed if the first message in $buf[m]$ (m's message buffer) has type *read_cache_freshQ* which indicates that processor p wants to read. The message is removed from the queue (received) and the body is executed. A message $read\_cache\_freshR(m, head_m, cv_m, gone)$ is sent to processor p, if some processor on the list had issued a write query (indicated by the argument *gone*). Otherwise, response $read\_cache\_freshR(m, head_m, cv_m, ok)$ is sent to p. (Argument *ok* indicates that no processors are on the list which have requested to modify the store.) Local variable $status_m$ is used by m to record whether some processor is on the list which has issued a write query — its value is then *Gone*; or whether processors on the list have issued read queries only — its value is then *Fresh*; or if no such queries have been issued and hence the list is empty — its value is then *Home*. Finally, local variable $head_m$ is maintained by m to record the head of the list. As shown by this example, bodies can contain assignments, conditionals, and sends. In addition to receives, guards can be boolean conditions.

The protocol is represented as 21 actions: 4 for memory including the one above and 17 for each processor. Communication is via 14 types of messages, made up of 7 pairs of query (Q) and response (R) messages. In addition to the above action, memory has two actions responding to write requests, one from a processor that is already on the doubly linked list because it is reading, and one from a processor that is not yet on the list. It also has an action responding to a processor that wants to go off the list. The 17 actions for each processor include one read request, two write requests, actions for requesting to go on the list or to go off the list (for example, after it has "accessed" the store), an action for purging others off the list when it has been given permission to write the store and decided that it is indeed going to do so, actions for modifying the cache, as well as actions that respond to each kind of request from another processor. This high degree of communication is a main complicating factor in the protocol. Several rounds of messages must be exchanged before a processor is on the list with $succ_p$ and $pred_p$ properly set. Thus, the doubly linked list is constantly modified and constitutes an abstraction of the structure which arises during an actual computation. A variable $status_p$ keeps track of a processor p's state with respect to the list and can take on one of 8 possible values.

2.2 Formalization in Nuprl

Our formalization of correctness follows closely the proof in [3]. Our embedding of the semantics of state transition systems in Nuprl is fairly straightforward. We define a state as a pair where the first component is the usual mapping from identifiers to values. The second component is a *history* variable that records the sequence of messages that have been sent and received during the entire execution. This history variable is important for reasoning about the program's communication behavior. The Nuprl definitions of the components of state are given below. Booleans (B), atoms, integers (Z), and lists are defined in the standard Nuprl libraries.

```
PId  ==  {k:Z| k ≥ 0 }          hist_el ==  B × PId × Z × mesg
id   ==  Atom × PId             hist    ==  hist_el List
mesg ==  Z × Z List             state   ==  (id → Z) × hist
```

For simplicity, the values of all identifiers (id) are assumed to be integers. The first component of an identifier is its name (type Atom) and the second is the process identifier (type PId) to which the variable belongs. The first component of a history element (hist_el) is a boolean value indicating whether the message is a send (tt) or a receive (ff). The remaining components are the sender, receiver, and message (type mesg). Message types such as *read_cache_freshQ* are encoded as integers as the first component of a message. The second component encodes the arguments.

Expressions and commands are defined as functions on state. As an example, we give the definition of the assignment command.

```
com ==  state → state
x:=e ==  λs.<λy.if (x = y) then (e·s) else (y·s), s.h>
```

Nuprl's display forms are used to define := and · as infix operators. The dot is used for evaluation in a state and is overloaded. Here e·s is expression evaluation defined as (e s) and (y·s) maps identifiers to values and is defined as (s.1 y) (where .1 denotes the projection of the first element of a pair). Other commands are defined similarly. Note that the assignment statement updates the first component of the state. The send command updates the second component by simply adding a history element to the front of the history with tt as its first component and the new message as its last component. (Histories and buffers are represented in reverse order.) The receive command also adds a history element to the front of the history, but is more complicated because it computes this element from the contents of the current history h. It uses an operation queue(p;h) which filters out those history elements that contain messages that have been sent and not yet received by process p. It then chooses the last (oldest) element and creates a new copy whose first component is ff. The message buffer of a process p in state s, denoted (buf[p])·s, is also computed using queue. In this case, the message components of the elements of list queue(p;s.2) are projected out.

