# Formal Verification of the HAL S1 System Cache Coherence Protocol


Alan J. Hu*
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
ajh@cs.ubc.ca

Masahiro Fujita
Fujitsu Laboratories of America
3350 Scott Blvd., Bldg. #34
Santa Clara, CA 95054-3104
fujita@fla.fujitsu.com

Chris Wilson
HAL Computer Systems
1315 Dell Avenue
Campbell, CA 95008
chrisw@hal.com



## Abstract

*This paper describes our experience applying formal verification to the cache coherence protocol of the HAL S1 System, a shared-memory and/or message-passing multiprocessor consisting of standard Intel Pentium® Pro symmetric multiprocessing (SMP) servers connected by HAL's proprietary Mercury Interconnect to create a cache-coherent, non-uniform memory access (CC-NUMA) machine. In recent years, several researchers have described the verification of cache coherence protocols to demonstrate the potential of formal verification. In this project, we sought to quantify this potential by carefully tracking the effort and results of applying formal verification, rather than simply demonstrating that verification was possible. Based on our records and experience, we show that protocol-level formal verification, properly applied, is sufficiently well-understood to be routinely undertaken, and we describe the techniques used to simplify the verification process. On the negative side, our formal verification methodology has limitations, so we outline the pitfalls we encountered and recommend ways to minimize them.*


## 1. Introduction

Formal design verification is attracting increasing interest as a tool to deal with the ever increasing cost and complexity of hardware designs and protocols. One application area that has been particularly promising is the use of automatic verification methods (finite state exploration and model checking) to debug multiprocessor cache coherence protocols. This is an attractive problem for formal verification for several reasons:

- The problem domain is important. Cache coherent multiprocessors are an increasingly common architecture for high-performance servers.
- Multiprocessor cache coherence protocols are hard to design and debug because they are intrinsically highly concurrent. Simulation is likely to miss important corner cases, and replicating and understanding any bugs that arise is extremely difficult.
- The protocols can be modeled with finite state, making them amenable to automatic formal verification.
- Several researchers have successfully used automatic formal verification to debug complicated, real cache coherence protocols (e.g., [13, 7, 4, 10, 15, 17, 3, 14]).

Unfortunately, in addition to the growing body of verification-of-cache-coherence-protocol success literature is also a growing oral tradition of verification failure stories. In the hallways at conferences or via private email, verification researchers informally trade stories of failed attempts at using formal verification. Clearly, there are pitfalls as well as promise, and the decision to try using formal verification can be difficult to justify based solely on a collection of "It worked." and "It didn't work." anecdotes.

The justification for using formal verification on cache coherence protocols is essentially economic, with failure and success defined by comparing the effort expended on formal verification versus the confidence gained in the design. The role of formal verification has been redefined from the traditional attempt to certify correctness to a more practical, less absolute goal of providing powerful debugging support to catch hard-to-find bugs. As cited above, previous research has clearly established that formal verification

*This work was done while the first author was a member of the research staff at Fujitsu Laboratories of America.

can help catch hard-to-find bugs. What's missing is the other side of the economic equation: How much effort is needed to use formal verification effectively? What kind of results can be expected for a given investment in time? Are there common rules-of-thumb and modeling techniques that reduce the effort required? In this paper, we address these questions in the context of a specific example, verifying the cache coherence protocol of the HAL S1 System. In short, this paper is not just another report of an industrial cache coherence protocol successfully verified; this paper is a report of the time, effort, and procedure required for protocol-level verification of a representative cache coherence protocol.

## 2. The HAL S1 System

The HAL S1 System is a multiprocessing computer that supports both shared-memory and message-passing paradigms. It consists of symmetric multiprocessing (SMP) nodes connected by HAL's Mercury Interconnect. Each node is a standard SMP set-up containing four Intel Pentium® Pro processors and up to four gigabytes of dynamic RAM. The interconnect is based on two chips: the Mesh Coherence Unit (MCU) and the Router. The MCU is the network interface for each node, translating bus transactions within each node to network messages and vice-versa. Zero or more Routers can be cascaded in arbitrary topologies to form the network fabric. The interconnect has low latency (42ns fall-through per Router) and high bandwidth (up to 9.6GBytes/s per Router). For a system with 4 nodes (16 processors), cache-coherent memory access time to a remote node is only about four to five times slower than access to memory local to the SMP node. Details on the S1 System and Mercury Interconnect are available elsewhere [16]; Figure 1 sketches the architecture of the system.

