TLA$^+$ Verification of Cache-Coherence Protocols

Homayoon Akhiani, Damien Doligez*, Paul Harter, Leslie Lamport,
Mark Tuttle, and Yuan Yu
Compaq

Joshua Scheid
University of Illinois

10 February 1999

*on leave from INRIA

Contents

1 Introduction

This paper describes two projects to formally specify and verify cache-coherence protocols for multiprocessor computers being built by Compaq. These protocols are significant components of major products, and the projects are steps towards moving formal verification from research to Compaq engineering.

The first project, undertaken by researchers, was for an Alpha EV6-based computer[1]. The protocol was too complex to be verified completely, but the results were encouraging enough to motivate the second project—formal verification of the protocol for the Alpha EV7 processor. This project, which is still underway, is being undertaken jointly by researchers and engineers. So far, it has been quite successful.

Verifications of several cache-coherence protocols have been described in the literature [10, 11, 12, 13]. However, the protocols we have attempted to verify are more complicated than any we know of. For example, the protocol specification verified by McMillan and Schwalbe consisted of 500 lines of "code" that was "written in a very simple description language based on Boolean logic" [10]. In contrast, our specifications each consist of about 1800 lines of mathematical formulas, written in a sophisticated high-level language based on first-order logic and set theory.

Both projects used the specification language TLA$^+$, described in Section 2. That section also describes the hierarchical proof style that we used. The EV7 project is also using TLC, a model checker being developed for a subclass of TLA$^+$ specifications. TLC is described in Section 4.

The project to verify the EV6 protocol began in the fall of 1996 and ended the following fall; it is described in Section 3. The project to verify the EV7 protocol began in the spring of 1998; it is described in Section 5. A concluding section summarizes what we have learned from these projects about the use of formal methods in industry.

2 Preliminaries

2.1 TLA$^+$

TLA$^+$ is a formal specification language based on (untyped) ZF set theory, first-order logic, and TLA (the Temporal Logic of Actions). TLA is a version of temporal logic developed for describing and reasoning about concurrent

[1] EV6 and EV7 are the internal names for the Alpha 21264 and 21364 processors.

and reactive systems [7]. TLA$^+$ includes modules and ways of combining them to form larger specifications. An introduction to TLA$^+$ appears in [4], further details can be found on the World Wide Web [5].

Although TLA$^+$ permits a wide variety of specification styles, the specifications discussed here all have the form $Init \land \Box Next \land Liveness$, where:[2]

Init is the initial-state predicate—a formula describing all legal initial states.

Next is the next-state relation, which specifies all possible steps (pairs of successive states) in a behavior of the system. It is a disjunction of actions that describe the different system operations. An action is a mathematical formula in which unprimed variables refer to the first state of a step and primed variables refer to its second state. For example, $(x' = x + 1) \land (y' = y)$ is an action describing steps in which the variable x is incremented by 1 and the variable y is unchanged.

Liveness is a temporal formula that specifies the liveness (progress) properties of the system as the conjunction of fairness conditions on actions. Although the specifications we wrote included liveness properties, these properties were not checked during the verification, so we ignore liveness here.

Such a specification can be viewed as describing a state machine. However, unlike specifications in methods based on abstract machines or programming languages, a TLA$^+$ specification consists of a single mathematical formula. Moreover, the state predicates and actions in TLA$^+$ specifications are arbitrary formulas written in a high-level language with the full power of set theory and predicate logic, making TLA$^+$ very expressive.

A state predicate I is an invariant of a specification S iff S implies $\Box I$, the formula asserting that I is always true. When S equals $Init \land \Box Next$, this is proved by finding a formula Inv, called an invariant of $Next$, that satisfies $Init \Rightarrow Inv$, $Inv \land Next \Rightarrow Inv'$, and $Inv \Rightarrow I$, where Inv' is the formula obtained from Inv by priming all variables. A particularly simple form of invariant is one that expresses type correctness, asserting that each variable is an element of some set (its "type").

To prove that a specification S_1 implements another specification S_2, we must express the variables of S_2 as functions of the variables of S_1 and prove $S_1 \Rightarrow \overline{S_2}$, where $\overline{S_2}$ is formula S_2 with its variables replaced by the

[2]The formula $\Box Next$ should actually be $\Box[Next]_v$, where v is the tuple of all variables; we drop the subscript for simplicity. We also ignore the hiding of internal variables, expressed with temporal existential quantification.

corresponding functions of the variables of S_1. A major part of this proof involves finding and verifying a suitable invariant of S_1. The proof may also require adding auxiliary variables to S_1, usually to capture history information [1].