A program is defined as a pair containing a list of commands and an initial condition which is a predicate on state (of type state → P_1 where P_1 is the type of Nuprl propositions). In our model, a command is enabled if it changes the state when applied. Thus commands whose guards are true but do not change the state are considered disabled. A trace is defined in the usual way as a function from natural numbers to states such that for any n, there is an action (enabled or not) such that when applied to state n results in state $n + 1$.

The correctness of the SCI cache coherence protocol is stated as five linear temporal logic formulas. The first, for example, expresses that there is always a unique cache owner. The notion of cache owner is fairly complex because of the distributed nature of the protocol. If no processor has requested to write to the cache, then memory is the owner. Otherwise, the owner roughly corresponds to the processor p whose variable cs_p has value *dirty*. However, there are various cases where 0 or more than 1 processor has this value. In such cases there is a always a message in some processor's buffer that will cause it to set its value of cs_p to *dirty* or to something else making it or some other processor the unique owner. In order to show that this uniqueness property and the other four properties hold, we prove a series of complex invariants from which these properties follow. These invariants are expressed as 14 lemmas (spanning several pages in [3]), each with several interdependent clauses. There are also many auxiliary concepts that appear in the invariants. For example, there are 6 predicates on processors indicating their degree of progress in getting on or off the doubly linked list. The most complex concept is a function called *rank* whose value reflects how close a process is to getting permission to write.

In related work, Stern and Dill [11] use Murϕ, a verification system that employs explicit state enumeration, to analyze SCI cache coherence. Their largest example included three processors with one cache line each, one memory with one address and two data values, and they reported finding several errors using a smaller example. The model they used was extracted from the C code describing the protocol in [8], whereas our model has been constructed from the informal English explanation. By abstracting at this level, inconsistencies in the lower-level description were removed. Our model also differs from theirs (and from the SCI protocol standard) in that we have assumed that messages sent from one processor to another processor are always received in the order sent. Stern and Dill check for certain safety properties, two of which are formulated as invariants. One of their invariants corresponds to one of our five correctness properties stating that processors in a certain state have a consistent view of the cache. The other is essentially the same as an invariant in one of our supporting lemmas stating at what point a processor is at the head of the linked list.

In [10], Park and Dill use PVS to verify the FLASH cache coherence protocol. Because the protocol uses directories instead of the distributed list of SCI, it seems simpler, and also it seems that the abstraction method they employ may not be applicable to SCI.

3 Imported Mathematics

In this section we describe the connection between HOL and Nuprl, and summarize how it was used in our proof.

3.1 The Importation Mechanism

We believe that much of the mathematics used in practical verification is highly sharable, including theories of basic data types, and also a good deal of the mathematics related to software modeling and semantic connections to external tools. We have taken a first step toward this kind of sharing by borrowing some of the mathematics we needed for our verification from HOL.

Importation of mathematics from HOL into Nuprl is done at the theory level. An HOL theory consists of some type and individual constants, some axioms (usually definitional) constraining the constants, and a set of theorems following from the axioms (and the axioms of ancestor theories). To import a theory, one *interprets* the type constants with Nuprl types and the term constants with members of the appropriate types, and then proves the axioms. When this is done, the theorems can then all be accepted immediately as Nuprl theorems. Typechecking is undecidable in Nuprl, so the well-typedness of interpreting terms must be proven explicitly.

Theorems directly imported from HOL are usually of a form that makes them useless for direct application in Nuprl proofs. It turns out that massaging the theorems into the desired form is possible, and is largely automatable.

To illustrate what kind of transformations are needed on directly imported mathematics, consider an example from list theory. The following is a raw import of a HOL theorem stating that a non-empty list is a cons. Because Nuprl currently has a single flat namespace, the names of all imported constants have an "h" prepended to avoid conflicts with Nuprl objects. The outermost quantifier quantifies over the type S of all (small) non-empty types (this quantifier is implicit in HOL).

```
∀'a:S ↑(hall (λl:hlist('a).
               himplies (hnot (hnull l))
                        (hequal (hcons (hhd l) (htl l)) l)))
```

Apart from the outermost quantifier, the logical connectives themselves are imported constants. The transformed, "Nuprl-friendly" theorem generated from the above is

```
∀'a:S. ∀l:'a List.  ¬mt(l) ⇒ hd(l)::tl(l) = l.
```

The logical connectives in HOL are all boolean-valued functions, possibly taking functional arguments, as in the case of the quantifiers. The interpretations of these connectives use boolean logic defined within Nuprl. The boolean connectives are rewritten in the second theorem to Nuprl's normal logical connectives, which are defined using a propositions-as-types correspondence. The operator

↑ in the imported theorem coerces a boolean into a Nuprl proposition. The imported list type is interpreted as Nuprl's list type, and the imported tail function is interpreted as Nuprl's tail function. Note however that `htl` is *applied*, as a function, to its argument, while the Nuprl `tl` is a defined operator with a single operand (Nuprl also has an operator for function application, of course). We have used a notational device to suppress type arguments in the (pre-rewrite) imported theorem. Each of the imported constants in the theorem actually has at least one type argument. In the rewritten theorem, there are no hidden type arguments (the Nuprl operations are "implicitly polymorphic").

