Automatic Verification of the SCI Cache Coherence Protocol[*]

Ulrich Stern[**] and David L. Dill

Department of Computer Science, Stanford University, Stanford, CA 94305

{uli@rutabaga, dill@cs}.stanford.edu

Abstract. This paper describes an ongoing effort to verify the cache coherence protocol of the IEEE/ANSI Standard for Scalable Coherent Interface using the Murφ verification system. A model of the typical set protocol was constructed in the Murφ description language. This model was augmented with a specification of properties necessary for cache coherence. The Murφ verification system automatically checks if all reachable states in the model satisfy the given specification. Although verification is still under way, we have already found several errors in the C-code defining the protocol. Finally, we elucidate the experiences gained in the verification project.

1 Introduction

The IEEE/ANSI Standard for Scalable Coherent Interface (SCI) includes a cache coherence protocol for distributed shared-memory multiprocessors. Designing a complex protocol – like this cache coherence protocol – is a challenging and difficult task. It is very hard for a designer to predict all possible interactions among the distributed system components. One way a computer can support the designer is by means of *simulating* random executions of the system. However, especially in complex systems, there is a high probability of missing executions containing errors using this simulation approach. Conversely, an automatic *verifier* tries to examine all states reachable from a given start state. The biggest obstacle in this exhaustive approach is the often unmanageably huge number of reachable states – the "state explosion problem".

We are currently using the Murφ verification system developed at Stanford to find errors in the SCI cache coherence protocol. In prior work, the Murφ system was successfully applied to several industrial protocols [2, 3, 9, 14]. For verifying the SCI cache coherence protocol, the typical set protocol was modeled with the Murφ description language. This model was augmented with a *specification* of

[*] A preliminary version of this paper was presented at the 2nd International Workshop on SCI-based High-Performance Low-Cost Computing, Santa Clara University, 1995.

[**] Ulrich Stern was supported during this research by a scholarship from the German Academic Exchange Service (DAAD-Doktorandenstipendium HSP-II).

necessary conditions for cache coherence. Together, the model and the specification form the input file for the verifier and consist of 3700 lines of Murφ code (not counting comments). The Murφ verification system automatically checks by explicit state enumeration if all reachable states in the model satisfy the given specification (*model checking*). Although verification is still under way, we have already found several errors in the C-code defining the protocol. Various of these errors affect both the typical set protocol and the full set protocol.

To alleviate the state explosion problem, the model was made *scalable*. By simply changing constant declarations, one is able to change the size of the model – and with that the number of reachable states. Since verification is usually possible only for "down-scaled" models, one cannot guarantee design correctness. Thus, we consider formal verification only as a debugging tool. However, since verification is completed for the down-scaled model of the system, it is likely to catch errors that are missed during simulation.

We tried to make the Murφ description of the model similar to the SCI C-code to prevent incurring additional errors in the translation process. At the same time, however, we had to abstract away from low-level details of the protocol that are not important for the verification. The resulting description should be easy to understand for someone familiar with the C-code. Furthermore, we tried to make it easy to add new features of the SCI cache coherence protocol to the current Murφ description.

The C-code includes a multi-threaded execution environment, which incurred many implicit state variables of the SCI protocol. These variables now occur explicitly in our Murφ description, which should help in implementations of the protocol. The Murφ system contains a simulator as well, allowing to run executions without the need to construct a multi-threaded execution environment.

The Scalable Coherent Interface is specified in the IEEE Standard 1596–1992 [7]. An easy-to-read introduction to the SCI cache coherence protocol can be found in [11]. An overview of the SCI and related standards projects is given in [6]. Previous work on formally verifying the SCI cache coherence protocol was done by Gjessing et al. [4, 5]. However, they did not use automatic methods and did not report finding any errors.