TLA$^+$ specifications are mathematical formulas, and correctness properties are expressed directly as mathematical theorems. There is no need to generate verification conditions; what you see is what you prove. Moreover, the proof rules of TLA reduce all proofs to reasoning about individual states or pairs of states, using ordinary, nontemporal mathematical reasoning. Over the years, such state-based reasoning has been found to be more reliable than behavioral reasoning, in which one reasons directly about the entire execution.

2.2 Hierarchical Proofs

The TLA$^+$ formulas that describe our protocols are about 1800 lines long. Correctness proofs of such systems are therefore long and complex. Handling complexity requires two things: hierarchical structure and the ability to refer to parts of that structure by name.

We use a hierarchical proof structure, in which a level n proof step consists of a numbered statement and its proof, which is a sequence of level $n+1$ proof steps [8]. The steps of a level n proof are named $\langle n \rangle 1$, $\langle n \rangle 2$, etc. (Even though all level n proofs use the same step names, references to those names are unambiguous in context.) At the lowest level are the leaf proofs. In a mechanically checked proof, they contain instructions for a proof checker. A hand proof is usually not carried out to so low a level, and the leaf proofs are short paragraphs containing ordinary mathematical reasoning. Greater rigor can be achieved by carrying out the proof to more levels. Structured proofs provide a continuum of tradeoffs between reliability and cost—the more levels of detail, the greater the rigor and the higher the cost.

Mathematical formulas are inherently structured—usually as nested conjunctions and disjunctions. A simple convention is used to name individual conjuncts and disjuncts; for example, the second conjunct of the third disjunct of the fourth conjunct of formula P is named $P.4.c.2$ [6].

The structure of the formula to be proved largely determines the structure of the proof. For example, the proof of $A \Rightarrow B \wedge C$ consists of the two lower-level steps $A \Rightarrow B$ and $A \Rightarrow C$, and the proof of $A \vee B \Rightarrow C$ consists of the steps $A \Rightarrow C$ and $B \Rightarrow C$. In this way, a large proof is quickly decomposed into a large number of smaller subproofs. Intelligence is needed to guide this decomposition, so reasoning common to several subproofs appears only once. Once the proof has been decomposed this far, further levels of decomposition depend on the details of the individual substeps.

This way of structuring proofs and formulas can be used informally as well as formally. A formula can be written informally as nested conjunctions and disjunctions of assertions that combine mathematics and English. Steps in a structured proof can likewise be informal assertions.

3 The EV6 Project

In a shared-memory multiprocessor, processors communicate by reading and writing shared memory locations. Each processor has a cache in which it stores the current value of recently-used memory locations for quick and easy access. A *memory model* defines the relationship between the values written by one processor and read by another—a relationship that must be preserved by a cache-coherence protocol.

The cache-coherence protocol is at the core of a multiprocessor's design, and proving its correctness seemed to be an ideal application for introducing formal methods into product engineering at Compaq.[3] Therefore, when we learned about the Alpha EV6-based multiprocessor, we were eager to use it as a way of bringing our research to bear on engineering problems.

The EV6 cache-coherence protocol[4] is highly optimized and is the most complicated one any of us has ever heard of. Its complexity made the designers welcome our help in checking its correctness.

In modern cache-coherence protocols, processors send messages to one another over a high-speed communication network, moving data from one cache to another and invalidating other copies of data when a processor updates its copy. TLA$^+$ was developed precisely for this type of asynchronous concurrent system, so its choice as our specification language was an obvious one.

We envisioned a verification of the cache-coherence protocol consisting of three parts:

- A specification of the Alpha memory model, which the protocol is supposed to implement.

[3]The EV6 project was undertaken at Digital, which was later acquired by Compaq.

[4]This protocol is for one particular EV6-based multiprocessor, but for brevity, we refer to it simply as the EV6 protocol.

- A specification of an abstract version of the protocol, and a proof that it implements the Alpha memory model.
- A specification of the lower-level protocol, and a proof that it implements the abstract version.

Given the amount of time and manpower available, a complete verification was infeasible. In the following three sections, we describe the parts of these three tasks that we did accomplish.

3.1 The Alpha Memory Model

When we began, there was already a carefully worked out, though somewhat informal, definition of the Alpha memory model [2]. However, it defines the model in terms of the sequences of operations (such as loads, stores, and memory barriers) that are allowed. To write the kind of proof described in Section 2.1, we needed a state-based definition of the memory model. So we defined the model as a TLA$^+$ specification that describes an abstract state machine.