In this paper, we are only concerned with the cache coherence protocol of the S1 System. Within each SMP node, the normal bus-based snooping protocol keeps the processor caches coherent. We will assume the snooping protocol works correctly. A directory-based cache coherence protocol maintains coherence between nodes. This protocol is the target of our verification effort.

The cache coherence protocol is loosely based on the Stanford DASH multiprocessor [11]. Main memory is physically distributed among the nodes, but each processor views all memory as a global, flat address space. To achieve good performance, processors cache the memory lines they access, whether the memory is located locally in the same node as the processor or in some other node. The cache coherence protocol must deliver data to processors that request it and ensure that all cached copies of a memory line are coherent, producing a cache-coherent, non-uniform memory access (CC-NUMA) machine. The cache coherence protocol maintains a directory that indicates which nodes have cached copies of which memory lines. This directory is physically distributed among the nodes, just as main memory is.

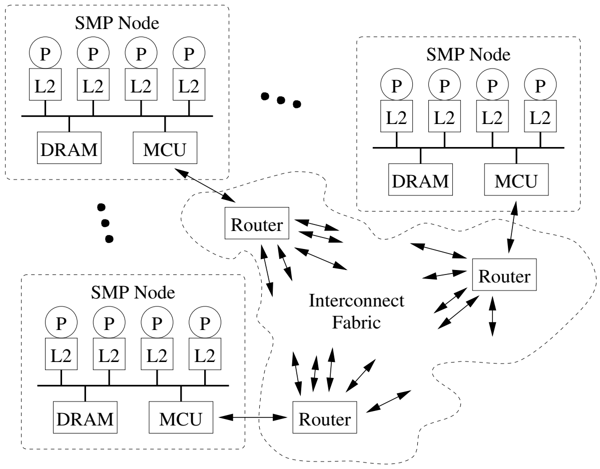


**Figure 1. Architecture of the HAL S1 System. Each node is a standard SMP configuration of Intel Pentium® Pro processors and local DRAM. The Mesh Coherence Unit (MCU) acts as a bridge between the internal bus of the SMP node and the network fabric. The current implementation supports up to 32 nodes (128 processors).**

For example, suppose a processor in node $A$ needs to read a memory location that is physically in node $B$. Node $A$ will send a message through the interconnect to node $B$, requesting the memory line. If the memory line is available, node $B$ will send a reply containing the requested memory line back to node $A$ and update its directory to indicate that node $A$ has the line cached. On the other hand, suppose the memory line requested by $A$ is actually cached dirty at another node $C$. In this case, node $B$ knows from its directory that node $C$ has the line, so it sends a message to node $C$ asking for the memory line back or for $C$ to forward the line to $A$. As before, $B$ must update its directory accordingly.

The scenario becomes more complex when we consider that multiple transactions on the same memory line can be initiated simultaneously from different processors. The protocol must prevent race conditions and different transactions interfering with each other. In general, designing directory-based cache coherence protocols is a subtle art, making it an attractive target for formal verification.

## 3. Formal Verification Methodology

Few resources were committed to the formal verification project: basically, one person part-time and some cycles on a workstation with a large amount of RAM. In our case, this level of commitment resulted from the confluence of a verification researcher looking to spend some of his time working on a real design as a case study and a design team with a complicated protocol that they felt would benefit from formal verification. In general, we expect a similarly low level of commitment to formal verification to be common in practice. For example, an architect, design engineer, or verification engineer might experiment with formal verification as just one of many responsibilities and as just one of many techniques to debug a complicated design.