The paper is organized as follows. Sections 2 and 3 present an overview of the Murφ verification system and the SCI cache coherence protocol, respectively. The modeling of the SCI cache coherence protocol is described in Sect. 4, while the specification of cache coherence properties can be found in Sect. 5. In the Sect. 6, we report on some of the errors we found in the protocol and how they were fixed. The experience gained during the course of the verification project is elucidated in Sect. 7. Finally, Sect. 8 contains some suggestions for future work.

2 The Murφ Verification System

The Murφ language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. Its unique features not found in a "typical" high-level language can be described as follows:

- The *state* of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to the global variables. The transition from one state to another is performed by *rules*. Each rule has a Boolean condition and an action. The action may be executed if the condition is true (i.e. the rule is enabled) and may change global variables. An action is a program segment that is executed atomically. In any state, there can be more than one enabled rule (*nondeterminism*).

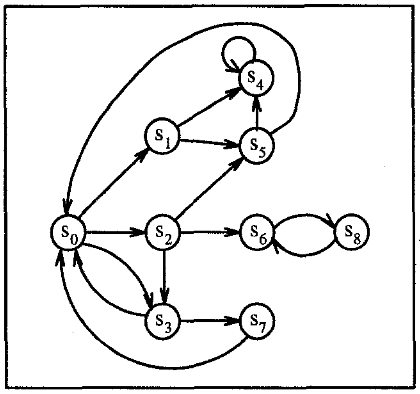

Fig. 1. Sample state graph

Figure 1 shows a simple sample state graph with nine states $(s_0, \ldots, s_8)$. The outgoing arcs in each state correspond to the rules that are enabled in that state. While a simulator chooses an outgoing arc at random, a verifier explores all reachable states from a given startstate (s_0). For the verification, either breadth-first or depth-first search can be selected in Murφ. Reached states are stored in a hash table to avoid double work when a state is revisited. In our SCI model, for example, each processor has the nondeterministic[3] choice of Load, Store, Delete or Flush for its next instruction.

- The *parallel composition* of two processes in Murφ is done by simply using the union of the rules of the two processes as rules for the composition. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of *asynchronous, interleaving* concurrency. Parallel processes communicate via shared variables. There are no special language constructs for communication.
- The Murφ language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. This down-scaling capability is important to reduce the number of reachable states and thus make verification feasible. In many cases, the errors in a

[3] A nondeterministic choice will also be called an *arbitrary* choice in the following.

system are also found in the down-scaled system. For example, in our SCI model the number of processors is scalable and defined by a constant.

- The Murφ verifier supports automatic *symmetry* reduction of models by special language constructs [8, 9]. For example, if we have two processors, the state where processor one is the head and two is the tail of a sharing list is – for verification purposes – the same as the state where processor one is the tail and two is the head.
- There are several ways the Murφ verifier detects design errors. First, the description is checked for *deadlocks*. Second, there is an *assert* statement, which causes the verifier to print an error trace if the assertion condition is violated. An error trace is a sequence of states from the start state to a state exhibiting the problem. Besides the assert statement, Murφ has an error statement that always prints an error trace. In the SCI model, for example, the error statement was used in the default case of switch statements to check for illegal cache line states. Finally, one may specify *invariants* (Boolean conditions) that have to be true in every reachable state. For example, invariants were added to the SCI model to specify cache coherence properties.

  With the methods for detecting design errors described above, one is *not* able to specify *fairness properties*. Thus, livelocks cannot be detected and forward progress cannot be guaranteed with Murφ. This limitation will be lifted in the future. Note that the system whose state graph is shown in Fig. 1 has one deadlock (s_4) and one livelock (s_6, s_8) assuming that the system should always return to the startstate (s_0).

3 Overview of the Protocol

Shared-memory multiprocessors are commonly deemed to be easier to program than distributed multiprocessors, where the communication takes place via message passing. However, the latter are easier to implement in hardware. A solution to this problem is a distributed shared-memory multiprocessor, which provides shared memory at the software level, while the actual hardware implementation is a distributed message passing system. The IEEE Standard for Scalable Coherent Interface (SCI) includes a protocol for maintaining cache coherence among the distributed components in such a distributed shared-memory multiprocessor.