We specified a memory model that is simpler than the actual Alpha memory model in two ways:

- It describes only operations to entire cache lines. Much of the complexity of the complete model arises from interactions among operations of different granularity—for example, if two processors are accessing different parts of a word while a third performs an operation on the entire word.
- It rules out certain types of behavior that are permitted by the general model, but that cannot occur in any currently envisioned protocols. In particular, our specification assumes that all remote accesses to any individual cache line have a common point of synchronization. (It is unclear how one could implement the Alpha model without such a synchronization point.)

The heart of the Alpha memory model is a *Before* relation that orders reads and writes and determines what values must be returned by the reads. The heart of our TLA$^+$ definition of the model is a predicate *GoodExecutionOrder* that specifies the required properties of the *Before* relation. The state machine itself has just four actions:

ReceiveRequest(p, r) Receive a load, store, or memory barrier request r from processor p.

ChooseNewData(p, i) Choose the data that will be returned to processor p in response to its ith request.

Respond(p, i) Send processor p the response to its ith request.

ExtendBefore Nondeterministically add new relations to the current *Before* order. This action makes it easier to verify correctness of an implementation that is more restrictive than the very permissive Alpha memory model.

Each action asserts that *GoodExecutionOrder* is true in the final state of a step (so it is true in all reachable states). In particular, the values chosen by *ChooseNewData* steps must be allowed by the *Before* order.

The TLA$^+$ specification is about 200 lines long.[5] The definition of *GoodExecutionOrder*, the interesting part of the specification, takes about 40 lines. Our specification is shorter than the definition in [2], illustrating the advantage of using a powerful, mathematics-based language like TLA$^+$.

3.2 The Abstract Protocol

As with many real systems, the EV6 cache-coherence protocol can be viewed as an abstract protocol together with complicated implementation details. In the case of the EV6 protocol, the correctness of even the abstract protocol was far from obvious. The combination of a complex protocol and the rather subtle requirements of the Alpha memory model made us feel that the abstract protocol could easily be incorrect. Indeed, we discovered that it allows a behavior that does not satisfy the original Alpha model. It was decided that what we had found was an error in the Alpha memory model, which was subsequently changed in [2].

To prove that a protocol implements our TLA$^+$ specification of the Alpha memory model, we must define the specification's variables in terms of the implementation's variables. (This requires adding an auxiliary variable to record the history of all operations that have ever been issued.) The key specification variable is the one that describes the *Before* order, and the key step in the proof is showing that the order always satisfies *GoodExecutionOrder*.

For the EV6 protocol, it was fairly clear how the *Before* order should be defined. The difficult part in proving *GoodExecutionOrder* is to show that the *Before* order is acyclic. One of the system designers had a good intuitive

[5] The lengths we give for all our formal specifications do not include comments. With comments, most of the specifications are about 2 or 3 times as long.

understanding of why the abstract protocol worked and had sketched an informal proof that the *Before* order is acyclic. We converted that sketch into a rigorous, though not formal, hierarchically structured behavioral proof. The proof consisted of about 100 lines of definitions and 350 lines of proof. The deepest portion of the proof had 8 levels.

Behavioral proofs are notoriously error-prone, so we wanted an invariance proof that the *Before* order is always acyclic. Although one of us has over 20 years of experience constructing invariance proofs of concurrent algorithms, this turned out to be extremely difficult. We produced a 35-line invariant, based on about 300 lines of definitions. The invariance proof was about 550 lines long and had a maximum depth of 10 levels.

The definitions and the invariant were informal. (They were structured formulas whose basic components were assertions in mathematical English.) To make them formal, we would have to write a TLA$^+$ specification of the abstract protocol. We believe that such a specification would be about 500 lines long. It should then be a straightforward exercise to translate our informal invariant into a formal one. The invariance proof should be the largest part of a complete proof that the abstract protocol implements the memory model.

3.3 The Lower-Level Protocol

3.3.1 Writing the Specification

We actually began the project by writing the specification of the lower-level protocol. To do this, we had to overcome two obstacles.

The first obstacle was the difficulty of obtaining a single, coherent description of the system. We received a massive amount of documentation—a pile about 8 cm. tall containing some 20 separate documents. For us, coming into the middle of the design process, it was a confusing array of descriptions of different parts of the system at different levels of abstraction. Often, documents written at different times would use different names for the same object. Fortunately, a few of the documents provided overviews of the system. We also obtained the Lisp code for a simulator written by one of the designers—code that proved to be crucial to understanding some of the system details. None of this documentation provided the complete, precise description of the system that we needed as the basis for a proof, so we had to write our own.