Given this minimal level of commitment, we chose to try formal verification at the protocol-level — verifying the cache coherence protocol itself at a very high level, rather than verifying implementations of the protocol [7]. This choice had several positive consequences:

- Protocol-level verification seemed feasible for one person working part-time.
- It kept formal verification off the critical path of the project. The design and verification engineers were already busy implementing and simulating RTL-level code, and we did not want to impede their progress.
- The formal verification was still useful. The protocol was sufficiently complex that the designers wanted further assurance of its correctness, and if formal verification discovered any bugs in the protocol, the cost savings would be enormous, since the bugs could be fixed before tapeout.

On the negative side, the same decoupling from the regular design flow that makes protocol-level formal verification attractive also renders the results less relevant to the real design — all of the bugs that occur while implementing the design are not modeled and hence cannot be caught by protocol-level verification. Nevertheless, the fact that protocol-level verification of cache coherence protocols can be very valuable has been made repeatedly [13, 7, 4, 10, 15, 17, 3, 14].

Another negative is that our methodology in applying protocol-level formal verification was far from ideal [7]:

- Ideally, protocol-level formal verification is done extremely early in the design cycle. In our case, we started protocol-level verification as RTL coding was nearing completion.
- Ideally, the designers/architects themselves use formal verification to explore preliminary ideas. In our case, an outsider came in to check a completed, mature design.
- Ideally, protocol-level formal verification catches bugs before much effort is spent on the design. In our case, any bugs found would require substantial effort to fix.

Unfortunately, we expect our scenario to be a common one in practice, so our experience with this less-than-ideal methodology is likely to be repeated by many others.

A final issue is the choice of verification system to use. The two verification systems that have the most published success stories on real cache coherence protocols are SMV [12, 5] and Murphi [7, 6]. Both would be appropriate to our problem.[1] SMV has two main advantages over Murphi: the use of BDDs potentially allows much larger state spaces to be explored, and the specification logic CTL allows the verification of more complex behaviors. Murphi, on the other hand, is praised for its ease-of-use (e.g., [17]) while still being powerful enough to handle practical problems. Furthermore, the potential of BDDs frequently doesn't materialize in practice; BDDs are notoriously unpredictable for protocol-level verification (e.g., [9]). Because of our limited time and resources, we chose Murphi for simplicity. We will discuss some of the limitations resulting from this choice in Section 6.

## 4. Modeling Directory-Based Cache Coherence

In their original paper on Murphi, Dill *et al.* noted:

> Verification goes through several stages: deciding how to model the problem (especially, what details to omit); writing the description; using verification to find description errors; and only then discovering genuine design errors. [7]

In this section, we will discuss how to model directory-based cache coherence and how to write the Murphi description.

The most salient feature of our modeling process is how much we were able to reuse the results of previous work on directory-based cache coherence. The difficult modeling decisions had been made already. The extent of our reuse of previous work saved considerable effort; we will discuss this point further in Section 5. The key insights for successful modeling were drawn from a several sources. We present the most important points below.

The original Murphi paper suggests several guidelines for protocol-level verification of hardware. The paper recommends that one should model the system at a very high level, design the model to be scalable (especially to verify scaled-down versions of the system), and insert in-line error statements that flag every unused branch of `if` or `case` statements. Additionally, the paper suggests using an array of messages to model a network and gives general properties to be verified for cache coherence (mutual exclusion on dirty data, coherence on shared data). All of these ideas are necessary for successful verification with Murphi.

[1] Of the cited cache coherence verification success stories, [13, 4, 3] use SMV, and [7, 10, 15, 17, 14] use Murphi.

The system we verified has a hierarchical cache coherence protocol, a feature common to many cutting-edge designs. A simpler (usually bus-based snooping) cache coherence protocol is used within nodes of a few processors each, and a more complex protocol maintains cache coherence between nodes. Successful formal verification projects typically abstract away the internals of each node, decoupling the verification of the two cache coherence protocols (and generally ignoring the protocol within the node). This decoupling unfortunately prevents the formal verification model from exploring the interaction between the two protocols, but is presumably necessary to keep the complexity of formal verification manageable.