An SCI *node* may contain a *processor* – consisting of (multiple) *execution units* and a *cache* – and may contain a *memory*. The SCI nodes communicate via transactions, each consisting of a *request* packet and a *response* packet. In this simplified description, echo packets are not taken into account. A distributed shared-memory multiprocessor can be assembled out of these nodes.

The SCI Standard consists of both an English language description and an accurate definition in the C programming language. This C-code was also used for debugging the protocol when it was developed. Therefore, the C-code contains a multi-threaded execution environment for running simulations of the protocol. The SCI Standard contains many options that can each be enabled or disabled in actual hardware implementations. Thus, the protocol can be tailored to meet the needs of a specific implementation. Furthermore, two subsets of the (*full set*) cache coherence protocol – the *minimal set* and the *typical set* protocol – are defined for reducing the complexity of early implementations.

In a cache coherent SCI system, where snooping is not possible, for each memory line a list of all caches that have a copy of this line has to be maintained. In an SCI system, this "sharing list" is distributed among the system components. This is illustrated in Fig. 2. The left-hand side of this figure shows a sharing list of cache lines in processors B and C and the corresponding memory line. The pointers for the sharing lists are stored in additional bits (*tags*) in each memory and cache line. The current states of the memory and cache lines are also stored in these tags.

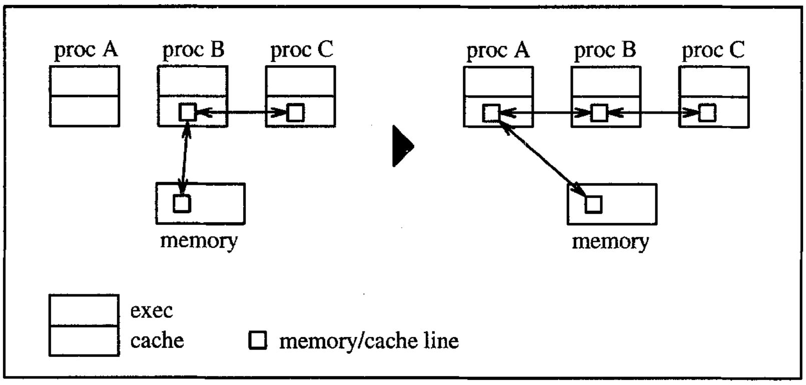

Fig. 2. A sharing list before and after a Load instruction

We now give an example of a typical execution sequence in the cache coherence protocol. If processor A on the left-hand side of Fig. 2 is executing a Load instruction and wants to read data from the memory line that is shared by processor B and C, it first issues an mread64 request packet to the memory and is notified in the response packet that processor B has the data. Assume the data in processor B's cache line is modified. Then, processor A sends a cread64 request packet to processor B's cache, obtains the data in the response packet and becomes the new *head* (owner) of the sharing list as shown on the right-hand side of Fig. 2.

In the typical set protocol, five instructions are defined by which a processor may access the shared memory. In addition to executing a *Load* or *Store* instruction, a processor may *Delete* itself from a sharing list, *Flush* (i.e. purge) the whole sharing list or *Lock* the memory line. According to the standard, these instructions are executed in four *phases* – namely allocate, setup, execute and cleanup.

The three distinct behaviors of processors, caches and memories are defined separately from each other in the C-code. According to this definition, the execution of the routines implementing cache and memory behavior is performed atomically. However, the execution of a routine modeling the execution of an instruction by a processor may be non-atomical. For example, after processor A in the above example sent out its mread64 request packet to the memory, processor B may start a Delete instruction, processor C may continue its Lock instruction in progress, etc.

4 The Modeling of the Protocol

The model of the SCI cache coherence protocol was constructed in three steps. These steps are clarified in the following subsections.