The second obstacle we faced was the sheer complexity of the algorithm. Taking into account the fact that some messages had slightly different semantics in different message queues, there were over 60 kinds of messages that could be flowing through the system. Finding the right abstractions for reasoning about this kind of system is essential, and is still an art requiring skill and experience. In our case, one of the most successful abstractions was to identify a collection of 15 units of functionality that we called *quarks*, and to represent each message in the system as a collection of quarks. In our specification, the individual quarks in a message are processed separately and independently when the message is received. In addition to simplifying the TLA$^+$ specification, this abstraction was helpful enough that the designers started using it, as we discovered on overhearing them discussing the system during a break in a meeting.

The complete specification was about 1900 lines long. Three of us spent about three months writing it, during which time we interacted frequently with the designers by email and telephone. After writing the specification, we went over it with the designers in a two-day meeting, making sure that they agreed that our specification accurately reflected their design. One discrepancy was discovered in that meeting, leading to a change in our specification.

3.3.2 The Low-Level Proof

After writing the specification, we faced the problem of verifying its correctness. It was evident that we did not have enough people and time to write a complete proof. We decided that the most efficient way to find errors was to write an invariance proof—to try to find an invariant of the next-state relation and prove its invariance.

Finding and checking a complete invariant would have been too time-consuming. We wrote about 1000 lines of informal state predicates that formed the major part of a complete invariant. Formalizing those 1000 lines would have been straightforward, but it would have made them about four times as long and we decided that it wasn't worth doing. We selected two conjuncts, each about 150 lines long, as the invariant's most important parts. Each of these conjuncts described all possible sequences of messages that could appear in a particular class of message queues. We felt that if there were an error in the protocol, it was most likely to be found by verifying these two parts of the invariant.

Writing an invariance proof is a tedious process of checking a large number of cases, and continually correcting the invariant when the proof fails. It was probably made easier by TLA$^+$ because we could reason mathematically about the specification itself without first having to generate verification conditions. The conjuncts of the invariant we were verifying were as much as 11 levels deep (alternations of conjuncts and disjuncts). One step of the proof might involve assuming that disjunct a.2.a.2.b.1.b.2.b of the invariant is true in the initial state and then, for each possible action in the next state relation, identifying the disjunct that should hold in the next state, and proving that it is indeed true.

We completed the proof for one of the conjuncts. The proof was about 2000 lines long and 13 levels deep. The proof of the second conjunct would have been about twice as long. However, about halfway through its proof, we decided that we had reached the point of diminishing returns, and the likelihood of finding an additional error was too small to justify further effort. In all, four of us spent a total of about seven man-months on these two proofs.

The result of this effort was the discovery of one bug in the design—an easily-fixed error in one entry of one table. The simplest scenario displaying the bug required four processors, two memory locations, and over 15 messages. We believe that this error could only have been found by writing a proof. The chances of it being found through testing are remote. The low-level protocol is so complex that we believe that any realistic finite-state model that exhibited the error would have too many states to be checked exhaustively by a model checker. (However, it may be possible to construct a small enough model whose sole purpose is to allow a model checker to exhibit this one particular bug.)

3.4 Lessons Learned

Four of us spent more than two man-years on the project. This was not enough to attempt anything close to a complete proof. However, even our incomplete proof subjected the algorithm to a level of rigorous analysis that significantly increased the designer's confidence in their protocol. During a meeting at the end of the project, the designers indicated that they were quite happy with our work. Finding a bug in the Alpha memory model was another accomplishment of the project.

Conventional wisdom, at least in the academic community, is that mathematical proofs usually discover many errors. We were therefore surprised to find only one small error in the design. This was a tribute to the skill of the designers, and a consequence of the fact that we started work on the proof quite late in the design cycle—some two years after the designers had begun work on the system. Apparently, all the easy bugs had already been found.

The one bug in the protocol we did find testifies to the efficacy of writing proofs, since it could almost certainly not have been discovered by simulation or model checking. Moreover, we don't think that any model checker could have discovered the discrepancy between the protocol and the original Alpha memory model. Because it records the history of all issued commands, the memory model has too many reachable states for serious model checking to be feasible. The hierarchical structuring techniques described above in Section 2.2 were crucial to our proof effort; it would have been impossible to manage the complexity of the proofs without them.