Abstracting the nodes even further creates two types of abstract nodes in the model. In one type of abstract node, we model the memory and directory, but no processor caches. In the other type of abstract node, we model a processor cache (trusting the intra-node protocol to keep the processors within the node coherent) but no memory or directory. Using these two types of abstract nodes generates a much smaller state space. Any bug that could have occurred had we modeled each node with both cache and memory can still occur in the restricted model, since the intra-node protocol is assumed to prevent problems between the cache and memory within a single node. The current Murphi distribution includes several examples of directory-based cache coherence protocols that demonstrate this modeling technique.[2]

Another modeling technique taken from the examples distributed with Murphi is to model the network not as a single array of messages, but as a separate array for messages sent between each pair of nodes. Doing so greatly reduces the number of reachable states of the system, because the verifier does not consider the different interleavings of messages between unrelated nodes to be distinct states.

Stern and Dill [15] recommend resetting to fixed values all state variables whose values are no longer needed, citing a factor of twenty reduction in reachable states for their example. Despite casual awareness of this idea, we discovered on a careful examination of our model that we violated this guideline in a few places. Fixing those places in the code resulted in an additional factor of ten reduction in reachable states. We recommend very strict adherence to this practice.

The Murphi system provides semi-automatic symmetry reduction: if the user identifies symmetric structures by using special data types, the verifier automatically reduces the state space with respect to that symmetry [10]. Although the verifier runs slower using symmetry, the reduction in reachable states more than compensates for the speed difference, and because of space constraints, symmetry reduction is absolutely necessary for successful verification. Symmetries in a model of directory-based cache coherence include the addresses, the data values, and each type of abstract node. One problem we encountered is that the cache coherence protocol takes advantage of the Mercury Interconnect's ability to guarantee in-order message delivery. This feature simplifies the protocol design, but is a major cause of blow-up in the number of reachable states during verification, because different orders of messages in the queue are counted as different states. In a network that does not preserve message order, the Murphi `Multiset` data type can be used to provide automatic symmetry reduction on the different orderings of messages.

Finally, one modeling technique that is frequently helpful and recommended for verifying communication protocols [7] was not useful for us at all. The idea is to force all data in a system to have one particular value, say 0, and then non-deterministically generate exactly one copy of a distinct value, say 1. In a link-level protocol, this technique allows us to spot any duplicate or dropped packets, because the verifier explores all possible executions, so the duplicated or dropped packet can be the packet with the 1. Using this technique greatly reduces the number of reachable states. In a cache coherence protocol, however, the special value can be copied and shared throughout the system, so we did not achieve a reduction in the set of reachable states.

## 5. Results

Prevailing wisdom on protocol-level formal verification suggests that the following milestones occur in roughly this order:

1. Obvious Documentation/Specification Errors — Formal verification requires rereading the documentation from a fresh perspective, and many errors become immediately apparent.
2. Bugs in the Formal Model — Writing the formal model for verification is much like writing a small program. Simple coding errors (unrelated to bugs in the real design being verified) creep in and require a bit of debugging.
3. Subtle Documentation/Specification Errors — Once the formal model is reasonably debugged, the verifier will flag many "errors" in the design. At first, these are the result of misreading the design documentation, but at some point, the errors in the formal model result from ambiguous or erroneous documentation.
4. Bugs in the Real Design That Were Fixed Already — Since other debugging efforts are underway simultaneously with formal verification, and since incorporating design changes into the formal model takes some time, the first real bugs discovered by formal verification are likely to be bugs that were discovered already by other methods.

[2] The current distribution of the Murphi system is available from `http://verify.stanford.edu/dill/murphi.html`.

5. Newly Discovered Bugs in the Real Design — This is the payoff. Every bug caught earlier by formal verification saves considerable time, effort, and money that would have been spent fixing the bug later.

Our experiences matched this pattern closely.

We reached Milestone 1 in the second week of the project, after less than ten hours in total (See Table 1 below.) had been spent on formal verification. An example of the obvious errors identified is that the documentation specified the wrong data structure to update in a certain situation.