Abstraction

The goal of the abstraction or modeling was to extract the details of the SCI Standard that are important for the cache coherence protocol. Equivalently, this means that unnecessary details of the standard were omitted. Figure 3 shows an abstraction (model) of the SCI configuration. Details of the internal structures of the SCI nodes (processors and memories) are omitted. The transfer cloud connecting the system components is reliable. However, the order of packets is not preserved. Echo packets were not modeled, so a transaction consists of a request packet and a response packet. Only the fields of request and response packets were modeled that are actually used in the cache coherence part of the SCI Standard. In fact, chapter 4 of the SCI Standard [7] uses a similar abstraction to describe the cache coherence protocol.

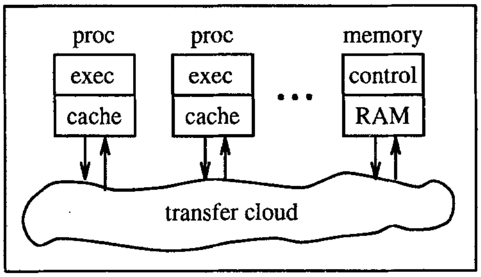

Fig. 3. Model of the SCI configuration

Simplification

The simplifications done were needed to make the model construction possible in a "finite" amount of time. The most significant of these simplifications was not modeling the full set cache coherence protocol, but restricting ourselves to the typical set protocol. In addition, only three of the processor/cache options were implemented in our model, namely DIRTY, FRESH and MODS. For the memory, the option MOP_FRESH was selected. The coherent instructions Load (with fetch options CO_FETCH, CO_LOAD and CO_STORE), Store, Delete and Flush were implemented.

Another simplification was not to model DMA reads and writes that are "allowed" in the typical set protocol. Furthermore, strong ordering constraints were assumed, so pipelining during the cleanup phase of an instruction was disabled. Finally, only one execution unit is attached to each processor.

Implementation

As mentioned before, scalability is crucial for successful verification. In implementing the model, we kept the following parameters scalable: the number of processors, the number of lines in each cache, the number of memories, the number of lines in each memory and the number of different data values. Besides that, each SCI processor/cache option, each instruction and each fetch option can be enabled or disabled by simply changing a constant declaration.

The model can be explained by three different *types* of behaviors or processes, namely memory, cache and processor. There can be many *individual* processes of each of these three types. For example, there is one individual process of type memory for each memory in the system. The model consists of all the resulting processes running (asynchronously) in parallel.

- Each *memory* has a simple request/response behavior, i.e. if there is a request packet for our memory in the transfer cloud, the memory reacts by sending a response packet. This is done atomically. However, before the memory responds to a request in the transfer cloud, any other process in the model may be active.
  The implementation in Murφ is done by using one rule, whose condition is true iff there is a request for the particular memory in the transfer cloud. The action of the rule deletes the request from the transfer cloud, performs the update of the accessed memory line's data field and tags and sends out a response on the transfer cloud.
- Each *cache* has a simple request/response behavior, similar to that of the memory.
- The *processor* behavior in our model is more complicated. A processor arbitrarily chooses a coherent instruction (Load, Store, Flush or Delete) to execute next, when the preceding instruction has completed. If the new instruction is, for example, a Load, the processor also chooses an arbitrary source address, an arbitrary cache line for cache misses and an arbitrary fetch option. Thus, we verified the cache coherence protocol while an arbitrary program is running in each processor.
  When a coherent instruction is in progress, the processor may several times send out a request on the transfer cloud to a cache or memory and then wait for the corresponding response. During this waiting time, any other system component may be active. Consequently, almost all SCI C-code routines

describing the processor behavior are often executed non-atomically[4]. For example, Table 1 shows a hierarchy of routines that may be called in a Flush instruction (main routine TypicalExecuteFlush()). The last routine called (CommonTransaction()) sends out the request packet and waits for the response packet. Thus, it is interruptable. Consequently, all routines shown in the table are interruptable, since each of them calls a subroutine that may be interrupted.