Tools are essential, and even simple tools are important. One tool that made our task easier was an Emacs-based system of TAGS files to help find definitions and move around the specification. Another was a collection of Emacs macros for expanding and hiding parts of hierarchically-structured proofs. The use of names like $Inv.a.2.a.2.b.1.b.2.b$ for parts of formulas completely obliterated any intuitive context. A simple Emacs package let us manually highlight the applicable assumptions and the current goal when writing a proof step. With only these primitive tools, we had to develop organizational techniques to cope with the complexity of the proofs. We had to keep track by hand of what had been proved when. (A proof that referred to a step that was later changed needed to be checked.) More sophisticated tools for managing large proofs would have been a big help.

The project was made even more interesting by geographical separation. Two of us were at one site in California; the other two were at two sites in the eastern United States, separated by 50 km. Participants at two or three different sites frequently worked together. For this long-distance collaboration, we used telephones (with headsets) and *shX*—an X-windows utility for running a single application, such as a text editor, on multiple computers. This setup worked quite well, as long as the network provided the necessary bandwidth. In addition to the shared windows, we each had our own private windows for viewing the specification and proof. This put more information at our fingertips than had we been working together in the same office.

4 TLC—The TLA$^+$ Model Checker

TLC is a new on-the-fly model checker for debugging a TLA$^+$ specification by checking invariance properties of a finite-state model. It is unique in that it works on specifications written in a very rich language, rather than in the primitive, low-level languages typical of model checkers.

The design and implementation of TLC was partly motivated by our experience in the EV6 project. When working on the correctness proof, we became convinced that a model checker could be used to catch errors in the design, the specification, and the proof assertions. We considered using some existing model checkers such as SMV [9] and Murφ [3], but their low-level input languages made it awkward, if not impossible, to use them to check TLA$^+$ specifications and proofs.

TLC checks invariance properties of the form $Init \wedge \Box Next \Rightarrow \Box I$. Its input is a TLA$^+$ specification and a configuration file. The configuration file provides the names of the initial condition ($Init$), the next-state relation ($Next$), and the invariance property (I). It also specifies a finite-state model that determines a finite subset of the potentially infinite set of reachable states of the TLA$^+$ specification. The model is specified by instantiating the specification's parameters and giving the name of a constraint. For example, in the TLA$^+$ specification of a cache-coherence protocol, the sets of processors and memory addresses are unspecified parameters, and queues can be unbounded. The configuration file instantiates the sets of processors and addresses with specific finite sets, and a constraint bounds the number of messages in each message queue, making the set of reachable states finite. TLC also requires a type invariant, which is identified by the configuration file.

TLC explores all reachable states in the model, looking for one in which (a) an invariant is not satisfied, (b) deadlock occurs (there is no possible next state), or (c) the type declaration is violated. When TLC detects an error, a minimal-length trace that leads from an initial state to the bad state is reported as part of the error report.

No model checker can handle all the specifications that can be written in a language as expressive as TLA$^+$. TLC can handle a subclass of TLA$^+$ specifications that we believe includes most specifications that describe actual systems. We expect it to be less likely to handle a more abstract, high-level specification, such as the specification of the Alpha memory model described in Section 3.1.

We began work on TLC in February of 1998, almost at the same time that the EV7 verification project was begun. By August of 1998, TLC was working on some toy TLA$^+$ specifications. Implementation and performance tuning continued, and in November of 1998, we tried applying TLC to the TLA$^+$ specification for the EV7 cache-coherence protocol. After two weeks of further debugging, TLC was able to handle the EV7 specification. Performance enhancement and debugging of TLC continue. We hope it will be ready for public release in the summer of 1999.

Our initial experience using TLC led us to add an assertion feature.

When TLC encounters the formula $Assert(P, e)$ while evaluating an expression, it evaluates the formula P. If P is false, TLC prints the value e and reports an error, printing a trace leading to the current state. (The *Assert* formula is defined in TLA$^+$ to equal TRUE.) For example, an assertion can assert that a condition P holds whenever some particular type of step is taken.

One other feature that we have found to be missing is coverage analysis. It is easy for a simple error in the specification to make it impossible for some action ever to be taken—a problem that cannot be detected by checking an invariant. We can verify that an action is taken by conjoining an $Assert(\text{FALSE}, \ldots)$ assertion to it and seeing if TLC finds the "error". However, we prefer not to have to modify the specification in this way. Moreover, adding these assertions is time-consuming and repetitive enough to be a good candidate for automation. We have not yet determined how this can best be done.