The most interesting point in this translation is the function for head of a list. In HOL, this is a *total* function on lists. When we import it into Nuprl, we must prove that the interpretation returns a value on every list, empty or not. Since `hhd` is polymorphic, given an arbitrary type and the empty list as an argument, it must choose some arbitrary member of the type as output. Thus we must give `hhd` a noncomputable definition in Nuprl. However, we can prove that this function is the same as Nuprl's `hd` when the list is non-empty. This gives us a conditional rewrite which goes through for this example theorem.

3.2 HOL Math Used in the SCI Verification

The main source of HOL theorems used in the SCI verification is a large body of theorems about lists. Lists are important in two central areas of the proof. First, the definition and proof of properties about the contents of buffers require sophisticated list manipulation since, as mentioned, they are computed from the history component of a state. For example, from the definition of buffer, it fairly is straightforward to prove that when a message `M` is sent to process `p` in state `s`, its buffer becomes `M::((buf[p])·s)` where `::` is the cons operator. The proof that `but_last_el((buf[p])·s)` is the contents of `p`'s buffer after `p` receives a message is significantly more complex. The operator `but_last_el` is defined in an HOL library in terms of the `lastn` operator (the operation which extracts the last n elements of a list) which is also defined in HOL. The `snoc` operator, which is the opposite of `cons` (in particular, the property `snoc(x;l) = l @ (x::[])` holds, where `@` is the append operator), is also defined in HOL and is useful for reasoning about these operators. The existing HOL theorems about these and a variety of other operators were directly usable in this and other proofs.

The above two theorems are examples of lemmas used as rewrite rules. Nuprl provides powerful automation for the application of rewrite lemmas and good use of this machinery is essential for a large proof such as the SCI verification. We proved and make extensive use of numerous other rewrite lemmas involving histories and buffers. A variety of other theorems about histories and buffers have also been proved and used as support for other kinds of rewrite lemmas.

One invariant (part of Lemma 9 [3]) states that any processor has at most one outstanding message. In particular, for any Q/R pair, there is at most one Q message for which a processor is waiting for the corresponding R message. This means that there is either 0 or 1 Q messages from a processor p in some q's buffer, or there is 0 or 1 R messages in p's buffer, but not both. Our rewrite lemmas along with various other list operators and properties from HOL play a central role in proving this fact.

The second area of the proof in which lists are important is in defining the notion of rank. Rank roughly corresponds to the order in which processors have requested to read or write to the cache. It is only defined for *active* processors, a property of processors that are on or "mostly on" the doubly linked list. An important property is the fact that for any processor, its rank does not increase. This property insures that the list does not contain circularities. As long as a process stays active (and a few other properties hold) its rank will decrease until it becomes 0 at which point it is allowed to write if it has requested to do so. Rank is defined by filtering from the history all read and write requests that memory has received, projecting out the sender, and keeping only the first occurrence of each active processor in the resulting list. The first occurrence corresponds to a processor's most recent request. We prove a variety of lemmas describing how a processor's rank changes with changes in the state. These lemmas are also used as rewrite rules in proving invariants.

4 A Type Annotation Scheme for Nuprl

Our type annotation scheme is a way of attaching type expressions, which we call *annotations*, to all (or only some) of the subterms of a term. Our scheme meets the following goals.

1. Annotations are optional. Terms that do not have annotations attached to them are treated as before by Nuprl's tactics.
2. If a term t is introduced into a proof as a member of a type T, and t occurs somewhere in the current goal with a compatible annotation, then the requirement to prove $t \in T$ is eliminated.
3. Annotations justify rewriting, so that a subterm with an annotation A can be replaced by an equal term (*qua* member of A) without further justification.
4. There are no heuristics in the scheme *per se*. Although type inference and checking are highly heuristic in Nuprl, this is independent of the annotation scheme. Annotations for terms are generated by examining the results of applying Nuprl's existing machinery.
5. Annotations can be effectively maintained. In principal, it is possible for annotations to be lost during inference. For example, the generalized term in the induction rule needs to reannotated (or left without annotations). However, such inference steps form a tiny fraction in practice. For example, annotations are almost never lost during equational rewriting.
6. There are no global tables. We retain the tree-structuring of proofs, with independence of proof branches, that allows us, among other things, to do dependency-directed backtracking, and selective replay of subproofs.
7. Soundness depends only on a fixed set of primitive inference rules that all proofs must reduce to.
8. The scheme is almost entirely invisible to users.