Table 1. Possible hierarchy of routines in a Flush instruction

TypicalExecuteFlush()
TypicalFlushSetup()
InvalidToOnlyDirty()
AttachLists()
CacheReadSrc()
CommonTransaction()

One goal in the implementation was to make the Murφ code similar to the SCI C-code. Specifically, we wanted to model each SCI routine by a Murφ routine. We implemented interruptable routines in Murφ by making it possible to re-enter them. If, for example, the Flush instruction initiates two transactions, the Murφ routine TypicalExecuteFlush() is called three times. One time initially and the other two times after a response packet arrived.
To implement routines that can be re-entered, the corresponding Murφ routines had to store their current state in special global variables. Corresponding state variables also have to occur in a hardware implementation of the protocol. Since they are explicit in our Murφ model, this model could be useful for hardware designers as well.

Finally, the implementation of the *transfer cloud* is described. We assumed that pipelining is disabled. Then, each processor can only have one outstanding request and (later) one non-processed response. Thus, in our implementation each processor has – as part of its state variables – a record for the "outgoing" request packet and another one for the "incoming" response packet. Each memory, for example, scans the request packet records of all processors to see if there is a request addressed to it in the transfer cloud.

5 Specifying Cache Coherence

The cache coherence property was specified in our Murφ model in two different ways:

[4] Routines that may be executed non-atomically will also be called *interruptable* in the following. Clearly, this is different from interrupts in the classical sense.

- First, the *SCI C-code* includes many assert statements to catch errors while running simulations with the built-in execution environment. Furthermore, the C-code contains several statements for the detection of memory-tag and cache-tag inconsistencies. We tried to include as many of these self-checks into our Murφ model as possible.
- Second, we added ***invariants*** to the Murφ model to specify more accurately cache coherence. These invariants imposed local conditions on the elements of sharing lists. We give two examples to clarify that:
  - If a cache line is in an unmodified stable state[5] (e.g. CS_ONLY_CLEAN), the data value in the cache line must be the same as the one in the corresponding memory line.
  - If a cache line is the head of a stable sharing list[6] (e.g. CS_HEAD_DIRTY), there must be a successor in the sharing list having the same memory address and pointing back to the head.

  Even though our conditions specifying cache coherence are not sufficient conditions for cache coherence, we expect them to be able to catch many of the errors that could occur. Furthermore, we are currently working to make the specification more accurate.

  In the process of specifying cache coherence with invariants, we first attempted relatively straightforward conditions. If these conditions were violated by any execution, we checked whether a protocol error was detected or whether a legal state violated our conditions. In the latter case, the conditions were relaxed to take into account this state.

  We would also like to specify fairness properties. For example, a processor who starts a Load instruction should finally get a copy of the data and finish the Load instruction. As mentioned in Sect. 2, specifying fairness properties is not possible in the current version of Murφ.

6 Errors Found During Verification

All the errors found so far occurred in system configurations with only two processors with one cache line each, one memory with one address and one data value ("zero bits of data") after examining a few thousand states in time on the order of minutes. Furthermore, only the protocol self-checks copied from the SCI C-code were triggered. None of our invariants was violated.

The largest example we ran had three processors with one cache line each, one memory with one address and two data values. The Load (fetch option CO_LOAD), Store, and Delete instruction were enabled. The cache/processor options DIRTY, FRESH and MODS were selected. The Murφ verifier examined 5.8 million states in 6.4 h, running on a Sun SPARCstation 20 and using 61 bytes per state. However, this example revealed no new errors.

We also ran examples in which we used more than one memory, address or cache line. None of these examples revealed new errors in the protocol. We only

[5] See Table 4–3 in [7] for a list of all stable cache-tag states.

[6] See Table 4–4 in [7] for a list of all stable sharing lists.

sometimes found errors in our Murφ model that were due to incorrect translation from the C-code to Murφ.

All protocol errors we found can be divided into three different classes, that are described in the following subsections. For each class, error examples are given. The full error list was sent to the SCI code-bugs reflector.