5 The EV7 Project

The next generation of Alpha processor after the EV6 is the EV7. The EV7's memory controller is on the chip, so there is a single protocol for all EV7-based computers. Embedding the protocol on the chip makes any error even more costly. The success of the EV6 project made the EV7 designers interested in having formal verification applied to their cache-coherence protocol. Despite reluctance to commit engineering resources, it was decided that the EV7 verification team—the engineers who run simulations and tests of the hardware—would work with the researchers on a project to verify the protocol.

The TLA$^+$ specification of the Alpha memory can be used for the EV7 as well as the EV6. Hence, we had to write a specification only of the cache-coherence protocol itself. Because it had to be implemented entirely on the processor chip, the EV7 protocol is significantly simpler than the EV6 protocol. We have not yet seen any need to verify an abstract version of the protocol, so we have been working only on a lower-level view.

5.1 Writing the Specification

Our goal is to move formal methods from research to engineering. The researchers felt that the engineers should be able to write the TLA$^+$ specification by themselves. It was agreed that the researchers would teach the verification engineers how to write the TLA$^+$ specification, and the formal verification would be based on their specification.

At the time, there was no introductory document on TLA$^+$. To learn to write TLA$^+$ specifications, the engineers received about eight hours of face-to-face instruction, read some example specifications, and had their specifications corrected by email. The current EV7 protocol specification was essentially written by one engineer, who is now comfortable writing TLA$^+$ specifications by himself.

As in the EV6 project, writing the specification required translating several high-level descriptions of the protocol into one formal description in TLA$^+$. The original descriptions were in various formats, including English mixed with tables and a specially formated pseudocode. Translating these descriptions, which contained varying levels of ambiguity, into a precise formal specification took some effort. It required determining the behavior of the protocol in many corner cases not mentioned by the informal descriptions. The engineer writing the specification decided that, in case of uncertainty about exactly what behaviors should be allowed, it was better to adopt a more restrictive approach. He felt that it would be easier at a later time to admit a new behavior than to argue that one that was previously allowed should be forbidden. (This was in contrast with the researchers' approach in writing the EV6 protocol specification, which was to check all such decisions with the designers.)

To convert the informal protocol description to a formal specification, we first had to determine the level of abstraction. A cycle-accurate specification of the protocol would have far too much detail to be tractable. We chose a level that was as abstract as possible, but still exhibited all the orderings of events at the communication ports that could occur in the actual system. Writing the actual TLA$^+$ specification involved the following steps.

Identifying the atomic actions. The atomic actions of the specification are the sending of one or more messages and the receipt of a single message. This assumes that the system handles all messages to a single processor one at a time. Systems in which a processor can receive several messages concurrently are not accurately described by this specification. One would presumably prove that such a system implements the current specification.

Determining the state. Most of the specification's state was modeled after the actual system state, as described by the design documents. However, their values were represented in the TLA$^+$ specification by more abstract data structures. Queues became sequences; an unordered interconnection network became an array of sets, each set containing the messages in transit to a particular processor.

Incrementally describing the protocol. We started by writing a specification that described only the simple case of a request by a processor for an address that is not cached by another processor. We then added other cases, including more complicated ones involving third parties and race conditions, a few at a time.

The current TLA$^+$ specification is about 1800 lines.

5.2 The Verification Effort

5.2.1 Model Checking

When the engineers were taught TLA$^+$, they were told to observe only one restriction for writing a specification that TLC could handle: to use bounded quantification (formulas of the form $\exists\, x \in S : P$) rather than unbounded quantification (formulas of the form $\exists\, x : P$). Guided by just this weak restriction, they wrote a TLA$^+$ specification that TLC could handle; no rewriting was needed to accommodate TLC. To apply TLC to the specification, we just had to provide a configuration file, as explained in Section 4—and fix a few TLC bugs, as expected when using a new program.

To reduce the state space to a tractable size, we use a model having only one cache line, two data values, and two or three processors. With these restrictions, there are 6 to 12 million reachable states, depending on three mode bits that are parameters of the algorithm. A complete TLC run then takes between 3 days and a week on a fast workstation.

The system is composed of a number of identical components. Hence, the state space has a lot of symmetry. Since the processors are identical, processor p sending message m and processor q sending message n is equivalent, from a correctness point of view, to p sending n and q sending m. Lacking any tools for exploiting symmetries, we hand-coded some symmetry-breaking constraints into the specification itself. By adding about 15 lines to the specification, we were able to reduce the state space by about 50%.