The type theory of the PVS system [9] has some similarities to Nuprl, such as subtypes, (a limited form of) dependent types, and undecidable typechecking. PVS uses a typing discipline that achieves most of the goals above, but it would only be applicable to an insufficiently small subtheory of Nuprl. Some complicating aspects of Nuprl, which aren't present in PVS, are: universe polymorphism; type-indexed equality, so that two terms may both be in two types, but be equal in one type and not in the other; contravariant subtyping, where a function type is enlarged when its domain is shrunk; and general dependent types. In addition, the PVS scheme does not address 7 above.

Nuprl terms have the form $\theta(\overline{x}_1.\, e_1; \ldots; \overline{x}_n.\, e_n)$ where θ is an *operator* and in each operand $\overline{x}_i.e_i$, each of the variables in the sequence $\overline{x}_i$ binds in e_i. Note that no types are associated with the variables in this syntax. An *annotated term* has the form

$$\theta(\ldots; \overline{x}_i.\ e_i : [\phi_i]A_i; \ldots) : B$$

where the e_i are also annotated terms. The expressions $[\phi_i]A_i$ are the *subannotations* of the term, and can be thought of as the expected types for the operands, and B is the annotation type of the term. Informally, $e_i : [\phi_i]A_i$ can be thought of as meaning that under assumption ϕ_i, e_i has type A_i. The ϕ_i can refer to the variables in $\overline{x}_i$, and can contain, for example, assertions of the form $x \in T$. Examples of annotated terms are $fact((3 : Z) : [true]N) : N$, where $fact$, N and Z are factorial, the natural numbers and the integers respectively, and *if(b:B; e_1:[b]A; e_2:[¬b]A): A*.

One of the key points is how the annotation type of a term relates to its subannotations and to the subannotations of an immediately surrounding term. We chose the minimal requirement that supports rewriting as described above, and so we require only respect for equality. For example, in $\theta((e : A) : [\phi]A') : B$, where the operand $e : A$ is itself an annotated term, we require, first, that for all $x \in A'$, if $x = e \in A'$ then $\theta(x) = \theta(e) \in B$, and, second, that for all $x \in A$, if $x = e \in A$ then $x = e \in A'$. The generalization of this requirement to the presence of binding variables is straightforward.

As with ordinary typing in Nuprl, the validity of an annotation of a term is undecidable, and must be proven. One possibility would be to generate "type checking conditions" as PVS does, which are side conditions generated whenever a new term is introduced. This is not workable for Nuprl because tactics work by putting together appropriate primitive inference rules, and need an opportunity to assemble proofs of annotation validity at the same time as the proofs justifying the main inference. Rewriting works, for example, by taking a term and producing a rewritten term along with a proof of equality. For annotated terms, it is natural to modify rewriting to take an annotated term, and produce a new term, an equality proof, and *also* a proof that the new term's annotations are correct. We therefore have two kinds of annotations: one kind we can assume are valid during the course of a proof, and the other must be proved to be valid.

The annotation scheme is justified semantically, and requires a re-interpretation of the semantics of sequents. A full report is in preparation.

5 The Correctness Proof in Nuprl

The definition below encodes the formula $\Box P$ from linear temporal logic and is central in proving invariants. A state `s` is in an execution of program `prg`, denoted `in_exec(s;prg)`, if `s` occurs in some trace of `prg`.

```
inv(prg;s.I[s]) ==  ∀s:state. in_exec(prg;s) ⇒ I[s]
```