We reached Milestone 3 in the seventh week of the project. An example of a subtle specification error discovered is that a certain state machine had eight possible states, named for example $a0$, $a1$, $a2$, $a3$, $b0$, $b1$, $b2$, and $c$. The protocol specification indicated a certain action if the state machine was in state "$ax$", but the correct specification was that the action be taken if the state machine was in any state other than $c$. This error is perhaps more intuitive if we explain that state $c$ was a distinguished state, and that the $a$ states and the $b$ states were conceptually similar.

We reached Milestone 4 in the tenth week of the project. Over the course of the project, protocol-level formal verification discovered only one real bug in the protocol: a complex interaction among various subunits of the MCU chip would result in incorrect behavior when a node with dirty data wrote back the data, but the message crossed with a request for that node to forward the data to another node. This bug, however, had been caught and fixed five weeks earlier, but the person performing the verification had been working from documentation just over five weeks old.

We did not find any bugs with formal verification that had not been caught by other methods. This failure was due in part to the limitations of our methodology (discussed more in Section 6), but was also due to the maturity and stability of the protocol. Indeed, the cache coherence protocol was essentially the second implementation of a protocol that had been extensively scrutinized (including some formal verification) previously. Most of the bugs, therefore, were at the implementation level, which protocol-level verification cannot discover.

In any case, failure to discover new bugs does not necessarily imply failure of the project. An analogy to insurance is perhaps appropriate. If the premium is sufficiently low, insurance is worth buying — even if the coverage is poor — because in the event of a disaster that is covered, it will mitigate the damage suffered. Similarly, if the cost of formal verification is sufficiently low, verification is worth doing because in the event of a serious bug that is discovered, it will mitigate the damage suffered. In either case, however, not collecting on the insurance is the best situation.

How much did the formal verification cost? Throughout the project, time spent on formal verification was carefully recorded. Table 1 shows a week-by-week breakdown of cumulative time spent.

The most striking feature of the data in Table 1 is that almost no time was spent deciding how to model the problem. This situation resulted mainly from our ability to reuse the results of previous work on verifying directory-based cache coherence protocols, as described in Section 4. Apparently, enough work has been done on protocol-level verification of cache coherence protocols that the problem domain is well-understood and new projects can be done with minimal effort. Furthermore, reusing modeling decisions from previous, successful verification projects minimizes the likelihood of state-space explosion resulting from bad modeling decisions.

A substantial amount of time was spent writing Murphi code. This is an area that could benefit greatly from increased automation. Most of the coding is routine and repetitive. For example, in most Murphi cache coherence verification examples, a large `switch` statement handles the messages from the network, with a different case for each possible message type. The code for each case involves looking up the correct action in the protocol specification. If the golden protocol specification is machine-readable (in our case, they were machine-readable tables), it should be possible in principle to write these case statements automatically. Furthermore, automatic generation of Murphi code would allow revisions to the protocol to be reflected immediately in the verification model, bypassing the "Bugs That Were Fixed Already" milestone. A different approach to this problem would be to use the formal verification model as the golden protocol specification.

Later in the project, a great deal of time was spent waiting for the verifier to run. Obviously, a faster machine or a faster verifier would improve this problem. However, note that all of the interesting feedback from the verifier came before lengthy runtimes were required. These long verification runs might be worthwhile only if there are plenty of machines sitting idle.

Finally, a large fraction of time was spent in meetings. While time devoted to meetings is generally the target of derision,[3] this time is actually crucial to the success of the verification project. Almost all of the time logged for meetings was from the regular status meetings of the group. Attending these meetings is necessary to keep the people doing formal verification up-to-date on the progress and any changes in the design. If a person is responsible for formal verification as well as other aspects of the project, he or she would be attending these meetings anyway, so almost no additional meeting time would be required to support the formal verification.