The model checker has also been used in one unexpected way. The error traces it produces can be transformed into input stimuli for the verification team's RTL simulator. Errors tend to occur on corner cases that are not found by random testing, and programming directed tests is time-consuming, so these inputs provide useful tests. One design error has been found by running these tests. We are planning to automate the translation from TLC's trace output to simulator input, and to use TLC to generate randomly chosen traces. Such traces should be better than purely random ones, because they satisfy the TLA$^+$ specification.

The RTL simulation also used the TLA$^+$ specification itself in a way we had not anticipated. When specifying the next-state relation with TLA$^+$, one must explicitly assert what variables an action leaves unchanged. These assertions were added as checks to the simulator.

5.2.2 Proof

Because our goal is to move formal methods into engineering, we originally intended to have the verification engineers write an invariance proof. However, a shortage of engineers on the verification team made this impossible. So, the task of writing proofs has fallen once again to the researchers.

This protocol is simpler than the EV6 protocol, so we expect to be able to get closer to a complete proof. However, largely because we have been so successful at finding errors in the specification—especially with TLC—we have not gotten very far yet with the proof. (There's no point trying to prove something that's not correct.) The proof effort has thus far consisted mainly of writing invariants and checking them with TLC. This process has helped us find errors in the specification.

The benefit of using TLC to check invariants comes from how invariants are found. An invariant *Inv* must satisfy $Inv \land Next \Rightarrow Inv'$, where *Next* is the next-state relation. (The other condition that *Inv* must satisfy, that it is implied by the initial predicate, is generally trivial.) In practice, *Inv* is a large conjunction of state predicates, each describing some part of the global state. Finding *Inv* is an iterative process of "guessing" what it should be and successively correcting that guess when the proof of $Inv \land Next \Rightarrow Inv'$ fails. The proof can fail for one of two reasons:

1. The invariant is too weak. It is true in all reachable states, but it is also true in some unreachable ones for which $Inv \land Next$ does not imply Inv'.

2. The invariant is too strong. It is not true of all reachable states, so there is a reachable state for which $Inv \land Next$ does not imply Inv'.

In case 1, we simply strengthen the invariant by conjoining some condition and continue. The proof just gets bigger, because we now have to prove the additional conjunct of Inv'. Case 2 is more problematic. It forces us to weaken the invariant by removing or weakening a conjunct. Any part of the proof that used this conjunct has to be modified accordingly. So, we need to look carefully at everything that has already been proved in order to find and modify all uses of the conjunct. Although made easier by the hierarchical structure of our proofs, this is still a time-consuming task. The model-checker helps avoid this case. By checking it on all reachable states, TLC can discover that the invariant is too strong before we try writing the invariance proof.

5.2.3 Results So Far

As soon as we started using TLC, we found many errors in the TLA$^+$ specification. Initially, most of the errors were typos or simple type errors that were easily caught, often by the parser. Starting from when the specification was free of these simple errors, we found about 66 bugs in successive versions. We can roughly categorize them as follows:

- 37 minor errors—typos, syntax errors, type errors, and other simple mistakes introduced when making changes. This kind of error is generally detected by TLC within a few seconds.
- 11 errors in which the wrong message is sent or the wrong state is set. Some of these errors were hard to understand and took more than one rewrite to correct. Three of them corresponded to one error in the design documents—an inconsistency between the descriptions of two components.
- 14 instances of missing cases, overly restrictive conditions, and assertions (*Assert* formulas) that were too strong. One of these was introduced following a discussion about the protocol, where we mistakenly decided that a case could not occur. Most of the rest were due to the restrictive approach taken when writing the specification.
- 4 miscellaneous errors: one caused by misunderstanding the semantics of TLA$^+$, one in which two different messages were represented by the same element in a set of messages, an incorrect use of an *Assert*, and one apparently useless disjunct of the next-state relation. (We are not yet completely sure that the disjunct is useless.)

About 90% of these errors were found by TLC. The rest were found by a human reading the specification. One of these could not have been found by model checking because it occurs only in complex configurations, for which the state space is too big to be exhaustively explored.

So far, we have found two design errors: the inconsistency, mentioned above, in the design documents and the one mentioned in Section 5.2.1 that was found using an error trace produced by TLC.