Omissions in the typical set protocol

The typical set protocol can be considered as a simplification of the full set protocol. All errors in the first class have in common that there were some program segments missing in the C-code of a routine of the typical set protocol but not in the corresponding routine of the full set protocol. Thus, these errors were easy to fix. The missing program segments were copied from the full set routine into the corresponding typical set routine.

For example, a processor executing a Load instruction may set the current cache line state to CI_ONLY_EXCL. This intermediate cache line state is not considered in the routine TypicalLoad(), which reports an error instead. However, the routine FullLoad() considers this case and correctly changes the cache line state to CS_ONLY_DIRTY[7].

Uninitialized variables

At some places in the SCI C-code uninitialized variables are accessed. During simulation runs using the C-code execution environment, these variables were presumably initialized to zero by code generated by the C-compiler – thus causing no problem. However, hardware implementations are less error-prone if all initializations are made explicit.

Instead of describing the situations when access to uninitialized variables occurs, we give two examples where variables have to be set to a defined value to avoid problems later.

- First, the routine MemoryAccessCoherent() should not return without setting the command nullified (cn) bit in the response packet to a defined value. We added an assignment to set the cn bit by default to zero.
- Second, the routine CacheRamAccess() should set the command.cmd field in the response packet to the default value SC_RESP00. This is especially important since the routine CommonTransaction() copies the incoming data into the cache line data field dependent on the command.cmd field of the incoming packet.

Logical protocol errors

The logical protocol errors we found required more subtle changes in the cache coherence protocol. These errors can be characterized as revealing flaws in the

[7] Actually, the state is set to CS_ONLYP_DIRTY_POP. However, this equals CS_ONLY_DIRTY since we assume pairwise sharing being disabled.

logical structure of the protocol. Note, that this error class and the previous one not only affect the correct operation of the typical set protocol, but also the full set protocol.

So far, we found the two logical protocol errors explained in the following. Only the first error has been fixed, the second one is currently being discussed with SCI working group members.

- When a Flush instruction is in progress in a processor, the current cache line state may be set to CS_HX_INVAL_OX. Assume a second processor who is a "TAIL_VALID" member of the sharing list now also starts a Flush instruction. Then, he sends a cread00.CC_PREV_VTAIL request to the first cache. When the first cache tries to respond to this request, an assert statement is violated in the cache's routine CacheTagUpdate() because none of the CacheTag...Update() routines has processed the request. The error can be fixed by adding CS_HX_INVAL_OX to the "blocking states" in the routine CacheTagBasicUpdate().
- While the errors described so far occurred with the three processor/cache options DIRTY, FRESH and MODS enabled, the following error was found with only options FRESH and MODS enabled. Table 2 shows the trace for this error, consisting of *actions*, starting from a state where both processors have invalid caches and ending in the error state.

Table 2. Error trace for the second logical protocol error

1.	proc1 starts Flush instruction, sends mread64.CACHE_DIRTY to memory
2.	proc2 starts Store instruction, sends mread64.CACHE_DIRTY to memory
3.	memory responds to proc2
4.	proc2 finishes Store instruction, cache2 becomes ONLY_DIRTY
5.	memory responds to proc1
6.	proc1 sends cread64.COPY_VALID to cache2
7.	cache2 responds to proc1, becomes TAIL_VALID
8.	proc1 continues Flush instruction, assertion POP_DIRTY is violated in HeadDirtyToFlushed()

Usually, the protocol leaves several choices at each state for the successor state. Thus, the longer an error trace, the more unlikely it becomes to detect that error by simulation means. The error trace in Table 2 was found by breadth-first search and is therefore as short as possible.