[3] In the words of Scott Adams, "If you're new to the business world, you might mistakenly think that meetings are boring, sadistic hell, populated by galactic-level morons." [1, p. 220]

| Week | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Overhead | 2 | 2 | 2.75 | 2.75 | 2.75 | 4.75 | 4.75 | 4.75 | 4.75 | 4.75 | 4.75 | 4.75 | 4.75 |
| Meetings | 1.25 | 6.75 | 8.75 | 9.25 | 9.75 | 11 | 11 | 12 | 16 | 18.75 | 18.75 | 19.75 | 19.75 |
| Study | 0 | 3 | 4.75 | 5.5 | 8 | 8.25 | 8.25 | 8.25 | 8.25 | 8.25 | 8.25 | 8.25 | 8.25 |
| Model | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.5 | 0.5 | 0.5 | 0.5 | 1 |
| Code | 0 | 0 | 0.5 | 1.75 | 3.25 | 4.25 | 8.25 | 13 | 15.75 | 20 | 22 | 22 | 22 |
| Run | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3.25 | 14 | 19.75 |
| Total | 3.25 | 11.75 | 16.75 | 19.25 | 23.75 | 28.25 | 32.25 | 38 | 45.25 | 52.25 | 57.5 | 69.25 | 75.5 |

| Week | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Overhead | 7.75 | 7.75 | 7.75 | 7.75 | 7.75 | 7.75 | 7.75 | 7.75 | 7.75 | 10.25 | 10.25 |
| Meetings | 26.25 | 26.75 | 26.75 | 28 | 28 | 28.75 | 29.5 | 29.5 | 29.5 | 29.5 | 29.5 |
| Study | 8.25 | 8.25 | 8.25 | 8.25 | 8.75 | 8.75 | 8.75 | 8.75 | 8.75 | 8.75 | 8.75 |
| Model | 1 | 1 | 1.75 | 2.75 | 2.75 | 3.25 | 4 | 4 | 4 | 4 | 4 |
| Code | 24.5 | 24.5 | 25.5 | 26.5 | 29.25 | 29.75 | 29.75 | 29.75 | 29.75 | 29.75 | 29.75 |
| Run | 19.75 | 22.25 | 23.25 | 27 | 29 | 34.25 | 36 | 36.5 | 36.75 | 36.75 | 38.25 |
| Total | 87.5 | 90.5 | 93.25 | 100.25 | 105.5 | 112.5 | 115.75 | 116.25 | 116.5 | 119 | 120.5 |

**Table 1. Verification Effort over Time. This table shows the cumulative amount of time spent on formal verification up to the $n$th week of the project. Times are in hours, recorded in 15 minute increments. "Overhead" indicates time spent on general overhead, such as setting up computers or installing software. "Meetings" indicates time spent in meetings; the time was multiplied by the number of people present if the meeting was for the formal verification project. "Study" indicates time spent studying the design. "Model" indicates time spent deciding how to model the design in Murphi. "Code" indicates time spent actually coding the Murphi model. "Run" indicates time spent waiting for verification jobs to run; it does not include the time taken by jobs left unattended. Most interesting verification traces and possible bugs occurred around the tenth week, as the basic model was completed and before lengthy runs were required. After the twentieth week, the project started winding down, with very large verification jobs left to run unsupervised on idle machines.**

## 6. Pitfalls

The fundamental pitfall we encountered was the gap between protocol-level formal verification and the overall verification effort. Many bugs were found and corrected, even after the formal verification was completed. Obviously, some bugs were implementation errors that we cannot expect protocol-level verification to catch. Other bugs occurred in parts of the system that we did not verify (e.g., the snooping protocol within each node). Most disturbing were bugs that were intuitively protocol bugs, but the formal model had abstracted away too much detail to find them. For example, one bug occurred as the result of an error in a table look-up; the Murphi model represented this table in a fully-associative way, which hid the bug. Clearly, protocol-level formal verification is not a cure-all for the overall verification effort, but better connection between the formal and conventional verification efforts would be valuable.

The most obvious way to help bridge the gap between protocol-level formal verification and the overall verification effort is simply to acknowledge that the gap exists. Thus, the best time for protocol-level formal verification is extremely early in the design cycle, when we focus on finding protocol-level bugs. Later in the design cycle, we emphasize that protocol-level formal verification is no substitute for conventional simulation and emulation.