5.3 Lessons Learned

Our experience shows that learning TLA$^+$ is not a major task. An engineer can learn to read TLA$^+$ specifications with an hour of instruction. An engineer who is not intimidated by mathematical notation can learn to write TLA$^+$ specifications with just a few hours of instruction. But, writing good specifications does require practice and experience. Even though the EV7 protocol is simpler, the specifications of the EV6 and EV7 protocols are about the same size. To some extent, this is because much of their size reflects the complexity of the interface, not of the underlying protocol. But, the EV6 specification is also more concise because it was written by researchers with a lot of experience writing formal specifications; the EV7 specification was written by an engineer new to formal methods.

Most of the errors found so far in the EV7 protocol's TLA$^+$ specification were introduced when translating the informal documents into a formal specification. The EV7 specification was written by a fairly junior engineer who was somewhat diffident about asking the designers for clarification. Some of these errors could probably have been avoided if he had been more aggressive in this respect. However, the best way to avoid such errors is for the designers, rather than the verification team, to write the TLA$^+$ specification.

Formal specification has been viewed as part of the verification process, not as part of the design process. The verification team proposed that, because it clarified and made precise many details of the system, the TLA$^+$ specification should be made part of the design documentation. The designers rejected this idea because they felt that improving the specification had too low a priority, so they were unwilling to learn how to read a TLA$^+$ specification. However, after they received the first report of errors found with TLC, we were able to convince the designers to take the time to learn how to read a TLA$^+$ specification.

We believe that our experience shows that formal specification should be part of the design process. It should not be postponed until a separate verification phase. Making a formal specification part of the design documentation has three benefits:

- It removes ambiguity and forces the designers to confront all the subtle corner cases.
- It forces the designers to provide a description of the entire system at a single level of abstraction. Informal engineering documents typically contain a melange of ideas ranging from high-level algorithm descriptions to low-level data formats.
- It allows the use of tools such as TLC to find errors early in the design stage, when they are easier to correct.

We hope that, in future projects, the formal specifications will be written by the designers.

6 Conclusion

We have described two projects, both verifications of cache-coherence protocols, but quite different from one another. The EV6 project was our first attempt to verify a complete subsystem of an actual product under development. We would have preferred to cut our teeth on a simpler problem. However, verifying the EV6 protocol presented not just a research challenge, but also an opportunity to do something useful for the company.

The results of the EV6 project were both humbling and encouraging. They were humbling because, contrary to our expectations, we found no serious errors—just a single low-level bug. They were encouraging because the basic verification method, refined through years of research, worked pretty much as expected. Writing the proofs was somewhat harder than anticipated, and we learned some new things about managing very large proofs. But there were no big surprises. We were also encouraged because the system's designers found what we did to be valuable.

The results so far of the EV7 project have been very satisfying. We have shown that engineers can write TLA$^+$ specifications. The TLC model checker has performed beyond our expectations. After only preliminary debugging, it was able to handle an unmodified, industrial-sized TLA$^+$ specification. TLC has already helped find two design errors. We have taken a large step towards introducing formal verification as a part of system development at Compaq. There are three reasons why the EV7 project has been more successful than the EV6 project: we started earlier in the design phase, we have been able to use TLC, and the protocol is simpler.

An important lesson to be learned from our experience is the need for patience and perseverance in moving formal methods from the research community to the engineering community. Engineers are under severe constraints when designing a system. Speed is of the essence, and they never have as many people as they can use. In this situation, it is reasonable to do what has worked before, and not to risk a project by using untried methods. Before introducing formal methods into the process, engineers must be convinced that they will help.

A fortuitous set of circumstances led to the EV6 project, one done completely by researchers with the cooperation of the engineers. The success of that project made the engineers willing to commit people to the EV7 project. The results of these two projects have made other engineers within Compaq interested in applying the methods, and we hope TLA$^+$ will play a role in other projects in the near future.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.

[3] David L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–393, 1996.

[4] Peter Ladkin, Leslie Lamport, Bryan Olivier, and Denis Roegel. Lazy caching in TLA. To appear in *Distributed Computing*.

[5] Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web. It can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

[6] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994. First appeared as Research Report 119, Digital Equipment Corporation, Systems Research Center.

[7] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[8] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.

[9] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[10] K. L. McMillan and J. Schwalbe. Formal verification of the encore gigamax cache consistency protocols. In *International Symposium on Shared Memory Multiprocessors*, Apr 1991.

[11] Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, July/August 1998.

[12] M. Plakal, D. Sorin, A. Condon, and M. Hill. Lamport clocks: Verifying a directory cache coherence protocol. To appear in *Symposium on Parallel Algorithms and Architectures.*, June 1998.

[13] Ulrich Stern and David L. Dill. Verifying the SCI cache coherence protocol. In *2nd International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 75–82, 1995.