7 Conclusion

The most important experiences gained in verifying the SCI cache coherence protocol can be summarized as follows:

- The *abstraction* done in the modeling was relatively simple and straightforward. Modeling at a higher level of abstraction – for example, by mapping the many possible cache line states onto fewer abstract states – would have incurred the problem of comparing the abstract model with the real protocol defined in the C-code. To avoid this (severe) difficulty, we opted for simple abstractions.
- A careful *implementation* of the model in Murφ is important in fighting the state explosion problem. For example, we were able to reduce the number of reachable states by a factor of over 20 by just setting all state variables whose current values were no longer needed to fixed values.
- The Murφ system for formal verification should be viewed as a *debugging tool.* Verification was only possible for down-scaled versions of the model and thus total correctness cannot be guaranteed.
- It seems to be advantageous to *design and specify* complex protocols with the help of formal verification tools. First, the quality of the system is increased. Second, the time-consuming task of translating the description of the system into a "formal model" would be eliminated. Finally, our Murφ description is deemed to be easier to implement in hardware and not more complicated to understand than the original C-code.

8 Future Work

Our model of the SCI cache coherence protocol could be extended in several ways. However, one should keep in mind that these extensions worsen the state explosion.

- First, the model could be enlarged to cover the full set cache coherence protocol and all of the processor/cache options defined in the SCI Standard. Furthermore, the extensions of the SCI cache coherence protocol currently under development (for an overview see [10]) could be included in the model.
- Second, in the current version of our model, the processor/cache options have to be enabled/disabled by hand and they are identical for all nodes. For automatic verification, they should be selected automatically, arbitrarily and separately for each node.
- Finally, the model could be altered to allow multiple execution units for each processor and pipelining during the cleanup phase of an instruction. However, unlike the first two suggestions, this would require significant changes in the current model.

In our verification project, some errors were revealed that had not been found before. Verification methods are able to help in constructing better systems, but they have to keep pace with the increasing size of the systems. There are several ways by which the verifiable size of the model could be increased. First, it might be possible to abstract the sharing list from a low-level doubly-linked list to an abstract list. This way, the number of reachable states could be reduced. Second, there are some ways to increase the number of explorable states in the current Murφ system. Examples would be state compression [13] and on-the-fly methods [12]. Finally, symbolic methods to represent the set of reachable states [1] could yield further progress in the SCI verification.

Acknowledgements

We would like to thank Dave James for help with understanding details of the SCI cache coherence protocol and for discussing our results. Dave Gustavson helped in all other SCI related issues. Furthermore, we would like to thank Norris Ip for his assistance with symmetry problems, Irene Shen for her comments on a draft of this paper, and David Long for sharing his SCI description with us at the beginning of this project, which helped us with some initial modeling decisions.

References

1. J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.
2. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–5, 1992.
3. D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.
4. S. Gjessing, S. Krogdahl, and E. Munthe-Kaas. A top down approach to the formal specification of SCI cache coherence. In *Computer Aided Verification. 3rd International Workshop*, pages 83–91, 1991.
5. S. Gjessing and E. Munthe-Kaas. Formal specification of cache coherence in a shared memory multiprocessor. Research Report 158, Department of Informatics, University of Oslo, 1991.
6. D. B. Gustavson. The Scalable Coherent Interface and related standards projects. *IEEE Micro*, 12(1):10–22, 1992.
7. *IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI).*
8. C. N. Ip and D. L. Dill. Better verification through symmetry. In *11th International Conference on Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.
9. C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, 1993.
10. D. V. James. The Scalable Coherent Interface: Scaling to high-performance systems. In *Spring COMPCON*, pages 64–71, 1994.

11. D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi. Distributed-directory scheme: Scalable Coherent Interface. *Computer*, 23(6):74–7, 1990.
12. C. Jard and T. Jéron. Bounded-memory algorithms for verification on-the-fly. In *Computer Aided Verification. 3rd International Workshop*, pages 192–202, 1991.
13. U. Stern and D. L. Dill. Improved probabilistic verification by hash compaction. In *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, 1995.
14. L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC^{TM}-I. In *32nd Design Automation Conference*, pages 7–12, 1995.