In a proof of this magnitude, it was essential to provide a high degree of automation. Our automation falls roughly into two categories: tactics that decompose reasoning modularly, and properties expressing equality and equivalence that can be used by Nuprl's rewriting machinery such as those mentioned in Sect. 3.2. Both the decomposition properties and rewrite theorems include general theorems and theorems specific to SCI. The rewrites for message buffers discussed in Sect. 3.2, for example, are not specific to SCI, while the notion of rank is. The decomposition tactics rely on lemmas that we have proven, such as one stating that to show that `inv(prg;s.I[s])` holds, it suffices to consider one case for each action of the program and to show that the initial condition holds in the initial state. From this general lemma, we proved decomposition lemmas for SCI which decompose reasoning into 21 cases, one for each memory action and one for each processor action for some arbitrary processor p. We chose to further decompose conditional statements into cases so that each case contains only send, receive, and assignment statements. Rewriting operates on these simplified cases. Although these decomposition properties are specific to SCI, we automated the generation of their statements — as well as a variety of other properties specific to SCI — from the definitions of the actions. Their proofs were often largely automatic also. We also automated the application of many of these lemmas by writing tactics which apply them and solve various subgoals automatically.

Of the 14 lemmas expressing invariants, the first 8 (roughly 2.5 pages in [3]) are fairly simple and express properties about the values that various variables can take on during execution. For example, we prove:

$$\begin{array}{l} \mathit{read\_cache\_freshR}(p, r, cv, arg) \in \mathit{buf}[p] \Rightarrow \\ \quad [p = m \wedge q \in \mathcal{P}(n) \wedge (r = \mathit{nil} \vee r \in \mathcal{P}(n)) \wedge (arg = ok \vee arg = gone)]. \end{array}$$

Here $\mathcal{P}(n)$ denotes the set of processors involved in the protocol, with process identifiers $1, \ldots, n$.

The 9^{th} lemma contains five statements which together express the property of outstanding messages described in Sect. 3.2 as well as eight statements expressing which kind of outstanding message a processor p has depending on the value of status_p. Lemmas 10 and 11 express a variety of properties of the form $\Box(P\ W\ Q)$ (where W is the weak until operator). We proved a general decomposition theorem for formulas of this form which makes the structure of these proofs similar to those for the other invariants. Lemma 12 expresses some basic properties about rank including two which follow directly from the definition (which is slightly different but equivalent to the one given in [3]) and two which must be proven as invariants. While the invariants up to this point are large and detailed, they are fairly straightforward to prove. The main difficulty in the proof is found in the 13^{th} and 14^{th} lemmas. Lemma 13 has 17 clauses and one assumption which later gets discharged and Lemma 14 has 7 clauses. They state the complex invariants about rank that are required to prove correctness of the protocol.

The proofs up through and including Lemma 11 are completed, as well as the two properties of Lemma 12 that follow from the definition of rank. We have also proven 5 and nearly completed 2 more of the 17 clauses of Lemma 13. For example, we have proven the invariant:

$$purgeQ(q) \in buf[p] \Rightarrow (visiting(p) \wedge rank(q) = rank(p) + 1)$$

where *visiting* processors are a subset of the *active* ones. In doing so, we have developed all of the rewrite lemmas about the rank function and all other auxilliary predicates that we need to complete the remainder of Lemmas 12, 13, and 14. The reasoning needed to complete the proof by showing that the desired safety properties follow from these invariants will be detailed but straightforward.

Because we started from a proof of correctness [3], we did not expect to find errors in the protocol. However, we have found two errors in the proof. Two of the conjuncts of the first clause of Lemma 13 could not be proved using the assertions we had formulated, although they are true. To prove these conjuncts, we had to add and prove some additional clauses. One is an invariant explicitly stating that two particular messages sent from one processor to another are received in the order sent.

References

1. C.-T. Chou and D. Peled. Verifying a model-checking algorithm. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 241–257. Springer-Verlag, 1996.
2. R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
3. A. Felty and F. Stomp. A correctness proof of a cache coherence protocol. 1997. Available at www.cs.bell-labs.com/~felty/sci/. An earlier version appears in *Proceedings of the 11th Annual Conference on Computer Assurance*, 1996.
4. A. P. Felty and D. J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In *Fourteenth International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 1997.
5. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.
6. D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.
7. D. J. Howe. Importing mathematics from HOL into Nuprl. In *Theorem Proving in Higher Order Logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 267–281. Springer-Verlag, 1996.
8. IEEE-P1596-05Nov90-doc197-iii. *Part IIIA: SCI Coherence Overview*, 1990. Unapproved Draft. Approved standard is described in IEEE Std. 1596-1992 "The Scalable Coherent Interface".

9. S. Owre and N. Shankar. The formal semantics of PVS. Technical report, SRI, August 1997.
10. S. Park and D. L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, 1996.
11. U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods*, 1995.
12. B. Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.