Another approach to bridging the gap is to use formal verification at a lower level or to write more detailed formal models that are closer to the actual implementation. This approach quickly exceeds the limitations of explicit-state verification tools like Murphi. With a BDD-based verifier, much larger state spaces are potentially within reach. Eiríksson [2] reports using SMV to verify a cache coherence protocol and then using refinement checking to extend the verification result down to RTL. We believe this is a promising approach.

In a different vein, Ho *et al.* [8] describe a method to use a very high-level formal verification model to generate good test vectors to test all transitions of the control logic during simulation. Their application was for processor verification, but the method holds potential for other applications as well.

## 7. Conclusion

Overall, we have shown that protocol-level formal verification of directory-based cache coherence protocols has become sufficiently well-understood to be reasonably straightforward. The protocol we verified was of representative complexity, and the time and resources required were demonstrably small. Automatic translation of the protocol specification seems feasible and would further reduce the time required to use formal verification. Similar verification projects should be easy to justify in the future: the cost is minimal, and the potential value is good.

The main weakness to protocol-level formal verification is the lack of connection between the formal verification and the overall validation effort. Not finding bugs at the protocol level does not imply an error-free implementation. Nevertheless, finding any bug faster via formal verification than by other methods is an enormous benefit, making formal verification worthwhile if it's sufficiently easy or sufficiently likely to find difficult bugs. In the future, superior verification tools that allow verifying models much closer to the actual hardware, or new verification techniques that use the information gleaned from higher-level verification to help test lower-level implementations will greatly enhance the value of formal verification.


## Acknowledgments

We would like to thank the MCU team at HAL Computer Systems for putting up with this verification experiment despite very tight deadlines. Special thanks to Frank Chen, Hungwen Li, and Wolf-Dietrich Weber for their indispensible help and support throughout the project.


## References


[1] S. Adams. *The Dilbert Principle: A Cubicle's Eye View of Bosses, Meetings, Management Fads, and Other Workplace Afflictions*. HarperBusiness, 1996.

[2] Ásgeir Th. Eiríksson. Integrating formal verification methods with a conventional project design flow. In *33rd Design Automation Conference*, pages 666–671. ACM/IEEE, 1996.

[3] Ásgeir Th. Eiríksson and K. L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In P. Wolper, editor, *Computer-Aided Verification: Seventh International Conference*, pages 367–380. Springer-Verlag, July 1995. Lecture Notes in Computer Science Number 939.

[4] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K. McMillan, and L. Ness. Verification of the Futurebus+ cache coherence protocol. Technical Report CMU-CS-92-206, Carnegie Mellon University, October 1992.

[5] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification: Eighth International Conference*, pages 419–422. Springer-Verlag, July 1996. Lecture Notes in Computer Science Number 1102.

[6] D. L. Dill. The Mur$\varphi$ verification system. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification: Eighth International Conference*, pages 390–393. Springer-Verlag, July 1996. Lecture Notes in Computer Science Number 1102.

[7] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*. IEEE, October 1992.

[8] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *International Symposium on Computer Architecture*, 1995.

[9] A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282. ACM/IEEE, 1994.

[10] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *International Conference on Computer Design*, pages 230–234. IEEE, October 1993.

[11] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[12] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie-Mellon University, May 1992. Published as CMU Tech Report CMU-CS-92-131.

[13] K. L. McMillan and J. Schwalbe. Formal verification of the Gigamax cache-consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[14] F. Pong, A. Nowatzyk, G. Aybay, and M. Dubois. Verifying distributed directory-based cache coherence protocols: S3.mp, a case study. In *International Conference on Parallel Processing, EuroPar '95*, August 1995.

[15] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods, CHARME '95*, pages 21–34. IFIP WG 10.5 Advanced Research Working Conference, 1995.

[16] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. The Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers. In *International Symposium on Computer Architecture*, 1997.

[17] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC-I. In *32nd Design Automation Conference*, pages 7–12. ACM/IEEE, 1995.